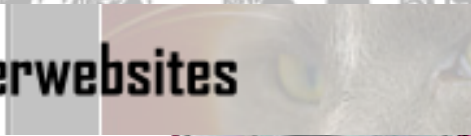


a-font



Download the coolest fonts for PC & MAC at a-font [\[Click Here\]](#)

bestwallpaperwebsites



Top 40 Wallpaper Websites on the Web [\[Click Here\]](#)

cellphonereviewed



Latest Cell Phones reviewed plus video reviews [\[Click Here\]](#)

coolestwebgamez



Coolest Online Web Flash Games, Addictive & Fun [\[Click Here\]](#)

desktop-it



High resolution wallpapers, the best online.. [\[Click Here\]](#)

ebook-portal



Free Ebooks & Magazines For download [\[Click Here\]](#)

For Vista Wallpapers



Amazing Wallpapers to go with your Windows Vista [\[Click Here\]](#)

geeKiee



Cool Fun Tech News & Bookmarks [\[Click Here\]](#)

latestsoftware



Latest Software Available For Download For Free [\[Click Here\]](#)

readytemplates



The Best Collection of Free Professional Website Templates for your website [\[Click Here\]](#)

for car wallpapers



A Collection of the Best Car Wallpapers Updated Often [\[Click Here\]](#)

Vista-Supported Software
download the latest vista software



Download Vista-Supported Software [\[Click Here\]](#)

Programmer to Programmer™



Excel® 2007 VBA

Programmer's Reference

John Green, Stephen Bullen, Rob Bovey, Michael Alexander



Updates, source code, and Wrox technical support at www.wrox.com

Excel® 2007 VBA Programmer's Reference

John Green
Stephen Bullen
Rob Bovey
Michael Alexander



Wiley Publishing, Inc.

Excel[®] 2007 VBA Programmer's Reference

Excel® 2007 VBA Programmer's Reference

John Green
Stephen Bullen
Rob Bovey
Michael Alexander



Wiley Publishing, Inc.

Excel® 2007 VBA Programmer's Reference

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2007 by Wiley Publishing, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-0-470-04643-2

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Legal Department, Wiley Publishing, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, (317) 572-3447, fax (317) 572-4355, or online at <http://www.wiley.com/go/permissions>.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: THE PUBLISHER AND THE AUTHOR MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION WARRANTIES OF FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES OR PROMOTIONAL MATERIALS. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR EVERY SITUATION. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING LEGAL, ACCOUNTING, OR OTHER PROFESSIONAL SERVICES. IF PROFESSIONAL ASSISTANCE IS REQUIRED, THE SERVICES OF A COMPETENT PROFESSIONAL PERSON SHOULD BE SOUGHT. NEITHER THE PUBLISHER NOR THE AUTHOR SHALL BE LIABLE FOR DAMAGES ARISING HEREFROM. THE FACT THAT AN ORGANIZATION OR WEBSITE IS REFERRED TO IN THIS WORK AS A CITATION AND/OR A POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE AUTHOR OR THE PUBLISHER ENDORSES THE INFORMATION THE ORGANIZATION OR WEBSITE MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. FURTHER, READERS SHOULD BE AWARE THAT INTERNET WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ.

For general information on our other products and services or to obtain technical support, please contact our Customer Care Department within the U.S. at (800) 762-2974, outside the U.S. at (317) 572-3993 or fax (317) 572-4002.

Library of Congress Cataloging-in-Publication Data

Excel 2007 VBA programmer's reference / John Green ... [et al.].

p. cm.

Includes index.

ISBN 978-0-470-04643-2 (paper/website)

1. Microsoft Excel (Computer file) 2. Business—Computer programs. I. Green, John, 1945-

HF5548.4.M523E92988 2007

005.54—dc22

2007004976

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

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

About the Authors

John Green lives and works in Sydney, Australia, as an independent computer consultant, specializing in Excel and Access. He has 35 years of computing experience, a Chemical Engineering degree, and an MBA.

He wrote his first programs in FORTRAN, took a part in the evolution of specialized planning languages on mainframes and, in the early '80s, became interested in spreadsheet systems, including 1-2-3 and Excel.

John established his company, Execuplan Consulting, in 1980, specializing in developing computer-based planning applications and in training. He has led training seminars for software applications and operating systems both in Australia and overseas.

John has had regular columns in a number of Australian magazines and has contributed chapters to a number of books including *Excel Expert Solutions* and *Using Visual Basic for Applications 5*. He also co-authored *Professional Excel Development* with Stephen Bullen and Rob Bovey.

From 1995 to 2005 he was accorded the status of MVP (Most Valuable Professional) by Microsoft for his contributions to the CompuServe Excel forum and MS Internet newsgroups.

John Green contributed the Introduction, Chapters 1–11, 13, 15–17, and 19 to this book.

Stephen Bullen lives in Woodford Green, London, England, with his partner Clare, daughter Becky, and their dogs, Fluffy and Charlie. He has two other daughters, Jane and Katie, from his first marriage.

A graduate of Oxford University, Stephen has an MA in Engineering, Economics, and Management, providing a unique blend of both business and technical skills. He has been providing Excel consulting and application development services since 1994, originally as an employee of Price Waterhouse Management Consultants and later as an independent consultant trading under the names of Business Modelling Solutions Limited and Office Automation Limited. Stephen now works for Barclays Capital in London, developing trading systems for complex exotic derivative products.

The Office Automation web site, www.oaltd.co.uk, provides a number of helpful and interesting utilities, examples, tips and techniques to help in your use of Excel and development of Excel applications.

As well as co-authoring previous editions of the *Excel VBA Programmer's Reference*, Stephen co-authored *Professional Excel Development*.

In addition to his consulting and writing assignments, Stephen actively supports the Excel user community in Microsoft's peer-to-peer support newsgroups and the Daily Dose of Excel blog. In recognition of his knowledge, skills and contributions, Microsoft has awarded him the title of Most Valuable Professional each year since 1996.

Stephen Bullen contributed Chapters 14, 18, 24–27, and Appendix B to this book.

Rob Bovey is president of Application Professionals, a software development company specializing in Microsoft Office, Visual Basic, and SQL Server applications. He brings many years' experience creating financial, accounting, and executive information systems for corporate users to Application Professionals. You can visit the Application Professionals web site at www.appspro.com.

Rob developed several add-ins shipped by Microsoft for Microsoft Excel and co-authored the *Microsoft Excel 97 Developers Kit* and *Professional Excel Development*. He earned his Bachelor of Science degree from The Rochester Institute of Technology and his MBA from the University of North Carolina at Chapel Hill. He is a Microsoft Certified Systems Engineer (MCSE) and a Microsoft Certified Solution Developer (MCSD). Microsoft has awarded him the title of Most Valuable Professional each year since 1995.

Rob Bovey contributed Chapters 20–22 to this book.

Michael Alexander is a Microsoft Certified Application Developer (MCAD) with more than 14 years' experience consulting and developing office solutions. He parlayed his experience with VBA and VB into a successful consulting practice in the private sector, developing middleware and reporting solutions for a wide variety of industries. He currently lives in Frisco, Texas, where he serves as a Senior Program Manager for a top technology firm. Michael is the author of several books on Microsoft Access and Excel, and is the principle behind DataPig Technologies, where he shares Access and Excel knowledge with the Office community.

Michael Alexander contributed Chapters 12 and 23 and Appendices A and C to this book.

Credits

Acquisitions Editor

Katie Mohr

Development Editor

Brian Herrmann

Technical Editor

Dick Kusleika

Production Editor

William A. Barton

Copy Editor

Kim Cofer

Editorial Manager

Mary Beth Wakefield

Production Manager

Tim Tate

Vice President and Executive Group Publisher

Richard Swadley

Vice President and Executive Publisher

Joseph B. Wikert

Project Coordinator

Jennifer Theriot

Graphics and Production Specialists

Carrie A. Foster

Denny Hager

Joyce Haughey

Jennifer Mayberry

Barbara Moore

Barry Offringa

Heather Ryan

Quality Control Technicians

Jessica Kramer

Christine Pingleton

Proofreading and Indexing

Kevin Broccoli

Sean Medlock

Contents

Acknowledgments	xxi
Introduction	xxiii
Chapter 1: Primer in Excel VBA	1
Using the Macro Recorder	2
Recording Macros	2
Running Macros	6
The Visual Basic Editor	8
Other Ways to Run Macros	11
User-Defined Functions	17
Creating a UDF	18
What UDFs Cannot Do	21
The Excel Object Model	21
Objects	22
Getting Help	27
Experimenting in the Immediate Window	29
The VBA Language	30
Basic Input and Output	30
Calling Functions and Sub Procedures	35
Parentheses and Argument Lists	37
Variable Declaration	38
Scope and Lifetime of Variables	40
Variable Type	42
Object Variables	45
Making Decisions	47
Looping	50
Arrays	55
Run-Time Error-Handling	59
Summary	62
Chapter 2: The Application Object	63
Globals	63
The Active Properties	64
Display Alerts	65
Screen Updating	66

Contents

Evaluate	66
InputBox	68
StatusBar	70
SendKeys	70
OnTime	71
OnKey	72
Worksheet Functions	73
Caller	74
Summary	75
Chapter 3: Workbooks and Worksheets	77
<hr/>	
The Workbooks Collection	77
Getting a Filename from a Path	78
Files in the Same Directory	81
Overwriting an Existing Workbook	81
Saving Changes	82
The Sheets Collection	83
Worksheets	83
Copy and Move	85
Grouping Worksheets	87
The Window Object	89
Synchronizing Worksheets	90
Summary	91
Chapter 4: Using Ranges	93
<hr/>	
Activate and Select	93
Range Property	95
Shortcut Range References	96
Ranges on Inactive Worksheets	96
Range Property of a Range Object	97
Cells Property	97
Cells Used in Range	98
Ranges of Inactive Worksheets	99
More on the Cells Property of the Range Object	99
Single-Parameter Range Reference	101
Offset Property	102
Resize Property	103
SpecialCells Method	105
Last Cell	105
Deleting Numbers	107

CurrentRegion Property	108
End Property	110
Referring to Ranges with End	110
Summing a Range	111
Columns and Rows Properties	112
Areas	113
Union and Intersect Methods	115
Empty Cells	115
Transferring Values between Arrays and Ranges	118
Deleting Rows	121
Summary	123
Chapter 5: Using Names	125
Naming Ranges	127
Using the Name Property of the Range Object	128
Special Names	128
Storing Values in Names	129
Storing Arrays	130
Hiding Names	131
Working with Named Ranges	132
Searching for a Name	133
Searching for the Name of a Range	135
Determining which Names Overlap a Range	136
Summary	139
Chapter 6: Data Lists	141
Structuring the Data	141
Sorting a Range	142
Older Excel Versions	144
Creating a Table	144
Sorting a Table	145
AutoFilter	146
AutoFilter Object	147
Filter Object	148
Date Custom Filter	148
Adding Combo Boxes	149
Copying the Visible Rows	153
Finding the Visible Rows	154
Advanced Filter	156
Data Form	158
Summary	159

Contents

Chapter 7: PivotTables	161
Creating a PivotTable Report	162
PivotCaches	165
PivotTables Collection	165
PivotFields	166
CalculatedFields	170
PivotItems	171
Grouping	171
Visible Property	175
CalculatedItems	176
PivotCharts	177
External Data Sources	178
Summary	180
Chapter 8: Charts	181
Chart Sheets	182
The Recorded Macro	184
Adding a Chart Sheet Using VBA Code	184
Embedded Charts	185
Using the Macro Recorder	186
Adding an Embedded Chart Using VBA Code	186
Editing Data Series	187
Defining Chart Series with Arrays	190
Converting a Chart to Use Arrays	193
Determining the Ranges Used in a Chart	194
Chart Labels	195
Summary	196
Chapter 9: Event Procedures	199
Worksheet Events	199
Enable Events	200
Worksheet Calculate	201
Chart Events	202
Before Double Click	202
Workbook Events	205
Save Changes	206
Headers and Footers	207
Summary	208

Chapter 10: Adding Controls	209
Form and ActiveX Controls	209
ActiveX Controls	210
Scrollbar Control	211
Spin Button Control	211
CheckBox Control	212
Option Button Controls	212
Forms Controls	214
Dynamic ActiveX Controls	216
Controls on Charts	220
Summary	221
Chapter 11: Text Files and File Dialog	223
Opening Text Files	223
Writing to Text Files	224
Reading Text Files	226
Writing to Text Files Using Print	227
Reading Data Strings	229
Flexible Separators and Delimiters	230
FileDialog	233
FileDialogFilters	235
FileDialogSelectedItems	235
Dialog Types	235
Execute Method	235
MultiSelect	236
Summary	238
Chapter 12: Working with XML and the Open XML File Formats	239
The Basics of Using XML Data in Excel	240
XML Fundamentals	240
Consuming XML Data Directly	246
Creating and Managing Your Own XML Maps	249
Using VBA to Program XML Processes	253
Programming XML Maps	253
Leveraging DOM and XPath to Manipulate XML Files	258
Using VBA to Program Open XML Files	265
Programming Open XML Files with VBA	266
Programmatically Zipping an Excel Container	267
Summary	272

Chapter 13: UserForms	273
Displaying a UserForm	273
Creating a UserForm	275
Directly Accessing Controls in UserForms	277
Stopping the Close Button	281
Maintaining a Data List	282
Modeless UserForms	288
Progress Indicator	288
Variable UserForm Name	291
Summary	291
Chapter 14: RibbonX	293
Overview	293
Prerequisites	294
Adding the Customizations	294
XML Structure	295
RibbonX and VBA	298
Control Types	299
Basic Controls	299
Container Controls	300
Control Attributes	301
Control Callbacks	303
Managing Control Images	305
Other RibbonX Elements, Attributes, and Callbacks	307
Sharing Controls among Multiple Workbooks	308
Updating Controls at Run Time	309
Hooking Built-In Controls	311
RibbonX in Dictator Applications	312
Customizing the Office Menu	312
Customizing the QAT	313
Controlling Tabs, Tab Sets, and Groups	313
Dynamic Controls	314
dropDown, comboBox, and gallery	315
dynamicMenu	315
CommandBar Extensions for the Ribbon	316
RibbonX Limitations	317
Summary	318

Chapter 15: Command Bars	319
Toolbars, Menu Bars, and Popups	320
Excel's Built-in Command Bars	322
Controls at All Levels	325
Facelds	328
Creating New Menus	330
The OnAction Macros	332
Passing Parameter Values	333
Deleting a Menu	334
Creating a Toolbar	335
Popup Menus	338
Showing Popup Command Bars	342
Table-Driven Command Bar Creation	344
Summary	354
Chapter 16: Class Modules	355
Creating Your Own Objects	356
Property Procedures	357
Creating Collections	359
Class Module Collection	360
Encapsulation	363
Trapping Application Events	363
Embedded Chart Events	365
A Collection of UserForm Controls	368
Referencing Classes Across Projects	370
Summary	371
Chapter 17: Add-ins	373
Hiding the Code	374
Creating an Add-in	374
Closing Add-ins	375
Code Changes	376
Saving Changes	377
Interface Changes	377
Installing an Add-in	379
AddinInstall Event	381
Removing an Add-in from the Add-ins List	381
Summary	382

Contents

Chapter 18: Automation Add-Ins and COM Add-Ins	383
Automation Add-Ins	383
A Simple Add-In — Sequence	384
Registering Automation Add-Ins with Excel	385
Using Automation Add-Ins	386
Introducing the IDTExtensibility2 Interface	388
COM Add-Ins	394
The IDTExtensibility2 Interface (Continued)	395
Registering a COM Add-In with Excel	395
The COM Add-In Designer	396
Summary	409
Chapter 19: Interacting with Other Office Applications	411
Establishing the Connection	411
Late Binding	412
Early Binding	414
Opening a Document in Word	416
Accessing an Active Word Document	417
Creating a New Word Document	418
Access and ADO	419
Access, Excel, and, Outlook	420
Better than Mail Merge	423
Readable Document Variables	428
Summary	430
Chapter 20: Data Access with ADO	431
An Introduction to Structured Query Language (SQL)	431
The SELECT Statement	432
The INSERT Statement	434
The UPDATE Statement	434
The DELETE Statement	435
An Overview of ADO	436
The Connection Object	437
The Recordset Object	441
The Command Object	445
Using ADO in Microsoft Excel Applications	447
Using ADO with Microsoft Access	448
Using ADO with Microsoft SQL Server	454
Using ADO with Non-Standard Data Sources	463
Summary	468

Chapter 21: Managing External Data	469
The External Data User Interface	469
Get External Data	470
Manage Connections	471
The QueryTable and ListObject	472
A QueryTable from a Relational Database	472
A Query Table Associated with a ListObject	475
QueryTables and Parameter Queries	476
QueryTables from Web Queries	479
A QueryTable from a Text File	482
Creating and Using Connection Files	484
The WorkbookConnection Object and the Connections Collection	487
External Data Security Settings	489
Summary	490
Chapter 22: The Trust Center and Document Security	491
The Trust Center	491
Trusted Publishers	492
Trusted Locations	492
Add-ins	494
ActiveX Settings	495
Macro Settings	497
Message Bar	498
External Content	499
Privacy Options	501
Automating Document Inspection	503
The RemoveDocumentInformation Method	503
The DocumentInspectors Collection	505
Summary	506
Chapter 23: Browsing OLAP Data Sources with Excel	507
Analyzing OLAP Data via Pivot Tables	508
Connecting to an OLAP Data Source	508
Browsing the OLAP Data Source	510
Understanding the MDX behind OLAP-based Pivot Tables	512
The Basics of MDX	513
Browsing OLAP Data Sources without Pivot Tables	517
Using ADO to Return Flattened Recordsets	517
Using ADO MD to Get Cube Schema Information	518
Creating an Inventory of Dimensions, Hierarchies, and Levels	519

Contents

Creating Offline Cubes	521
Creating an Offline Cube Manually	521
Using the CreateCubeFile Method	521
Creating an Offline Cube Using ADO MD and VBA	522
Summary	523
Chapter 24: Excel and the Internet	525
What Can the Internet Do for You?	526
Using the Internet for Storing Workbooks	526
Using the Internet as a Data Source	527
Opening Web Pages as Workbooks	528
Using Web Queries	528
Parsing Web Pages for Specific Information	530
Using the Internet to Publish Results	531
Setting Up a Web Server	532
Saving Worksheets as Web Pages	532
Creating Interactive Web Pages	533
Using the Internet as a Communication Channel	533
Communicating with a Web Server	534
Summary	536
Chapter 25: International Issues	537
Changing Windows Regional Settings and the Office 2007 UI Language	537
Responding to Regional Settings and the Windows Language	538
Identifying the User's Regional Settings and Windows Language	538
VBA Conversion Functions from an International Perspective	539
Interacting with Excel	545
Sending Data to Excel	545
Reading Data from Excel	548
The Rules for Working with Excel	548
Interacting with Users	549
Paper Sizes	549
Displaying Data	549
Interpreting Data	550
The xxxLocal Properties	550
The Rules for Working with Your Users	551
Excel 2007's International Options	552
Features That Don't Play by the Rules	554
The OpenText Function	555
The SaveAs Function	556
The ShowDataForm Sub Procedure	556

Pasting Text	557
PivotTable Calculated Fields and Items, and Conditional Format and Data Validation Formulas	557
Web Queries	558
=TEXT() Worksheet Function	558
The Range.Value, Range.Formula, and Range.FormulaArray Properties	559
The Range.AutoFilter Method	559
The Range.AdvancedFilter Method	559
The Application.Evaluate, Application.ConvertFormula, and Application.ExecuteExcel4Macro Functions	560
Responding to Office 2007 Language Settings	560
Where Does the Text Come From?	560
Identifying the Office UI Language Settings	562
Creating a Multilingual Application	562
Working in a Multilingual Environment	564
The Rules for Developing a Multilingual Application	565
Some Helpful Functions	565
The bWinToNum Function	566
The bWinToDate Function	566
The sFormatDate Function	567
The ReplaceHolders Function	568
Summary	568
Chapter 26: Programming the VBE	571
<hr/>	
Identifying VBE Objects in Code	572
The VBE Object	572
The VBProject Object	572
The VBComponent Object	573
The CodeModule Object	574
The CodePane Object	574
The Designer Object	574
Starting Up	575
Adding Menu Items to the VBE	576
Working with Workbooks	580
Working with Code	589
Working with UserForms	594
Working with References	598
COM Add-ins	599
Summary	600

Chapter 27: Programming with the Windows API	601
Anatomy of an API Call	602
Interpreting C-Style Declarations	603
Constants, Structures, Handles, and Classes	606
What If Something Goes Wrong?	609
Wrapping API Calls in Class Modules	611
Some Example Classes	616
A High-Resolution Timer Class	616
Class Module CHighResTimer	616
Freeze a UserForm	618
A System Info Class	619
Modifying UserForm Styles	622
Window Styles	623
The CFormChanger Class	624
Resizable UserForms	625
Absolute Changes	626
Relative Changes	627
The CFormResizer Class	628
Summary	634
Appendix A: Excel 2007 Object Model	635
Appendix B: VBE Object Model	971
Appendix C: Office 2007 Object Model	995
Index	1079

Acknowledgments

John Green

Thanks to Katie Mohr and Michael Alexander for getting us back together, and thanks to Brian Herrmann for melding us into a coherent whole.

Dick Kusleika deserves special mention as our technical editor. He has saved us from some embarrassment and suggested numerous improvements in the examples and text. Thank you, Dick.

I would like to thank Michael Beale for seeding some of the examples of interaction with other Office applications.

Finally, a heartfelt thank you to my fellow authors. I have handled the basics and Michael, Rob, and Stephen have supplied the benefits of their specialized knowledge in the higher-level topics to take us further than I would have ever dared on my own.

Stephen Bullen

First and foremost, I'd like to thank my long-suffering girlfriend, Clare, for putting up with all the late nights and lonely evenings she endured while I wrote this update. Thanks also goes to Mike Alexander and Katie Mohr for their efforts in resurrecting the original author team to write this update to the book, and to John and Rob for agreeing to do it—your professionalism leaves me humbled.

Dick Kusleika is the unsung hero of this book. While the four authors could concentrate on our own chapters, Dick had to carefully read every word and check its accuracy. The credit for the amazingly high quality of this work goes to him, while any remaining errors are ours.

Of course, without the Excel team at Microsoft, we wouldn't have had anything to write about, so thanks goes to David Gainer and his team for crafting an amazing update to a quite mature product, and for being so open with the Excel MVPs and wider public over the past few years. The Ribbon is the biggest change that has happened to Office for many years and Jensen Harris and Savraj Dhanjal and their teams have done a brilliant job in designing the Ribbon's UI and programmability model, respectively. I'd particularly like to thank them for listening to the (sometimes harsh) criticism from the beta testers, and for updating their designs in response.

Last, I'd like to thank you, the reader, for buying this book, writing the five-star reviews on Amazon and recommending it to all your friends and colleagues!

Mike Alexander

I would like to first thank the original authors—John Green, Stephen Bullen, and Rob Bovey—for agreeing to reclaim their work. Believe me when I say that these men are very well respected among professional Excel developers, and it is an absolute honor to be associated with their work.

Acknowledgments

A big thank you goes to Katie Mohr for joining me in lobbying to get the original author team back on board. It is safe to say that without her efforts, this title would not be the superb product it is today. I would also like to thank Brian Herrmann and the professionals at Wiley for all of their time and resources in helping this ambitious title come to fruition.

Dick Kusleika is definitely the “the fifth Beatle” of this book. Dick clearly put a lot of time and effort into keeping us honest and ensuring that our work is as clean as possible. A solid technical editor is paramount for an all-encompassing reference like this one, and Dick Kusleika really came through for all of us.

A very special thank you to Mary for putting up with all of my crazy projects. The royalty checks are in the mail, my love.

Introduction

Excel made its debut on the Macintosh in 1985 and has never lost its position as the most popular spreadsheet application in the Mac environment. In 1987, Excel was ported to the PC, running under Windows. It took many years for Excel to overtake Lotus 1-2-3, which was one of the most successful software systems in the history of computing at that time.

A number of spreadsheet applications enjoyed success prior to the release of the IBM PC in 1981. Among these were VisiCalc and Multiplan. VisiCalc started it all, but fell by the wayside early on. Multiplan was Microsoft's predecessor to Excel, using the R1C1 cell addressing which is still available as an option in Excel. But it was 1-2-3 that shot to stardom very soon after its release in 1982 and came to dominate the PC spreadsheet market.

Early Spreadsheet Macros

1-2-3 was the first spreadsheet application to offer spreadsheet, charting, and database capabilities in one package. However, the main reason for its runaway success was its macro capability. Legend has it that the 1-2-3 developers set up macros as a debugging and testing mechanism for the product. It is said that they only realized the potential of macros at the last minute, and included them in the final release pretty much as an afterthought.

Whatever their origins, macros gave non-programmers a simple way to become programmers and automate their spreadsheets. They grabbed the opportunity and ran. At last they had a measure of independence from the computer department.

The original 1-2-3 macros performed a task by executing the same keystrokes that a user would use to carry out the same task. It was, therefore, very simple to create a macro because there was virtually nothing new to learn to progress from normal spreadsheet manipulation to programmed manipulation. All you had to do was remember what keys to press and write them down. The only concessions to traditional programming were eight extra commands, the `/x` commands. The `/x` commands provided some primitive decision-making and branching capabilities, a way to get input from a user, and a way to construct menus.

One major problem with 1-2-3 macros was their vulnerability. The multi-sheet workbook had not yet been invented and macros had to be written directly into the cells of the spreadsheet they supported, along with input data and calculations. Macros were at the mercy of the user. For example, they could be inadvertently disrupted when a user inserted or deleted rows or columns. Macros were also at the mercy of the programmer. A badly designed macro could destroy itself quite easily while trying to edit spreadsheet data.

Despite the problems, users reveled in their newfound programming ability and millions of lines of code were written in this cryptic language, using arcane techniques to get around its many limitations. The world came to rely on code that was often badly designed, nearly always poorly documented, and at all times highly vulnerable, often supporting enterprise-critical control systems.

The XLM Macro Language

The original Excel macro language required you to write your macros in a macro sheet that was saved in a file with an `.xlm` extension. In this way, macros were kept separate from the worksheet, which was saved in a file with an `.xls` extension. These macros are now often referred to as XLM macros, or Excel 4 macros, to distinguish them from the VBA macro language introduced in Excel Version 5.

The XLM macro language consisted of function calls, arranged in columns in the macro sheet. There were many hundreds of functions necessary to provide all the features of Excel and allow programmatic control. The XLM language was far more sophisticated and powerful than the 1-2-3 macro language, even allowing for the enhancements made in 1-2-3 Releases 2 and 3. However, the code produced was not much more intelligible.

The sophistication of Excel's macro language was a two-edged sword. It appealed to those with high programming aptitude, who could tap the language's power, but was a barrier to most users. There was no simple relationship between the way you manually operated Excel and the way you programmed it. There was a very steep learning curve involved in mastering the XLM language.

Another barrier to Excel's acceptance on the PC was that it required Windows. The early versions of Windows were restricted by limited access to memory, and Windows required much more horsepower to operate than DOS. The Graphical User Interface was appealing, but the tradeoffs in hardware cost and operating speed were perceived as problems.

Lotus made the mistake of assuming that Windows was a flash in the pan, soon to be replaced by OS/2, and did not bother to plan a Windows version of 1-2-3. Lotus put its energy into 1-2-3/G, a very nice GUI version of 1-2-3 that only operated under OS/2. This one-horse bet was to prove the undoing of 1-2-3.

By the time it became clear that Windows was here to stay, Lotus was in real trouble as it watched users flocking to Excel. The first attempt at a Windows version of 1-2-3, released in 1991, was really 1-2-3 Release 3 for DOS in a thin GUI shell. Succeeding releases have closed the gap between 1-2-3 and Excel, but have been too late to stop the almost universal adoption of Microsoft Office by the market.

Excel 5

Microsoft made a brave decision to unify the programming code behind its Office applications by introducing *VBA* (Visual Basic for Applications) as the common macro language in Office. Excel 5, released in 1993, was the first application to include VBA. It was gradually introduced into the other Office applications in subsequent versions of Office. Excel, Word, Access, PowerPoint, and Outlook all use VBA as their macro language in Office.

Since the release of Excel 5, Excel has supported both the XLM and the VBA macro languages, and the support for XLM should continue into the foreseeable future, but has decreased in significance as users switch to VBA.

VBA is an object-oriented programming language that is identical to the Visual Basic programming language in the way it is structured and in the way it handles objects. If you learn to use VBA in Excel, you know how to use it in the other Office applications.

The Office applications differ in the objects they expose to VBA. To program an application, you need to be familiar with its *object model*. The object model is a hierarchy of all the objects that you find in the application. For example, part of the Excel object model tells us that there is an `Application` object that contains a `Workbook` object that contains a `Worksheet` object that contains a `Range` object.

VBA is somewhat easier to learn than the XLM macro language, is more powerful, is generally more efficient, and allows you to write well-structured code. You can also write badly structured code, but by following a few principles, you should be able to produce code that is readily understood by others and is reasonably easy to maintain.

In Excel 5, VBA code was written in modules, which were sheets in a workbook. Worksheets, chart sheets, and dialog sheets were other types of sheets that could be contained in an Excel 5 workbook.

A module is really just a word-processing document with some special characteristics that help you write and test code.

Excel 97

In Excel 97, Microsoft introduced some dramatic changes in the VBA interface and some changes in the Excel object model. From Excel 97 onward, modules are not visible in the Excel application window and modules are no longer objects contained by the `Workbook` object. Modules are contained in the VBA project associated with the workbook and can only be viewed and edited in the Visual Basic Editor (VBE) window.

In addition to the standard modules, class modules were introduced, which allow you to create your own objects and access application events. `CommandBars` were introduced to replace menus and toolbars, and `UserForms` replaced dialog sheets. Like modules, `UserForms` can only be edited in the VBE window. As usual, the replaced objects are still supported in Excel, but are considered to be hidden objects and are not documented in the Help screens.

In previous versions of Excel, objects such as buttons embedded in worksheets could only respond to a single event, usually the `Click` event. Excel 97 greatly increased the number of events that VBA code can respond to and formalized the way in which this is done by providing event procedures for the workbook, worksheet, and chart sheet objects. For example, in Excel 2007 workbooks have 29 events they can respond to, such as `BeforeSave`, `BeforePrint`, and `BeforeClose`. Excel 97 also introduced ActiveX controls that can be embedded in worksheets and `UserForms`. ActiveX controls can respond to a wide range of events such as `GotFocus`, `MouseMove`, and `DoubleClick`.

The VBE provides users with much more help than was previously available. For example, as you write code, pop-ups appear with lists of appropriate methods and properties for objects, and arguments and parameter values for functions and methods. The *Object Browser* is much better than previous versions, allowing you to search for entries, for example, and providing comprehensive information on intrinsic constants.

Introduction

Microsoft has provided an Extensibility library that makes it possible to write VBA code that manipulates the VBE environment and VBA projects. This makes it possible to write code that can directly access code modules and UserForms. It is possible to set up applications that indent module code or export code from modules to text files, for example.

Excel 2000

Excel 2000 did not introduce dramatic changes from a VBA programming perspective. There were a large number of improvements in the Office 2000 and Excel 2000 user interfaces and improvements in some Excel features such as PivotTables. A new PivotChart feature was added. Web users benefited the most from Excel 2000, especially through the ability to save workbooks as web pages. There were also improvements for users with a need to share information, through new online collaboration features.

One long-awaited improvement for VBA users was the introduction of modeless UserForms. Previously, Excel only supported modal dialog boxes, which take the focus when they are onscreen so that no other activity can take place until they are closed. Modeless dialog boxes allow the user to continue with other work while the dialog box floats above the worksheet. Modeless dialog boxes can be used to show a “splash” screen when an application written in Excel is loaded and to display a progress indicator while a lengthy macro runs.

Excel 2002

Excel 2002 also introduced only incremental changes. Once more, the major improvements were in the user interface rather than in programming features. Microsoft continued to concentrate on improving web-related features to make it easier to access and distribute data using the Internet. New features that can be useful for VBA programmers included a new `Protection` object, SmartTags, RTD (Real Time Data), and improved support for XML.

The `Protection` object allows selective control over the features that are accessible to users when you protect a worksheet. You can decide whether users can sort, alter cell formatting, or insert and delete rows and columns, for example. There is also a new `AllowEditRange` object that you can use to specify which users can edit specific ranges and whether they must use a password to do so. You can apply different combinations of permissions to different ranges.

SmartTags allow Excel to recognize data typed into cells as having special significance. For example, Excel 2002 can recognize stock market abbreviations, such as MSFT for Microsoft Corporation. When Excel sees an item like this, it displays a SmartTag symbol that has a pop-up menu. You can use the menu to obtain related information, such as the latest stock price or a summary report on the company. Microsoft provides a kit that allows developers to create new SmartTag software to make data available throughout an organization or across the Internet.

RTD allows developers to create sources of information that users can draw from. Once you establish a link to a worksheet, changes in the source data are automatically passed on. An obvious use for this is to obtain stock prices that change in real time during the course of trading. Other possible applications include the ability to log data from scientific instruments or industrial process controllers.

Improved XML support meant that it was getting easier to create applications that exchange data through the Internet and intranets. As everyone becomes more dependent on these burgeoning technologies, XML support becomes of increasing importance.

Excel 2003

Excel 2003 continued to introduce new web-orientated features, including improved support for XML and improved online help and the ability to share and update data using Windows SharePoint Services.

Excel 2003 introduced corrected versions of a number of Excel's statistical functions.

The List feature was introduced to allow easier management of a database table. Lists make it easier to sort, filter, and edit data. Lists can also be integrated into SharePoint to share data via the Internet.

New features were introduced to enhance document sharing and management of access rights. Side-by-side comparison of workbooks was also introduced.

Excel 2007

Excel 2007 represents the greatest change in Excel since Excel 97. The most impact will be made by the new user interface, which uses the Ribbon as the primary navigation tool, replacing menus and toolbars. Although the Ribbon is probably much easier to digest for new users, it means that experienced users need to be re-educated. From a developer's point of view, the Ribbon is a major challenge requiring a whole new approach in application interfaces and a completely new set of programming rules.

Excel 2007 lifts many of the old limits, supporting 1,048,576 rows and 16,384 columns, for example. There are many changes to the way features are accessed so that PivotTables and charts are more accessible and easier to manipulate, as are many other features.

The List feature of Excel 2003, which handles database tables, has become the Table feature in Excel 2007 and is easier to use and has more capabilities. Sorting and filtering have been redesigned. You can sort on up to 64 keys simultaneously, for example. Enhancements have also been made in the range of external data sources that are now accessible, and the ways in which the data is accessed have been improved.

New file formats are used in Excel 2007, which are not compatible with previous versions although data can be saved back to older formats with the loss of any new features. If you want to have VBA code saved with a workbook, the format of the file is different compared with a standard workbook file.

Security concepts have been redesigned, introducing the Trust Center. You can now designate folders as "trusted," and macros in these folders will be allowed to run without needing digital certificates.

For a VBA programmer there are a number of new objects to be discovered and new concepts to be learned.

Excel 2007 VBA Programmer's Reference

This book is aimed squarely at Excel users who want to harness the power of the VBA language in their Excel applications. At all times, the VBA language is presented in the context of Excel, not just as a general application programming language.

The pages that follow have been loosely divided into three sections:

- ❑ Primer (Chapter 1)
- ❑ Working with Specific Objects (Chapters 2–27)
- ❑ Object Model References (Appendices A–C)

The Primer has been written for those who are new to VBA programming and the Excel object model. It introduces the VBA language and the features of the language that are common to all VBA applications. It explains the relationship between collections, objects, properties, methods, and events and shows how to relate these concepts to Excel through its object model. It also shows how to use the Visual Basic Editor and its multitude of tools, including how to obtain help.

The middle section of the book takes the key objects in Excel and shows, through many practical examples, how to go about working with those objects. The techniques presented have been developed through the exchange of ideas of many talented Excel VBA programmers over many years and show the best way to gain access to workbooks, worksheets, charts, ranges, and so on. The emphasis is on efficiency—that is, how to write code that is readable and easy to maintain and that runs at maximum speed. In addition, the chapters devoted to accessing external databases detail techniques for accessing data in a range of formats.

The final four chapters of the book address the following advanced issues: linking Excel to the Internet, writing code for international compatibility, programming the Visual Basic Editor, and how to use the functions in the Win32 API (Windows 32-bit Application Programming Interface).

Finally, the appendices are a comprehensive reference to the Excel 2007 object model, as well as the Visual Basic Editor and Office object models. All the objects in the models are presented together with all their properties, methods, and events. I trust that this book will become a well-thumbed resource that you can dig into, as needed, to reveal that elusive bit of code that you must have right now.

Version Issues

Previous editions of this book were able to cover all versions of Excel from Excel 97 onward, because the changes in the Excel object model and user interface were relatively minor. The changes in Excel 2007 have meant that it is no longer possible to do this without filling the book with complicated alternatives. This book applies to Excel 2007.

What You Need to Use this Book

Nearly everything discussed in this book has examples with it. All the code is written out and there are plenty of screenshots where they are appropriate. The version of Windows you use is not important. It is

important to have a full installation of Excel and, if you want to try the more advanced chapters involving communication between Excel and other Office applications, you will need a full installation of Office. Make sure your installation includes access to the Visual Basic Editor and the VBA Help files. It is possible to exclude these items during the installation process.

Note that Chapter 18 requires you to have VB6 installed because it covers the topics of COM Addins. Chapter 23 requires you to have IIS 5.0, SQL Server 2000, and SQL Server 2005 installed in order to interact with OLAP data sources.

Conventions Used

This book uses a number of different styles of text and layout in the book, to help differentiate between different kinds of information. Here are some of the styles and an explanation of what they mean:

These boxes hold important, not-to-be forgotten, mission-critical details that are directly relevant to the surrounding text.

Background information, asides, and references appear in text like this.

- ❑ *Important words* are italicized
- ❑ Words that appear on the screen, such as menu options, are capitalized—for example, the Tools menu.
- ❑ All object names, function names, and other code snippets are in this style: `SELECT`.

Code that is new or important is presented like this:

```
SELECT CustomerID, ContactName, Phone
FROM Customers
```

Code that you've seen before or has little to do with the matter being discussed, looks like this:

```
SELECT ProductName FROM Products
```

In Case of a Crisis...

There are a number of places you can turn to if you encounter a problem. The best source of information on all aspects of Excel is your peers. You can find them in a number of newsgroups across the Internet. Try pointing your newsreader to the following site where you will find all of the authors actively participating:

- ❑ `msnews.microsoft.com`

Subscribe to `microsoft.public.excel.programming` or any of the groups that appeal to you. You can submit questions and generally receive answers within an hour or so.

Introduction

Stephen Bullen and Rob Bovey maintain very useful web sites, where you will find a great deal of information and free downloadable files, at the following addresses:

- ❑ www.oaltd.co.uk
- ❑ www.appspro.com

John Walkenbach maintains another useful site at:

- ❑ www.j-walk.com

Wrox can be contacted directly at:

- ❑ www.wrox.com—for downloadable source code and support
- ❑ http://p2p.wrox.com/list.asp?list=vba_excel—for open Excel VBA discussion

Other useful Microsoft information sources can be found at:

- ❑ www.microsoft.com/office/—for up-to-the-minute news and support
- ❑ <http://msdn.microsoft.com/office/>—for developer news and good articles about how to work with Microsoft products
- ❑ www.microsoft.com/technet—for Microsoft Knowledge Base articles, security information, and a bevy of other more admin-related items

Feedback

We've tried, as far as possible, to write this book as though we were sitting down next to each other. We've made a concerted effort to keep it from getting "too heavy" while still maintaining a fairly quick pace. We'd like to think that we've been successful at it, but encourage you to e-mail us and let us know what you think one way or the other. Constructive criticism is always appreciated, and can only help future versions of this book. You can contact us either by e-mail (support@wrox.com) or via the Wrox web site.

Questions?

Seems like there are always some, eh? From the previous edition of this book, we received hundreds of questions. We have tried to respond to every one of them as best as possible. What we ask is that you give it your best shot to understand the problem based on the explanations in the book.

If the book fails you, then you can either e-mail Wrox (support@wrox.com) or us personally (greenj@bigpond.net.au, RobBovey@AppsPro.com, Stephen@oaltd.co.uk). You can also ask questions on the `vba_excel` list at <http://p2p.wrox.com>. Wrox has a dedicated team of support staff and we personally *try* (no guarantees!) to answer all the mail that comes to them. For the previous book, we responded to about 98% of the questions asked—but life sometimes becomes demanding enough that we can't get to them all. Just realize that the response may take a few days (because we get an awful lot of mail).

Primer in Excel VBA

This chapter is intended for those who are not familiar with Excel and the Excel macro recorder, or who are inexperienced with programming using the Visual Basic language. If you are already comfortable with navigating around the features provided by Excel, have used the macro recorder, and have a working knowledge of Visual Basic and the Visual Basic Editor, you might want to skip straight to Chapter 2.

If this is not the case, this chapter has been designed to provide you with the information you need to be able to move on comfortably to the more advanced features presented in the following chapters. Specifically, this chapter covers the following topics:

- The Excel macro recorder
- User-defined functions
- The Excel object model
- VBA programming concepts

Excel VBA is a programming application that allows you to use Visual Basic code to run the many features of the Excel package, thereby allowing you to customize your Excel applications. Units of VBA code are often referred to as *macros*. More formal terminology is covered in this chapter, but you will continue to see the term *macro* as a general way to refer to any VBA code.

In your day-to-day use of Excel, if you carry out the same sequence of commands repetitively, you can save a lot of time and effort by automating those steps using macros. If you are setting up an application for other users who don't know much about Excel, you can use macros to create buttons and dialog boxes to guide them through your application as well as automate the processes involved.

If you are able to perform an operation manually, you can use the *macro recorder* to capture that operation. This is a very quick and easy process and requires no prior knowledge of the VBA language. Many Excel users record and run macros and feel no need to learn about VBA.

Chapter 1: Primer in Excel VBA

However, the recorded results might not be very flexible, in that the macro can only be used to carry out one particular task on one particular range of cells. In addition, the recorded macro is likely to run much more slowly than code written by someone with knowledge of VBA. To set up interactive macros that can adapt to change and also run quickly, and to take advantage of more advanced features of Excel such as customized dialog boxes, you need to learn about VBA.

Don't get the impression that we are dismissing the macro recorder. The macro recorder is one of the most valuable tools available to VBA programmers. It is the fastest way to generate working VBA code, but you must be prepared to apply your own knowledge of VBA to edit the recorded macro to obtain flexible and efficient code. A recurring theme in this book is recording an Excel macro and then showing how to adapt the recorded code.

In this chapter, you learn how to use the macro recorder and you see all the ways Excel provides to run your macros. You see how to use the *Visual Basic Editor* to examine and change your macros, thus going beyond the recorder and tapping into the power of the VBA language and the *Excel object model*.

You can also use VBA to create your own worksheet functions. Excel comes with hundreds of built-in functions, such as `SUM` and `IF`, which you can use in cell formulas. However, if you have a complex calculation that you use frequently and that is not included in the set of standard Excel functions — such as a tax calculation or a specialized scientific formula — you can write your own *user-defined function*.

Using the Macro Recorder

Excel's macro recorder operates very much like the recorder that stores the greeting on your telephone answering machine. To record a greeting, you first prepare yourself by rehearsing the greeting to ensure that it says what you want. Then you switch on the recorder and deliver the greeting. When you have finished, you switch off the recorder. You now have a recording that automatically plays when you leave a call unanswered.

Recording an Excel macro is very similar. You first rehearse the steps involved and decide at what points you want to start and stop the recording process. You prepare your spreadsheet, switch on the Excel recorder, carry out your Excel operations, and switch off the recorder. You now have an automated procedure that you and others can reproduce at the press of a button.

Recording Macros

Say you want a macro that types six month names as three-letter abbreviations, Jan to Jun, across the top of your worksheet, starting in cell B1. I know this is rather a silly macro because you could do this easily with an AutoFill operation, but this example will serve to show you some important general concepts:

- ❑ First, think about how you are going to carry out this operation. In this case, it is easy — you will just type the data across the worksheet. Remember, a more complex macro might need more rehearsals before you are ready to record it.

- ❑ Next, think about when you want to start recording. In this case, you should include the selection of cell B1 in the recording, because you want to always have Jan in B1. If you don't select B1 at the start, you will record typing Jan into the active cell, which could be anywhere when you play back the macro.
- ❑ Next, think about when you want to stop recording. You might first want to include some formatting such as making the cells bold and italic, so you should include that in the recording. Where do you want the active cell to be after the macro runs? Do you want it to be in the same cell as Jan, or would you rather have the active cell in column A or column B, ready for your next input? Assume that you want the active cell to be A2, at the completion of the macro, so you will select A2 before turning off the recorder.
- ❑ Now you can set up your screen, ready to record.

In this case, start with an empty worksheet with cell A1 selected. If you can't see the Developer tab above the Ribbon, you will need to click the round Microsoft Office button that you can see in the top-left corner of the Excel screen shown in Figure 1-1. Click Excel Options at the bottom of the dialog box and select Personalize. Select the checkbox for Show Developer tab in the Ribbon and click OK. Now you can select the Developer section of the Ribbon and click Record Macro to display the Record Macro dialog box, shown in Figure 1-1.

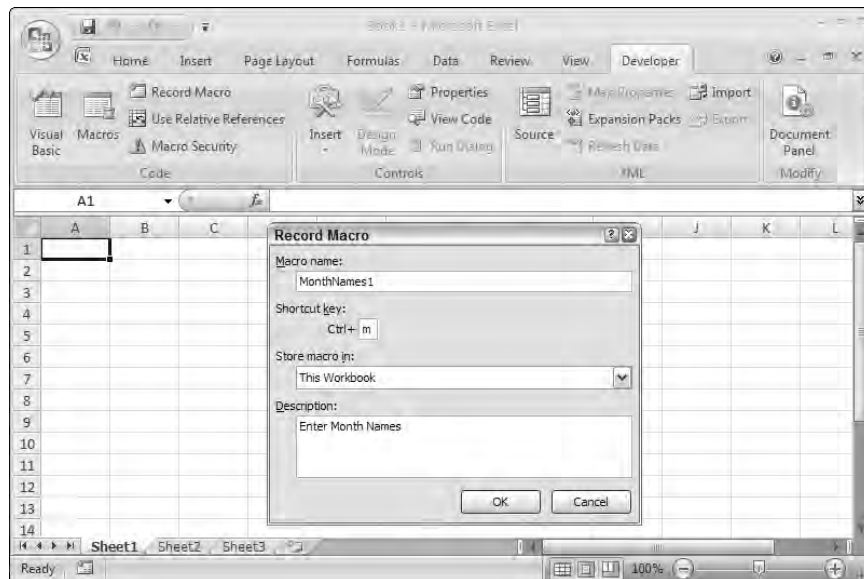


Figure 1-1

In the Macro name: box, replace the default entry, such as Macro1, with the name you want for your macro. The name should start with a letter and contain only letters, numbers, and the underscore character, with a maximum length of 255 characters. The macro name must not contain special characters such as exclamation points (!) or question marks (?), nor should it contain blank spaces. It is also best to use a short but descriptive name that you will recognize later. You can use the underscore character to separate words, but it is easy to just use capitalization to distinguish words.

Chapter 1: Primer in Excel VBA

Call the macro `MonthNames1`, because you will create another version later.

In the **Shortcut key:** box, you can type in a single letter. This key can be pressed later, while holding down the **Ctrl** key, to run the macro. Use a lowercase `m`. Alternatively, you can use an uppercase `M`. In this case, when you later want to run the macro, you need to use the keystroke combination **Ctrl+Shift+M**. It is not mandatory to provide a shortcut key; you can run a macro in a number of other ways, as you will see.

In the **Description:** box, you can add text that will be added as comments to the macro. These lines will appear at the top of your macro code. They have no significance to VBA, but provide you and others with information about the macro.

All Excel macros are stored in workbooks. You are given a choice regarding where the recorded macro will be stored. The **Store macro in:** combo box lists three possibilities. If you choose **New Workbook**, the recorder will open a new empty workbook for the macro. **Personal Macro Workbook** refers to a special hidden workbook, which is discussed in a moment. Choose **This Workbook** to store the macro in the currently active workbook.

When you have filled in the **Record Macro** dialog box, click the **OK** button. You will see a new **Stop Recording** button appear on the left side of the status bar at the bottom of the screen, as shown in **Figure 1-2**. You will also notice that the **Start Recording** button in the Ribbon has been replaced by a new **Stop Recording** button.

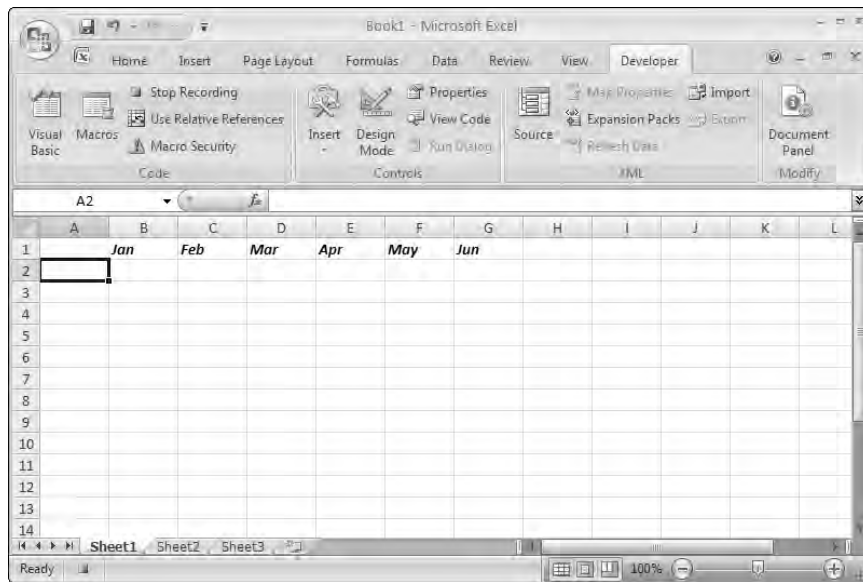


Figure 1-2

You should now click cell **B1**, type in `Jan`, and fill in the rest of the cells as shown in **Figure 1-2**. Then select **B1:G1** and click the **Bold** and **Italic** buttons on the **Home** tab of the **Ribbon**. Click the **A2** cell and then stop the recorder. You can stop the recorder by clicking the **Stop Recording** button on the **Ribbon** or by clicking the **Stop Recording** button on the status bar.

It is important to remember to stop the recorder. If you leave the recorder on and try to run the recorded macro, you can go into a loop where the macro runs itself over and over again. If this does happen to you, or any other error occurs while testing your macros, hold down the Ctrl key and press the Break key to interrupt the macro. You can then end the macro or go into debug mode to trace errors. You can also interrupt a macro with the Esc key, but it is not as effective as Ctrl+Break for a macro that is pausing for input.

You could now save the workbook, but before you do so, you should determine the file type you need and consider the security issues covered in the next section.

You can't save the workbook as the default Excel Workbook (*.xlsx) type. This file format does not allow macros to be included. You can save the workbook as an Excel Macro-Enabled Workbook (*.xlsm) type, which is in XLM format, or you can save it as an Excel Binary Workbook (*.xlsb) type, which is in a binary format. Neither of these file types is compatible with previous versions of Excel. Another alternative is to save the workbook as an Excel 97-2003 Workbook (*.xls) type, which produces a workbook compatible with Excel versions from Excel 97 through Excel 2003.

Macro Security

To develop macros with minimum interruption, work with Office 2007's security restrictions. Without getting into the complications of digitally signing your workbooks, you have a couple of simple options. Select the Developer tab on the Ribbon and click the Macro Security button. You will see the Trust Center dialog box, where you can select Macro Settings. Here you can enable all macros. This is not recommended because it leaves you wide open to macro viruses.

A better alternative is to nominate a specific directory as a trusted location. Click Trusted Locations to the left of the Trust Center dialog box. You probably already have a number of trusted locations, including your XLSTART directory and templates directories. Use the Add new location button to specify a suitable directory for storing your workbooks.

You should now save the workbook containing the newly recorded macro into the trusted location. Click the Microsoft Office button and select Save As. In the Save as type drop-down, select the .xlsm type and save the workbook in the trusted location as Recorder.xlsm.

If you can't see the file extensions, such as .xlsm, in the Save As dialog box, you should open Windows Explorer, click the Tools menu, and choose Folder Options. In the View tab, remove the check against Hide extensions for known file types.

The Personal Macro Workbook

If you choose to store your recorded macro in the Personal Macro Workbook, the macro is added to a special file called Personal.xlsm, which is a hidden file that is saved in your Excel Startup directory when you close Excel. This means that Personal.xlsm is automatically loaded when you launch Excel and, therefore, its macros are always available for any other workbook to use.

Chapter 1: Primer in Excel VBA

If `Personal.xlsb` does not already exist, the recorder will create it for you. You can use the Unhide button on the View tab of the Ribbon to see this workbook in the Excel window, but it is seldom necessary or desirable to do this because you can examine and modify the `Personal.xlsb` macros in the Visual Basic Editor window.

An exception where you might want to make `Personal.xlsb` visible is if you need to store data in its worksheets. You can hide it again, after adding the data, with the Hide button on the View tab of the Ribbon. If you are creating a general-purpose utility macro, which you want to be able to use with any workbook, store it in `Personal.xlsb`. If the macro relates to just the application in the current workbook, store the macro with the application.

Running Macros

To run the macro, either use another worksheet in the `Recorder.xlsm` workbook or open a new empty workbook, leaving `Recorder.xlsm` open in memory. You can only run macros that are in open workbooks, but they can be run from within any other open workbook.

You can run the macro by pressing `Ctrl+M`, the shortcut you assigned at the start of the recording process. You can also run the macro by clicking the Macros button in the View tab of the Ribbon or by clicking the Macros button in the Developer tab of the Ribbon. Both buttons open the dialog box shown in Figure 1-3. You can run the macro by double-clicking the macro name, or by selecting the macro name and clicking Run.



Figure 1-3

The same dialog box can be opened by pressing `Alt+F8`.

Shortcut Keys

You can change the shortcut key assigned to a macro using the Macro dialog box shown in Figure 1-3. Select the macro name and click Options. This opens the dialog box shown in Figure 1-4.



Figure 1-4

It is possible to assign the same shortcut key to more than one macro in the same workbook using this dialog box (although the dialog box that appears when you start the macro recorder will not let you assign a shortcut that is already in use).

It is also quite likely that two different workbooks could contain macros with the same shortcut key assigned. If this happens, which macro runs when you use the shortcut? The macro that comes first alphabetically.

Shortcuts are appropriate for macros that you use frequently, especially if you prefer to keep your hands on the keyboard. It is worth memorizing the shortcuts so you won't forget them if you use them regularly. Shortcuts are *not* appropriate for macros that are run infrequently or are intended to make life easier for less experienced users of your application. It is better to assign meaningful names to those macros and run them from the Macro dialog box. Alternatively, they can be run from buttons that you add to the worksheet. You learn how to do this shortly.

Absolute and Relative Recording

When you run `MonthNames1`, the macro returns to the same cells you selected while typing in the month names. It doesn't matter which cell is active when you start; if the macro contains the command to select cell B1, that is what it selects. The macro selects B1 because you recorded in absolute record mode. The alternative, relative record mode, remembers the position of the active cell relative to its previous position. If you have cell A10 selected, turn on the recorder, and go on to select B10, the recorder notes that you moved one cell to the right, rather than noting that you selected cell B10.

Record a second macro called `MonthNames2`. There will be three differences in this macro compared with the previous one:

- Click the Use Relative References button on the Developer tab of the Ribbon. You can do this before you start recording or while you are recording.
- Do not select the Jan cell before typing. You want your recorded macro to type Jan into the active cell when you run the macro.
- Finish by selecting the cell under Jan, rather than A2, just before turning off the recorder.

Chapter 1: Primer in Excel VBA

Start with an empty worksheet and select the B1 cell. Turn on the macro recorder and specify the macro name as `MonthNames2`. Enter the shortcut as uppercase M—the recorder won't let you use lowercase m again. Click OK and select the Use Relative References button on the Developer tab of the Ribbon.

Type Jan and the other month names, as you did when recording `MonthNames1`. Select cells B1:G1 and click the Bold and Italic buttons on the Home tab of the Ribbon.

Make sure you select B1:G1 from left to right, so that B1 is the active cell. There is a small kink in the recording process that can cause errors in the recorded macro if you select cells from right to left or from bottom to top. Always select from the top-left corner when recording relatively. This has been a problem with all versions of Excel VBA.

Finally, select cell B2, the cell under Jan, and turn off the recorder.

Before running `MonthNames2`, select a starting cell, such as A10. You will find that the macro now types the month names across row 10, starting in column A, and finishes by selecting the cell under the starting cell.

Before you record a macro that selects cells, you need to think about whether to use absolute or relative reference recording. If you are selecting input cells for data entry, or for a print area, you will probably want to record with absolute references. If you want to be able to run your macro in different areas of your worksheet, you will probably want to record with relative references.

If you are trying to reproduce the effect of the Ctrl+arrow keys to select the last cell in a column or row of data, you should record with relative references. You can even switch between relative and absolute reference recording in the middle of a macro, if you want. You might want to select the top of a column with an absolute reference, switch to relative references, and use Ctrl+down arrow to get to the bottom of the column and an extra down arrow to go to the first empty cell.

Excel 2000 was the first version of Excel to let you successfully record selecting a block of cells of variable height and width using the Ctrl key. If you start at the top left-hand corner of a block of data, you can hold down the Shift and Ctrl keys and press the down arrow and then the right arrow to select the whole block (as long as there are no gaps in the data). If you record these operations with relative referencing, you can use the macro to select a block of different dimensions. Previous versions of Excel recorded an absolute selection of the original block size, regardless of recording mode.

The Visual Basic Editor

It is now time to see what has been going on behind the scenes. If you want to understand macros, be able to modify your macros, and tap into the full power of VBA, you need to know how to use the Visual Basic Editor (VBE). The VBE runs in its own window, separate from the Excel window. You can activate it in many ways.

First, you can activate it by clicking the Visual Basic button on the Developer tab of the Ribbon. You can also activate it by holding down the Alt key and pressing the F11 key. Alt+F11 acts as a toggle, taking

you between the Excel window and the VBE window. If you want to edit a specific macro, you can use the Macros button on the Developer tab of the Ribbon or the Play Macro button on the left of the status bar to open the Macro dialog box, select the macro, and click the Edit button. The VBE window will look something like Figure 1-5.

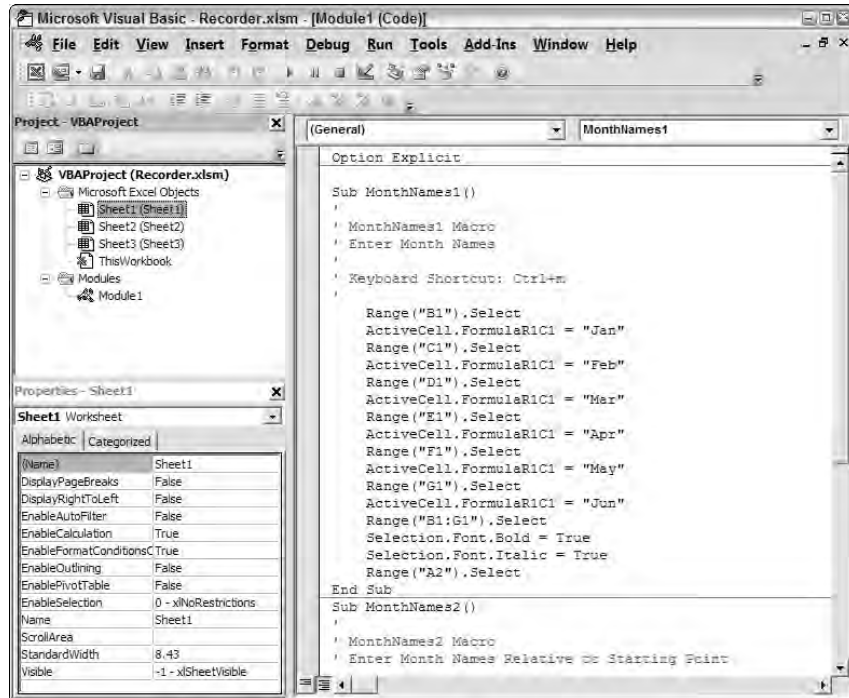


Figure 1-5

It is quite possible that you will see nothing but the menu bar when you switch to the VBE window. If you can't see the toolbars, use View ⇄ Toolbars and click the Standard toolbar. Use View ⇄ Project Explorer and View ⇄ Properties Window to show the windows on the left. If you can't see the code module on the right, double-click the icon for Module1 in the Project Explorer window.

Code Modules

All macros reside in code modules like the one on the right of the VBE window in Figure 1-5. There are two types of code modules — standard modules and class modules. The one you see on the right is a standard module. You can use class modules to create your own objects. You won't need to know much about class modules until you are working at a very advanced level. See Chapter 15 for more details on how to use class modules.

Some class modules have already been set up for you. They are associated with each worksheet in your workbook, and there is one for the entire workbook. You can see them in the Project Explorer window, in the folder called Microsoft Excel Objects. You will find out more about them later in this chapter.

Chapter 1: Primer in Excel VBA

You can add as many code modules to your workbook as you like. The macro recorder has inserted the one named `Module1`. Each module can contain many macros. For a small application, you would probably keep all your macros in one module. For larger projects, you can organize your code better by filing unrelated macros in separate modules.

Procedures

In VBA, macros are referred to as procedures. There are two types of procedures — sub procedures and function procedures. You will find out about function procedures in the next section. The macro recorder can only produce sub procedures. You can see the `MonthNames1` sub procedure set up by the recorder in Figure 1-5.

Sub procedures start with the keyword `Sub`, followed by the name of the procedure and opening and closing parentheses. The end of a sub procedure is marked by the keywords `End Sub`. Although it is not mandatory, the code within the sub procedure is normally indented to make it stand out from the start and end of the procedure, so that the whole procedure is easier to read. Further indentation is normally used to distinguish sections of code such as `If` tests and looping structures. For example:

```
If ActiveCell.Value = 10 Then
    ActiveCell.Font.Bold = True
End If
```

Any lines starting with a single quote are comment lines, which are ignored by VBA. They are added to provide documentation, which is a very important component of good programming practice. You can also add comments to the right of lines of code. For example:

```
Range("B1").Select 'Select the B1 cell
```

At this stage, the code may not make perfect sense, but you should be able to make out roughly what is going on. If you look at the code in `MonthNames1`, you will see that cells are being selected and then the month names are assigned to the active cell formula. You can edit some parts of the code, so if you had spelled a month name incorrectly, you could fix it; or you could identify and remove the line that sets the font to bold; or you can select and delete an entire macro.

Notice the differences between `MonthNames1` and `MonthNames2`. `MonthNames1` selects specific cells such as `B1` and `C1`. `MonthNames2` uses `Offset` to select a cell that is zero rows down and one column to the right from the active cell. Already, you are starting to get a feel for the VBA language.

The Project Explorer

The Project Explorer is an essential navigation tool. In VBA, each workbook contains a project. The Project Explorer displays all the open projects and the component parts of those projects, as you can see in Figure 1-6.

You can use the Project Explorer to locate and activate the code modules in your project. You can double-click a module icon to open and activate that module. You can also insert and remove code modules in the Project Explorer. Right-click anywhere in the Project Explorer window, and from the context menu select `Insert` to add a new standard module, class module, or `UserForm`.

To remove `Module1`, right-click it and choose `Remove Module1`. Note that you can't do this with the modules associated with workbook or worksheet objects. You can also export the code in a module to a separate text file, or import code from a text file.

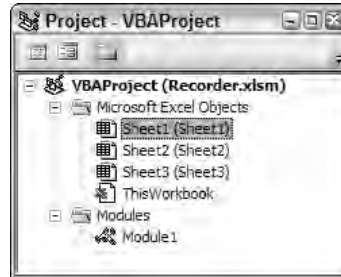


Figure 1-6

The Properties Window

The Properties window shows you the properties that can be changed at design time for the currently active object in the Project Explorer window. For example, if you click Sheet1 in the Project Explorer, the Sheet1 properties are displayed in the Properties window, as shown in Figure 1-7. The `ScrollArea` property has been set to `A1:D10`, to restrict users to that area of the worksheet.

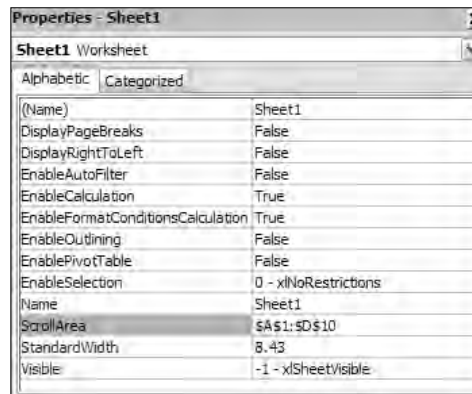


Figure 1-7

You can get to the help screen associated with any property very easily. Just select the property, such as the `ScrollArea` property, which is selected in Figure 1-7, and press F1.

Other Ways to Run Macros

You have seen how to run macros with shortcuts and how to run them from the Ribbon and status bar macro buttons. Neither method is particularly friendly. You need to be very familiar with your macros to be comfortable with these techniques. You can make your macros much more accessible by attaching them to buttons.

If the macro is worksheet-specific, and will only be used in a particular part of the worksheet, then it is suitable to use a button that has been embedded in the worksheet at the appropriate location. If you want to be able to use a macro in any worksheet or workbook and in any location in a worksheet, it is appropriate to attach the macro to a button on the Quick Access Toolbar.

Chapter 1: Primer in Excel VBA

There are many other objects that you can attach macros to, including combo boxes, list boxes, scrollbars, checkboxes, and option buttons. These are all referred to as controls. (See Chapter 11 for more information on controls.) You can also attach macros to graphic objects in the worksheet, such as shapes created with the Shapes button on the Insert tab of the Ribbon.

Worksheet Buttons

Excel 2007 has two different sets of controls that can be embedded in worksheets. One set has been inherited from the Forms toolbar in previous versions, and the other has been inherited from the Control ToolBox toolbar in previous versions. The Forms toolbar appeared in Excel 5 and 95. The Forms controls can be embedded in a worksheet and are also used with Excel 5 and 95 dialog sheets to create dialog boxes. Excel 97 introduced the newer ActiveX controls on the Control ToolBox toolbar. You can embed ActiveX controls in a worksheet or use them on UserForms, in the VBE, to create dialog boxes.

To create controls in Excel 2007, select the Developer tab on the Ribbon. In the Controls group, click the Insert button to open the window shown in Figure 1-8.

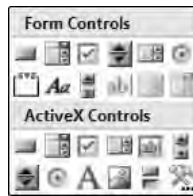


Figure 1-8

For compatibility with the older versions of Excel, both sets of controls and techniques for creating dialog boxes are supported in Excel 97 and higher. If you have no need to maintain backward compatibility with Excel 5 and 95, you can use just the ActiveX controls.

Forms Controls

A good reason for using the Forms controls is that they are simpler to use than the ActiveX controls, because they do not have all the features of ActiveX controls. For example, Forms controls can only respond to a single, predefined event, which is usually the mouse-click event. ActiveX controls can respond to many events, such as a mouse click, a double-click, or pressing a key on the keyboard. If you have no need of such features, you might prefer the simplicity of Forms controls. To create a Forms button in a worksheet, click the top-left button in the Controls dialog box, opened from the Insert button on the Developer tab of the Ribbon.

You can now draw the button in your worksheet by clicking where you want a corner of the button to appear and dragging to where you want the diagonally opposite corner to appear. The Assign Macro dialog box will appear as shown in Figure 1-9, and you can select the macro to attach to the button.

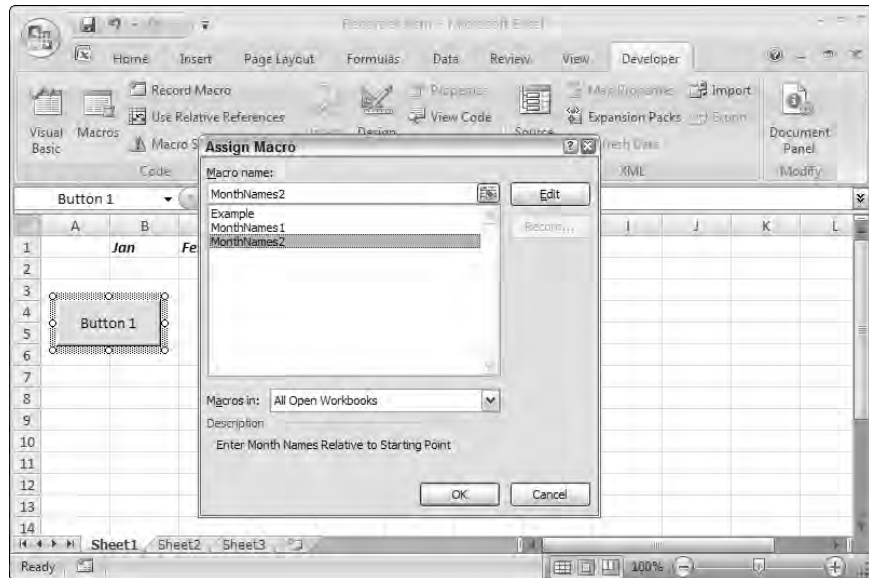


Figure 1-9

Click OK to complete the assignment. You can then edit the text on the button to give a more meaningful indication of its function. After you click a worksheet cell, you can click the button to run the attached macro. If you need to edit the button and it is not already selected, right-click it to select the control and display a context menu. If you don't want the context menu, hold down Ctrl and left-click or right-click the button to select it. (Don't drag the mouse while you hold down Ctrl, or you will create a copy of the button.)

If you want to align the button with the worksheet gridlines, hold down Alt as you draw it with the mouse. If you have already drawn the button, select it and hold down Alt as you drag any of the white boxes that appear on the corners and edges of the button. The edge or corner you drag will snap to the nearest gridline.

ActiveX Controls

To create an ActiveX command button control, click the top-left button in the ActiveX Controls section of the Controls dialog box, opened from the Insert button on the Developer tab of the Ribbon. When you draw your button in the worksheet, you enter into design mode. When you are in design mode, you can select a control with a left-click and edit it. You must turn off design mode if you want the new control to respond to events. You can do this by clicking the Design Mode button on the Developer tab of the Ribbon so it is no longer highlighted. Figure 1-10 shows the Design Mode button as it appears when design mode is active, after the insertion of the ActiveX control.

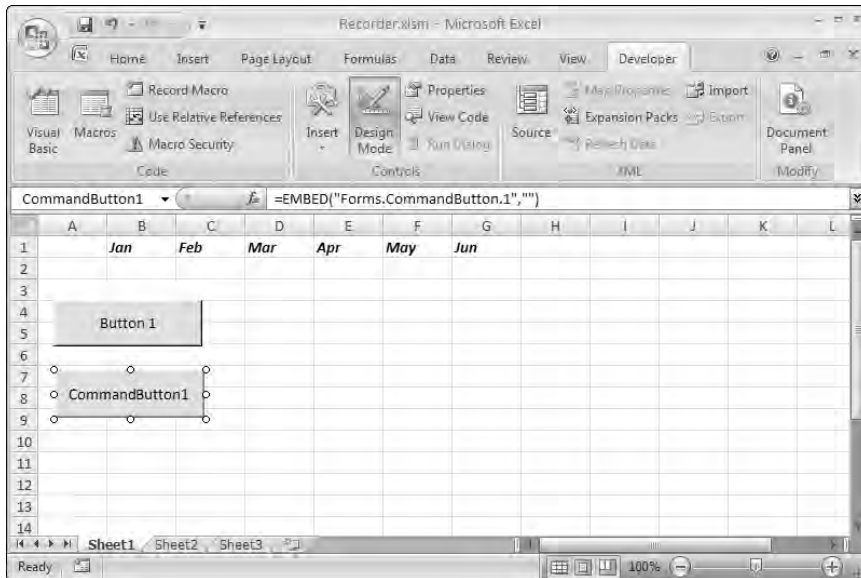


Figure 1-10

You are not prompted to assign a macro to the ActiveX command button, but you do need to write a click-event procedure for the button. An event procedure is a sub procedure that is executed when, for example, you click a button. To do this, make sure you are still in design mode and double-click the command button to open the VBE window and display the code module behind the worksheet. The Sub and End Sub statement lines for your code will have been inserted in the module, and you can add in the code necessary to run the `MonthNames2` macro, as shown in Figure 1-11.

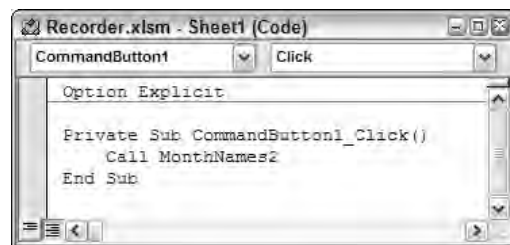


Figure 1-11

To run this code, switch back to the worksheet, turn off design mode, and click the command button.

If you want to make changes to the command button, you need to return to design mode by clicking the Design Mode button. You can then select the command button and change its size and position on the worksheet. You can also display its properties by right-clicking it and choosing Properties to display the window shown in Figure 1-12.

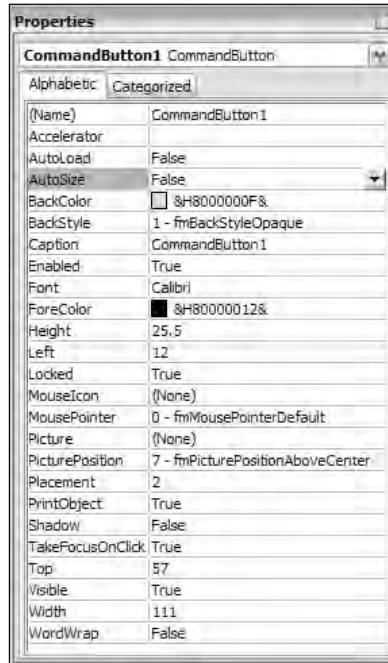


Figure 1-12

To change the text on the command button, change the `Caption` property. You can also set the font for the caption and the foreground and background colors. If you want the button to work satisfactorily in Excel 97, it is a good idea to change the `TakeFocusOnClick` property from its default value of `True` to `False`. If the button takes the focus when you click it, Excel 97 does not allow you to assign values to some properties, such as the `NumberFormat` property of the `Range` object.

Quick Access Toolbar

In versions of Excel prior to Excel 2007, you can attach macros to toolbar buttons. Because toolbars and menus have been replaced by the Ribbon in Excel 2007, this ability no longer exists, with the exception of the Quick Access Toolbar. The Quick Access Toolbar sits either above or below the Ribbon, and you can add any button from the Ribbon to it to give you direct access to the button. When you right-click a Ribbon button, you can choose `Add to Quick Access Toolbar` from the pop-up menu. The same pop-up menu offers a second choice, which is `Customize Quick Access Toolbar`. This choice opens the dialog box shown in Figure 1-13.

Select `Macros` from the `Choose commands from:` drop-down menu. You can now assign macros from open workbooks to the Quick Access Toolbar by selecting them and clicking the `Add` button. The icon associated with the macro can be changed by clicking the `Modify` button, which provides a selection of icons and a text box where you can enter a quick tip for the button.



Figure 1-13

Event Procedures

Event procedures are special macro procedures that respond to the events that occur in Excel. Events include user actions, such as clicking the mouse on a button, and system actions, such as the recalculation of a worksheet. Versions of Excel since Excel 97 expose a wide range of events for which you can write code.

The click-event procedure for the ActiveX command button that ran the `MonthNames2` macro, which you have already seen, is a good example. You entered the code for this event procedure in the code module behind the worksheet where the command button was embedded. All event procedures are contained in the class modules behind the workbook, worksheets, charts, and UserForms.

You can see the events that are available by activating a module, such as the `ThisWorkbook` module, choosing an object, such as `Workbook`, from the left drop-down list at the top of the module, and then activating the right drop-down, as shown in Figure 1-14.

The `Workbook_Open()` event can be used to initialize the workbook when it is opened. The code could be as simple as activating a particular worksheet and selecting a range for data input. The code could also be more sophisticated and construct new buttons in the Ribbon.

For compatibility with Excel 5 and 95, you can still create a sub procedure called `Auto_Open()`, in a standard module, that runs when the workbook is opened. If you also have a `Workbook_Open()` event procedure, the event procedure runs first.

As you can see, there are many events to choose from. Some events, such as the `BeforeSave` and `BeforeClose` events, allow cancellation of the event. The following event procedure stops the workbook from being closed until cell A1 in Sheet1 contains the value `True`:

```

Private Sub Workbook_BeforeClose(Cancel As Boolean)
    If ThisWorkbook.Sheets("Sheet1").Range("A1").Value <> True Then
        Cancel = True
    End If
End Sub

```

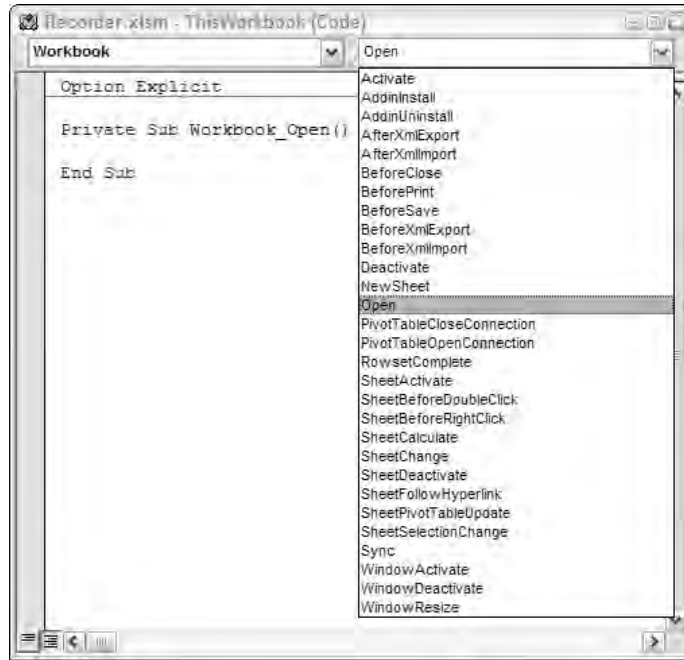


Figure 1-14

This code even prevents the closure of the Excel window.

User-Defined Functions

Excel has hundreds of built-in worksheet functions that you can use in cell formulas. You can select an empty worksheet cell, select the Formulas tab of the Ribbon, and click one of the buttons in the Function Library chunk to see a list of functions. Among the most frequently used functions are SUM, IF, and VLOOKUP. If the function you need is not already in Excel, you can write your own user-defined function (or UDF) using VBA.

UDFs can reduce the complexity of a worksheet. It is possible to reduce a calculation that requires many cells of intermediate results down to a single function call in one cell. UDFs can also increase productivity when many users have to repeatedly use the same calculation procedures. You can set up a library of functions tailored to your organization.

Creating a UDF

Unlike manual operations, UDFs cannot be recorded—you have to write them from scratch using a standard module in the VBE. If necessary, you can insert a standard module by right-clicking in the Project Explorer window and choosing Insert⇨Module. A simple example of a UDF is shown here:

```
Function Fahrenheit(Centigrade)
    Fahrenheit = Centigrade * 9 / 5 + 32
End Function
```

Here, a function called `Fahrenheit()` is created that converts degrees Centigrade to degrees Fahrenheit. In the worksheet, you could have column A containing degrees Centigrade and column B using the `Fahrenheit()` function to calculate the corresponding temperature in degrees Fahrenheit. You can see the formula in cell B2 by looking at the Formula bar in Figure 1-15.

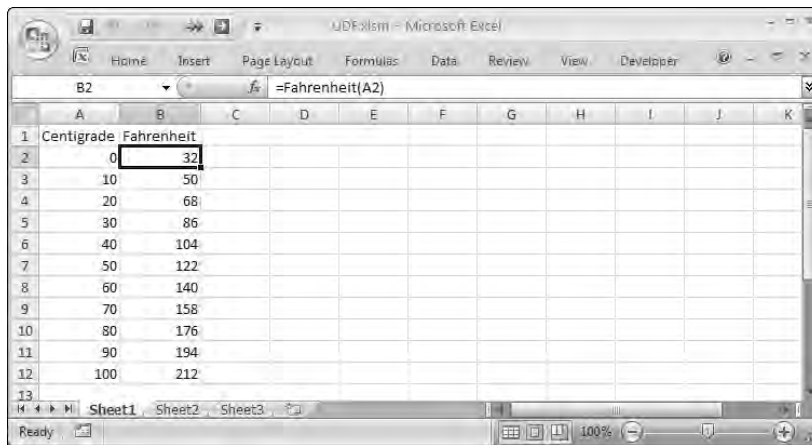


Figure 1-15

The formula has been copied into cells B3:B12.

The key difference between a sub procedure and a function procedure is that a function procedure returns a value. `Fahrenheit()` calculates a numeric value, which is returned to the worksheet cell where `Fahrenheit()` is used. A function procedure indicates the value to be returned by setting its own name equal to the return value.

Function procedures normally have one or more input parameters. `Fahrenheit()` has one input parameter called `Centigrade`, which is used to calculate the return value. When you enter the formula, `Fahrenheit(A2)`, the value in cell A2 is passed to `Fahrenheit()` through `Centigrade`. In the case where the value of `Centigrade` is 0, `Fahrenheit()` sets its own name equal to the calculated result, which is 32. The result is passed back to cell B2, as shown in Figure 1-15. The same process occurs in each cell that contains a reference to `Fahrenheit()`.

A different example that shows how you can reduce the complexity of spreadsheet formulas for users is shown in Figure 1-16. The lookup table in cells A1:D5 gives the price of each product, the discount sales

volume (above which a discount will be applied), and the percent discount for units above the discount volume. Using normal spreadsheet formulas, users would have to set up three lookup formulas together with some logical tests to calculate the invoice amount.

Product	Price	Discount Volume	Discount
Apples	10	100	5%
Mangoes	30	50	10%
Oranges	8	100	5%
Pears	12	100	5%

Product	Volume	Invoice Amount
Apples	50	500
Pears	200	2340
Apples	150	1475
Mangoes	50	1500
Apples	20	200

Figure 1-16

The InvoiceAmount() function has three input parameters: Product is the name of the product, Volume is the number of units sold, and Table is the lookup table. The formula in cell C9, in Figure 1-16, defines the ranges to be used for each input parameter:

```
Function InvoiceAmount(Product, Volume, Table)
'Find price in table
Price = WorksheetFunction.VLookup(Product, Table, 2)

'Find discount volume threshold
DiscountVolume = WorksheetFunction.VLookup(Product, Table, 3)

'Apply discount if volume above threshold
If Volume > DiscountVolume Then
'Calculate invoice with discount
DiscountPct = WorksheetFunction.VLookup(Product, Table, 4)
InvoiceAmount = Price * DiscountVolume + Price * _
(1 - DiscountPct) * (Volume - DiscountVolume)
Else
'Calculate invoice without discount
InvoiceAmount = Price * Volume
End If
End Function
```

The range for the table is absolute so that the copies of the formula below cell C8 refer to the same range. The first calculation in the function uses the VLookup function to find the product in the lookup table and return the corresponding value from the second column of the lookup table, which it assigns to the variable Price.

Chapter 1: Primer in Excel VBA

If you want to use an Excel worksheet function in a VBA procedure, you need to tell VBA where to find it by preceding the function name with `WorksheetFunction` and a period. For compatibility with Excel 5 and 95, you can use `Application` instead of `WorksheetFunction`. Not all worksheet functions are available this way. In these cases, VBA has equivalent functions, or mathematical operators, to carry out the same calculations.

In the next line of the function, the discount volume is found in the lookup table and assigned to the variable `DiscountVolume`. The `If` test on the next line compares the sales volume in `Volume` with `DiscountVolume`. If `Volume` is greater than `DiscountVolume`, the calculations following it, down to the `Else` statement, are carried out. Otherwise, the calculation after the `Else` is carried out.

If `Volume` is greater than `DiscountVolume`, the percent discount rate is found in the lookup table and assigned to the variable `DiscountPct`. The invoice amount is then calculated by applying the full price to the units up to `DiscountVolume` plus the discounted price for units above `DiscountVolume`. Note the use of the underscore character, preceded by a blank space, to indicate the continuation of the code on the next line.

The result is assigned to the name of the function, `InvoiceAmount`, so that the value will be returned to the worksheet cell. If `Volume` is not greater than `DiscountVolume`, the invoice amount is calculated by applying the price to the units sold, and the result is assigned to the name of the function.

Direct Reference to Ranges

When you define a UDF, it is possible to directly refer to worksheet ranges rather than through the input parameters of the UDF. This is illustrated in the following version of the `InvoiceAmount()` function:

```
Function InvoiceAmount2(Product, Volume)
    'Create object variable referring to table in worksheet
    Set Table = ThisWorkbook.Worksheets("Sheet2").Range("A2:D5")

    'Find price in table
    Price = WorksheetFunction.VLookup(Product, Table, 2)

    'Find discount volume threshold
    DiscountVolume = WorksheetFunction.VLookup(Product, Table, 3)

    'Apply discount if volume above threshold
    If Volume > DiscountVolume Then
        'Calculate invoice with discount
        DiscountPct = WorksheetFunction.VLookup(Product, Table, 4)
        InvoiceAmount2 = Price * DiscountVolume + Price * _
            (1 - DiscountPct) * (Volume - DiscountVolume)
    Else
        'Calculate invoice without discount
        InvoiceAmount2 = Price * Volume
    End If
End Function
```

Note that `Table` is no longer an input parameter. Instead, the `Set` statement defines `Table` with a direct reference to the worksheet range. Although this method still works, the return value of the function will not be recalculated if you change a value in the lookup table. Excel does not realize that it needs to recalculate the function when a lookup table value changes, because it does not see that the table is used by the function.

Excel only recalculates a UDF when it sees its input parameters change. If you want to remove the lookup table from the function parameters and still have the UDF recalculate automatically, you can declare the function to be volatile on the first line of the function, as shown here:

```
Function InvoiceAmount2(Product, Volume)
    Application.Volatile
    Set Table = ThisWorkbook.Worksheets("Sheet2").Range("A2:D5")
    ...
End Function
```

However, you should be aware that this feature comes at a price. If a UDF is declared volatile, the UDF is recalculated every time any value changes in the worksheet. This can add a significant recalculation burden to the worksheet if the UDF is used in many cells.

What UDFs Cannot Do

A common mistake made by users is to attempt to create a worksheet function that changes the structure of the worksheet by, for example, copying a range of cells. Such attempts will fail. No error messages are produced because Excel simply ignores the offending code lines, so the reason for the failure is not obvious.

UDFs, used in worksheet cells, are not permitted to change the structure of the worksheet, meaning that a UDF cannot return a value to any other cell than the one it is used in, and it cannot change a physical characteristic of a cell, such as the font color or background pattern. In addition, UDFs cannot carry out actions such as copying or moving spreadsheet cells. They cannot even carry out some actions that imply a change of cursor location, such as an Edit ⇨ Find. A UDF can call another function procedure, or even a sub procedure, but that procedure will be under the same restrictions as the UDF. It will still not be permitted to change the structure of the worksheet.

A distinction is made (in Excel VBA) between UDFs that are used in worksheet cells and function procedures that are not connected with worksheet cells. As long as the original calling procedure was not a UDF in a worksheet cell, a function procedure can carry out any Excel action, just like a sub procedure.

It should also be noted that UDFs are not as efficient as the built-in Excel worksheet functions. If UDFs are used extensively in a workbook, recalculation time will be greater compared with a similar workbook using the same number of built-in functions.

The Excel Object Model

The Visual Basic for Applications programming language is common across all the Microsoft Office applications. In addition to Excel, you can use VBA in Word, Access, PowerPoint, and Outlook. Once you learn it, you can apply it to any of these. However, to work with an application, you need to learn about the objects it contains. In Word, you deal with documents, paragraphs, and words. In Access, you deal with databases, recordsets, and fields. In Excel, you deal with workbooks, worksheets, and ranges.

Chapter 1: Primer in Excel VBA

Unlike many programming languages, you don't have to create your own objects in Office VBA. Each application has a clearly defined set of objects that are arranged according to the relationships between them. This structure is referred to as the application's object model. This section is an introduction to the Excel object model, which is fully documented in Appendix A.

Objects

First up, this section covers a few basics about Object-Oriented Programming (OOP). This not a complete formal treatise on the subject, but it covers what you need to know to work with the objects in Excel.

OOP's basic premise is that you can describe everything known to us as objects. You and I are objects, the world is an object, and the universe is an object. In Excel, a workbook is an object, a worksheet is an object, and a range is an object. These objects are only a small sample of around two hundred object types available to us in Excel. Look at some examples of how to refer to Range objects in VBA code. One simple way to refer to cells B2:C4 is as follows:

```
Range("B2:C4")
```

If you give the name `Data` to a range of cells, you can use that name in a similar way:

```
Range("Data")
```

There are also ways to refer to the currently active cell and selection using shortcuts.

In Figure 1-17, `ActiveCell` refers to the B2 cell, and `Selection` refers to the range B2:E6. For more information on `ActiveCell` and `Selection`, see Chapter 3.

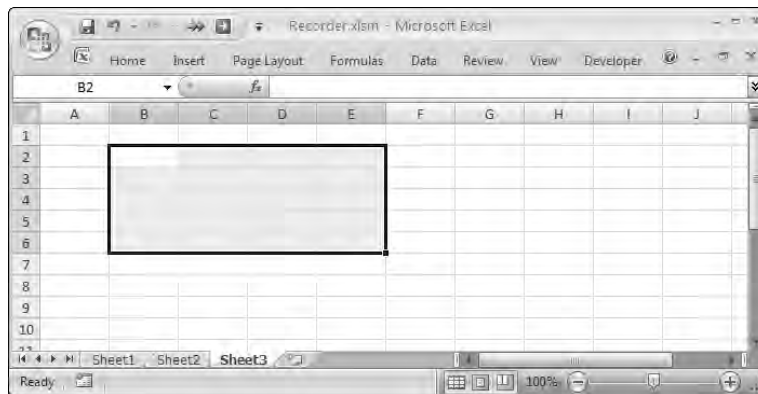


Figure 1-17

Collections

Many objects belong to collections. A city block is a collection of high-rise buildings. A high-rise building is a collection of floor objects. A floor is a collection of room objects. Collections are objects themselves — objects that contain other objects that are closely related. Collections and objects are often related in a hierarchical or tree structure.

Excel is an object itself, called the `Application` object. In the Excel `Application` object, there is a `Workbooks` collection that contains all the currently open `Workbook` objects. Each `Workbook` object has a `Worksheets` collection that contains the `Worksheet` objects in that workbook.

Note that you need to make a clear distinction between the plural `Worksheets` object, which is a collection, and the singular `Worksheet` object. They are quite different objects.

If you want to refer to a member of a collection, you can refer to it by its position in the collection, as an index number starting with 1, or by its name, as quoted text. If you have opened just one workbook called `Data.xls`, you can refer to it by either of the following:

```
Workbooks(1)
Workbooks("Data.xls")
```

If you have three worksheets in the active workbook that have the names `North`, `East`, and `South`, in that order, you can refer to the second worksheet by either of the following:

```
Worksheets(2)
Worksheets("East")
```

If you want to refer to a worksheet called `DataInput` in a workbook called `Sales.xls`, and `Sales.xls` is not the active workbook, you must qualify the worksheet reference with the workbook reference, separating them with a period, as follows:

```
Workbooks("Sales.xls").Worksheets("DataInput")
```

When you refer to the `B2` cell in `DataInput`, while another workbook is active, you use:

```
Workbooks("Sales.xls").Worksheets("DataInput").Range("B2")
```

The following section examines objects more closely and explains how you can manipulate them in VBA code. You need to be aware of two key characteristics of objects to do this. They are the properties and methods associated with an object.

Properties

Properties are the physical characteristics of objects, and can be measured or quantified. You and I have a height property, an age property, a bank balance property, and a name property. Some of our properties can be changed fairly easily, such as our bank balance. Other properties are more difficult or impossible to change, such as our name and age.

A worksheet `Range` object has a `RowHeight` property and a `ColumnWidth` property. A `Workbook` object has a `Name` property, which contains its filename. Some properties can be changed easily, such as the `Range` object's `ColumnWidth` property, by assigning the property a new value. Other properties, such as the `Workbook` object's `Name` property, are read-only. You can't change the `Name` property by simply assigning a new value to it.

Chapter 1: Primer in Excel VBA

You refer to the property of an object by referring to the object, then the property, separated by a period. For example, to change the width of the column containing the active cell to 20 points, you would assign the value to the `ColumnWidth` property of the `ActiveCell` using:

```
ActiveCell.ColumnWidth = 20
```

To enter the name `Florence` into cell `C10`, you assign the name to the `Value` property of the `Range` object:

```
Range("C10").Value = "Florence"
```

If the `Range` object is not in the active worksheet in the active workbook, you need to be more specific:

```
Workbooks("Sales.xls").Worksheets("DataInput").Range("C10").Value = 10
```

VBA can do what is impossible to do manually. It can enter data into worksheets that are not visible on the screen. It can copy and move data without having to make the sheets involved active. Therefore, it is very seldom necessary to activate a specific workbook, worksheet, or range to manipulate data using VBA. The more you can avoid activating objects, the faster your code will run. Unfortunately, the macro recorder can only record what you do and uses activation extensively.

In the previous examples, you have seen how to assign values to the properties of objects. You can also assign the property values of objects to variables or to other objects' properties. You can directly assign the column width of one cell to another cell on the active sheet, using:

```
Range("C1").ColumnWidth = Range("A1").ColumnWidth
```

You can assign the value in `C1` in the active sheet to `D10` in the sheet named `Sales`, in the active workbook, using:

```
Worksheets("Sales").Range("D10").Value = Range("C1").Value
```

You can assign the value of a property to a variable so it can be used in later code. This example stores the current value of cell `M100`, sets `M100` to a new value, prints the auto-recalculated results, and sets `M100` back to its original value:

```
OpeningStock = Range("M100").Value  
Range("M100").Value = 100  
ActiveSheet.PrintOut  
Range("M100").Value = OpeningStock
```

Some properties are read-only, which means that you can't assign a value to them directly. Sometimes there is an indirect way. One example is the `Text` property of a `Range` object. You can assign a value to a cell using its `Value` property, and you can give the cell a number format using its `NumberFormat` property. The `Text` property of the cell gives you the formatted appearance of the cell. The following example displays \$12,345.60 in a Message box:

```
Range("B10").Value = 12345.6  
Range("B10").NumberFormat = "$#,##0.00"  
MsgBox Range("B10").Text
```

This is the only means by which you can set the value of the `Text` property.

Methods

Whereas properties are the quantifiable characteristics of objects, methods are the actions that can be performed by objects or on objects. If you have a linguistic bent, you might like to think of objects as nouns, properties as adjectives, and methods as verbs. Methods often change the properties of objects. I have a walking method that takes me from A to B, changing my location property. I have a spending method that reduces my bank balance property and a working method that increases my bank balance property. My dieting method reduces my weight property, temporarily.

A simple example of an Excel method is the `Select` method of the `Range` object. To refer to a method, as with properties, put the object first, add a period, and then add the method. The following selects cell G4:

```
Range("G4").Select
```

Another example of an Excel method is the `Copy` method of the `Range` object. The following copies the contents of range A1:B3 to the clipboard:

```
Range("A1:B3").Copy
```

Methods often have parameters that you can use to modify the way the method works. For example, you can use the `Paste` method of the `Worksheet` object to paste the contents of the clipboard into a worksheet, but if you do not specify where the data is to be pasted, it is inserted with its top-left corner in the active cell. This can be overridden with the `Destination` parameter (parameters are discussed later in this section):

```
ActiveSheet.Paste Destination:=Range("G4")
```

Note that the value of a parameter is specified using `:=`, not just `=`.

Often, Excel methods provide shortcuts. The previous examples of `Copy` and `Paste` can be carried out entirely by the `Copy` method:

```
Range("A1:B3").Copy Destination:=Range("G4")
```

This is far more efficient than the code produced by the macro recorder:

```
Range("A1:B3").Select  
Selection.Copy  
Range("G4").Select  
ActiveSheet.Paste
```

Events

Another important concept in VBA is that objects can respond to events. A mouse click on a command button, a double-click on a cell, a recalculation of a worksheet, and the opening and closing of a workbook are examples of events.

All of the ActiveX controls can respond to events. These controls can be embedded in worksheets and in UserForms to enhance the functionality of those objects. Worksheets and workbooks can also respond to a wide range of events. If you want an object to respond to an event, enter VBA code into the appropriate event procedure for that object. The event procedure resides in the code module behind the `Workbook`, `Worksheet`, or `UserForm` object concerned.

For example, you might want to detect that a user has selected a new cell and highlight the cell's complete row and column. You can do this by entering code in the `Worksheet_SelectionChange()` event procedure:

1. First activate the VBE window and double-click the worksheet in the Project Explorer.
2. From the drop-down lists at the top of the worksheet code module, choose `Worksheet` and `SelectionChange`, and enter the following code:

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
    Rows.Interior.ColorIndex = xlColorIndexNone
    Target.EntireColumn.Interior.ColorIndex = 36
    Target.EntireRow.Interior.ColorIndex = 36
End Sub
```

This event procedure runs every time the user selects a new cell, or block of cells. The parameter, `Target`, refers to the selected range as a `Range` object. The first statement sets the `ColorIndex` property of all the worksheets cells to no color, to remove any existing background color. The second and third statements set the entire columns and entire rows that intersect with the selected cells to a background color of pale yellow. This color can be different, depending on the color palette set up in your workbook.

The use of properties in this example is more complex than you have seen before. Now analyze the component parts. If you assume that `Target` is a `Range` object referring to cell B10, then the following code uses the `EntireColumn` property of the B10 `Range` object to refer to the entire B column, which is the range B1:B1048576, or B:B for short:

```
Target.EntireColumn.Interior.ColorIndex = 36
```

Similarly, the next line of code changes the color of row 10, which is the range A10:XFD10, or 10:10 for short:

```
Target.EntireRow.Interior.ColorIndex = 36
```

The `Interior` property of a `Range` object refers to an `Interior` object, which is the background of a range. Finally, set the `ColorIndex` property of the `Interior` object equal to the index number for the required color.

This code might appear to many to be far from intuitive. So how do you go about figuring out how to carry out a task involving an Excel object?

Getting Help

The easiest way to discover the required code to perform an operation is to use the macro recorder. The recorded code is likely to be inefficient, but it will indicate the objects required and the properties and methods involved. If you turn on the recorder to find out how to color the background of a cell, you will get something like the following:

```
With Selection.Interior
    .Pattern = xlSolid
    .PatternColorIndex = 56
    .Color = 65535
    .TintAndShade = 0
    .PatternTintAndShade = 0
End With
```

This `With...End With` construction is discussed in more detail later in this chapter. It is equivalent to:

```
Selection.Interior.Pattern = xlSolid
Selection.Interior.PatternColorIndex = 56
Selection.Interior.Color = 65535
Selection.Interior.TintAndShade = 0
Selection.Interior.PatternTintAndShade = 0
```

The lines of code that specify `Pattern`, `TintAndShade`, and `PatternTintAndShade` are unnecessary, because they specify default values. The macro recorder is not sophisticated enough to know what the user does or doesn't want, so it includes everything. However, the recorded code provides the clues you need to get started. You only need to figure out how to change the `Range` object, `Selection`, into a complete row or complete column. If this can be done, it will be accomplished by using a property or method of the `Range` object.

The Object Browser

The Object Browser is a valuable tool for discovering the properties, methods, and events applicable to Excel objects. To display the Object Browser, you need to be in the VBE window. You can use `View ⇨ Object Browser`, press `F2`, or click the Object Browser button on the Standard toolbar to see the window shown in Figure 1-18.

The objects are listed in the window with the title `Classes`. Objects are instances of classes. You can click in this window and type an `r` to get quickly to the `Range` object.

Alternatively, you can click in the search box, second from the top with the binoculars to its right, and type in `range`. When you press `Enter` or click the binoculars, you will see a list of items containing this text. When you click `Range`, under the `Class` heading in the Search Results window, `Range` will be highlighted in the `Classes` window below. This technique is handy when you are searching for information on a specific property, method, or event.

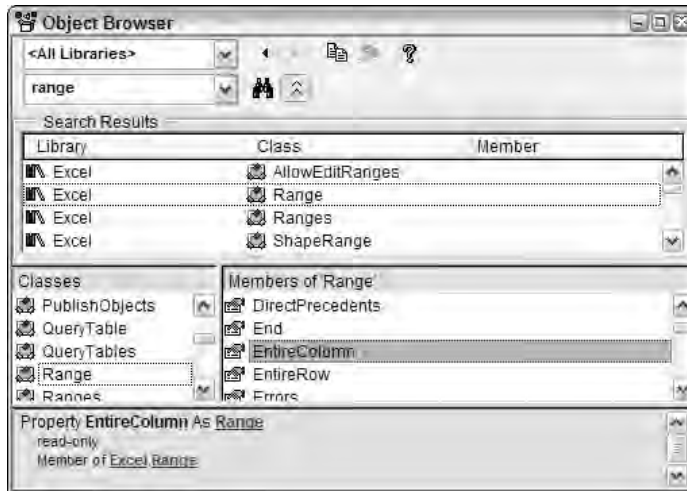


Figure 1-18

You now have a list of all the properties, methods, and events (if applicable) for this object, sorted alphabetically. If you right-click this list, you can choose Group Members to separate the properties, methods, and events, which makes it easier to read. If you scan through this list, you will see the `EntireColumn` and `EntireRow` properties, which look to be likely candidates for your requirements. To confirm this, select `EntireColumn` and click the question mark icon at the top of the Object Browser window to go to the window in Figure 1-19.

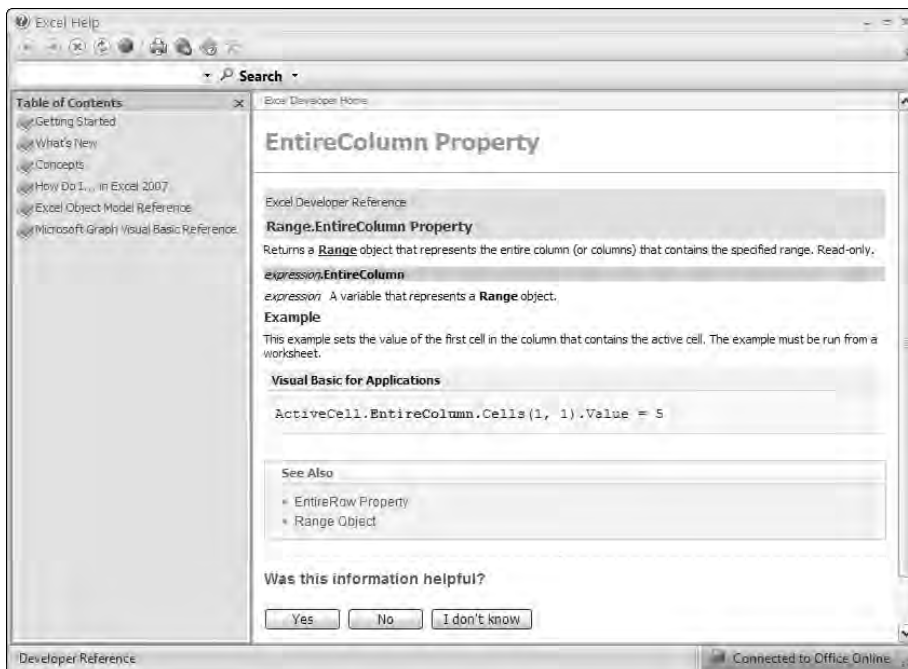


Figure 1-19

See Also can often lead to further information on related objects and methods. Now, all that remains to do is connect the properties you found and apply them to the right object.

Experimenting in the Immediate Window

If you want to experiment with code, you can use the VBE's Immediate window. Use View ⇄ Immediate Window, press Ctrl+G, or click the Immediate Window button on the Debug toolbar to make the Immediate window visible. You can tile the Excel window and the VBE window so you can type commands into the Immediate window and see the effects in the Excel window, as shown in Figure 1-20.

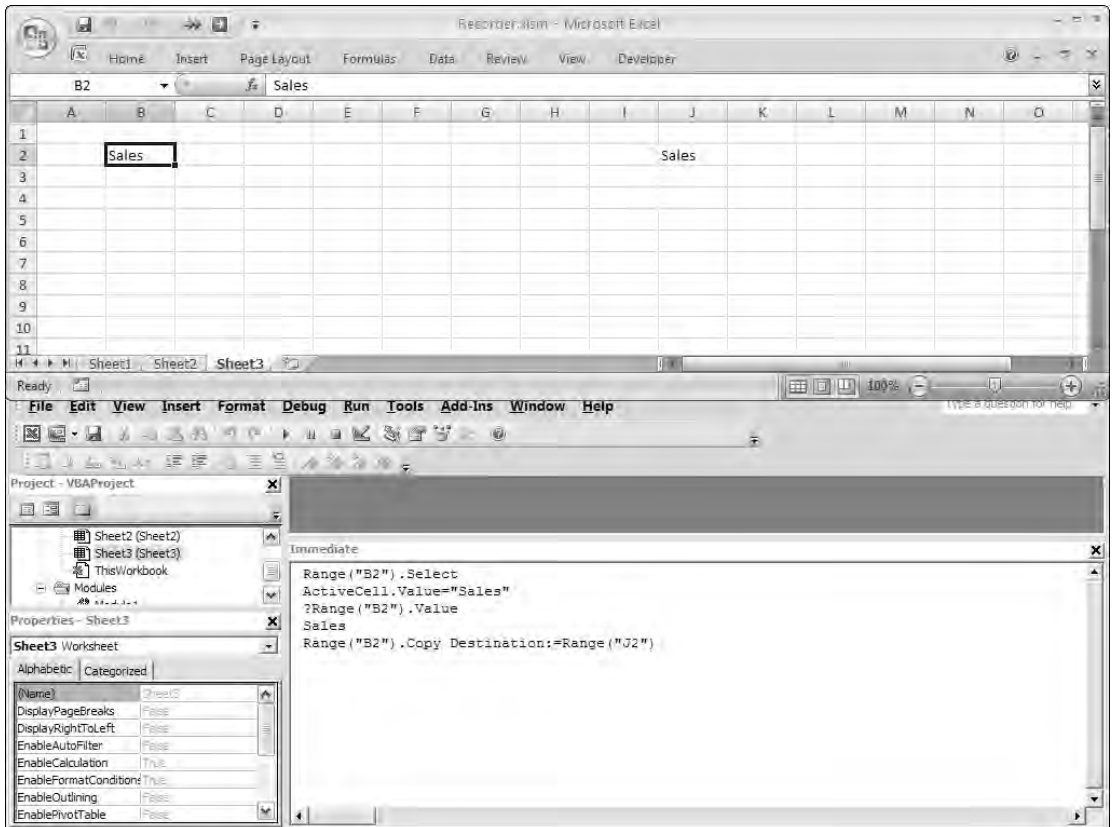


Figure 1-20

When a command is typed in and Enter is pressed, the command is immediately executed. To execute the same command again, click anywhere in the line with the command and press Enter again.

Here, the Value property of the ActiveCell object has been assigned the text "Sales". If you want to display a value, you precede the code with a question mark, which is a shortcut for Print:

```
?Range("B2").Value
```

Chapter 1: Primer in Excel VBA

This code has printed `Sales` on the next line of the Immediate window. The last command has copied the value in B2 to J2.

The VBA Language

In this section, you see the elements of the VBA language that are common to all versions of Visual Basic and the Microsoft Office applications. The section uses examples that employ the Excel object model, but the aim is to examine the common structures of the language. Many of these structures and concepts are common to other programming languages, although the syntax and keywords can vary. This section examines the following:

- Storing information in variables and arrays
- Decision-making in code
- Using loops
- Basic error-handling

Basic Input and Output

First, look at some simple communication techniques you can use to make your macros more flexible and useful. If you want to display a message, use the `MsgBox` function, which is useful if you want to display a warning message or ask a simple question.

In the first example, you want to make sure that the printer is switched on before a print operation. The following code generates the dialog box in Figure 1-21, giving the user a chance to check the printer. The macro pauses until the OK button is clicked:

```
MsgBox "Please make sure that the printer is switched on"
```

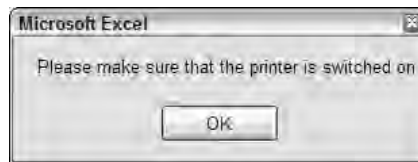


Figure 1-21

If you want to experiment, you can use the Immediate window to execute single lines of code. Alternatively, you can insert your code into a standard module in the VBE window. In this case, you need to include `Sub` and `End Sub` lines as follows:

```
Sub Test1()  
    MsgBox "Please make sure that the printer is switched on"  
End Sub
```


An easy way to execute a sub procedure is to click somewhere in the code to create an insertion point, then press F5.

`MsgBox` has many options that control the types of buttons and icons that appear in the dialog box. If you want to get help with this, or any VBA word, just click somewhere in the word and press the F1 key. The Help screen for the word will immediately appear. Among other details, you will see the input parameters accepted by the function:

```
MsgBox(prompt[, buttons] [, title] [, helpfile, context])
```

Parameters in square brackets are optional, so only the `prompt` message is required. If you want to have a `title` at the top of the dialog box, you can specify the third parameter. There are two ways to specify parameter values: by position and by name.

Parameters Specified by Position

If you specify a parameter by position, you need to make sure that the parameters are entered in the correct order. You also need to include extra commas for missing parameters. The following code provides a title for the dialog box, specifying the title by position and producing the result shown in Figure 1-22:

```
MsgBox "Is the printer on?", , "Caution!"
```



Figure 1-22

Parameters Specified by Name

There are some advantages and some special considerations required when specifying parameters by name:

- ❑ You can enter them in any order and do not need to include extra commas with nothing between them to allow for undefined parameters.
- ❑ You do need to use `:=` rather than just `=` between the parameter name and the value, as already pointed out.

The following code generates the same dialog box as in Figure 1-22:

```
MsgBox Title:="Caution!", Prompt:="Is the printer on?"
```

Another advantage of specifying parameters by name is that the code is better documented. Anyone reading the code is more likely to understand it.

If you want more information on the `buttons` parameter, you will find a table of options in the help screen as follows:

Constant	Value	Description
vbOKOnly	0	Display OK button only
vbOKCancel	1	Display OK and Cancel buttons
vbAbortRetryIgnore	2	Display Abort, Retry, and Ignore buttons
vbYesNoCancel	3	Display Yes, No, and Cancel buttons
vbYesNo	4	Display Yes and No buttons
vbRetryCancel	5	Display Retry and Cancel buttons
vbCritical	16	Display Critical Message icon
vbQuestion	32	Display Warning Query icon
vbExclamation	48	Display Warning Message icon
vbInformation	64	Display Information Message icon
vbDefaultButton1	0	First button is default
vbDefaultButton2	256	Second button is default
vbDefaultButton3	512	Third button is default
vbDefaultButton4	768	Fourth button is default
vbApplicationModal	0	Application modal; the user must respond to the message box before continuing work in the current application
vbSystemModal	4096	System modal; all applications are suspended until the user responds to the message box
vbMsgBoxHelpButton	16384	Adds Help button to the message box
vbMsgBoxSetForeground	65536	Specifies the message box window as the foreground window
vbMsgBoxRight	524288	Text is right-aligned
vbMsgBoxRtlReading	1048576	Specifies text should appear as right-to-left reading on Hebrew and Arabic systems

Values 0 to 5 control the buttons that appear. A value of 4 gives Yes and No buttons, as shown in Figure 1-23:

```
MsgBox Prompt:="Delete this record?", Buttons:=4
```



Figure 1-23

Values 16 to 64 control the icons that appear; 32 gives a question mark icon. If you want both value 4 and value 32, add them to see the dialog box in Figure 1-24:

```
MsgBox Prompt:="Delete this record?", Buttons:=36
```



Figure 1-24

Constants

Specifying a `Buttons` value of 36 ensures that your code is indecipherable to all but the most battle-hardened programmer. This is why VBA provides the constants shown to the left of the button values in the help screen. Rather than specifying `Buttons` by numeric value, you can use the constants, which provide a better indication of the choice behind the value. The following code generates the same dialog box as the previous example:

```
MsgBox Prompt:="Delete this record?", Buttons:=vbYesNo + vbQuestion
```

The VBE helps you as you type by providing a pop-up list of the appropriate constants after you type `Buttons:=`. Point to the first constant and press the plus key (+), and you will be prompted for the second constant. Choose the second and press the spacebar or Tab to finish the line. If there is another parameter to be specified, enter a comma rather than a space or a Tab.

Constants are a special type of variable that do not change, if that makes sense. They are used to hold key data and, as you have seen, provide a way to write more understandable code. VBA has many built-in constants that are referred to as intrinsic constants. You can also define your own constants, as you will see later in this chapter.

Return Values

There is something missing from the previous examples of `MsgBox`. You are asking a question, but failing to capture the user's response to the question. That is because you have been treating `MsgBox` as a statement, rather than a function. This is perfectly legal, but you need to know some rules if you are to avoid syntax errors. You can capture the return value of the `MsgBox` function by assigning it to a variable.

However, if you try the following, you will get a syntax error:

```
Answer = MsgBox Prompt:="Delete this record?", Buttons:=vbYesNo + vbQuestion
```

Chapter 1: Primer in Excel VBA

The error message, `Expected: End of Statement`, is not really very helpful. You can click the Help button on the error message to get a more detailed description of the error, but even then you might not understand the explanation.

Parentheses

The problem with the previous line of code is that there are no parentheses around the function arguments. It should read as follows:

```
Answer = MsgBox(Prompt:="Delete this record?", Buttons:=vbYesNo + vbQuestion)
```

The general rule is that if you want to capture the return value of a function, you need to put any arguments in parentheses. If you don't want to use the return value, you should not use parentheses, as with the original examples of using `MsgBox`.

The parentheses rule also applies to methods used with objects. Many methods have return values that you can ignore or capture. See the section on object variables later in this chapter for an example.

Now that you have captured the return value of `MsgBox`, how do you interpret it? Once again, the help screen provides the required information in the form of the following table of return values:

Constant	Value	Description
<code>vbOK</code>	1	OK
<code>vbCancel</code>	2	Cancel
<code>vbAbort</code>	3	Abort
<code>vbRetry</code>	4	Retry
<code>vbIgnore</code>	5	Ignore
<code>vbYes</code>	6	Yes
<code>vbNo</code>	7	No

If the Yes button is clicked, `MsgBox` returns a value of 6. You can use the constant `vbYes`, instead of the numeric value, in an `If` test:

```
Answer = MsgBox(Prompt:="Delete selected Row?", Buttons:=vbYesNo + vbQuestion)
If Answer = vbYes Then ActiveCell.EntireRow.Delete
...
```

InputBox

Another useful VBA function is `InputBox`, which allows you to get input data from a user in the form of text. The following code generates the dialog box shown in Figure 1-25:

```
UserName = InputBox(Prompt:="Please enter your name")
```

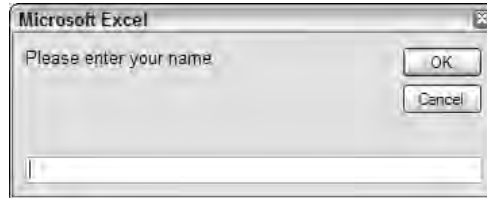


Figure 1-25

`InputBox` returns a text (string) result. Even if a numeric value is entered, the result is returned as text. If you click `Cancel` or `OK` without typing anything into the text box, `InputBox` returns a zero-length string. It is a good idea to test the result before proceeding so this situation can be handled. In the following example, the sub procedure does nothing if `Cancel` is clicked. The `Exit Sub` statement stops the procedure at that point. Otherwise, it places the entered data into cell B2:

```
Sub GetData()
    Sales = InputBox(Prompt:="Enter Target Sales")
    If Sales = "" Then Exit Sub
    Range("B2").Value = Sales
End Sub
```

In this code, the `If` test compares `Sales` with a zero-length string. There is nothing between the two double quote characters. Don't be tempted to put a blank space between the quotes.

There is a more powerful version of `InputBox` that is a method of the Excel Application object. It has the ability to restrict the type of data that you can enter. It is covered in Chapter 2.

Calling Functions and Sub Procedures

When you develop an application, you should not attempt to place all your code in one large procedure. You should write small procedures that carry out specific tasks, and test each procedure independently. You can then write a master procedure that runs your task procedures. This approach makes the testing and debugging of the application much simpler, and also makes it easier to modify the application later.

The following code illustrates this modular approach, although in a practical application your procedures would have many more lines of code:

```
Sub Master()
    SalesData = GetInput("Enter Sales Data")
    If SalesData = False Then Exit Sub
    PostInput SalesData, "B3"
End Sub

Function GetInput(Message)
    Data = InputBox(Message)
    If Data = "" Then GetInput = False Else GetInput = Data
End Function

Sub PostInput(InputData, Target)
    Range(Target).Value = InputData
End Sub
```

Chapter 1: Primer in Excel VBA

Master uses the `GetInput` function and the `PostInput` sub procedure. `GetInput` has one input parameter, which passes the prompt message for the `InputBox` function and tests for a zero-length string in the response. A value of `False` is returned if this is found. Otherwise, `GetInput` returns the response.

Master tests the return value from `GetInput` and exits if it is `False`. Otherwise, Master calls `PostInput`, passing two values that define the data to be posted and the cell the data is to be posted to.

Note that sub procedures can accept input parameters, just like function procedures, if they are called from another procedure. You can't run a sub procedure with input parameters directly.

Also note that, when calling `PostInput` and passing two parameters to it, Master does not place parentheses around the parameters. Because sub procedures do not generate a return value, you should not put parentheses around the arguments when one is called, except when using the `Call` statement that is discussed next.

When calling your own functions and subs, you can specify parameters by name, just as you can with built-in procedures. The following version of Master uses this technique:

```
Sub Master()  
    SalesData = GetInput(Message:="Enter Sales Data")  
    If SalesData = False Then Exit Sub
```

```
    PostInput Target:="B3", InputData:=SalesData
```

```
End Sub
```

The Call Statement

When running a sub procedure from another procedure, you can use the `Call` statement. There is no particular benefit in doing this; it is just an alternative to the previous method. Master can be modified as follows:

```
Sub Master()  
    SalesData = GetInput("Enter Sales Data")  
    If SalesData = False Then Exit Sub
```

```
    Call PostInput(SalesData, "B3")
```

```
End Sub
```

Note that if you use `Call`, you must put parentheses around the parameters passed to the called procedure, regardless of the fact that there is no return value from the procedure. You can also use `Call` with a function, but only if the return value is not used.

Parentheses and Argument Lists

As you have seen, the use of parentheses around arguments when calling procedures is a bit of a mine-field, so the following sections summarize when to use them, at the risk of opening a can of worms and getting lost in mixed metaphors. Bear in mind that the same rules apply to argument lists of methods.

Without the Call Statement

Only place parentheses around the arguments when you are calling a function procedure and are also making use of the return value from the function procedure:

```
SalesData = GetInput("Enter Sales Data")
```

Don't place parentheses around the arguments when you are calling a function procedure and are not making use of the return value from the function procedure:

```
GetInput "Enter Sales Data"
```

Don't place parentheses around the arguments when you are calling a sub procedure:

```
PostInput SalesData, "B3"
```

An Important Subtlety Regarding Parentheses

The following is correct syntax and leads to untold confusion:

```
MsgBox ("Insert Disk")
```

It is not what it appears and it is not a negation of the parentheses rules. VBA has inserted a space between `MsgBox` and the left parenthesis, which it does not insert in the following:

```
Response = MsgBox("Insert Disk")
```

The extra space indicates that the parentheses are around the argument, not around the argument list. If you pass two input parameters, the following is not valid syntax:

```
MsgBox ("Insert Disk", vbExclamation)
```

The following is valid syntax:

```
MsgBox ("Insert Disk"), (vbExclamation)
```

It is fine to place parentheses around individual arguments, but not around the argument list. However, you might not get the result you expect.

Apologies if you are bored, but this is important stuff. It is more important when you get to refer to objects in parameter lists. Placing an object reference in parentheses causes VBA to convert the object reference to the object's default property. For example, `(Range("B1"))` is converted to the value in the B1 cell and is not a reference to a Range object. The following is valid syntax to copy A1 to B1:

```
Range("A1").Copy Range("B1")
```

Chapter 1: Primer in Excel VBA

The following is valid syntax but causes a run-time error:

```
Range("A1").Copy (Range("B1"))
```

With the Call Statement

If you use the `Call` statement, you must place parentheses around the arguments you pass to the called procedure:

```
Call PostInput(SalesData, "B3")
```

Because `Call` is of limited use, not being able to process a return value, and muddies the water with its own rules, it is preferable not to use it.

Variable Declaration

You have seen many examples of the use of variables for storing information. It is now time to discuss the rules for creating variable names, look at different types of variables, and talk about the best way to define variables.

Variable names can be constructed from letters and numbers and the underscore character. The name must start with a letter and can be up to 255 characters in length. It is a good idea to avoid using any special characters in variable names. To be on the safe side, you should only use the letters of the alphabet (upper- and lowercase), plus the numbers 0–9 and the underscore (_). Also, variable names can't be the same as VBA keywords, such as `Sub` and `End`, or VBA function names.

So far you have been creating variables simply by using them. This is referred to as implicit variable declaration. Most computer languages require you to employ explicit variable declaration. This means that you must define the names of all the variables you are going to use before using them in code. VBA allows both types of declarations. If you want to declare a variable explicitly, do so using a `Dim` statement or one of its variations, which is discussed shortly. The following `Dim` statement declares a variable called `SalesData`:

```
Sub GetData()  
    Dim SalesData  
    SalesData = InputBox(Prompt:="Enter Target Sales")  
    ...
```

Most users find implicit declaration easier than explicit declaration, but there are many advantages to being explicit. One advantage is the preservation of capitalization. More important advantages are discussed later in this chapter.

You might have noticed that if you enter VBA words, such as `inputbox`, in lowercase, they are automatically converted to VBA's standard capitalization when you move to the next line. This is a valuable form of feedback that tells you the word has been recognized as valid VBA code. It is a good idea to always type VBA words in lowercase and look for the change.

If you do not explicitly declare a variable name, you can get odd effects regarding its capitalization. Say you write the following code:

```
Sub GetData()
    SalesData = InputBox(Prompt:="Enter Target Sales")
    If salesdata = "" Then Exit Sub
    ...

```

You will find that when you press Enter at the end of line 3, the original occurrence of `SalesData` loses its capitalization and the procedure reads as follows:

```
Sub GetData()
    salesdata = InputBox(Prompt:="Enter Target Sales")
    If salesdata = "" Then Exit Sub
    ...

```

In fact, any time you edit the procedure and alter the capitalization of `salesdata`, the new version will be applied throughout the procedure. If you declare `SalesData` in a `Dim` statement, the capitalization you use on that line will prevail throughout the procedure. You can now type the variable name in lowercase in the body of the code and obtain confirmation that it has been correctly spelled as you move to a new line.

Option Explicit

There is a way to force explicit declaration in VBA. Place the statement `Option Explicit` in the declarations section of your module, which is at the very top of your module, before any procedures, as shown in Figure 1-26.

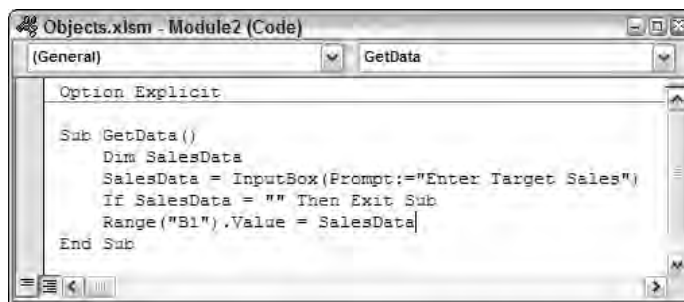


Figure 1-26

`Option Explicit` only applies to the module it appears in. Each module requiring explicit declaration of variables must repeat the statement in its declarations section.

When you try to compile your module or run a procedure using explicit variable declaration, VBA will check for variables that have not been declared, highlight them, and show an error message. This has an enormous benefit. It picks up spelling mistakes, which are among the most common errors in programming. Consider the following version of `GetData`, where there is no `Option Explicit` at the top of the module and, therefore, implicit declaration is used:

```
Sub GetData()  
    SalesData = InputBox(Prompt:="Enter Target Sales")  
    If SaleData = "" Then Exit Sub  
    Range("B2").Value = SalesData  
End Sub
```

This code will never enter any data into cell B2. VBA happily accepts the misspelled `SaleData` in the `If` test as a new variable that is empty, and thus is considered to be a zero-length string for the purposes of the test. Consequently, the `Exit Sub` is always executed and the final line is never executed. This type of error, especially when embedded in a longer section of code, can be very difficult to see.

If you include `Option Explicit` in your declarations section, and `Dim SalesData` at the beginning of `GetData`, you will get an error message, `Variable not defined`, immediately after you attempt to run `GetData`. The undefined variable will be highlighted so that you can see exactly where the error is.

You can have `Option Explicit` automatically added to any new modules you create. In the VBE, use `Tools` ⇄ `Options` and click the `Editor` tab. Check the box against `Require Variable Declaration`. This is a highly recommended option. Note that setting this option will not affect any existing modules, where you will need to insert `Option Explicit` manually.

Scope and Lifetime of Variables

There are two important concepts associated with variables:

- The scope of a variable defines which procedures can use that variable
- The lifetime of a variable defines how long that variable retains the values assigned to it

The following procedure illustrates the lifetime of a variable:

```
Sub LifeTime()  
    Dim Sales  
    Sales = Sales + 1  
    MsgBox Sales  
End Sub
```

Every time `LifeTime` is run, it displays a value of one. This is because the variable `Sales` is only retained in memory until the end of the procedure. The memory `Sales` uses is released when the `End Sub` is reached. Next time `LifeTime` is run, `Sales` is re-created and treated as having a 0 value. The lifetime of `Sales` is the time taken to run the procedure. You can increase the lifetime of `Sales` by declaring it in a `Static` statement:

```
Sub LifeTime()  
    Static Sales  
    Sales = Sales + 1  
    MsgBox Sales  
End Sub
```

The lifetime of `Sales` is now extended to the time that the workbook is open. The more times `LifeTime` is run, the higher the value of `Sales` will become.

The following two procedures illustrate the scope of a variable:

```
Sub Scope1()  
    Static Sales  
    Sales = Sales + 1  
    MsgBox Sales  
End Sub  
  
Sub Scope2()  
    Static Sales  
    Sales = Sales + 10  
    MsgBox Sales  
End Sub
```

The variable `Sales` in `Scope1` is not the same variable as the `Sales` in `Scope2`. Each time `Scope1` is executed, the value of its `Sales` will increase by one, independently of the value of `Sales` in `Scope2`. Similarly, the `Sales` in `Scope2` will increase by 10 with each execution of `Scope2`, independently of the value of `Sales` in `Scope1`. Any variable declared within a procedure has a scope that is confined to that procedure. A variable that is declared within a procedure is referred to as a procedure-level variable.

Variables can also be declared in the declarations section at the top of a module, as shown in the following version of the code:

```
Option Explicit  
Dim Sales  
  
Sub Scope1()  
    Sales = Sales + 1  
    MsgBox Sales  
End Sub  
  
Sub Scope2()  
    Sales = Sales + 10  
    MsgBox Sales  
End Sub
```

Chapter 1: Primer in Excel VBA

Scope1 and Scope2 are now processing the same variable, Sales. A variable declared in the declarations section of a module is referred to as a module-level variable, and its scope is now the whole module. Therefore, it is visible to all the procedures in the module. Its lifetime is now the time that the workbook is open.

If a procedure in the module declares a variable with the same name as a module-level variable, the module-level variable will no longer be visible to that procedure. It will process its own procedure-level variable.

Module-level variables, declared in the declarations section of the module with a Dim statement, are not visible to other modules. If you want to share a variable between modules, you need to declare it as Public in the declarations section:

```
Public Sales
```

Public variables can also be made visible to other workbooks, or VBA projects. To accomplish this, a reference to the workbook containing the Public variable is created in the other workbook, using Tools → References in the VBE.

Variable Type

Computers store different types of data in different ways. The way a number is stored is quite different from the way text, or a character string, is stored. Different categories of numbers are also stored in different ways. An integer (a whole number with no decimals) is stored differently from a number with decimals. Most computer languages require that you declare the type of data to be stored in a variable. VBA does not require this, but your code will be more efficient if you do declare variable types. It is also more likely that you will discover any problems that arise when data is converted from one type to another, if you have declared your variable types.

The following table has been taken directly from the VBA Help files. It defines the various data types available in VBA and their memory requirements. It also shows you the range of values that each type can handle:

Data type	Storage size	Range
Byte	1 byte	0 to 255
Boolean	2 bytes	True or False
Integer	2 bytes	-32,768 to 32,767
Long (long integer)	4 bytes	-2,147,483,648 to 2,147,483,647
Single (single-precision floating-point)	4 bytes	-3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values
Double (double-precision floating-point)	8 bytes	-1.79769313486231E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values

Data type	Storage size	Range
Currency (scaled integer)	8 bytes	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
Decimal	14 bytes	+/-79,228,162,514,264,337,593,543,950,335 with no decimal point; +/-7.9228162514264337593543950335 with 28 places to the right of the decimal; the smallest non-0 number is +/-0.00000000000000000000000000000001
Date	8 bytes	January 1, 100 to December 31, 9999
Object	4 bytes	Any Object reference
String (variable-length)	10 bytes + string length	0 to approximately 2 billion characters
String (fixed-length)	Length of string	1 to approximately 65,400 characters
Variant (with numbers)	16 bytes	Any numeric value up to the range of a Double
Variant (with characters)	22 bytes + string length	Same range as for variable-length String
User-defined (using Type)	Number required by elements	The range of each element is the same as the range of its data type

If you do not declare a variable's type, it defaults to the `Variant` type. `Variant`s take up more memory than any other type because each `Variant` has to carry information with it that tells VBA what type of data it is currently storing, as well as store the data itself.

`Variant`s use more computer overhead when they are processed. VBA has to figure out what types it is dealing with and whether it needs to convert between types in order to process the number. If maximum processing speed is required for your application, you should declare your variable types, taking advantage of those types that use less memory when you can. For example, if you know your numbers will be whole numbers in the range of -32000 to +32000, you would use an `Integer` type.

Declaring Variable Type

You can declare a variable's type on a `Dim` statement, or related declaration statements such as `Public`. The following declares `Sales` to be a double precision floating-point number:

```
Dim Sales As Double
```

You can declare more than one variable on a `Dim`:

```
Dim SalesData As Double, Index As Integer, StartDate As Date
```

The following can be a trap:

```
Dim Col, Row, Sheet As Integer
```

Chapter 1: Primer in Excel VBA

Many users assume that this declares each variable to be `Integer`. This is not true. `Col` and `Row` are `Variant` because they have not been given a type. To declare all three as `Integer`, the line should be as follows:

```
Dim Col As Integer, Row As Integer, Sheet As Integer
```

Declaring Function and Parameter Types

If you have input parameters for sub procedures or function procedures, you can define each parameter type in the first line of the procedure as follows:

```
Function IsHoliday(WhichDay As Date)
Sub Marine(CrewSize As Integer, FuelCapacity As Double)
```

You can also declare the return value type for a function. The following example is for a function that returns a value of `True` or `False`:

```
Function IsHoliday(WhichDay As Date) As Boolean
```

Constants

You have seen that many intrinsic constants are built into VBA, such as `vbYes` and `vbNo`, discussed previously. You can also define your own constants. Constants are handy for holding numbers or pieces of text that do not change while your code is running, but that you want to use repeatedly in calculations and messages. Constants are declared using the `Const` keyword, as follows:

```
Const Pi = 3.14159265358979
```

You can include the constant's type in the declaration:

```
Const Version As String = "Release 3.9a"
```

Constants follow the same rules regarding scope as variables. If you declare a constant within a procedure, it will be local to that procedure. If you declare it in the declarations section of a module, it will be available to all procedures in the module. If you want to make it available to all modules, you can declare it to be `Public` as follows:

```
Public Const Error666 As String = "You can't do that"
```

Variable Naming Conventions

You can call your variables and user-defined functions anything you want, except where there is a clash with VBA keywords and function names. However, many programmers adopt a system whereby the variable or object type is included, in abbreviated form, in the variable name, usually as a prefix, so instead of declaring:

```
Dim SalesData As Double
```

you can use:

```
Dim dSalesData As Double
```

Wherever `dSalesData` appears in your code, you will be reminded that the variable is of type `Double`. Alternatively, you could use this line of code:

```
Dim dblSalesData As Double
```

For the sake of simplicity, this approach has not been used so far in this chapter, but from here onward, the examples will use a system to create variable names. This is the convention used in this book:

- ❑ One-letter prefixes for the common data types:

```
Dim iColumn As Integer
Dim lRow As Long
Dim dProduct As Double
Dim sName As String
Dim vValue As Variant
Dim bChoice As Boolean
```

- ❑ Two- or three-letter prefixes for object types:

```
Dim objExcel As Object
Dim rngData As Range
Dim wkbSales As Workbook
```

In addition to these characters, a lowercase `a` will be inserted in front of array variables, which are discussed later in this chapter. If the variable is a module-level variable, it will also have a lowercase `m` placed in front of it. If it is a public variable, it will have a lowercase `g` (for global) placed in front of it. For example, `malEffect` would be a module-level array variable containing long integer values.

Object Variables

The variables you have seen so far have held data such as numbers and text. You can also create object variables to refer to objects such as worksheets and ranges. The `Set` statement is used to assign an object reference to an object variable. Object variables should also be declared and assigned a type as with normal variables. If you don't know the type, you can use the generic term `Object` as the type:

```
Dim objWorkbook As Object
Set objWorkbook = ThisWorkbook
MsgBox objWorkbook.Name
```

It is more efficient to use the specific object type if you can. The following code creates an object variable `rng`, referring to cell B10 in Sheet1, in the same workbook as the code. It then assigns values to the object and the cell above:

```
Sub ObjectVariable()
    Dim rng As Range
    Set rng = ThisWorkbook.Worksheets("Sheet1").Range("C10")
    rng.Value = InputBox("Enter Sales for January")
    rng.Offset(-1, 0).Value = "January Sales"
End Sub
```

If you are going to refer to the same object more than once, it is more efficient to create an object variable than to keep repeating a lengthy specification of the object. It also makes code easier to read and write.

Chapter 1: Primer in Excel VBA

Object variables can also be very useful for capturing the return values of some methods, particularly when you are creating new instances of an object. For example, with either the `Workbooks` object or the `Worksheets` object, the `Add` method returns a reference to the new object. This reference can be assigned to an object variable so that you can easily refer to the new object in later code:

```
Sub NewWorkbook()  
    Dim wkb As Workbook, wks As Worksheet  
  
    Set wkb = Workbooks.Add  
    Set wks = wkb.Worksheets.Add(After:=wkb.Sheets(wkb.Sheets.Count))  
    wks.Name = "January"  
    wks.Range("A1").Value = "Sales Data"  
    wkb.SaveAs Filename:="JanSales.xlsx"  
End Sub
```

This example creates a new empty workbook and assigns a reference to it to the object variable `wkb`. A new worksheet is added to the workbook, after any existing sheets, and a reference to the new worksheet is assigned to the object variable `wks`. The name on the tab at the bottom of the worksheet is then changed to January, and the heading Sales Data is placed in cell A1. Finally, the new workbook is saved as `JanSales.xlsx`.

Note that the parameter after the `Worksheets.Add` is in parentheses. Because you are assigning the return value of the `Add` method to the object variable, any parameters must be in parentheses. If the return value of the `Add` method were ignored, the statement would be without parentheses, as follows:

```
wkb.Worksheets.Add After:=wkb.Sheets(wkb.Sheets.Count)
```

With...End With

Object variables provide a useful way to refer to objects in shorthand, and are also more efficiently processed by VBA than fully qualified object strings. Another way to reduce the amount of code you write, and also increase processing efficiency, is to use a `With...End With` structure. The final example in the previous section could be rewritten as follows:

```
With wkb  
    .Worksheets.Add After:=.Sheets(.Sheets.Count)  
End With
```

VBA knows that anything starting with a period is a property or a method of the object following the `With`. You can rewrite the entire `NewWorkbook` procedure to eliminate the `wkb` object variable, as follows:

```
Sub NewWorkbook()  
    Dim wks As Worksheet  
    With Workbooks.Add  
        Set wks = .Worksheets.Add(After:=.Sheets(.Sheets.Count))  
        wks.Name = "January"  
        wks.Range("A1").Value = "Sales Data"  
        .SaveAs Filename:="JanSales.xlsx"  
    End With  
End Sub
```

You can take this a step further and eliminate the `wks` object variable:


```

Sub NewWorkbook()
  With Workbooks.Add
    With .Worksheets.Add(After:=.Sheets(.Sheets.Count))
      .Name = "January"
      .Range("A1").Value = "Sales Data"
    End With
    .SaveAs Filename:="JanSales.xlsx"
  End With
End Sub

```

If you find this confusing, you can compromise with a combination of object variables and `With...End With`:

```

Sub NewWorkbook()
  Dim wkb As Workbook, wks As Worksheet

  Set wkb = Workbooks.Add
  With wkb
    Set wks = .Worksheets.Add(After:=.Sheets(.Sheets.Count))
    With wks
      .Name = "January"
      .Range("A1").Value = "Sales Data"
    End With
    .SaveAs Filename:="JanSales.xlsx"
  End With
End Sub

```

`With...End With` is useful when references to an object are repeated in a small section of code.

Making Decisions

VBA provides two main structures for making decisions and carrying out alternative processing, represented by the `If` and `Select Case` statements. `If` is the more flexible one, but `Select Case` is better when you are testing a single variable.

If Statements

`If` comes in three forms: the `IIf` function, the one-line `If` statement, and the block `If` structure. The following `dTax` function uses the `IIf` (Immediate `If`) function:

```

Function dTax(dProfitBeforeTax As Double) As Double
  dTax = IIf(dProfitBeforeTax > 0, 0.3 * dProfitBeforeTax, 0)
End Function

```

`IIf` is similar to the Excel worksheet `IF` function. It has three input arguments: the first is a logical test, the second is an expression that is evaluated if the test is true, and the third is an expression that is evaluated if the test is false.

In this example, the `IIf` function tests that the `dProfitBeforeTax` value is greater than 0. If the test is true, `IIf` calculates 30% of `dProfitBeforeTax`. If the test is false, `IIf` calculates 0. The calculated `IIf` value is then assigned to the return value of the `Tax` function. The `Tax` function can be rewritten using the single-line `If` statement as follows:

Chapter 1: Primer in Excel VBA

```
Function dTax(dProfitBeforeTax As Double) As Double
    If dProfitBeforeTax > 0 Then dTax = 0.3 * dProfitBeforeTax Else dTax = 0
End Function
```

One difference between `IIf` and the single-line `If` is that the `Else` section of the single-line `If` is optional. The third parameter of the `IIf` function must be defined. In VBA, it is often useful to omit the `Else`:

```
If dProfitBeforeTax < 0 Then MsgBox "A Loss has occurred", , "Warning"
```

Another difference is that, whereas `IIf` can only return a value to a single variable, the single-line `If` can assign values to different variables:

```
If iJohnsScore > iMarysScore Then iJohn = iJohn + 1 Else iMary = iMary + 1
```

Block If

If you want to carry out more than one action when a test is `true`, you can use a block `If` structure, as follows:

```
If iJohnsScore > iMarysScore Then
    iJohn = iJohn + 1
    iMary = iMary - 1
End If
```

Using a block `If`, you must not include any code after the `Then`, on the same line. You can have as many lines after the test as required, and you must terminate the scope of the block `If` with an `End If` statement. A block `If` can also have an `Else` section, as follows:

```
If iJohnsScore > iMarysScore Then
    iJohn = iJohn + 1
    iMary = iMary - 1
Else
    iJohn = iJohn - 1
    iMary = iMary + 1
End If
```

A block `If` can also have as many `ElseIf` sections as required:

```
If iJohnsScore > iMarysScore Then
    iJohn = iJohn + 1
    iMary = iMary - 1
ElseIf iJohnsScore < iMarysScore Then
    iJohn = iJohn - 1
    iMary = iMary + 1
Else
    iJohn = iJohn + 1
    iMary = iMary + 1
End If
```

When you have a block `If` followed by one or more `ElseIf` tests, VBA keeps testing until it finds a `true` section. It executes the code for that section and then proceeds directly to the statement following the `End If`. If no test is `true`, the `Else` section is executed.

A block `If` does nothing when all tests are false and the `Else` section is missing. Block `If` tests can be nested, one inside the other. You should make use of indenting to show the scope of each block. This is vital—you can get into an awful muddle with the nesting of `If` blocks within other `If` blocks, and `If` blocks within `Else` blocks, and so on. If code is unindented, it isn't easy, in a long series of nested `If` tests, to match each `End If` with each `If`:

```
If Not ThisWorkbook.Saved Then
    lAnswer = MsgBox("Do you want to save your changes", vbQuestion + _
                                                             vbYesNo)

    If lAnswer = vbYes Then
        ThisWorkbook.Save
        MsgBox ThisWorkbook.Name & " has been saved"
    End If
End If
```

This code uses the `Saved` property of the `Workbook` object containing the code to see if the workbook has been saved since changes were last made to it. If changes have not been saved, the user is asked if they want to save changes. If the answer is yes, the inner block `If` saves the workbook and informs the user.

Select Case

The following block `If` is testing the same variable value in each section:

```
Function vPrice(sProduct As String) As Variant
    If sProduct = "Apples" Then
        vPrice = 12.5
    ElseIf sProduct = "Oranges" Then
        vPrice = 15
    ElseIf sProduct = "Pears" Then
        vPrice = 18
    ElseIf sProduct = "Mangoes" Then
        vPrice = 25
    Else
        vPrice = CVErr(xlErrNA)
    End If
End Function
```

If `sProduct` is not found, the `vPrice` function returns an Excel error value of `#NA`. Note that `vPrice` is declared as a `Variant` so it can handle the error value as well as numeric values. For a situation like this, `Select Case` is a more elegant construction. It looks like this:

```
Function vPrice(sProduct As String) As Variant
    Select Case sProduct
        Case "Apples"
            vPrice = 12.5
        Case "Oranges"
            vPrice = 15
        Case "Pears"
            vPrice = 18
        Case "Mangoes"
            vPrice = 25
        Case Else
            vPrice = CVErr(xlErrNA)
    End Select
End Function
```

Chapter 1: Primer in Excel VBA

If you have only one statement per case, the following format works quite well. You can place multiple statements on a single line by placing a colon between statements:

```
Function vPrice(sProduct As String) As Variant
    Select Case sProduct
        Case "Apples": vPrice = 12.5
        Case "Oranges": vPrice = 15
        Case "Pears": vPrice = 18
        Case "Mangoes": vPrice = 25
        Case Else: vPrice = CVErr(xlErrNA)
    End Select
End Function
```

Select Case can also handle ranges of numbers or text, as well as comparisons using the keyword `Is`. The following example calculates a fare of 0 for infants up to 3 years old and anyone older than 65, with two ranges between. Negative ages generate an error:

```
Function vFare(iAge As Integer) As Variant
    Select Case iAge
        Case 0 To 3, Is > 65
            vFare = 0
        Case 4 To 15
            vFare = 10
        Case 16 To 65
            vFare = 20
        Case Else
            vFare = CVErr(xlErrNA)
    End Select
End Function
```

Looping

All computer languages provide a mechanism for repeating the same, or similar, operations in an efficient way. VBA has two main structures that allow you to loop through the same code over and over again. They are the `Do...Loop` and the `For...Next` loop.

The `Do...Loop` is for those situations where the loop will be terminated when a logical condition applies, such as reaching the end of your data. The `For...Next` loop is for situations where you can predict in advance how many times you want to loop, such as when you want to enter expenses for the 10 people in your department.

VBA also has an interesting variation on the `For...Next` loop that is used to process all the objects in a collection—the `For Each...Next` loop. You can use it to process all the cells in a range or all the sheets in a workbook, for example.

Do...Loop

To illustrate the use of a `Do...Loop`, construct a sub procedure to shade every second line of a worksheet, as shown in Figure 1-27, to make it more readable. You want to apply the macro to different report sheets with different numbers of products, so the macro will need to test each cell in the A column until it gets to an empty cell to determine when to stop.

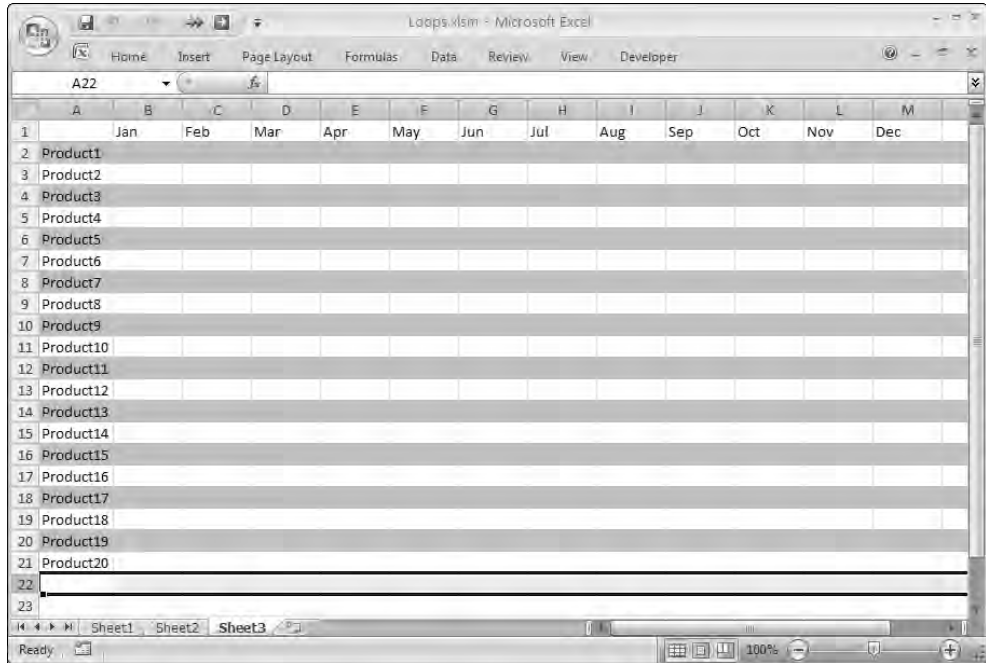


Figure 1-27

The first macro will select every other row and apply the formatting:

```
Sub ShadeEverySecondRow()
    Range("A2").EntireRow.Select
    Do While ActiveCell.Value <> ""
        Selection.Interior.ColorIndex = 15
        ActiveCell.Offset(2, 0).EntireRow.Select
    Loop
End Sub
```

`ShadeEverySecondRow` begins by selecting row 2 in its entirety. When you select an entire row, the left-most cell (in column A) becomes the active cell. The code between the `Do` and `Loop` statements is then repeated `While` the value property of the active cell is not a zero-length string, that is, the active cell is not empty. In the loop, the macro sets the interior color index of the selected cells to 15, which is gray. Then the macro selects the entire row, two rows under the active cell. When a row is selected that has an empty cell in column A, the `While` condition is no longer true and the loop terminates.

You can make `ShadeEverySecondRow` run faster by avoiding selecting. It is seldom necessary to select cells in VBA, but you are led into this way of doing things because that's the way you do it manually, and that's what you get from the macro recorder.

The following version of `ShadeEverySecondRow` does not select cells, and it runs considerably faster. It sets up an index `1Row`, which indicates the row of the worksheet and is initially assigned a value of 2. The `Cells` property of the worksheet allows you to refer to cells by row number and column number, so

Chapter 1: Primer in Excel VBA

when the loop starts, `Cells(1Row, 1)` refers to cell A2. Each time around the loop, `i` is increased by two. You can, therefore, change any reference to the active cell to a `Cells(1Row, 1)` reference and apply the `EntireRow` property to `Cells(1Row, 1)` to refer to the complete row:

```
Sub ShadeEverySecondRow()  
    Dim lRow As Long  
    lRow= 2  
    Do Until IsEmpty(Cells(lRow, 1))  
        Cells(lRow, 1).EntireRow.Interior.ColorIndex = 15  
        lRow= lRow+ 2  
    Loop  
End Sub
```

To illustrate some alternatives, two more changes have been made on the `Do` statement line in the previous code. Either `While` or `Until` can be used after the `Do`, so the test has been changed to an `Until` and you have used the VBA `IsEmpty` function to test for an empty cell.

The `IsEmpty` function is the best way to test that a cell is empty. If you use `If Cells(1Row, 1) = ""`, the test will be true for a formula that calculates a zero-length string.

It is also possible to exit a loop using a test within the loop and the `Exit Do` statement, as follows, which also shows another way to refer to entire rows:

```
Sub ShadeEverySecondRow()  
    Dim lRow as Long  
    lRow= 0  
    Do  
        lRow= lRow+ 2  
        If IsEmpty(Cells(lRow, 1)) Then Exit Do  
        Rows(lRow).Interior.ColorIndex = 15  
    Loop  
End Sub
```

Yet another alternative is to place the `While` or `Until` on the `Loop` statement line. This ensures that the code in the loop is executed at least once. When the test is on the `Do` line, it is possible that the test will be false to start with, and the loop will be skipped.

Sometimes, it makes more sense if the test is on the last line of the loop. In the following example, it seems more sensible to test `sPassword` after getting input from the user, although the code would still work if the `Until` statement were placed on the `Do` line:

```
Sub GetPassword()  
    Dim sPassword As String, i As Integer  
    i = 0  
    Do  
        i = i + 1  
        If i > 3 Then  
            MsgBox "Sorry, Only three tries"  
            Exit Sub  
        End If  
    Loop
```

```

    sPassWord = InputBox("Enter Password")
    Loop Until sPassWord = "XXX"
    MsgBox "Welcome"
End Sub

```

GetPassword loops until the password XXX is supplied, or the number of times around the loop exceeds three.

For...Next Loop

The For...Next loop differs from the Do...Loop in two ways. It has a built-in counter that is automatically incremented each time the loop is executed, and it is designed to execute until the counter exceeds a predefined value, rather than depending on a user-specified logical test. The following example places the full file path and name of the workbook into the center footer for each worksheet in the active workbook:

```

Sub FilePathInFooter()
    Dim i As Integer, sFilePath As String

    sFilePath = ActiveWorkbook.FullName
    For i = 1 To Worksheets.Count Step 1
        Worksheets(i).PageSetup.CenterFooter = sFilePath
    Next i
End Sub

```

Versions of Excel prior to Excel 2002 do not have an option to automatically include the full file path in a custom header or footer, so this macro inserts the information as text. It begins by assigning the FullName property of the active workbook to the variable sFilePath. The loop starts with the For statement and loops on the Next statement. i is used as a counter, starting at 1 and finishing when i exceeds Worksheets.Count, which uses the Count property of the Worksheets collection to determine how many worksheets there are in the active workbook.

The Step option defines the amount that i will be increased each time around the loop. Step 1 could be left out of this example, because a step of 1 is the default value. In the loop, i is used as an index to the Worksheets collection to specify each individual Worksheet object. The PageSetup property of the Worksheet object refers to the PageSetup object in that worksheet, so that the CenterFooter property of the PageSetup object can be assigned the sFilePath text.

The following example shows how you can step backwards. It takes a complete file path and strips out the filename, excluding the file extension. The example uses the FullName property of the active workbook as input, but the same code could be used with any file path. It starts at the last character in the file path and steps backwards until it finds the period between the filename and its extension, and then the backslash character before the filename. It then extracts the characters between the two:

```

Sub GetFileName()
    Dim iBackSlash As Integer, iPoint As Integer
    Dim sFilePath As String, sFileName As String
    Dim i As Integer

    sFilePath = ActiveWorkbook.FullName
    For i = Len(sFilePath) To 1 Step -1

```

```
    If Mid$(sFilePath, i, 1) = "." Then
        iPoint = i
        Exit For
    End If
Next i
If iPoint = 0 Then iPoint = Len(sFilePath) + 1
For i = iPoint - 1 To 1 Step -1
    If Mid$(sFilePath, i, 1) = "\" Then
        iBackSlash = i
        Exit For
    End If
Next i
sFileName = Mid$(sFilePath, iBackSlash + 1, iPoint - iBackSlash - 1)
MsgBox sFileName
End Sub
```

The first `For...Next` loop uses the `Len` function to determine how many characters are in the `sFilePath` variable, and `i` is set up to step backwards, counting from the last character position, working toward the first character position. The `Mid$` function extracts the character from `sFilePath` at the position defined by `i` and tests it to see if it is a period.

When a period is found, the position is recorded in `iPoint` and the first `For...Next` loop is exited. If the filename has no extension, no period is found and `iPoint` will have its default value of 0. In this case, the `If` test records an imaginary period position in `iPoint` that is one character beyond the end of the filename.

The same technique is used in the second `For...Next` loop as the first, starting one character before the period, to find the position of the backslash character, and storing the position in `iBackSlash`. The `Mid$` function is then used to extract the characters between the backslash and the period.

For Each...Next Loop

When you want to process every member of a collection, you can use the `For Each...Next` loop. The following example is a rework of the `FilePathInFooter` procedure:

```
Sub FilePathInFooter()
    Dim sFilePath As String, wks As Worksheet

    sFilePath = ActiveWorkbook.FullName
    For Each wks In Worksheets
        wks.PageSetup.CenterFooter = sFilePath
    Next wks
End Sub
```

The loop steps through all the members of the collection. During each pass, a reference to the next member of the collection is assigned to the object variable `wks`.

The following example lists all the files in the root directory of the C: drive. It uses the Windows Scripting `FileSystemObject` to create a reference to the C drive root directory. The example uses a `For Each...Next` loop to display the names of all the files in the directory:


```

Sub FileList()
    'Listing files with a For...Each loop

    Dim objFSO As Object
    Dim objFolder As Object
    Dim objFile As Object

    'Create a reference to the FileSystemObject
    Set objFSO = CreateObject("Scripting.FileSystemObject")

    'Create a folder reference
    Set objFolder = objFSO.GetFolder("C:\")

    'List files in folder
    For Each objFile In objFolder.Files
        MsgBox objFile.Name
    Next objFile

End Sub

```

The code uses techniques that are discussed in Chapter 19 to reference objects outside the Excel object model. If you test this procedure on a directory with lots of files, and get tired of clicking OK, don't forget that you can break out of the code with Ctrl+Break.

Arrays

Arrays are VBA variables that can hold more than one item of data. An array is declared by including parentheses after the array name. An integer is placed within the parentheses, defining the number of elements in the array:

```
Dim avData(2)
```

You assign values to the elements of the array by indicating the element number as follows:

```

avData(0) = 1
avData(1) = 10
avData(2) = 100

```

The number of elements in the array depends on the array base. The default base is 0, which means that the first data element is item 0. `Dim avData(2)` declares a three-element array if the base is 0. Alternatively, you can place the following statement in the declarations section at the top of your module to declare that arrays are 1-based:

```
Option Base 1
```

With a base of 1, `Dim avData(2)` declares a two-element array. Item 0 does not exist.

You can use the following procedure to test the effect of the `Option Base` statement:

```
Sub Array1()  
    Dim aiData(10) As Integer  
    Dim sMessage As String, i As Integer  
  
    For i = LBound(aiData) To UBound(aiData)  
        aiData(i) = i  
    Next i  
    sMessage = "Lower Bound = " & LBound(aiData) & vbCr  
    sMessage = sMessage & "Upper Bound = " & UBound(aiData) & vbCr  
    sMessage = sMessage & "Num Elements = " & WorksheetFunction.Count(aiData) & vbCr  
    sMessage = sMessage & "Sum Elements = " & WorksheetFunction.Sum(aiData)  
    MsgBox sMessage  
End Sub
```

Array1 uses the `LBound` (lower bound) and `UBound` (upper bound) functions to determine the lowest and highest index values for the array. It uses the `Count` worksheet function to determine the number of elements in the array. If you run this code with `Options Base 0`, or no `Options Base` statement, in the declarations section of the module, it will show a lowest index number of 0 and 11 elements in the array. With `Options Base 1`, it shows a lowest index number of 1 and 10 elements in the array.

Note the use of the intrinsic constant `vbCr`, which contains a carriage return character. `vbCr` is used to break the message text to a new line.

If you want to make your array size independent of the `Option Base` statement, you can explicitly declare the lower bound as well as the upper bound as follows:

```
Dim avData(1 To 2)
```

Arrays are very useful for processing lists or tables of items. If you want to create a short list, you can use the `Array` function as follows:

```
Dim avData As Variant  
avData = Array("North", "South", "East", "West")
```

You can then use the list in a `For...Next` loop. For example, you could open and process a series of workbooks called `North.xls`, `South.xls`, `East.xls`, and `West.xls`:

```
Sub Array2()  
    Dim avData As Variant, wkb As Workbook  
    Dim i As Integer  
  
    avData = Array("North", "South", "East", "West")  
    For i = LBound(avData) To UBound(avData)  
        Set wkb = Workbooks.Open(FileName:=avData(i) & ".xls")  
        'Process data here  
        wkb.Close SaveChanges:=True  
    Next i  
End Sub
```

Multi-Dimensional Arrays

So far you have only looked at arrays with a single dimension. You can actually define arrays with up to 60 dimensions, although few people would use more than two or three dimensions. The following statements declare two-dimensional arrays:

```
Dim avData(10,20)
Dim avData(1 To 10,1 to 20)
```

You can think of a two-dimensional array as a table of data. The preceding example defines a table with 10 rows and 20 columns.

Arrays are very useful in Excel for processing the data in worksheet ranges. It can be far more efficient to load the values in a range into an array, process the data, and write it back to the worksheet, than to access each cell individually.

The following procedure shows how you can assign the values in a range to a `Variant`. The code uses the `LBound` and `UBound` functions to find the number of dimensions in `avData`. Note that there is a second parameter in `LBound` and `UBound` to indicate which index you are referring to. If you leave this parameter out, the functions refer to the first index:

```
Sub Array3()
    Dim avData As Variant, vUBound As Variant
    Dim Message As String, i As Integer

    avData = Range("A1:A20").Value
    i = 1
    Do
        Message = "Lower Bound = " & LBound(avData, i) & vbCrLf
        Message = Message & "Upper Bound = " & UBound(avData, i) & vbCrLf
        MsgBox Message, , "Index Number = " & i
        i = i + 1
        On Error Resume Next
        vUBound = UBound(avData, i)
        If Err.Number <> 0 Then Exit Do
        On Error GoTo 0
    Loop
    Message = "Number of Non Blank Elements =" & _
        & WorksheetFunction.CountA(avData) & vbCrLf
    MsgBox Message
End Sub
```

The first time around, the `Do . . . Loop`, `Array3` determines the upper and lower bounds of the first dimension of `avData`, as `i` has a value of 1. It then increases the value of `i` to look for the next dimension. It exits the loop when an error occurs, indicating that no more dimensions exist.

By substituting different ranges into `Array3`, you can determine that the array created by assigning a range of values to a `Variant` is two-dimensional, even if there is only one row or one column in the range. You can also determine that the lower bound of each index is 1, regardless of the `Option Base` setting in the declarations section.

Dynamic Arrays

When writing your code, it is sometimes not possible to determine the size of the array that will be required. For example, you might want to load the names of all the `.xls` files in the current directory into an array. You won't know in advance how many files there will be. One alternative is to declare an array that is big enough to hold the largest possible amount of data—but this would be inefficient. Instead, you can define a dynamic array and set its size when the procedure runs.

You declare a dynamic array by leaving out the dimensions:

```
Dim avData()
```

You can declare the required size at run time with a `ReDim` statement, which can use variables to define the bounds of the indexes:

```
ReDim avData(iRows, iColumns)
ReDim avData(iminRow to imaxRow, iminCol to imaxCol)
```

`ReDim` will re-initialize the array and destroy any data in it, unless you use the `Preserve` keyword. `Preserve` is used in the following procedure that uses a `Do...Loop` to load the names of files into the dynamic array called `asFNames`, increasing the upper bound of its index by one each time to accommodate the new name.

The `Dir` function returns the first filename found that matches the wildcard specification in `sFType`. Subsequent usage of `Dir`, with no parameter, repeats the same specification, getting the next file that matches, until it runs out of files and returns a zero-length string:

```
Sub FileNames()
    Dim sFName As String
    Dim asFNames() As String
    Dim sFType As String
    Dim i As Integer

    sFType = "*.xls"
    sFName = Dir(sFType)
    Do Until sFName = ""
        i = i + 1
        ReDim Preserve asFNames(1 To i)
        asFNames(i) = sFName
        sFName = Dir
    Loop
    If i = 0 Then
        MsgBox "No files found"
    Else
        For i = 1 To UBound(asFNames)
            MsgBox asFNames(i)
        Next i
    End If
End Sub
```

If you intend to work on the files in a directory and save the results, it is a good idea to get all the filenames first, as in the `FileNames` procedure, and use that list to process the files. It is not a good idea to rely on the `Dir` function to give you an accurate file list while you are in the process of reading and overwriting files.

Run-Time Error-Handling

When you are designing an application, you should try to anticipate any problems that could occur when the application is used in the real world. You can remove all the bugs in your code and have flawless logic that works with all permutations of conditions, but a simple operational problem could still bring your code crashing down with a less than helpful message displayed to the user.

For example, if you try to save a workbook file to the floppy disk in the A: drive, and there is no disk in the A: drive, your code will grind to a halt and display a message that will probably not mean anything to the average user.

If you anticipate this particular problem, you can set up your code to gracefully deal with the situation. VBA allows you to trap error conditions using the following statement:

```
On Error GoTo LineLabel
```

LineLabel is a marker that you insert at the end of your normal code, as shown in the following code with the line label `errTrap`. Note that a colon follows the line label. The line label marks the start of your error recovery code and should be preceded by an `Exit` statement to prevent execution of the error recovery code when no error occurs:

```
Sub ErrorTrap1()
    Dim lAnswer As Long, sMyFile As String
    Dim sMessage As String, sCurrentPath As String

    On Error GoTo errTrap
    sCurrentPath = CurDir$

    ChDrive "A"
    ChDrive sCurrentPath
    ChDir sCurrentPath
    sMyFile = "A:\Data.xls"
    Application.DisplayAlerts = False
    ActiveWorkbook.SaveAs Filename:=sMyFile
TidyUp:
    ChDrive sCurrentPath
    ChDir sCurrentPath
Exit Sub
errTrap:
    sMessage = "Error No: = " & Err.Number & vbCr
    sMessage = sMessage & Err.Description & vbCr & vbCr
    sMessage = sMessage & "Please place a disk in the A: drive" & vbCr
    sMessage = sMessage & "and press OK" & vbCr & vbCr
    sMessage = sMessage & "Or press Cancel to abort File Save"
    lAnswer = MsgBox(sMessage, vbQuestion + vbOKCancel, "Error")
    If lAnswer = vbCancel Then Resume TidyUp
    Resume
End Sub
```

Once the `On Error` statement is executed, error trapping is enabled. If an error occurs, no message is displayed and the code following the line label is executed. You can use the `Err` object to obtain information about the error. The `Number` property of the `Err` object returns the error number, and the `Description` property returns the error message associated with the error. You can use `Err.Number` to

Chapter 1: Primer in Excel VBA

determine the error when it is possible that any of a number of errors could occur. You can incorporate `Err.Description` into your own error message, if appropriate.

In Excel 5 and 95, `Err` was not an object, but a function that returned the error number. Because `Number` is the default property of the `Err` object, using `Err` by itself is equivalent to using `Err.Number`, and the code from the older versions of Excel still works in Excel 97 and later versions.

The code in `ErrorTrap1`, after executing the `On Error` statement, saves the current directory drive and path into the variable `sCurrentPath`. It then executes the `ChDrive` statement to try to activate the A: drive. If there is no disk in the A: drive, error 68—(Device unavailable) occurs and the error recovery code executes. For illustration purposes, the error number and description are displayed and the user is given the opportunity to either place a disk in the A: drive and continue, or abort the save.

If the user wishes to stop, you branch back to `TidyUp` and restore the original drive and directory settings. Otherwise the `Resume` statement is executed. This means that execution returns to the statement that caused the error. If there is still no disk in the A: drive, the error recovery code is executed again. Otherwise the code continues normally.

The only reason for the `ChDrive "A"` statement is to test the readiness of the A: drive, so the code restores the stored drive and directory path. The code sets the `DisplayAlerts` property of the `Application` object to `False`, before saving the active workbook. This prevents a warning if an old file called `Data.xls` is being replaced by the new `Data.xls`. (See Chapter 3 for more on `DisplayAlerts`.)

The `Resume` statement comes in three forms:

- `Resume` causes execution of the statement that caused the error.
- `Resume Next` returns execution to the statement following the statement that caused the error, so the problem statement is skipped.
- `Resume LineLabel` jumps back to any designated line label in the code, so you can decide to resume where you want.

The following code uses `Resume Next` to skip the `Kill` statement, if necessary. The charmingly named `Kill` statement removes a file from disk. The following code removes any file with the same name as the one you are about to save, so there will be no need to answer the warning message about overwriting the existing file.

The problem is that `Kill` will cause a fatal error if the file does not exist. If `Kill` does cause a problem, the error recovery code executes and you use `Resume Next` to skip `Kill` and continue with `SaveAs`. The `MsgBox` is there for educational purposes only. You would not normally include it:

```
Sub ErrorTrap2()  
    Dim sMyFile As String, sMessage As String  
    Dim sAnswer As String  
  
    On Error GoTo errTrap  
  
    Workbooks.Add  
    sMyFile = "C:\Data.xls"  
    Kill sMyFile
```

```

ActiveWorkbook.SaveAs Filename:=sMyFile
ActiveWorkbook.Close

Exit Sub
errTrap:
    sMessage = "Error No: = " & Err.Number & vbCrLf
    sMessage = sMessage & Err.Description & vbCrLf & vbCrLf
    sMessage = sMessage & "File does not exist"
    sAnswer = MsgBox(sMessage, vbInformation, "Error")
    Resume Next
End Sub

```

On Error Resume Next

As an alternative to On Error GoTo, you can use:

```
On Error Resume Next
```

This statement causes errors to be ignored, so it should be used with caution. However, it has many uses. The following code is a rework of ErrorTrap2:

```

Sub ErrorTrap3()
    Dim sMyFile As String

    Workbooks.Add
    sMyFile = "C:\Data.xls"
    On Error Resume Next
    Kill sMyFile
    On Error GoTo 0
    ActiveWorkbook.SaveAs Filename:=sMyFile
    ActiveWorkbook.Close
End Sub

```

Use On Error Resume Next just before the Kill statement. If C:\Data.xls does not exist, the error caused by Kill is ignored and execution continues on the next line. After all, you don't care if the file does not exist. That's the situation you are trying to achieve.

On Error GoTo 0 is used to turn on normal VBA error-handling again. Otherwise, any further errors would be ignored. It is best not to try to interpret this statement, which appears to be directing error-handling to line 0. Just accept that it works.

You can use On Error Resume Next to write code that would otherwise be less efficient. The following sub procedure determines whether a name exists in the active workbook:

```

Sub TestForName()
    If bNameExists("SalesData") Then
        MsgBox "Name Exists"
    Else
        MsgBox "Name does not exist"
    End If
End Sub

Function bNameExists(sMyName As String) As Boolean

```

```
Dim sName As String
On Error Resume Next
sName = Names(sMyName).RefersTo
If Err.Number <> 0 Then
    bNameExists = False
    Err.Clear
Else
    bNameExists = True
End If
End Function
```

`TestForName` calls the `bNameExists` function, which uses `On Error Resume Next` to prevent a fatal error when it tries to assign the name's `RefersTo` property to a variable. There is no need for `On Error GoTo 0` here, because error-handling in a procedure is disabled when a procedure exits, although `Err.Number` is not cleared.

If no error occurred, the `Number` property of the `Err` object is 0. If `Err.Number` has a non-0 value, an error occurred, presumably because the name did not exist, so `bNameExists` is assigned a value of `False` and the error is cleared. The alternative to this single pass procedure is to loop through all the names in the workbook, looking for a match. If there are lots of names, this can be a slow process.

Summary

In this chapter, you have seen those elements of the VBA language that enable you to write useful and efficient procedures. You have seen how to add interaction to macros with the `MsgBox` and `InputBox` functions, how to use variables to store information, and how to get help with VBA keywords.

You have seen how to declare variables and define their type, and the effect on variable scope and lifetime of different declaration techniques. In addition, you used the block `If` and `Select Case` structures to perform tests and carry out alternative calculations, and `Do . . . Loop` and `For . . . Next` loops that allow you to efficiently repeat similar calculations. You have seen how arrays can be used, particularly with looping procedures. Moreover, you learned how to use `On Error` statements to trap errors.

When writing VBA code for Excel, the easiest way to get started is to use the macro recorder. You can then modify that code, using the VBE, to better suit your purposes and to operate efficiently. Using the Object Browser, Help screens, and the reference section of this book, you can discover objects, methods, properties, and events that can't be found with the macro recorder. Using the coding structures provided by VBA, you can efficiently handle large amounts of data and automate tedious processes.

You now have the knowledge required to move on to the next chapter, where you will find a rich set of practical examples showing you how to work with key Excel objects. You will discover how to create your own user interface, setting up your own Ribbon buttons and dialog boxes, and embedding controls in your worksheets to enable yourself and others to work more productively.

2

The Application Object

This chapter examines a range of Excel functionality, looking at features that are not necessarily related to each other. In general, the Excel object model contains objects designed to address quite specific tasks. The `Application` object sits at the top of the Excel object model hierarchy and contains all the other objects in Excel. It also acts as a catch-all area for properties and methods that do not fall neatly into any other object, but are necessary for programmatic control of Excel. There are `Application` properties that control screen updating and toggle alert messages, for example. There is an `Application` method that calculates the formulas in the open workbooks.

Globals

Many of the `Application` object's methods and properties are also members of `<globals>`, which can be found at the top of the list of classes in the Object Browser, as shown in Figure 2-1.

If a property or method is in `<globals>`, you can refer to that property or method without a preceding reference to an object. For example, the following two references are equivalent:

```
Application.ActiveCell  
ActiveCell
```

However, you do need to be careful. It is easy to assume that frequently used `Application` object properties, such as `ScreenUpdating`, are `<globals>` when they are not. The following code is correct:

```
Application.ScreenUpdating = False
```

You will get unexpected results with the following:

```
ScreenUpdating = False
```

This code sets up a new variable and assigns the value `False` to it. You can easily avoid this error by having the line of code `Option Explicit` at the top of each module so that such references are flagged as undefined variables when your code is compiled.

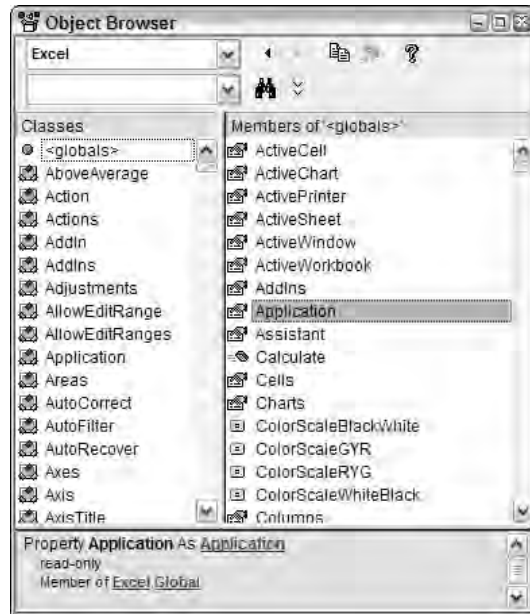


Figure 2-1

Remember that you can have `Option Explicit` automatically inserted in new modules if you use `Tools > Options` in the VBE window and, under the Editor tab, tick the `Require Variable Declaration` checkbox.

The Active Properties

The `Application` object provides many shortcuts that allow you to refer to active objects without naming them explicitly. This makes it possible to discover what is currently active when your macro runs. It also makes it easy to write generalized code that can be applied to objects of the same type with different names.

The following `Application` object properties are global properties that allow you to refer to active objects:

- `ActiveCell`
- `ActiveChart`
- `ActivePrinter`
- `ActiveSheet`
- `ActiveWindow`

- ❑ `ActiveWorkbook`
- ❑ `Selection`

If you have just created a new workbook and want to save it with a specific filename, using the `ActiveWorkbook` property is an easy way to return a reference to the new `Workbook` object:

```
Workbooks.Add  
ActiveWorkbook.SaveAs Filename:="C:\Data.xls"
```

If you want to write a macro that can apply a bold format to the currently selected cells, you can use the `Selection` property to return a reference to the `Range` object containing the selected cells:

```
Selection.Font.Bold = True
```

Be aware that `Selection` will not refer to a `Range` object if another type of object, such as a `Shape` object, is currently selected or the active sheet is not a worksheet. You might want to build a check into a macro to ensure that a worksheet is selected before attempting to enter data into it:

```
If TypeName(ActiveSheet) <> "Worksheet" Or _  
    TypeName(Selection) <> "Range" Then  
    MsgBox "You can only run this macro in a range", vbCritical  
    Exit Sub  
End If
```

Display Alerts

It can be annoying to have to respond to system alerts while a macro runs. For example, if a macro deletes a worksheet, an alert message appears and you have to click the OK button to continue. However, there is also the possibility of a user clicking the Cancel button, which would abort the delete operation and could adversely affect subsequent code where the delete operation was assumed to have been carried out.

You can suppress most alerts by setting the `DisplayAlerts` property to `False`. When you suppress an alert dialog box, the action that is associated with the default button in that box is automatically carried out, as follows:

```
Application.DisplayAlerts = False  
ActiveSheet.Delete  
Application.DisplayAlerts = True
```

It is not necessary to reset `DisplayAlerts` to `True` at the end of your macro because VBA does this automatically. However, it is usually a good idea, after suppressing a particular message, to turn the alerts back on so that any unexpected warnings do appear on screen.

`DisplayAlerts` is commonly used to suppress the warning that you are about to overwrite an existing file using `File` ⇨ `SaveAs`. When you suppress this warning, the default action is taken and the file is overwritten without interrupting the macro.

Screen Updating

It can likewise be annoying to see the screen change and flicker while a macro is running. This happens with macros that select or activate objects and is typical of the code generated by the macro recorder.

It is better to avoid selecting objects in VBA. It is seldom necessary to do this, and your code will run faster if you can avoid selecting or activating objects. Most of the code in this book avoids selecting where possible.

If you want to freeze the screen while your macro runs, you use the following line of code:

```
Application.ScreenUpdating = False
```

The screen remains frozen until you assign the property a value of `True`, or when your macro finishes executing and returns control to the user interface. There is no need to restore `ScreenUpdating` to `True`, unless you want to display screen changes while your macro is still running.

There is one situation where it is a good idea to set `ScreenUpdating` to `True` while your macro is running. If you display a user form or built-in dialog box while your macro is running, you should make sure screen updating is on before showing the object. If screen updating is off and the user drags the user form around the screen, the user form will act as an eraser on the screen behind it. You can turn screen updating off again after showing the object.

A beneficial side effect of turning off screen updating is that your code runs faster. It will even speed up code that avoids selecting objects, where little screen updating is required. Your code runs at maximum speed when you avoid selecting and turn off screen updating.

Evaluate

The `Evaluate` method can be used to calculate Excel worksheet formulas and generate references to `Range` objects. The normal syntax for the `Evaluate` method is as follows:

```
Evaluate("Expression")
```

You can also use a shortcut format where you omit the quotes and place square brackets around the expression, as follows:

```
[Expression]
```

`Expression` can be any valid worksheet calculation, with or without the equal sign on the left, or it can be a reference to a range of cells. The worksheet calculations can include worksheet functions that are not made available to VBA through the `WorksheetFunction` object, or they can be worksheet array formulas. You will find more information about the `WorksheetFunction` object later in this chapter.

For instance, the `ISBLANK` function, which you can use in your worksheet formulas, is not available to VBA through the `WorksheetFunction` object, because the VBA equivalent function `IsEmpty` provides the same functionality. All the same, you can use `ISBLANK` if you need to. The following two examples are equivalent and return `True` if `A1` is empty or `False` if `A1` is not empty:

```
MsgBox Evaluate("=ISBLANK(A1) ")
MsgBox [ISBLANK(A1)]
```

The advantage of the first technique is that you can generate the string value using code, which makes it very flexible. The second technique is shorter, but you can only change the expression by editing your code. The following procedure displays a `True` or `False` value to indicate whether or not the active cell is empty, and illustrates the flexibility of the first technique:

```
Sub IsActiveCellEmpty()
    Dim sFunctionName As String, sCellReference As String
    sFunctionName = "ISBLANK"
    sCellReference = ActiveCell.Address
    MsgBox Evaluate(sFunctionName & "(" & sCellReference & ")")
End Sub
```

Note that you cannot evaluate an expression containing variables using the second technique.

The following two lines of code show you two ways you can use `Evaluate` to generate a reference to a `Range` object, and assign a value to that object:

```
Evaluate("A1").Value = 10
[A1].Value = 10
```

The first expression is unwieldy and is rarely used, but the second is a convenient way to refer to a `Range` object, although it is not very flexible. You can further shorten the expressions by omitting the `Value` property, because this is the default property of the `Range` object:

```
[A1] = 10
```

More interesting uses of `Evaluate` include returning the contents of a workbook's `Names` collection and efficiently generating arrays of values. The following code creates a hidden name to store a password. Hidden names cannot be seen in the `Insert` ⇨ `Name` ⇨ `Define` dialog box, so they are a convenient way to store information in a workbook without cluttering the user interface:

```
Names.Add Name:="PassWord", RefersTo:="Bazonkas", Visible:=False
```

You can then use the hidden data in expressions like the following:

```
sUserInput = InputBox("Enter Password")
If sUserInput = [PassWord] Then
    ...
```

The use of names for storing data is discussed in more detail in Chapter 5.

Chapter 2: The Application Object

The `Evaluate` method can also be used with arrays. The following expression generates a `Variant` array with two dimensions, 100 rows and one column, containing the values from 101 to 200. This process is carried out more efficiently than using a `For . . . Next` loop:

```
vRowArray = [ROW(101:200)]
```

Similarly, the following code assigns the values 101 to 200 to the range `B1:B100`, and again does it more efficiently than a `For . . . Next` loop:

```
[B1:B100] = [ROW(101:200)]
```

InputBox

VBA has an `InputBox` function that provides an easy way to prompt for input data. There is also the `InputBox` method of the `Application` object that produces a very similar dialog box for obtaining data, but is more powerful. It allows you to control the type of data that must be supplied by the user, and allows you to detect when the `Cancel` button is clicked.

If you have an unqualified reference to `InputBox` in your code, as follows, you are using the VBA `InputBox` function:

```
sAnswer = InputBox(prompt:="Enter range")
```

The user can only type data into the dialog box. It is not possible to point to a cell with the mouse. The return value from the `InputBox` function is always a string value, and there is no check on what that string contains. If the user enters nothing, a zero-length string is returned. If the user clicks the `Cancel` button, a zero-length string is also returned. Your code cannot distinguish between no entry and the result of clicking `Cancel`.

The following example uses the `Application` object's `InputBox` method to prompt for a range:

```
vAnswer = Application.InputBox(Prompt:="Enter range", Type:=8)
```

The `Type` parameter can take the following values, or any sum of the following values if you want to allow for multiple types.

Value of Type	Meaning
0	A formula
1	A number
2	Text (a string)
4	A logical value (<code>True</code> or <code>False</code>)
8	A cell reference, as a <code>Range</code> object
16	An error value, such as <code>#N/A</code>
64	An array of values

The user can point to cells with the mouse or type in data. If the input is of the wrong type, the `InputBox` method displays an error message and prompts for the data again. If the user clicks the Cancel button, the `InputBox` method returns a value of `False`.

If you assign the return value to a `Variant`, you can check to see if the value is `False`, for most return types, to detect a Cancel. If you are prompting for a range, the situation is not so simple. You need to use code like the following:

```
Sub GetRange()  
    Dim rng As Range  
  
    On Error Resume Next  
    Set rng = Application.InputBox(prompt:="Enter range", Type:=8)  
    If rng Is Nothing Then  
        MsgBox "Operation Cancelled"  
    Else  
        rng.Select  
    End If  
End Sub
```

When you run this code, use the mouse to select the range. The output should look something like Figure 2-2.

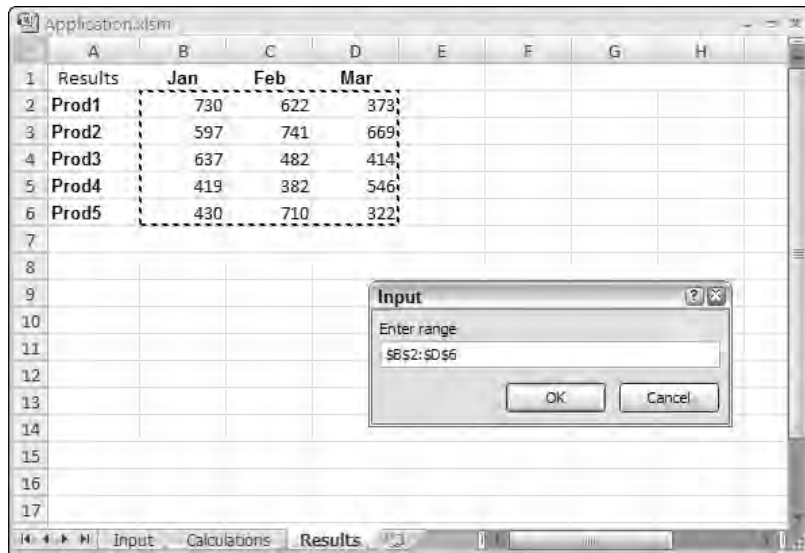


Figure 2-2

The problem is that you must use the `Set` statement to assign a range object to an object variable. If the user clicks Cancel and a `False` value is returned, the `Set` fails and you get a run-time error. Using the `On Error Resume Next` statement, you can avoid the run-time error and then check to see if a valid range was generated. You know that the built-in type checking of the `InputBox` method ensures a valid range will be returned if the user clicks OK, so an empty range indicates that Cancel was clicked.

StatusBar

The `StatusBar` property allows you to assign a text string to be displayed at the left-hand side of the Excel status bar at the bottom of the screen. This is an easy way to keep users informed of progress during a lengthy macro operation. It is a good idea to keep users informed, particularly if you have screen updating turned off and there is no sign of activity on the screen. Even though you have turned off screen updating, you can still display messages on the status bar.

The following code shows how you can use this technique in a looping procedure:

```
Sub ShowMessage()  
    Dim lCounter As Long  
    For lCounter = 0 To 100000000  
        If lCounter Mod 1000000 = 0 Then  
            Application.StatusBar = "Processing Record " & lCounter  
        End If  
    Next lCounter  
    Application.StatusBar = False  
End Sub
```

At the end of your processing, you must set the `StatusBar` property to `False` so that it returns to normal operation. Otherwise, your last message will stay on the screen.

SendKeys

`SendKeys` allows you to send keystrokes to the currently active window. It is used to control applications that do not support any other form of communication, such as DDE (Dynamic Data Exchange) or OLE. It is generally considered a last-resort technique.

The following example opens the Notepad application, which does not support DDE or OLE, and writes a line of data to the Notepad document:

```
Sub SKeys()  
    Dim dReturnValue As Double  
    dReturnValue = Shell("NOTEPAD.EXE", vbNormalFocus)  
    AppActivate dReturnValue  
    Application.SendKeys "Copy Data.xlsx c:\", True  
    Application.SendKeys "~", True  
    Application.SendKeys "%FABATCH%S", True  
End Sub
```

This example might not execute correctly from the VBE. Run it from the Excel window.

`SKeys` uses `Alt+F+A` to perform a `File` ⇄ `SaveAs` and enters the filename as `BATCH` and then enters `Alt+S` to save the text file. The percent symbol (%) is used to represent `Alt` and the tilde (~) represents `Enter`. The caret symbol (^) is used to represent `Ctrl`, and other special keys are specified by putting their names in curly braces. For example, the `Delete` key is represented by `{Del}`, as shown in the following example.

You can also send keystrokes directly to Excel. The following procedure clears the VBE's Immediate window. If you have been experimenting in the Immediate window or using `Debug.Print` to write to the Immediate window, it can get cluttered with old information. This procedure switches focus to the Immediate window and sends `Ctrl+a` to select all the text in the window. The text is then deleted by sending `Del`:

```
Sub ImmediateWindowClear()  
    Application.VBE.Windows.Item("Immediate").SetFocus  
    Application.SendKeys "^a"  
    Application.SendKeys "{Del}"  
End Sub
```

It is necessary for you to have programmatic access to your Visual Basic project for this macro to work. This can be set from the Excel Ribbon. Select the Developer tab, select Macro Security, and check the box against Trust access to the VBA project object model.

OnTime

You can use the `OnTime` method to schedule a macro to run sometime in the future. You need to specify the date and time for the macro to run, and the name of the macro. If you use the `Wait` method of the `Application` object to pause a macro, all Excel activity, including manual interaction, is suspended. The advantage of `OnTime` is that it allows you to return to normal Excel interaction, including running other macros, while you wait for the scheduled macro to run.

Say you have an open workbook with links to `Data.xls`, which exists on your network server but is not currently open. At 3 p.m. you want to update the links to `Data.xls`. The following example schedules the `RefreshData` macro to run at 3 p.m., which is 15:00 hours using a 24-hour clock, on the current day. `Date` returns the current date, and the `TimeSerial` function is used to add the necessary time:

```
Sub RunOnTime()  
    Application.OnTime Date + TimeSerial(15, 0, 0), "RefreshData"  
End Sub
```

It is worth noting that if you attempt to run this macro when it is currently after 3 p.m., you will receive an error message because you cannot schedule a task to run in the past. If necessary, change the time to one in the future.

The following `RefreshData` macro updates the links to `Data.xlsx` that exist in `ThisWorkbook` using the `UpdateLink` method. `ThisWorkbook` is a convenient way to refer to the workbook containing the macro:

```
Sub RefreshData()  
    ThisWorkbook.UpdateLink Name:="C:\Data.xlsx", Type:=xlExcelLinks  
End Sub
```

Chapter 2: The Application Object

If you want to keep refreshing the data on a regular basis, you can make the macro run itself as follows:

```
Dim mdteScheduledTime As Date

Sub RefreshData()
    ThisWorkbook.UpdateLink Name:="C:\Data.xlsx", Type:=xlExcelLinks
    mdteScheduledTime = Now + TimeSerial(0, 1, 0)
    Application.OnTime mdteScheduledTime, "RefreshData"
End Sub

Sub StopRefresh()
    Application.OnTime mdteScheduledTime, "RefreshData", , False
End Sub
```

Once you run `RefreshData`, it will keep scheduling itself to run every minute. In order to stop the macro, you need to know the scheduled time, so the module-level variable `mdteScheduledTime` is used to store the latest scheduled time. `StopRefresh` sets the fourth parameter of `OnTime` to `False` to cancel the scheduled run of `RefreshData`.

When you schedule a macro to run at a future time using the `OnTime` method, you must make sure that Excel keeps running in memory until the scheduled time occurs. It is not necessary to leave the workbook containing the `OnTime` macro open. Excel will open it, if it needs to.

The `OnTime` method is also useful when you want to introduce a delay in macro processing to allow an event to occur that is beyond your control. For example, you might want to send data to another application through a DDE link and wait for a response from that application before continuing with further processing. To do this, you would create two macros. The first macro sends the data and schedules the second macro (which processes the response) to run after sufficient time has passed. The second macro could keep running itself until it detected a change in the worksheet or the environment caused by the response from the external application.

OnKey

You can use the `OnKey` method to assign a macro procedure to a single keystroke or any combination of `Ctrl`, `Shift`, and `Alt` with another key. You can also use the method to disable key combinations.

The following example shows how to assign the `DownTen` macro to the down arrow key. Once `AssignDown` has been run, the down arrow key will run the `DownTen` macro and move the cell pointer down ten rows instead of one:

```
Sub AssignDown()
    Application.OnKey "{Down}", "DownTen"
End Sub

Sub DownTen()
    ActiveCell.Offset(10, 0).Select
```

```
End Sub

Sub ClearDown()
    Application.OnKey "{Down}"
End Sub
```

ClearDown returns the down arrow key to its normal function.

OnKey can be used to disable existing keyboard shortcuts. You can disable the Ctrl+c shortcut, normally used to copy, with the following code that assigns a null procedure to the key combination:

```
Sub StopCopyShortCut()
    Application.OnKey "^c", ""
End Sub
```

Note that a lowercase c is used. If you used an uppercase C, it would apply to Ctrl+Shift+c. Once again, you can restore the normal operation of Ctrl+c with the following code:

```
Sub ClearCopyShortCut()
    Application.OnKey "^c"
End Sub
```

The key assignments made with the OnKey method apply to all open workbooks and only persist during the current Excel session.

Worksheet Functions

You can use two sources of built-in functions directly in your Excel VBA code. One group of functions is part of the VBA language. The other group of functions is a subset of the Excel worksheet functions.

Excel and the Visual Basic language, in the form of VBA, were not merged until Excel 5. Each system independently developed its own functions, so there are inevitably some overlaps and conflicts between the two series of functions. For example, Excel has a DATE function and VBA also has a Date function. The Excel DATE function takes three input arguments (year, month, and day) to generate a specific date. The VBA Date function takes no input arguments and returns the current date from the system clock. In addition, VBA has a DateSerial function that takes the same input arguments as the Excel DATE function and returns the same result as the Excel DATE function. Finally, Excel's TODAY function takes no arguments and returns the same result as the VBA Date function.

As a general rule, if a VBA function serves the same purpose as an Excel function, the Excel function is not made directly available to VBA macros (although you can use the Evaluate method to access any Excel function, as pointed out previously in this chapter). There is also a special case regarding the Excel MOD function. MOD is not directly available in VBA, but VBA has a Mod operator that serves the same purpose. The following line of code uses the Evaluate method shortcut and displays the day of the week as a number, using the Excel MOD function and the Excel TODAY function:

```
MsgBox [MOD(TODAY(),7)]
```

Chapter 2: The Application Object

The same result can be achieved more simply with the VBA `Date` function and the `Mod` operator, as follows:

```
MsgBox Date Mod 7
```

The Excel `CONCATENATE` function is also not available in VBA. You can use the concatenation operator (`&`) as a substitute, just as you can in an Excel worksheet formula. If you insist on using the `CONCATENATE` function in VBA, you can write code like the following:

```
Sub ConcatenateExample1()  
    Dim s1 As String, s2 As String  
    s1 = "Jack "  
    s2 = "Smith"  
    MsgBox Evaluate("CONCATENATE("" & s1 & """, "" & s2 & """)")  
End Sub
```

On the other hand, you can avoid being absurd and get the same result with the following code:

```
Sub ConcatenateExample2()  
    Dim s1 As String, s2 As String  
    s1 = "Jack "  
    s2 = "Smith"  
    MsgBox s1 & s2  
End Sub
```

The VBA functions, such as `Date`, `DateSerial`, and `IsEmpty`, can be used without qualification, because they are members of `<globals>`. For example, you can use the following:

```
StartDate = DateSerial(1999, 6, 1)
```

The Excel functions, such as `VLOOKUP` and `SUM`, are methods of the `WorksheetFunction` object and are used with the following syntax:

```
Total = WorksheetFunction.Sum(Range("A1:A10"))
```

For compatibility with Excel 5 and Excel 95, you can use `Application` rather than `WorksheetFunction`:

```
Total = Application.Sum(Range("A1:A10"))
```

For a complete list of the worksheet functions directly available in VBA, see the `WorksheetFunction` object in Appendix A.

Caller

The `Caller` property of the `Application` object returns a reference to the object that called or executed a macro procedure. It had a wide range of uses in Excel 5 and Excel 95, where it was used with menus and controls on dialog sheets. From Excel 97 onward, command bars and ActiveX controls on user forms have replaced menus and controls on dialog sheets, and the Ribbon and Quick Access Menu have now replaced command bars. The `Caller` property does not apply to these new features.

Caller still applies to the Forms toolbar controls, drawing objects that have macros attached and user-defined functions. It is particularly useful in determining the cell that called a user-defined function. The worksheet in Figure 2-3 uses the WorksheetName function to display the name of the worksheet in B2.

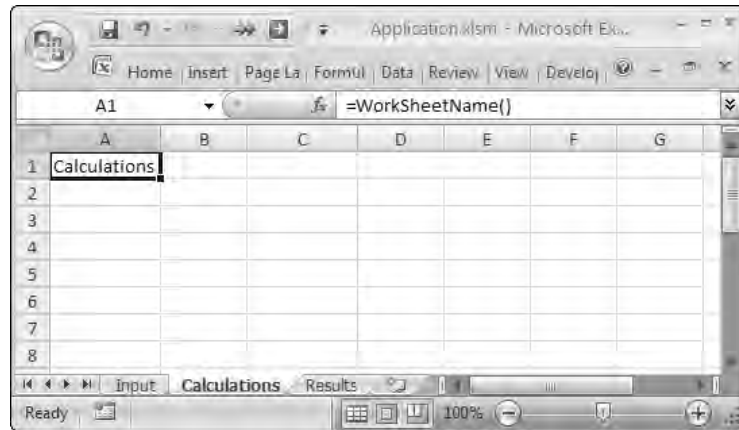


Figure 2-3

When used in a function, `Application.Caller` returns a reference to the cell that called the function, which is returned as a `Range` object. The following `WorksheetName` function uses the `Parent` property of the `Range` object to generate a reference to the `Worksheet` object containing the `Range` object. It assigns the `Name` property of the `Worksheet` object to the return value of the function. The `Volatile` method of the `Application` object forces Excel to recalculate the function every time the worksheet is recalculated, so that if you change the name of the sheet, the new name is displayed by the function:

```
Function WorksheetName()  
    Application.Volatile  
    WorksheetName = Application.Caller.Parent.Name  
End Function
```

It would be a mistake to use the following code in the `WorksheetName` function:

```
WorksheetName = ActiveSheet.Name
```

If a recalculation takes place while a worksheet is active that is different from the one containing the formula, the wrong name will be returned to the cell.

Summary

This chapter highlighted some of the more useful properties and methods of the `Application` object. Because `Application` is used to hold general-purpose functionality that does not fall clearly under other objects, it is easy to miss some of these very useful capabilities.

Chapter 2: The Application Object

The following properties and methods were covered:

- ❑ `ActiveCell`: Contains a reference to the active cell.
- ❑ `ActiveChart`: Contains a reference to the active chart.
- ❑ `ActivePrinter`: Contains a reference to the active printer.
- ❑ `ActiveSheet`: Contains a reference to the active worksheet.
- ❑ `ActiveWindow`: Contains a reference to the active window.
- ❑ `ActiveWorkbook`: Contains a reference to the active workbook.
- ❑ `Caller`: Contains reference to the object that called a macro.
- ❑ `DisplayAlerts`: Determines whether or not alert dialogs are displayed.
- ❑ `Evaluate`: Used to calculate Excel functions and generate Range objects.
- ❑ `InputBox`: Used to prompt a user for input.
- ❑ `OnKey`: Assigns a macro to a single keystroke, or a combination (with Ctrl, Alt, and so on).
- ❑ `OnTime`: Used to set the time for a macro to run.
- ❑ `ScreenUpdating`: Determines whether screen updating is turned on or off.
- ❑ `Selection`: Contains a reference to the selected range.
- ❑ `SendKeys`: Sends keystrokes to the active window.
- ❑ `StatusBar`: Allows messages to be displayed on the status bar.
- ❑ `WorksheetFunction`: Contains the Excel functions available to VBA.

This is but a small sample of the total number of properties and methods of the `Application` object — there are more than 200 of them in Excel 2007. A full list is given in Appendix A.

3

Workbooks and Worksheets

In this chapter, you learn how to create new `Workbook` objects and how to interact with the files that you use to store those workbooks. To do this, some basic utility functions are presented. You also see how to handle the `Sheet` objects within the workbook, and how some important features must be handled through the `Window` object. Finally, you learn how to synchronize your worksheets as you move from one worksheet to another.

The Workbooks Collection

The `Workbooks` collection consists of all the currently open `Workbook` objects in memory. Members can be added to the `Workbooks` collection in a number of ways. You can create a new empty workbook based on the default properties of the `Workbook` object, or you can create a new workbook based on a template file. Finally, you can open an existing workbook file.

To create a new empty workbook based on the default workbook, use the `Add` method of the `Workbooks` collection:

```
Workbooks.Add
```

The new workbook will be the active workbook, so you can refer to it in the following code as `ActiveWorkbook`. If you immediately save the workbook, using the `SaveAs` method, you can give it a filename that can be used to refer to the workbook in later code, even if it is no longer active. Before you try the following code, make sure you have a `C:\Data` directory or change the directory name used in the code:

```
Workbooks.Add  
ActiveWorkbook.SaveAs Filename:="C:\Data\SalesData1.xlsx"  
Workbooks.Add  
ActiveWorkbook.SaveAs Filename:="C:\Data\SalesData2.xlsx"  
Workbooks("SalesData1.xlsx").Activate
```

Chapter 3: Workbooks and Worksheets

However, a better technique is to use the return value of the `Add` method to create an object variable that refers to the new workbook. This provides a shortcut to refer to your workbook, and you can keep track of a temporary workbook without the need to save it:

```
Sub NewWorkbooks()  
    Dim wkb1 As Workbook  
    Dim wkb2 As Workbook  
  
    Set wkb1 = Workbooks.Add  
    Set wkb2 = Workbooks.Add  
    wkb1.Activate  
End Sub
```

The `Add` method allows you to specify a template for the new workbook. The template does not need to be a file saved as a template, with an `.xlt` extension — it can be a normal workbook file with an `.xlsx` extension. The following code creates a new, unsaved workbook called `SalesDataX`, where `X` is a sequence number that increments as you create more workbooks based on the same template, in the same way that Excel creates workbooks called `Book1`, `Book2`, and so forth when you create new workbooks through the user interface:

```
Set wkb1 = Workbooks.Add(Template:="C:\Data\SalesData.xlsx")
```

To add an existing workbook file to the `Workbooks` collection, you use the `Open` method. Once again, it is a good idea to use the return value of the `Open` method to create an object variable that you can use later in your code to refer to the workbook:

```
Set wkb1 = Workbooks.Open(Filename:="C:\Data\SalesData1.xlsx")
```

Many of the examples have data, such as filenames, hard coded. That is, data is placed inside the code instead of putting it into a variable and using the variable in the code. This is not good programming practice, in general, and isn't to be recommended. However, examples will continue in this format in order to simplify the code.

Getting a Filename from a Path

When you deal with workbooks in VBA, you often need to specify directory paths and filenames. Some tasks require that you know just the path — for example, if you set a default directory. Some tasks require you to know just the filename — for example, if you want to activate an open workbook. Other tasks require both path and filename — for example, if you want to open an existing workbook file that is not in the active directory.

Once a workbook is open, there is no problem getting its path, getting its full path and filename, or just getting the filename. For example, the following code displays `SalesData1.xlsx` in the message box:

```
Set wkb = Workbooks.Open(Filename:="C:\Data\SalesData1.xlsx")  
MsgBox wkb.Name
```

`wkb.Path` returns `"C:\Data"` and `wkb.FullName` returns `"C:\Data\SalesData1.xlsx"`.

However, if you are trying to discover whether a certain workbook is already open, and you have the full path information, you need to extract the filename from the full path to get the value of the `Name` property of the `Workbook` object. The following `GetFileName` function returns the name "SalesData1.xlsx" from the full path "C:\Data\SalesData1.xlsx":

```
Function sGetFileName(sFullName As String) As String
    'sGetFileName returns the file name, such as Cash.xlsx from
    'the end of a full path such as C:\Data\Project1\Cash.xlsx
    'sFullName is returned if no path separator is found
    Dim sPathSeparator As String      'Path Separator Character
    Dim iFNLength As Integer 'Length of FullName
    Dim i As Integer

    sPathSeparator = Application.PathSeparator
    iFNLength = Len(sFullName)
    'Find last path separator character, if any
    For i = iFNLength To 1 Step -1
        If Mid(sFullName, i, 1) = sPathSeparator Then Exit For
    Next i
    sGetFileName = Right(sFullName, iFNLength - i)
End Function
```

So that `sGetFileName` works on the Macintosh as well as under Windows, the path separator character is obtained using the `PathSeparator` property of the `Application` object. This returns `:` on the Macintosh and `\` under Windows. The `Len` function returns the number of characters in `sFullName`, and the `For...Next` loop searches backwards from the last character in `sFullName`, looking for the path separator. If it finds one, it exits the `For...Next` loop, and the index `i` is equal to the character position of the separator. If it does not find a separator, `i` will have a value of 0 when the `For...Next` loop is completed.

When a `For...Next` loop is permitted to complete normally, the index variable will not be equal to the `Stop` value. It will have been incremented past the end value.

`sGetFileName` uses the `Right` function to extract the characters to the right of the separator in `sFullName`. If there is no separator, all the characters from `sFullName` are returned. Once you have the filename of a workbook, you can use the following `bIsWorkbookOpen` function to see if the workbook is already a member of the `Workbooks` collection:

```
Function bIsWorkbookOpen(wkbName As String) As Boolean
    'bIsWorkbookOpen returns True if wkbName is a member
    'of the Workbooks collection. Otherwise, it returns False
    'wkbName must be provided as a file name without path
    Dim wkb As Workbook

    On Error Resume Next
    Set wkb = Workbooks(wkbName)
    If Not wkb Is Nothing Then
        bIsWorkbookOpen = True
    End If
End Function
```

Chapter 3: Workbooks and Worksheets

In this code, `bIsWorkbookOpen` tries to assign a reference to the workbook to an object variable, and then sees whether or not that attempt was successful. An alternative way to achieve the same result would be to search through the `WorkBooks` collection to see if any `Workbook` object had the name required.

In the preceding code, the `On Error Resume Next` ensures that no run-time error occurs when the workbook is not open. If the named document is found, `bIsWorkbookOpen` returns a value of `True`. If you do not define the return value of a Boolean function, it will return `False`. In other words, if no open workbook of the given name is found, `False` is returned.

You might prefer to use the following more lengthy but more explicit code to define the return value of `bIsWorkbookOpen`. It is also easier to understand because it avoids the double negative. Despite this, my own preference is for the shorter code as just presented, because it is shorter.

```
If wkb Is Nothing Then
    bIsWorkbookOpen = False
Else
    bIsWorkbookOpen = True
End If
```

The following code uses the user-defined `sGetFileName` and `bIsWorkbookOpen` functions described earlier. `ActivateWorkbook1` is designed to activate the workbook file in the path assigned to the variable `sFullName`:

```
Sub ActivateWorkbook1()
    Dim sFullName As String
    Dim sFileName As String
    Dim wkb As Workbook

    sFullName = "C:\Data\SalesData1.xlsx"
    sFileName = sGetFileName(sFullName)
    If bIsWorkbookOpen(sFileName) Then
        Set wkb = Workbooks(sFileName)
        wkb.Activate
    Else
        Set wkb = Workbooks.Open(FileName:=sFullName)
    End If
End Sub
```

`ActivateWorkbook1` first uses `sGetFileName` to extract the workbook filename, `SalesData1.xlsx`, from `sFullName` and assigns it to `sFileName`. Then it uses `bIsWorkbookOpen` to determine whether `SalesData1.xlsx` is currently open. If the file is open, it assigns a reference to the `Workbook` object to the `wkb` object variable and activates the workbook. If the file is not open, it opens the file and assigns the return value of the `Open` method to `wkb`. When the workbook is opened, it will automatically become the active workbook.

Note that the preceding code assumes that the workbook file exists at the specified location. It will fail if this is not the case. You will find a function, called `bFileExists`, in the “Overwriting an Existing Workbook” section later in the chapter that you can use to test for the file’s existence.

Files in the Same Directory

It is common practice to break up an application into a number of workbooks and keep the related workbook files in the same directory, including the workbook containing the code that controls the application. In this case, you could use the common directory name in your code when opening the related workbooks. However, if you “hard wire” the directory name into your code, you will have problems if the directory name changes, or if you copy the files to another directory on the same PC or another PC. You will have to edit the directory path in your macros.

To avoid maintenance problems in this situation, you can make use of `ThisWorkbook.Path`. `ThisWorkbook` is a reference to the workbook that contains the code. No matter where the workbook is located, the `Path` property of `ThisWorkbook` gives you the required path to locate the related files, as demonstrated in the following code:

```
Sub ActivateWorkbook2()
    Dim sPath As String
    Dim sFileName As String
    Dim sFullName As String
    Dim wkb As Workbook

    sFileName = "SalesData1.xlsx"
    If bIsWorkbookOpen(sFileName) Then
        Set wkb = Workbooks(sFileName)
        wkb.Activate
    Else
        sPath = ThisWorkbook.Path
        sFullName = sPath & "\" & sFileName
        Set wkb = Workbooks.Open(FileName:=sFullName)
    End If
End Sub
```

Overwriting an Existing Workbook

When you want to save a workbook using the `SaveAs` method and using a specific filename, there is the possibility that a file with that name will already exist on disk. If the file does already exist, the user receives an alert message and has to make a decision about overwriting the existing file. If you want, you can avoid the alert and take control programmatically.

If you want to overwrite the existing file every time, you can just suppress the alert with the following code:

```
Set wkb1 = Workbooks.Add
Application.DisplayAlerts = False
wkb1.SaveAs Filename:="C:\Data\SalesData1.xlsx"
Application.DisplayAlerts = True
```

If you want to check for the existing file and take alternative courses of action, you can use the `Dir` function. If this is a test that you need to perform often, you can create the following `bFileExists` function:

```
Function bFileExists(sFile As String) As Boolean
    If Dir(sFile) <> "" Then bFileExists = True
End Function
```

Chapter 3: Workbooks and Worksheets

The `Dir` function attempts to match its input argument against existing files. `Dir` can be used with wildcards under Windows for matches such as `*.xlsx`. If it finds a match, it returns the first match found and can be called again without an input argument to get subsequent matches. Here, you are trying for an exact match that will either return the same value as the input argument or a zero-length string if there is no match. The `bFileExists` function has been declared to return a `Boolean` type value and, as explained earlier, is set to the default value of `False` if no return value is defined. The `If` test assigns a value of `True` to the return value if `Dir` does not return a zero-length string.

The following code shows how you can use the `bFileExists` function to test for a specific filename and take alternative courses of action:

```
Sub TestForFile()  
    Dim sFileName As String  
  
    sFileName = "C:\Data\SalesData1.xlsx"  
    If bFileExists(sFileName) Then  
        MsgBox sFileName & " exists"  
    Else  
        MsgBox sFileName & " does not exist"  
    End If  
End Sub
```

What you actually do in each alternative depends very much on the situation you are dealing with. One alternative could be to prompt the user for a new filename if the name already exists. Another approach could be to compute a new filename by finding a new sequence number to be appended to the end of the text part of the filename, as shown here:

```
Sub CreateNextFileName()  
    Dim wkb1 As Workbook  
    Dim i As Integer  
    Dim sFName As String  
  
    Set wkb1 = Workbooks.Add(Template:="C:\Data\SalesData.xlsx")  
    i = 0  
    Do  
        i = i + 1  
        sFName = "C:\Data\SalesData" & i & ".xlsx"  
    Loop While bFileExists(sFName)  
  
    wkb1.SaveAs FileName:=sFName  
End Sub
```

Here, the code in the `Do . . . Loop` is repeated, increasing the value of `i` by one for each loop, as long as the filename generated exists. When `i` reaches a value for which there is no matching filename, the loop ends and the file is saved using the new name.

Saving Changes

You can close a workbook using the `Close` method of the `Workbook` object, as shown here:

```
ActiveWorkbook.Close
```

If changes have been made to the workbook, the user will be prompted to save the changes when an attempt is made to close the workbook. If you want to avoid this prompt, you can use several techniques, depending on whether or not you want to save the changes.

If you want to save changes automatically, you can specify this as a parameter of the `Close` method:

```
Sub CloseWorkbook()  
    Dim wkb1 As Workbook  
  
    Set wkb1 = Workbooks.Open(FileName:="C:\Data\SalesData1.xlsx")  
    Range("A1").Value = Format(Date, "ddd mmm dd, yyyy")  
    Range("A1").EntireColumn.AutoFit  
    wkb1.Close SaveChanges:=True  
End Sub
```

If you don't want to save changes, you can set the `SaveChanges` parameter of the `Close` method to `False`.

Another situation that could arise is where you want to leave a changed workbook open to view, but you don't want to save those changes or be prompted to save the changes when you close the workbook or Excel. In this situation, you can set the `Saved` property of the workbook to `True` and Excel will think that there are no changes to be saved. You should make doubly sure you would want to do this before you add this line of code:

```
ActiveWorkbook.Saved = True
```

The Sheets Collection

Within a `Workbook` object, there is a `Sheets` collection whose members can be either `Worksheet` objects or `Chart` objects. For compatibility with older versions of Excel, they can also be `DialogSheets`, `Excel4MacroSheets`, and `Excel4InternationalMacroSheets`. Excel 5 and Excel 95 included modules as part of the `Sheets` collection, but since Excel 97, modules have moved to the VBE.

Modules in workbooks created under Excel 5 or Excel 95 are considered by later versions of Excel to belong to a hidden Modules collection and can still be manipulated by the code originally set up in the older versions.

`Worksheet` and `Chart` objects also belong to their own collections — the `Worksheets` collection and the `Charts` collection, respectively. The `Charts` collection only includes chart sheets — that is, charts that are embedded in a worksheet are *not* members of the `Charts` collection. Charts embedded in worksheets are contained in `ChartObject` objects, which are members of the `ChartObjects` collection of the worksheet. See Chapter 9 for more details.

Worksheets

You can refer to a worksheet by its name or index number in the `Sheets` collection and the `Worksheets` collection. If you know the name of the worksheet you want to work on, it is appropriate, and usually

Chapter 3: Workbooks and Worksheets

safer, to use that name to specify the required member of the `Worksheets` collection. If you want to process all the members of the `Worksheets` collection — in a `For . . . Next` loop, for example — you would usually reference each worksheet by its index number.

The index number of a worksheet in the `Worksheets` collection can be different from the index number of the worksheet in the `Sheets` collection. In the workbook shown in Figure 3-1, `Sheet1` can be referenced by any of the following:

```
ActiveWorkbook.Sheets("Sheet1")
ActiveWorkbook.Worksheets("Sheet1")
ActiveWorkbook.Sheets(2)
ActiveWorkbook.Worksheets(1)
```

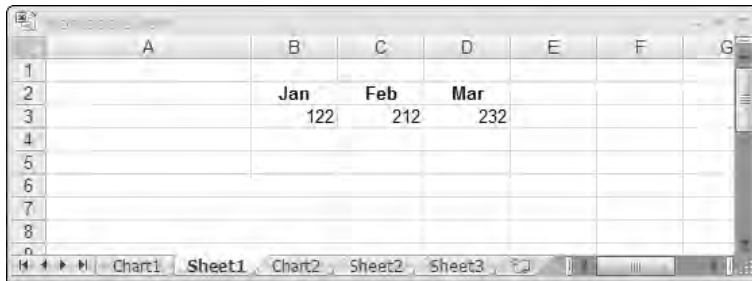


Figure 3-1

There is a trap, however, concerning the `Index` property of the `Worksheet` object in that it returns the value of the index in the `Sheets` collection, not the `Worksheets` collection. The following code tells you that `Worksheets(1)` is `Sheet1` with index 2, `Worksheets(2)` is `Sheet2` with index 4, and `Worksheets(3)` is `Sheet3` with index 5. You can see the message for `Sheet2` in Figure 3-2.

```
Sub WorksheetIndex()
    Dim i As Integer

    For i = 1 To ThisWorkbook.Worksheets.Count
        MsgBox ThisWorkbook.Worksheets(i).Name & _
            " has Index = " & _
            ThisWorkbook.Worksheets(i).Index
    Next i
End Sub
```

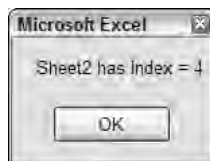


Figure 3-2

You should avoid using the `Index` property of the worksheet, if possible, because it leads to confusing code. The following example shows how you must use the worksheet `Index` as an index in the `Sheets` collection, not the `Worksheets` collection. The macro adds a new empty chart sheet to the left of every worksheet in the active workbook:

```
Sub InsertChartsBeforeWorksheets()  
    Dim wks As Worksheet  
  
    For Each wks In Worksheets  
        Charts.Add Before:=Sheets(wks.Index)  
    Next wks  
End Sub
```

In most cases you can avoid using the worksheet `Index` property. The preceding code should have been written as follows:

```
Sub InsertChartsBeforeWorksheets2()  
    Dim wks As Worksheet  
  
    For Each wks In Worksheets  
        Charts.Add Before:=wks  
    Next wks  
End Sub
```

Strangely enough, Excel will not allow you to add a new chart after the last worksheet, although it will let you move a chart after the last worksheet. If you want to insert chart sheets after each worksheet, you can use code like the following:

```
Sub InsertChartsAfterWorksheets()  
    Dim wks As Worksheet  
    Dim cht As Chart  
  
    For Each wks In Worksheets  
        Set cht = Charts.Add  
        cht.Move After:=wks  
    Next wks  
End Sub
```

Chart sheets are covered in more detail in Chapter 8.

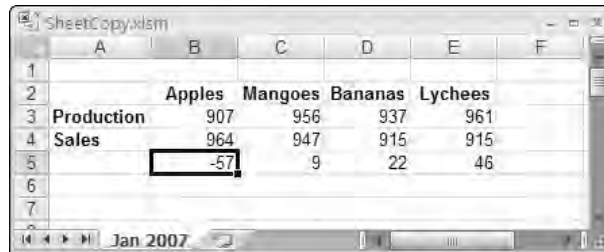
Copy and Move

The `Copy` and `Move` methods of the `Worksheet` object allow you to copy or move one or more worksheets in a single operation. They both have two optional parameters that allow you to specify the destination of the operation. The destination can be either before or after a specified sheet. If you do not use one of these parameters, the worksheet will be copied or moved to a new workbook.

`Copy` and `Move` do not return any value or reference, so you have to rely on other techniques if you want to create an object variable referring to the copied or moved worksheets. This is not generally a problem, because the first sheet created by a `Copy` operation, or the first sheet resulting from moving a group of sheets, will be active immediately after the operation.

Chapter 3: Workbooks and Worksheets

Say you have a workbook like that shown in Figure 3-3 and want to add another worksheet for February — and then more worksheets for the following months. The numbers on rows 3 and 4 are the input data, but row 5 contains calculations to give the difference between rows 3 and 4. When you copy the worksheet, you will want to clear the input data from the copies but retain the headings and formulas.



	A	B	C	D	E	F
1						
2		Apples	Mangoes	Bananas	Lychees	
3	Production	907	956	937	961	
4	Sales	964	947	915	915	
5		-57	9	22	46	
6						
7						

Figure 3-3

The following code creates a new monthly worksheet that is inserted into the workbook after the latest month. It copies the first worksheet, removes any numeric data from it but leaves any headings or formulas in place, and then renames the worksheet to the new month and year:

```
Sub NewMonth()  
    'Copy the first worksheet in the active workbook  
    'to create a new monthly sheet with name of format "mmm yyyy".  
    'The first worksheet must have a name that is in a recognizable  
    'date format.  
    Dim wks As Worksheet  
    Dim dteFirstDate As Date  
    Dim iFirstMonth As Integer  
    Dim iFirstYear As Integer  
    Dim iCount As Integer  
  
    'Initialize counter to number of worksheets  
    iCount = Worksheets.Count  
  
    'Copy first worksheet after last worksheet and increase counter  
    Worksheets(1).Copy After:=Worksheets(iCount)  
    iCount = iCount + 1  
  
    'Assign last worksheet to wks  
    Set wks = Worksheets(iCount)  
  
    'Calculate date from first worksheet name  
    dteFirstDate = DateValue(Worksheets(1).Name)  
  
    'Extract month and year components  
    iFirstMonth = Month(dteFirstDate)  
    iFirstYear = Year(dteFirstDate)  
  
    'Compute and assign new worksheet name
```



```

wks.Name = _
    Format(DateSerial(iFirstYear, iFirstMonth + iCount - 1, 1), "mmm yyyy")

'Clear data cells in wks, avoiding error if there is no data
wks.Cells.SpecialCells(xlCellTypeConstants, 1).ClearContents
End Sub

```

The result of the copy is shown in Figure 3-4.

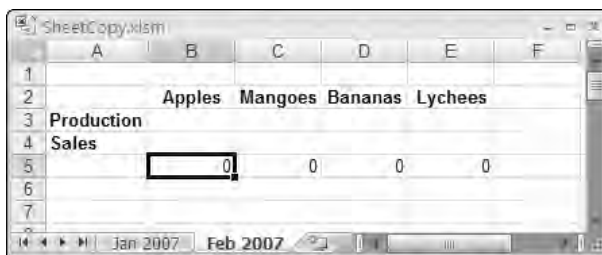


Figure 3-4

`NewMonth` first determines how many worksheets are in the workbook and then copies the current worksheet, appending it to the workbook. It updates the number of worksheets in `iCount` and creates an object variable `wks` that refers to the copied sheet. It then uses the `DateValue` function to convert the name of the January worksheet to a date.

`NewMonth` extracts the month and year of the date into the two integer variables `iFirstMonth` and `iFirstYear` using the `Month` and `Year` functions. It then uses the `DateSerial` function to calculate a new date that follows on from the last one. This calculation is valid even when new years are created, because `DateSerial`, like the worksheet `DATE` function, treats month numbers greater than 12 as the appropriate months in the following year.

`NewMonth` uses the VBA `Format` function to convert the new date into "mmm yyyy" format as a string. It assigns the text to the `Name` property of the new worksheet. Finally, `NewMonth` clears the contents of any cells containing numbers, using the `SpecialCells` method to find the numbers. `SpecialCells` is discussed in more detail in the following chapter on the `Range` object. The `On Error Resume Next` statement suppresses a run-time error when there is no numeric data to be cleared.

Grouping Worksheets

You can manually group the sheets in a workbook by clicking a sheet tab, then holding down `Shift` or `Ctrl` and clicking on another sheet tab. `Shift` groups all the sheets between the two tabs. `Ctrl` adds just the new sheet to the group. You can also group sheets in VBA by using the `Select` method of the `Worksheets` collection in conjunction with the `Array` function. The following code groups the first, third, and fifth worksheets and makes the third worksheet active:

```

Worksheets(Array(1, 3, 5)).Select
Worksheets(3).Activate

```

Chapter 3: Workbooks and Worksheets

In addition to this, you can also create a group using the `Select` method of the `Worksheet` object. The first sheet is selected in the normal way. Other worksheets are added to the group by using the `Select` method while setting its `Replace` parameter to `False`:

```
Sub GroupSheets()  
    Dim asNames(1 To 3) As String  
    Dim i As Integer  
  
    asNames(1) = "Jan 2007"  
    asNames(2) = "Mar 2007"  
    asNames(3) = "May 2007"  
  
    Worksheets(asNames(1)).Select  
  
    For i = 2 To 3  
        Worksheets(asNames(i)).Select Replace:=False  
    Next i  
  
End Sub
```

This technique is particularly useful when the names have been specified by user input, via a multi-select list box, for example.

One benefit of grouping sheets manually is that any data inserted into the active sheet and any formatting applied to the active sheet is automatically copied to the other sheets in the group. However, only the active sheet is affected when you apply changes to a grouped sheet using VBA code. If you want to change the other members of the group, you need to set up a `For Each...Next` loop and carry out the changes on each member.

The following code places the value 100 into the A1 cell of worksheets with index numbers 1, 3, and 5 and bolds the numbers:

```
Sub FormatGroup()  
    Dim shts As Sheets  
    Dim wks As Worksheet  
  
    Set shts = Worksheets(Array(1, 3, 5))  
  
    For Each wks In shts  
        wks.Range("A1").Value = 100  
        wks.Range("A1").Font.Bold = True  
    Next wks  
  
End Sub
```

The Window Object

In VBA, if you want to detect what sheets are currently grouped, you use the `SelectedSheets` property of the `Window` object. You might think that `SelectedSheets` should be a property of the `Workbook` object, but that is not the case. `SelectedSheets` is a property of the `Window` object, because you can open many windows on the same workbook and each window can have different groups, as Figure 3-5 shows.

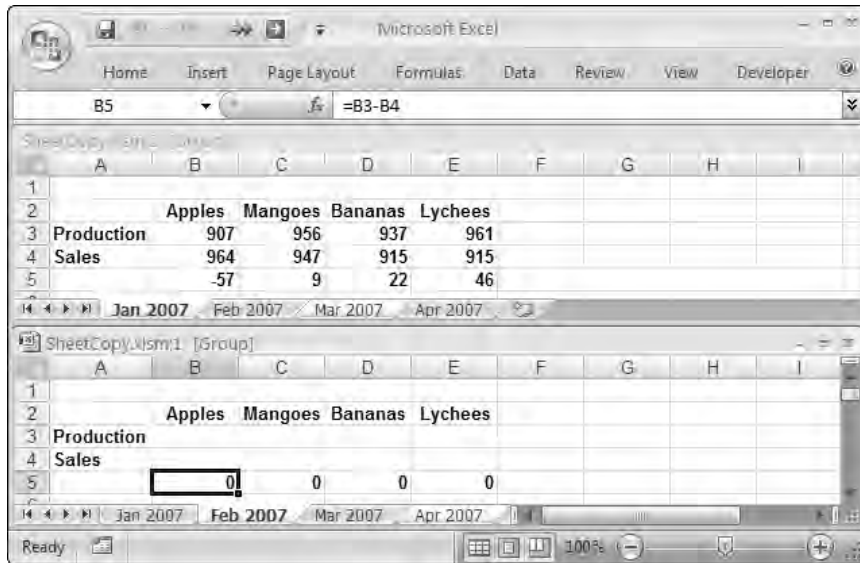


Figure 3-5

There are many other common workbook and worksheet properties that you might presume to be properties of the `Workbook` object or the `Worksheet` object, but which are actually `Window` object properties. Some examples of these are `ActiveCell`, `DisplayFormulas`, `DisplayGridlines`, `DisplayHeadings`, and `Selection`. See the `Window` object in Appendix A for a full list.

The following code determines which cells are selected on the active sheet, makes them bold, and then goes on to apply bold format to the corresponding ranges on the other sheets in the group:

```
Sub FormatSelectedGroup()
    Dim sht As Object
    Dim sRangeAddress As String

    sRangeAddress = Selection.Address

    For Each sht In ActiveWindow.SelectedSheets
        If TypeName(sht) = "Worksheet" Then
            sht.Range(sRangeAddress).Font.Bold = True
        End If
    Next sht

End Sub
```

Chapter 3: Workbooks and Worksheets

The address of the selected range on the active sheet is captured in `sRangeAddress` as a string. It is possible to activate only the selected sheets and apply bold format to the selected cells. Group mode ensures that the selections are the same on each worksheet. However, activating sheets is a slow process. By capturing the selection address as a string, you can generate references to the same range on other sheets using the `Range` property of the other sheets. The address is stored as a string in the form “\$B\$2:\$E\$2,\$A\$3:\$A\$4”, for example, and need not be a single contiguous block.

`FormatSelectedGroup` allows for the possibility that the user can include a chart sheet or another type of sheet in the group of sheets. It checks that the `TypeName` of the sheet is indeed “Worksheet” before applying the new format.

It is necessary to declare `sht` as the generic `Object` type if you want to allow it to refer to different sheet types. There is a `Sheets` collection in the Excel object model, but there is no `Sheet` object.

Synchronizing Worksheets

When you move from one worksheet in a workbook to another, the sheet you activate will be configured as it was when it was last active. The top-left corner cell, the selected range of cells, and the active cell will be in exactly the same positions as they were the last time the sheet was active, unless you are in Group mode. In Group mode, the selection and active cell are synchronized across the group. However, the top-left corner cell is not synchronized in Group mode, and it is possible that you will not be able to see the selected cells and the active cell when you activate a worksheet.

If you want to synchronize your worksheets completely, even out of Group mode, you can add the following code to the `ThisWorkbook` module of your workbook:

```
Dim mshtOldSheet As Object

Private Sub Workbook_SheetDeactivate(ByVal Sht As Object)
    'If the deactivated sheet is a worksheet,
    'store a reference to it in mshtOldSheet
    If TypeName(Sht) = "Worksheet" Then Set mshtOldSheet = Sht
End Sub

Private Sub Workbook_SheetActivate(ByVal NewSheet As Object)
    Dim lCurrentCol As Long
    Dim lCurrentRow As Long
    Dim sCurrentCell As String
    Dim sCurrentSelection As String

    On Error GoTo Fin
    If mshtOldSheet Is Nothing Then Exit Sub
    If TypeName(NewSheet) <> "Worksheet" Then Exit Sub
    Application.ScreenUpdating = False
    Application.EnableEvents = False

    mshtOldSheet.Activate    'Get the old worksheet configuration
    lCurrentCol = ActiveWindow.ScrollColumn
    lCurrentRow = ActiveWindow.ScrollRow
```

```
sCurrentSelection = Selection.Address
sCurrentCell = ActiveCell.Address

NewSheet.Activate 'Set the new worksheet configuration
ActiveWindow.ScrollColumn = lCurrentCol
ActiveWindow.ScrollRow = lCurrentRow
Range(sCurrentSelection).Select
Range(sCurrentCell).Activate
Fin:
Application.EnableEvents = True
End Sub
```

The `Dim mshtOldSheet as Object` statement must be at the top of the module in the declarations area, so that `mshtOldSheet` is a module-level variable that will retain its value while the workbook is open and can be accessed by the two event procedures. The `Workbook_SheetDeactivate` event procedure is used to store a reference to any worksheet that is deactivated. The `Deactivate` event occurs after another sheet is activated, so it is too late to store the active window properties. The procedure's `Sht` parameter refers to the deactivated sheet and its value is assigned to `mshtOldSheet`.

The `Workbook_SheetActivate` event procedure executes after the `Deactivate` procedure. The `On Error GoTo Fin` statement ensures that, if an error occurs, there are no error messages displayed and that control jumps to the `Fin:` label where event processing is enabled, just in case event processing has been switched off.

The first `If` tests check that `mshtOldSheet` has been defined, indicating that a worksheet has been deactivated during the current session. The second `If` test checks that the active sheet is a worksheet. If either `If` test fails, the procedure exits. These tests allow for other types of sheets, such as charts, being deactivated or activated.

Next, screen updating is turned off to minimize screen flicker. It is not possible to eliminate all flicker, because the new worksheet has already been activated and the user will get a brief glimpse of its old configuration before it is changed. Then, event processing is switched off so that no chain reactions occur. To get the data it needs, the procedure has to reactivate the deactivated worksheet, which would trigger the two event procedures again.

After reactivating the old worksheet, the `ScrollRow` (the row at the top of the screen), the `ScrollColumn` (the column at the left of the screen), the addresses of the current selection, and the active cell are stored. The new worksheet is then reactivated and its screen configuration is set to match the old worksheet. Because there is no `Exit Sub` statement before the `Fin:` label, the final statement is executed to make sure event processing is enabled again.

Summary

In this chapter you saw many techniques for handling workbooks and worksheets in VBA code. You have seen how to:

- ❑ Create new workbooks and open existing workbooks.
- ❑ Handle saving workbook files and overwriting existing files.
- ❑ Move and copy worksheets and interact with Group mode.

Chapter 3: Workbooks and Worksheets

You have also seen that you access some workbook and worksheet features through the `Window` object, and have been shown that you can synchronize your worksheets using workbook events procedures. See Chapter 9 for more discussion on this topic.

In addition, a number of utility macros have been presented, including routines to check that a workbook is open and to extract a filename from the full file path, and a simple macro that confirms that a file does indeed exist.

4

Using Ranges

The `Range` object is probably the object you will utilize the most in your VBA code. A `Range` object can be a single cell, a rectangular block of cells, or the union of many rectangular blocks (a non-contiguous range). A `Range` object is contained within a `Worksheet` object.

The Excel object model does not support three-dimensional `Range` objects that span multiple worksheets—every cell in a single `Range` object must be on the same worksheet. If you want to process 3D ranges, you must process a `Range` object in each worksheet separately.

This chapter examines the most useful properties and methods of the `Range` object.

Activate and Select

The `Activate` and `Select` methods cause some confusion, and it is sometimes claimed that there is no difference between them. To understand the difference between them, you first need to understand the difference between the `ActiveCell` and `Selection` properties of the `Application` object. The screen in Figure 4-1 illustrates this.

`Selection` refers to B3:E10. `ActiveCell` refers to C5, the cell where data will be inserted if the user types something. `ActiveCell` only ever refers to a single cell, whereas `Selection` can refer to a single cell or a range of cells. The active cell is usually the top left-hand cell in the selection, but can be any cell in the selection, as shown in Figure 4-1. You can manually change the position of the active cell in a selection by pressing `Tab`, `Enter`, `Shift+Tab`, or `Shift+Enter`.

You can achieve the combination of selection and active cell shown in Figure 4-1 by using the following code:

```
Range("B3:E10").Select  
Range("C5").Activate
```

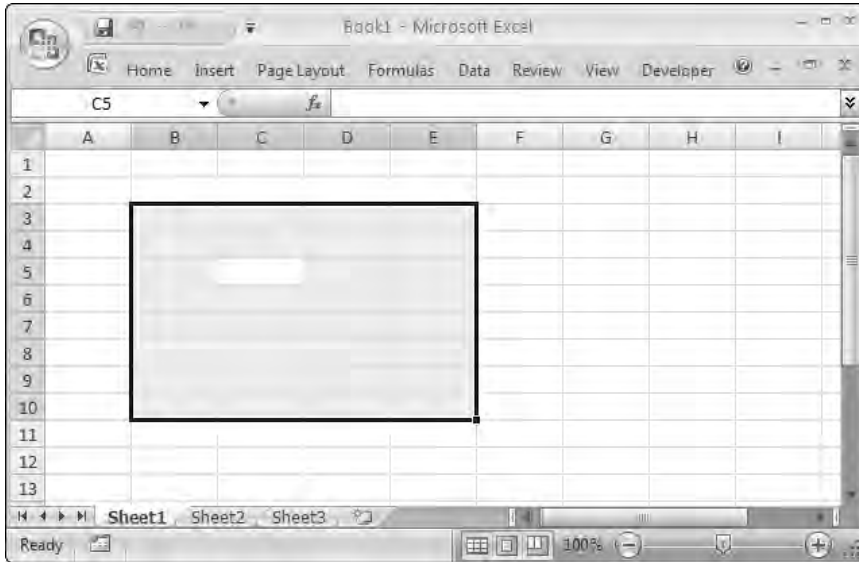


Figure 4-1

If you try to activate a cell that is outside the selection, you will change the selection, and the selection will become the activated cell.

Confusion also arises because you are permitted to specify more than one cell when you use the `Activate` method. Excel's behavior is determined by the location of the top-left cell in the range you activate. If the top-left cell is within the current selection, the selection does not change and the top-left cell becomes active. The following example creates the screen in Figure 4-1:

```
Range("B3:E10").Select  
Range("C5:Z100").Activate
```

If the top-left cell of the range you activate is not in the current selection, the range that you activate replaces the current selection, as shown by the following:

```
Range("B3:E10").Select  
Range("A2:C5").Activate
```

In this case, the `Select` is overruled by the `Activate` and A2:C5 becomes the selection.

To avoid errors, it is recommended that you don't use the `Activate` method to select a range of cells. If you get into the habit of using `Activate` instead of `Select`, you will get unexpected results when the top-left cell you activate is within the current selection.

Range Property

You can use the `Range` property of the `Application` object to refer to a `Range` object on the active worksheet. The following example refers to a `Range` object that is the B2 cell on the currently active worksheet:

```
Application.Range("B2")
```

Note that you can't test code examples like this one as they are presented. However, as long as you are referring to a range on the active worksheet, these examples can be tested by the Immediate window of the VBE, as follows:

```
Application.Range("B2").Select
```

It is important to note that the preceding reference to a `Range` object will cause an error if there is no worksheet currently active. For example, it will cause an error if you have a chart sheet active.

Because the `Range` property of the `Application` object is a member of `<globals>`, you can omit the reference to the `Application` object, as follows:

```
Range("B2")
```

You can refer to more complex `Range` objects than a single cell. The following example refers to a single block of cells on the active worksheet:

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

And this code refers to a non-contiguous range of cells:

```
Range("A1:A10,C1:C10,E1:E10")
```

The `Range` property also accepts two arguments that refer to diagonally opposite corners of a range. This gives you an alternative way to refer to the A1:D10 range:

```
Range("A1", "D10")
```

`Range` also accepts names that have been applied to ranges. If you have defined a range of cells with the name `SalesData`, you can use the name as an argument:

```
Range("SalesData")
```

The arguments can be objects as well as strings, which provides much more flexibility. For example, you might want to refer to every cell in column A, from cell A1 down to a cell that has been assigned the name `LastCell`:

```
Range("A1", Range("LastCell"))
```

Shortcut Range References

You can also refer to a range by enclosing an A1 style range reference or a name in square brackets, which is a shortcut form of the `Evaluate` method of the `Application` object. It is equivalent to using a single string argument with the `Range` property, but is shorter:

```
[B2]
[A1:D10]
[A1:A10, C1:C10, E1:E10]
[SalesData]
```

This shortcut is convenient when you want to refer to an absolute range. However, it is not as flexible as the `Range` property, because it cannot handle variable input as strings or object references.

Ranges on Inactive Worksheets

If you want to work efficiently with more than one worksheet at the same time, it is important to be able to refer to ranges on worksheets without having to activate those worksheets. Switching between worksheets is slow, and code that does this is more complex than it needs to be. This also leads to code that is harder to read and debug.

All the examples so far apply to the active worksheet, because they have not been qualified by any specific worksheet reference. If you want to refer to a range on a worksheet that is not active, simply use the `Range` property of the required `Worksheet` object:

```
Worksheets("Sheet1").Range("C10")
```

If the workbook containing the worksheet and range is not active, you need to further qualify the reference to the `Range` object as follows:

```
Workbooks("Sales.xls").Worksheets("Sheet1").Range("C10")
```

However, you need to be careful if you want to use the `Range` property as an argument to another `Range` property. Say you want to sum A1:A10 on Sheet1 while Sheet2 is the active sheet. You might be tempted to use the following code, which results in a run-time error:

```
MsgBox WorksheetFunction.Sum(Sheets("Sheet1").Range(Range("A1"), _
                                                                    Range("A10")))
```

The problem is that `Range("A1")` and `Range("A10")` refer to the active sheet, Sheet2. You need to use fully qualified properties:

```
MsgBox WorksheetFunction.Sum(Sheets("Sheet1").Range(_
                                                                    Sheets("Sheet1").Range("A1"), _
                                                                    Sheets("Sheet1").Range("A10")))
```

In this situation it is more elegant, and more efficient, to use a `With...End With` construct:

```
With Sheets("Sheet1")
    MsgBox WorksheetFunction.Sum(.Range(.Range("A1"), .Range("A10")))
End With
```

Range Property of a Range Object

The `Range` property is normally used as a property of the `Worksheet` object. You can also use the `Range` property of the `Range` object. In this case, it acts as a reference relative to the `Range` object itself. The following is a reference to the D4 cell:

```
Range("C3").Range("B2")
```

If you consider a virtual worksheet that has C3 as the top left-hand cell, and B2 is one column across and one row down on the virtual worksheet, you arrive at D4 on the real worksheet.

You will see this “Range in a Range” technique used in code generated by the macro recorder when relative recording is used (discussed in Chapter 2). For example, the following code was recorded when the active cell and the four cells to its right were selected while recording relatively:

```
ActiveCell.Range("A1:E1").Select
```

Because the preceding code is obviously very confusing, it is best to avoid this type of referencing. The `Cells` property is a much better way to reference relatively.

Cells Property

You can use the `Cells` property of the `Application`, `Worksheet`, or `Range` objects to refer to the `Range` object containing all the cells in a `Worksheet` object or `Range` object. The following two lines of code each refer to a `Range` object that contains all the cells in the active worksheet:

```
ActiveSheet.Cells  
Application.Cells
```

Because the `Cells` property of the `Application` object is a member of `<globals>`, you can also refer to the `Range` object containing all the cells on the active worksheet as follows:

```
Cells
```

You can use the `Cells` property of a `Range` object as follows:

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

However, this code achieves nothing because it simply refers to the original `Range` object it qualifies.

You can refer to a specific cell relative to the `Range` object by using the `Item` property of the `Range` object and specifying the relative row and column positions. The row parameter is always numeric. The column parameter can be numeric, or you can use the column letters entered as a string. The following are both references to the `Range` object containing the B2 cell in the active worksheet:

```
Cells.Item(2,2)  
Cells.Item(2,"B")
```

Because the `Item` property is the default property of the `Range` object, you can omit it as follows:

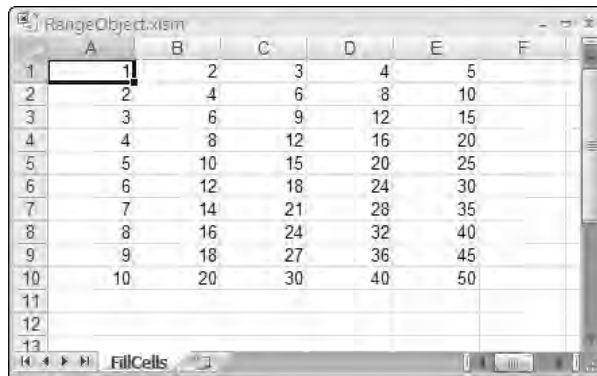
```
Cells(2,2)  
Cells(2,"B")
```

Chapter 4: Using Ranges

The numeric parameters are particularly useful when you want to loop through a series of rows or columns using an incrementing index number. The following example loops through rows 1 to 10 and columns A to E in the active worksheet, placing values in each cell:

```
Sub FillCells()  
    Dim lRow As Long, lColumn As Long  
  
    'Using Cells property to refer to range  
  
    'Loop through rows  
    For lRow = 1 To 10  
  
        'Loop through columns  
        For lColumn = 1 To 5  
  
            Cells(lRow, lColumn).Value = lRow * lColumn  
  
        Next lColumn  
  
    Next lRow  
End Sub
```

This gives the results shown in Figure 4-2.



	A	B	C	D	E	F
1	1	2	3	4	5	
2	2	4	6	8	10	
3	3	6	9	12	15	
4	4	8	12	16	20	
5	5	10	15	20	25	
6	6	12	18	24	30	
7	7	14	21	28	35	
8	8	16	24	32	40	
9	9	18	27	36	45	
10	10	20	30	40	50	
11						
12						
13						

Figure 4-2

Cells Used in Range

You can use the `Cells` property to specify the parameters within the `Range` property to define a `Range` object. The following code refers to A1:E10 in the active worksheet:

```
Range(Cells(1,1), Cells(10,5))
```

This type of referencing is particularly powerful because you can specify the parameters using numeric variables, as shown in the previous looping example.

Ranges of Inactive Worksheets

As with the `Range` property, you can apply the `Cells` property to a worksheet that is not currently active:

```
Worksheets("Sheet1").Cells(2,3)
```

If you want to refer to a block of cells on an inactive worksheet using the `Cells` property, the same precautions apply as with the `Range` property. You must make sure you qualify the `Cells` property fully. If `Sheet2` is active, and you want to refer to the range `A1:E10` on `Sheet1`, the following code will fail because `Cells(1,1)` and `Cells(10,5)` are properties of the active worksheet:

```
Sheets("Sheet1").Range(Cells(1,1), Cells(10,5)).Font.Bold = True
```

A `With...End With` construct is an efficient way to incorporate the correct sheet reference:

```
With Sheets("Sheet1")
    .Range(.Cells(1, 1), .Cells(10, 5)).Font.Bold = True
End With
```

More on the Cells Property of the Range Object

The `Cells` property of a `Range` object provides a nice way to refer to cells relative to a starting cell, or within a block of cells. The following refers to cell `F11`:

```
Range("D10:G20").Cells(2,3)
```

If you want to examine a range with the name `SalesData` and color any figure under 100 red, you can use the following code:

```
Sub ColorCells()
    Dim rngSales As Range
    Dim lRow As Long, lColumn As Long

    'Color cells using Cells property
    Set rngSales = Range("SalesData")

    For lRow = 1 To rngSales.Rows.Count

        For lColumn = 1 To rngSales.Columns.Count

            If rngSales.Cells(lRow, lColumn).Value < 100 Then
                rngSales.Cells(lRow, lColumn).Font.ColorIndex = 3
            Else
                rngSales.Cells(lRow, lColumn).Font.ColorIndex = 1
            End If

        Next lColumn

    Next lRow

End Sub
```

Chapter 4: Using Ranges

The result is shown in Figure 4-3.

	Jan	Feb	Mar	Apr	May	Jun	Jul
Total	1402	1455	1467	1508	1435	1592	1639
Prod1	13	104	70	183	97	10	174
Prod2	177	91	19	72	10	126	71
Prod3	177	111	127	33	145	150	164
Prod4	88	34	94	151	130	101	31
Prod5	7	106	126	116	157	1	61
Prod6	189	167	186	2	176	190	195
Prod7	47	188	36	141	77	130	193
Prod8	181	27	85	29	104	131	151
Prod9	5	182	120	103	181	48	134
Prod10	166	55	155	61	21	168	134
Prod11	24	121	172	124	103	145	132
Prod12	32	78	158	188	29	197	6
Prod13	166	53	45	131	47	191	38
Prod14	130	138	74	174	158	4	155

Figure 4-3

If you want to count all the cells in an Excel 2007 worksheet, you need to be aware that the `Count` property of the `Range` object is a Long Integer type and `Cells.Count` can't return the value of 17,179,869,184 cells, which exceeds the size of a Long. For compatibility with previous versions of Excel, `Count` is retained as it was and is supplemented by a new `CountLarge` property, which is a Variant that can return the larger value.

It is not, in fact, necessary to confine the referenced cells to the contents of the `Range` object. You can reference cells outside the original range. This means that you really only need to use the top-left cell of the `Range` object as a starting point. This code refers to F11, as in the earlier example:

```
Range("D10").Cells(2,3)
```

You can also use a shortcut version of this form of reference. The following is also a reference to cell F11:

```
Range("D10")(2,3)
```

Technically, this works because it is an allowable shortcut for the `Item` property of the `Range` object, rather than the `Cells` property, as described previously:

```
Range("D10").Item(2,3)
```

It is even possible to use zero or negative subscripts, as long as you don't attempt to reference outside the worksheet boundaries. This can lead to some odd results. The following code refers to cell C9:

```
Range("D10")(0,0)
```

The following refers to B8:

```
Range("D10")(-1,-1)
```

The previous `Font.ColorIndex` example using `rngSales` can be written as follows, using this technique:

```
Sub ColorCells2()
    Dim rngSales As Range
    Dim lRow As Long, lColumn As Long

    'Color cells using implied Item property
    Set rngSales = Range("SalesData")

    For lRow = 1 To rngSales.Rows.Count

        For lColumn = 1 To rngSales.Columns.Count

            If rngSales(lRow, lColumn).Value < 100 Then
                rngSales(lRow, lColumn).Font.ColorIndex = 3
            Else
                rngSales(lRow, lColumn).Font.ColorIndex = 1
            End If

        Next lColumn

    Next lRow
End Sub
```

There is actually a small increase in speed if you adopt this shortcut. Running the second example, the increase is about 5% on my PC when compared to the first example.

Single-Parameter Range Reference

The shortcut range reference accepts a single parameter as well as two. If you are using this technique with a range with more than one row, and the index exceeds the number of columns in the range, the reference wraps within the columns of the range, down to the appropriate row.

The following refers to cell E10:

```
Range("D10:E11")(2)
```

The following refers to cell D11:

```
Range("D10:E11")(3)
```

The index can exceed the number of cells in the `Range` object and the reference will continue to wrap within the `Range` object's columns. The following refers to cell D12:

```
Range("D10:E11")(5)
```

Chapter 4: Using Ranges

Qualifying a `Range` object with a single parameter is useful when you want to step through all the cells in a range without having to separately track rows and columns. The `ColorCells` example can be further rewritten as follows, using this technique:

```
Sub ColorCells3()  
    Dim rngSales As Range  
    Dim lCell As Long  
  
    'Color cells using single parameter range reference  
    Set rngSales = Range("SalesData")  
  
    For lCell = 1 To rngSales.Count  
  
        If rngSales(lCell).Value < 100 Then  
            rngSales(lCell).Font.ColorIndex = 3  
        Else  
            rngSales(lCell).Font.ColorIndex = 1  
        End If  
  
    Next lCell  
End Sub
```

In the fourth and final variation on the `ColorCells` theme, you can step through all the cells in a range using a `For Each...Next` loop, if you do not need the index value of the `For...Next` loop for other purposes:

```
Sub ColorCells4()  
    Dim rng As Range  
  
    'Color cells using For Each...Next loop  
  
    For Each rng In Range("SalesData")  
  
        If rng.Value < 100 Then  
            rng.Font.ColorIndex = 6  
        Else  
            rng.Font.ColorIndex = 1  
        End If  
  
    Next rng  
  
End Sub
```

Offset Property

The `Offset` property of the `Range` object returns a similar object to the `Cells` property, but is different in two ways. The first difference is that the `Offset` parameters are zero-based, rather than one-based, as the term *offset* implies. These examples both refer to the A10 cell:

```
Range("A10").Cells(1,1)  
Range("A10").Offset(0,0)
```


The second difference is that the `Range` object generated by `Cells` consists of one cell. The `Range` object referred to by the `Offset` property of a range has the same number of rows and columns as the original range. The following refers to B2:C3:

```
Range("A1:B2").Offset(1,1)
```

`Offset` is useful when you want to refer to ranges of equal sizes with a changing base point. For example, you might have sales figures for January to December in B1:B12 and want to generate a three-month moving average from March to December in C3:C12. The code to achieve this is:

```
Sub MovingAvgerage()
    Dim rng As Range
    Dim lRow As Long

    'Calculate moving average using Offset property
    Set rng = Range("B1:B3")

    For lRow = 3 To 12
        Cells(lRow, "C").Value = WorksheetFunction.Sum(rng) / 3
        Set rng = rng.Offset(1, 0)
    Next lRow

End Sub
```

The result of running the code is shown in Figure 4-4.

	A	B	C	D	E
1	Jan	100			
2	Feb	123			
3	Mar	115	113		
4	Apr	140	126		
5	May	120	125		
6	Jun	132	131		
7	Jul	124	125		
8	Aug	120	125		
9	Sep	115	120		
10	Oct	132	122		
11	Nov	143	130		
12	Dec	128	134		
13					
14					

Figure 4-4

Resize Property

You can use the `Resize` property of the `Range` object to refer to a range with the same top left-hand corner as the original range, but with a different number of rows and columns. The following refers to D10:E10:

```
Range("D10:F20").Resize(1,2)
```

Chapter 4: Using Ranges

Resize is useful when you want to extend or reduce a range by a row or column. For example, if you have a data list, which has been given the name `Database`, and you have just added another row at the bottom, you need to redefine the name to include the extra row. The following code extends the name by the extra row:

```
With Range("Database")
    .Resize(.Rows.Count + 1).Name = "Database"
End With
```

When you omit the second parameter, the number of columns remains unchanged. Similarly, you can omit the first parameter to leave the number of rows unchanged. The following refers to A1:C10:

```
Range("A1:B10").Resize(, 3)
```

You can use the following code to search for a value in a list and, having found it, copy it and the two columns to the right to a new location. The code to do this is:

```
Sub FindIt()
    Dim rng As Range

    'Find data and use Resize property to copy range
    Set rng = Range("A1:A12").Find(What:="Jun", _
        LookAt:=xlWhole, LookIn:=xlValues)

    If rng Is Nothing Then
        MsgBox "Data not found"
        Exit Sub
    Else
        rng.Resize(1, 3).Copy Destination:=Range("G1")
    End If
End Sub
```

And the result is shown in Figure 4-5.

	A	B	C	D	E	F	G	H	I	J
1	Jan	100					Jun	132	131	
2	Feb	123								
3	Mar	115	113							
4	Apr	140	126							
5	May	120	125							
6	Jun	132	131							
7	Jul	124	125							
8	Aug	120	125							
9	Sep	115	120							
10	Oct	132	122							
11	Nov	143	130							
12	Dec	128	134							
13										
14										
15										
16										

Figure 4-5

The `Find` method does not act like the `Edit ⇨ Find` command. It returns a reference to the found cell as a `Range` object, but it does not select the found cell. If `Find` does not locate a match, it returns a `null` object that you can test for with the `Is Nothing` expression. If you attempt to copy the `null` object, a run-time error occurs.

SpecialCells Method

When you press the `F5` key in a worksheet, the `Go To` dialog box appears. You can then click the `Special` button to show the dialog box in Figure 4-6.



Figure 4-6

This dialog allows you to do a number of useful things, such as find the last cell in the worksheet or all the cells with numbers rather than calculations. As you might expect, all these operations can be carried out in VBA code. Some have their own methods, but most of them can be performed using the `SpecialCells` method of the `Range` object.

Last Cell

The following code determines the last row and column in the worksheet:

```
Set rngLast = Range("A1").SpecialCells(xlCellTypeLastCell)
lLastRow = rngLast.Row
lLastCol = rngLast.Column
```

The last cell is considered to be the intersection of the highest-numbered row in the worksheet that contains information and the highest-numbered column in the worksheet that contains information. Excel also includes cells that have contained information during the current session, even if you have deleted that information. The last cell is not reset until you save the worksheet.

Chapter 4: Using Ranges

Excel considers formatted cells and unlocked cells to contain information. As a result, you will often find the last cell well beyond the region containing data, especially if the workbook has been imported from another spreadsheet application, such as Lotus 1-2-3. If you want to consider only cells that contain data in the form of numbers, text, and formulas, you can use the following code:

```
Sub GetRealLastCell()  
    Dim lRealLastRow As Long  
    Dim lRealLastColumn As Long  
  
    'Get bottom right corner of cells with data  
    Range("A1").Select  
  
    On Error Resume Next  
    lRealLastRow = Cells.Find("*", Range("A1"), xlFormulas, , xlByRows, _  
        xlPrevious).Row  
    lRealLastColumn = Cells.Find("*", Range("A1"), xlFormulas, , _  
        xlByColumns, xlPrevious).Column  
  
    Cells(lRealLastRow, lRealLastColumn).Select  
  
End Sub
```

In this example, the `Find` method searches backward from the A1 cell (which means that Excel wraps around the worksheet and starts searching from the last cell toward the A1 cell) to find the last row and column containing any characters. The `On Error Resume Next` statement is used to prevent a run-time error when the spreadsheet is empty.

Note that it is necessary to declare the row number variables as Long, rather than Integer, because integers can only be as high as 32,767 and Excel 2007 worksheets can contain 1,048,576 rows.

If you want to get rid of the extra rows containing formats, you should select the entire rows by selecting their row numbers and then clicking `Edit ⇄ Delete` to remove them. You can also select the unnecessary columns by their column letters and delete them. At this point, the last cell will not be reset. You can save the worksheet to reset the last cell, or execute `ActiveSheet.UsedRange` in your code to perform a reset. The following code will remove extraneous rows and columns and reset the last cell:

```
Sub DeleteUnusedFormats()  
    Dim lLastRow As Long, lLastColumn As Long  
    Dim lRealLastRow As Long, lRealLastColumn As Long  
  
    'Delete from used range rows & columns that have no data  
  
    'Detect end of used range including empty formatted cells  
    With Range("A1").SpecialCells(xlCellTypeLastCell)  
        lLastRow = .Row  
        lLastColumn = .Column  
    End With
```

```

End With

'Find end of cells with data
lRealLastRow = _
Cells.Find("*", Range("A1"), xlFormulas, , xlByRows, xlPrevious).Row

lRealLastColumn = _
Cells.Find("*", Range("A1"), xlFormulas, , _
xlByColumns, xlPrevious).Column

'If used range exceeds data, delete unused rows & columns
If lRealLastRow < lLastRow Then
    Range(Cells(lRealLastRow + 1, 1), Cells(lLastRow, 1)).EntireRow.Delete
End If

If lRealLastColumn < lLastColumn Then
    Range(Cells(1, lRealLastColumn + 1), _
Cells(1, lLastColumn)).EntireColumn.Delete
End If

ActiveSheet.UsedRange    'Resets LastCell
End Sub

```

The `EntireRow` property of a `Range` object refers to a `Range` object that spans the entire spreadsheet—that is, columns 1 to 16,384. (Or A to XFD on the rows contained in the original range. The `EntireColumn` property of a `Range` object refers to a `Range` object that spans the entire spreadsheet [rows 1 to 1,048,576] in the columns contained in the original object.)

Deleting Numbers

Sometimes it is useful to delete all the input data in a worksheet or template so it is more obvious where new values are required. The following code deletes all the numbers in a worksheet, leaving the formulas intact:

```

On Error Resume Next
Cells.SpecialCells(xlCellTypeConstants, xlNumbers).ClearContents

```

The preceding code should begin with the `On Error` statement if you want to prevent a run-time error when there are no numbers to be found.

Excel considers dates as numbers, and they will be cleared by the preceding code. If you have used dates as headings and want to avoid this, you can use the following code:

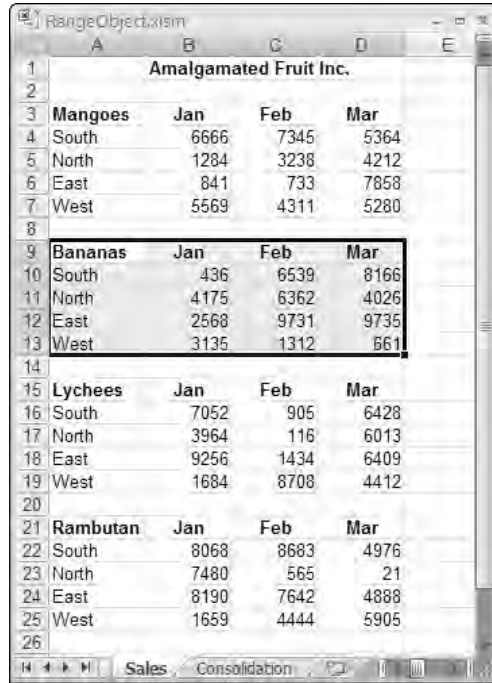
```

On Error Resume Next
For Each rng In Cells.SpecialCells(xlCellTypeConstants, xlNumbers)
    If Not IsDate(rng.Value) Then rng.ClearContents
Next rng

```

CurrentRegion Property

If you have tables of data that are separated from surrounding data by at least one empty row and one empty column, you can select an individual table using the `CurrentRegion` property of any cell in the table. It is equivalent to the manual `Ctrl+*` keyboard shortcut (or `Ctrl+A`). In the Figure 4-7 worksheet, you could select the Bananas table by clicking the A9 cell and pressing `Ctrl+*`.



Amalgamated Fruit Inc.				
	Mangoes	Jan	Feb	Mar
	South	6666	7345	5364
	North	1284	3238	4212
	East	841	733	7858
	West	5569	4311	5280
	Bananas	Jan	Feb	Mar
	South	436	6539	8166
	North	4175	6362	4026
	East	2568	9731	9735
	West	3135	1312	661
	Lychees	Jan	Feb	Mar
	South	7052	905	6428
	North	3964	116	6013
	East	9256	1434	6409
	West	1684	8708	4412
	Rambutan	Jan	Feb	Mar
	South	8068	8683	4976
	North	7480	565	21
	East	8190	7642	4888
	West	1659	4444	5905

Figure 4-7

The same result can be achieved with the following code, given that cell A9 has been named Bananas:

```
Range("Bananas").CurrentRegion.Select
```

This property is very useful for tables that change size over time. You can select all the months up to the current month as the table grows during the year, without having to change the code each month. Naturally, in your code, there is rarely any need to select anything. If you want to perform a consolidation of the fruit figures into a single table in a sheet called `Consolidation`, and you have named the top-left corner of each table with the product name, you can use the following code:

```
Sub Consolidate()  
    Dim vProducts As Variant  
    Dim rngCopy As Range 'Range to be copied  
    Dim rngDestination As Range
```

```

Dim iProductIndex As Integer

'Sum each product to give consolidated result
Application.ScreenUpdating = False

vProducts = Array("Mangoes", "Bananas", "Lychees", "Rambutan")

Set rngDestination = Worksheets("Consolidation").Range("B4")

For iProductIndex = LBound(vProducts) To UBound(vProducts)

    With Range(vProducts(iProductIndex)).CurrentRegion
        'Exclude headings from copy range
        Set rngCopy = .Offset(1, 1).Resize(.Rows.Count - 1, .Columns.Count - 1)
    End With

    rngCopy.Copy

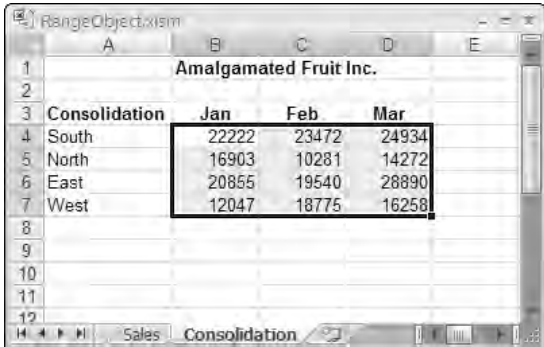
    If iProductIndex = LBound(vProducts) Then
        'Paste the first product values
        rngDestination.PasteSpecial xlPasteValues, xlPasteSpecialOperationNone
    Else
        'Add the other product values
        rngDestination.PasteSpecial xlPasteValues, xlPasteSpecialOperationAdd
    End If

Next iProductIndex

Application.CutCopyMode = False 'Clear the clipboard
End Sub

```

This gives the output in Figure 4-8.



Amalgamated Fruit Inc.			
Consolidation	Jan	Feb	Mar
South	22222	23472	24934
North	16903	10281	14272
East	20855	19540	28890
West	12047	18775	16258

Figure 4-8

Screen updating is suppressed to cut out screen flicker and speed up the macro. The `Array` function is a convenient way to define relatively short lists of items to be processed. The `LBound` and `UBound` functions are used to avoid worrying about which `Option Base` has been set in the declarations section of the module. The code can be reused in other modules without a problem.

Chapter 4: Using Ranges

The first product is copied and its values are pasted over any existing values in the destination cells. The other products are copied and their values added to the destination cells. The clipboard is cleared at the end to prevent users accidentally carrying out another paste by pressing the Enter key.

End Property

The `End` property emulates the operation of Ctrl+arrow key. If you have selected a cell at the top of a column of data, Ctrl+down arrow takes you to the next item of data in the column that is before an empty cell. If there are no empty cells in the column, you go to the last data item in the column. If the cell after the selected cell is empty, you jump to the next cell with data, if there is one, or the bottom of the worksheet.

The following code refers to the last data cell at the bottom of column A if there are no empty cells between it and A1:

```
Range("A1").End(xlDown)
```

To go in other directions, you use the constants `xlUp`, `xlToLeft`, and `xlToRight`.

If there are gaps in the data, and you want to refer to the last cell in column A, you can start from the bottom of the worksheet and go up, as long as data does not extend as far as A1048576:

```
Range("A1048576").End(xlUp)
```

In the section on rows later in this chapter, you will see a way to avoid the A1048576 reference and generalize the preceding code for different versions of Excel.

Referring to Ranges with End

You can refer to a range of cells from the active cell to the end of the same column with:

```
Range(ActiveCell, ActiveCell.End(xlDown)).Select
```

Say you have a table of data, starting at cell B3, which is separated from surrounding data by an empty row and an empty column. You can refer to the table, as long as it has continuous headings across the top and continuous data in the last column, using this line of code:

```
Range("B3", Range("B3").End(xlToRight).End(xlDown)).Select
```

The effect, in this case, is the same as using the `CurrentRegion` property, but `End` has many more uses, as you will see in the following examples.

As usual, there is no need to select anything if you want to operate on a `Range` object in VBA. The following code copies the continuous headings across the top of Sheet1 to the top of Sheet2:

```
With Worksheets("Sheet1").Range("A1")  
    .Range(.Cells(1), .End(xlToRight)).Copy Destination:= _  
        Worksheets("Sheet2").Range("A1")  
End With
```


This code can be executed, no matter what sheet is active, as long as the workbook that contains Sheet1 and Sheet2 is active.

Summing a Range

Say you want to place a SUM function in the active cell to add the values of the cells below it, down to the next empty cell. You can do that with the following code:

```
With ActiveCell
    Set rng = Range(.Offset(1), .Offset(1).End(xlDown))
    .Formula = "=SUM(" & rng.Address & ")"
End With
```

The Address property of the Range object returns an absolute address by default. If you want to be able to copy the formula to other cells and sum the data below them, you can change the address to a relative one and perform the copy as follows:

```
With ActiveCell
    Set rng = Range(.Offset(1), .Offset(1).End(xlDown))
    .Formula = "=SUM(" & rng.Address(RowAbsolute:=False, _
        ColumnAbsolute:=False) & ")"
    .Copy Destination:=Range(.Cells(1), .Offset(1).End(xlToRight).Offset(-1))
End With
```

The end of the destination range is determined by dropping down a row from the SUM, finding the last data column to the right, and popping back up a row.

Figure 4-9 shows what you get after selecting B2 and running the code.

	A	B	C	D	E	F	G	H	I
1		Jan	Feb	Mar	Apr	May	Jun	Jul	
2	Total	1402	1455	1467	1508	1435	1592	1639	
3	Prod1	13	104	70	183	97	10	174	
4	Prod2	177	91	19	72	10	126	71	
5	Prod3	177	111	127	33	145	150	164	
6	Prod4	88	34	94	151	130	101	31	
7	Prod5	7	106	126	116	157	1	61	
8	Prod6	189	167	186	2	176	190	195	
9	Prod7	47	188	36	141	77	130	193	
10	Prod8	181	27	85	29	104	131	151	
11	Prod9	5	182	120	103	181	48	134	
12	Prod10	166	55	155	61	21	168	134	
13	Prod11	24	121	172	124	103	145	132	
14	Prod12	32	78	158	188	29	197	6	
15	Prod13	166	53	45	131	47	191	38	
16	Prod14	130	138	74	174	158	4	155	
17									

Figure 4-9

Columns and Rows Properties

Columns and Rows are properties of the Application, Worksheet, and Range objects. They return a reference to all the columns or rows in a worksheet or range. In each case, the reference returned is a Range object, but this Range object has some odd characteristics that might make you think there are such things as a "Column object" and a "Row object," which do not exist in Excel. They are useful when you want to count the number of rows or columns, or process all the rows or columns of a range.

Excel 97 increased the number of worksheet rows from the 16,384 in previous versions to 65,536. Excel 2007 has increased the number to 1,048,576. If you want to write code to detect the number of rows in the active sheet, you can use the Count property of Rows:

```
Rows.Count
```

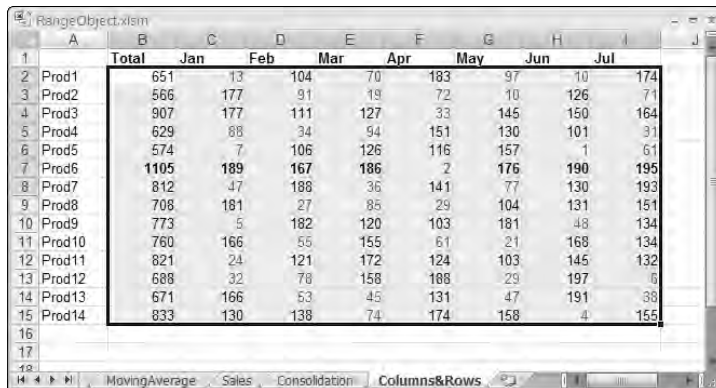
This is useful if you need a macro that will work with all versions of Excel VBA and detect the last row of data in a column, working from the bottom of the worksheet:

```
Cells(Rows.Count, "A").End(xlUp).Select
```

If you have a multi-column table of data in a range named Data, and you want to step through each row of the table, making every cell in each row bold where the first cell is greater than 1000, you can use:

```
For Each rngRow In Range("Data").Rows  
  
    If rngRow.Cells(1).Value > 1000 Then  
        rngRow.Font.Bold = True  
    Else  
        rngRow.Font.Bold = False  
    End If  
  
Next rngRow
```

This provides the result shown in Figure 4-10.



	Total	Jan	Feb	Mar	Apr	May	Jun	Jul
Prod1	651	13	104	70	183	97	10	174
Prod2	566	177	91	19	72	10	126	71
Prod3	907	177	111	127	33	145	150	164
Prod4	629	88	34	94	151	130	101	31
Prod5	574	7	106	126	116	157	1	61
Prod6	1105	189	167	186	2	176	190	195
Prod7	812	47	188	36	141	77	130	193
Prod8	708	181	27	85	29	104	131	151
Prod9	773	5	182	120	103	181	48	134
Prod10	760	166	55	155	61	21	168	134
Prod11	821	24	121	172	124	103	145	132
Prod12	688	32	78	158	188	29	197	6
Prod13	671	166	53	45	131	47	191	38
Prod14	833	130	138	74	174	158	4	155

Figure 4-10

Curiously, you cannot replace `rngRow.Cells(1)` with `rngRow(1)`, as you can with a normal `Range` object, because it returns a reference to the entire row and causes a run-time error. It seems that there is something special about the `Range` object referred to by the `Rows` and `Columns` properties. You may find it helps to think of them as `Row` and `Column` objects, even though such objects do not officially exist.

Areas

You need to be careful when using the `Columns` or `Rows` properties of non-contiguous ranges, such as those returned from the `SpecialCells` method when locating the numeric cells or blank cells in a worksheet, for example. Recall that a non-contiguous range consists of a number of separate rectangular blocks. If the cells are not all in one block, and you use the `Rows.Count` properties, you only count the rows from the first block. The following code generates an answer of 5, because only the first range, A1:B5, is evaluated:

```
Range("A1:B5,C6:D10,E11:F15").Rows.Count
```

The blocks in a non-contiguous range are `Range` objects contained within the `Areas` collection and can be processed separately. The following displays the address of each of the three blocks in the `Range` object, one at a time:

```
For Each rng In Range("A1:B5,C6:D10,E11:F15").Areas
    MsgBox rng.Address
Next rng
```

The worksheet shown in Figure 4-11 contains sales estimates that have been entered as numbers. The cost figures are calculated by formulas.

The screenshot shows an Excel spreadsheet with the following data:

	A	B	C	D	E	F
1						
2		Jan	Feb	Mar		
3	Sales1	100	120	130		
4						
5	Cost1	60	72	78		
6						
7	Sales2	50	60	70		
8						
9	Cost2	30	36	42		
10						
11	Sales3	40	35	55		
12						
13	Cost3	24	21	33		
14						
15						
16						

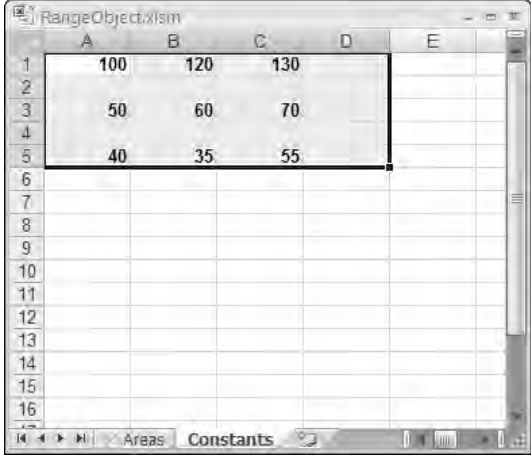
Figure 4-11

Chapter 4: Using Ranges

The following code copies all the numeric constants in the active sheet to blocks in the sheet named Constants, leaving an empty row between each block:

```
Sub CopyAreas()  
    Dim rng As Range, rngDestination As Range  
  
    'Copy the areas in a non-contiguous range  
  
    'Set the destination range  
    Set rngDestination = Worksheets("Constants").Range("A1")  
  
    'Process each non-contiguous area of numeric values  
    For Each rng In Cells.SpecialCells(xlCellTypeConstants, xlNumbers).Areas  
  
        rng.Copy Destination:=rngDestination  
  
        ' Set next destination under previous block copied  
        Set rngDestination = rngDestination.Offset(rng.Rows.Count + 1)  
  
    Next rng  
  
End Sub
```

This gives the result shown in Figure 4-12.



The screenshot shows a window titled 'RangeObject.xlsm' with a spreadsheet grid. The active sheet is 'Constants'. The grid shows columns A through E and rows 1 through 16. A range of cells is selected, containing the following data:

	A	B	C	D	E
1	100	120	130		
2					
3	50	60	70		
4					
5	40	35	55		
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					

Figure 4-12

Union and Intersect Methods

`Union` and `Intersect` are methods of the `Application` object, but they can be used without preceding them with a reference to `Application` because they are members of `<globals>`. They can be very useful tools, as you shall see.

Use `Union` when you want to generate a range from two or more blocks of cells. Use `Intersect` when you want to find the cells that are common to two or more ranges, or in other words, where the ranges overlap. The following event procedure, entered in the module behind a worksheet, illustrates how you can apply the two methods to prevent a user from selecting cells in two ranges B10:F20 and H10:L20. One use for this routine is to prevent a user from changing data in these two blocks:

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
    Dim rngForbidden As Range

    'Define forbidden range
    Set rngForbidden = Union(Range("B10:F20"), Range("H10:L20"))

    'If selection does not overlap forbidden areas, do nothing
    If Intersect(Target, rngForbidden) Is Nothing Then Exit Sub

    'Select A1 and issue warning
    Range("A1").Select
    MsgBox "You can't select cells in " & rngForbidden.Address, vbCritical
End Sub
```

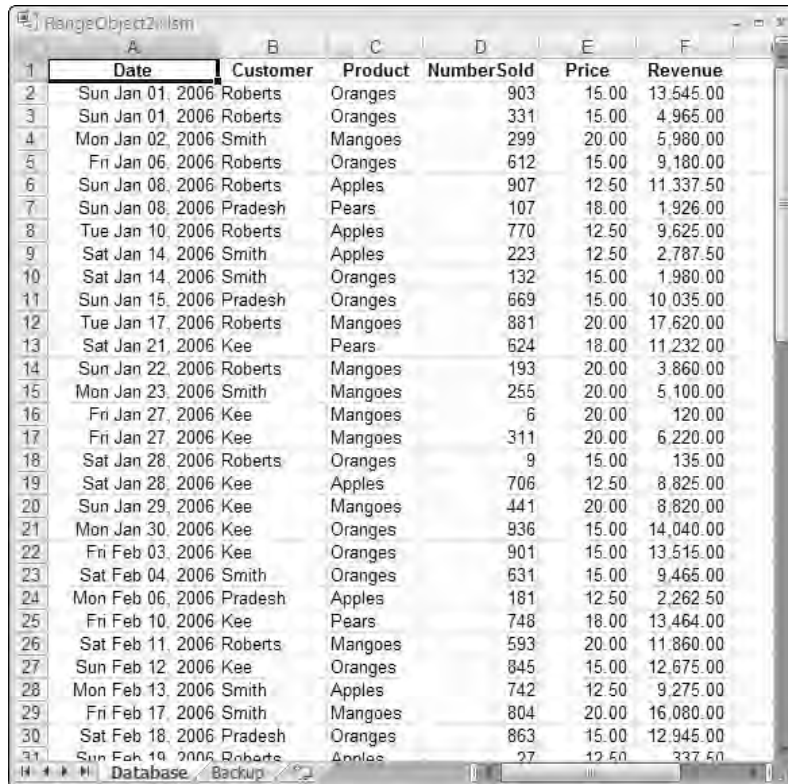
If you are not familiar with event procedures, refer to the “Events” section in Chapter 1. For more information on event procedures, see Chapter 10.

The `Worksheet_SelectionChange` event procedure is triggered every time the user selects a new range in the worksheet associated with the module containing the event procedure. The preceding code uses the `Union` method to define a forbidden range consisting of the two non-contiguous ranges. It then uses the `Intersect` method, in the `If` test, to see if the `Target` range, which is the new user selection, is within the forbidden range. `Intersect` returns `Nothing` if there is no overlap and the `Sub` exits. If there is an overlap, the code in the two lines following the `If` test are executed — cell A1 is selected and a warning message is issued to the user.

Empty Cells

You have seen that if you want to step through a column or row of cells until you get to an empty cell, you can use the `End` property to detect the end of the block. Another way is to examine each cell, one at a time, in a loop structure and stop when you find an empty cell. You can test for an empty cell with the VBA `IsEmpty` function.

In the spreadsheet shown in Figure 4-13, you want to insert blank rows between each week to produce a report that is more readable.



	A	B	C	D	E	F
1	Date	Customer	Product	NumberSold	Price	Revenue
2	Sun Jan 01, 2006	Roberts	Oranges	903	15.00	13,545.00
3	Sun Jan 01, 2006	Roberts	Oranges	331	15.00	4,965.00
4	Mon Jan 02, 2006	Smith	Mangoes	299	20.00	5,980.00
5	Fri Jan 06, 2006	Roberts	Oranges	612	15.00	9,180.00
6	Sun Jan 08, 2006	Roberts	Apples	907	12.50	11,337.50
7	Sun Jan 08, 2006	Pradesh	Pears	107	18.00	1,926.00
8	Tue Jan 10, 2006	Roberts	Apples	770	12.50	9,625.00
9	Sat Jan 14, 2006	Smith	Apples	223	12.50	2,787.50
10	Sat Jan 14, 2006	Smith	Oranges	132	15.00	1,980.00
11	Sun Jan 15, 2006	Pradesh	Oranges	669	15.00	10,035.00
12	Tue Jan 17, 2006	Roberts	Mangoes	881	20.00	17,620.00
13	Sat Jan 21, 2006	Kee	Pears	624	18.00	11,232.00
14	Sun Jan 22, 2006	Roberts	Mangoes	193	20.00	3,860.00
15	Mon Jan 23, 2006	Smith	Mangoes	255	20.00	5,100.00
16	Fri Jan 27, 2006	Kee	Mangoes	6	20.00	120.00
17	Fri Jan 27, 2006	Kee	Mangoes	311	20.00	6,220.00
18	Sat Jan 28, 2006	Roberts	Oranges	9	15.00	135.00
19	Sat Jan 28, 2006	Kee	Apples	706	12.50	8,825.00
20	Sun Jan 29, 2006	Kee	Mangoes	441	20.00	8,820.00
21	Mon Jan 30, 2006	Kee	Oranges	936	15.00	14,040.00
22	Fri Feb 03, 2006	Kee	Oranges	901	15.00	13,515.00
23	Sat Feb 04, 2006	Smith	Oranges	631	15.00	9,465.00
24	Mon Feb 06, 2006	Pradesh	Apples	181	12.50	2,262.50
25	Fri Feb 10, 2006	Kee	Pears	748	18.00	13,464.00
26	Sat Feb 11, 2006	Roberts	Mangoes	593	20.00	11,860.00
27	Sun Feb 12, 2006	Kee	Oranges	845	15.00	12,675.00
28	Mon Feb 13, 2006	Smith	Apples	742	12.50	9,275.00
29	Fri Feb 17, 2006	Smith	Mangoes	804	20.00	16,080.00
30	Sat Feb 18, 2006	Pradesh	Oranges	863	15.00	12,945.00
31	Sun Feb 19, 2006	Roberts	Apples	27	12.50	337.50

Figure 4-13

The following macro compares dates, using the VBA `weekday` function to get the day of the week as a number. By default, Sunday is day 1 and Saturday is day 7. If the macro finds that today's day number is less than yesterday's, it assumes a new week has started and inserts a blank row:

```
Sub ShowWeeks()  
    Dim iToday As Integer  
    Dim iYesterday As Integer  
  
    'Insert empty rows between weeks  
  
    Range("A2").Select  
  
    iYesterday = Weekday(ActiveCell.Value)  
  
    'Loop until an empty cell is found  
    Do Until IsEmpty(ActiveCell.Value)  
  
        'Select cell below  
        ActiveCell.Offset(1, 0).Select  
  
        'Calculate day of week from date in cell
```

```

iToday = Weekday(ActiveCell.Value)

'If day index has decreased, insert row
If iToday < iYesterday Then
    ActiveCell.EntireRow.Insert
    ActiveCell.Offset(1, 0).Select
End If

'Store latest week day index
iYesterday = iToday

Loop
End Sub

```

The result is shown in Figure 4-14.

	A	B	C	D	E	F
1	Date	Customer	Product	NumberSold	Price	Revenue
2	Sun Jan 01, 2006	Roberts	Oranges	903	15.00	13,545.00
3	Sun Jan 01, 2006	Roberts	Oranges	331	15.00	4,965.00
4	Mon Jan 02, 2006	Smith	Mangoes	299	20.00	5,980.00
5	Fri Jan 06, 2006	Roberts	Oranges	612	15.00	9,180.00
6						
7	Sun Jan 08, 2006	Roberts	Apples	907	12.50	11,337.50
8	Sun Jan 08, 2006	Pradesh	Pears	107	18.00	1,926.00
9	Tue Jan 10, 2006	Roberts	Apples	770	12.50	9,625.00
10	Sat Jan 14, 2006	Smith	Apples	223	12.50	2,787.50
11	Sat Jan 14, 2006	Smith	Oranges	132	15.00	1,980.00
12						
13	Sun Jan 15, 2006	Pradesh	Oranges	669	15.00	10,035.00
14	Tue Jan 17, 2006	Roberts	Mangoes	881	20.00	17,620.00
15	Sat Jan 21, 2006	Kee	Pears	624	18.00	11,232.00
16						
17	Sun Jan 22, 2006	Roberts	Mangoes	193	20.00	3,860.00
18	Mon Jan 23, 2006	Smith	Mangoes	255	20.00	5,100.00
19	Fri Jan 27, 2006	Kee	Mangoes	6	20.00	120.00
20	Fri Jan 27, 2006	Kee	Mangoes	311	20.00	6,220.00
21	Sat Jan 28, 2006	Roberts	Oranges	9	15.00	135.00
22	Sat Jan 28, 2006	Kee	Apples	706	12.50	8,825.00
23						
24	Sun Jan 29, 2006	Kee	Mangoes	441	20.00	8,820.00
25	Mon Jan 30, 2006	Kee	Oranges	936	15.00	14,040.00
26	Fri Feb 03, 2006	Kee	Oranges	901	15.00	13,515.00
27	Sat Feb 04, 2006	Smith	Oranges	631	15.00	9,465.00
28						
29	Mon Feb 06, 2006	Pradesh	Apples	181	12.50	2,262.50
30	Fri Feb 10, 2006	Kee	Pears	748	18.00	13,464.00
31	Sat Feb 11, 2006	Roberts	Mangoes	593	20.00	11,860.00

Figure 4-14

Note that many users detect an empty cell by testing for a zero-length string:

```
Do Until ActiveCell.Value = ""
```

Chapter 4: Using Ranges

This test works in most cases, and would have worked in the previous example, had it been used. However, problems can occur if you are testing cells that contain formulas that can produce zero-length strings, such as the following:

```
=IF (B2="Kee", "Trainee", "")
```

The zero-length string test does not distinguish between an empty cell and a zero-length string resulting from a formula. It is better practice to use the VBA `IsEmpty` function when testing for an empty cell.

Transferring Values between Arrays and Ranges

If you want to process all the data values in a range, it is much more efficient to assign the values to a VBA array and process the array rather than process the `Range` object itself. You can then assign the array back to the range.

You can assign the values in a range to an array very easily, as follows:

```
vSalesData = Range("A2:F10000").Value
```

The transfer is very fast compared with stepping through the cells one at a time. Note that this is quite different from creating an object variable referring to the range using:

```
Set rngSalesData = Range("A2:F10000")
```

When you assign range values to a variable such as `vSalesData`, the variable must have a `Variant` data type. VBA copies all the values in the range to the variable, creating an array with two dimensions. The first dimension represents the rows and the second dimension represents the columns, so you can access the values by their row and column numbers in the array. To assign the value in the first row and second column of the array to `sCustomer`, use:

```
sCustomer = vSalesData(1, 2)
```

When the values in a range are assigned to a `Variant`, the indexes of the array that is created are always one-based, not zero-based, regardless of the `Option Base` setting in the declarations section of the module. Also, the array always has two dimensions, even if the range has only one row or one column. This preserves the inherent column and row structure of the worksheet in the array and is an advantage when you write the array back to the worksheet.

For example, if you assign the values in A1:A10 to `vSalesData`, the first element is `vSalesData(1, 1)` and the last element is `vSalesData(10, 1)`. If you assign the values in A1:E1 to `vSalesData`, the first element is `vSalesData(1, 1)` and the last element is `vSalesData(1, 5)`.

You might want a macro that sums all the Revenues for Kee in the previous example. The following macro uses the traditional method to directly test and sum the range of data:


```

Sub KeeTotal()
    Dim dTotal As Double
    Dim lRow As Long

    'Specify data range
    With Range("A2:F54")

        'Loop through rows
        For lRow = 1 To .Rows.Count

            'Sum rows for Kee
            If .Cells(lRow, 2) = "Kee" Then dTotal = dTotal + .Cells(lRow, 6)

        Next lRow

    End With

    'Display result
    MsgBox "Kee Total = " & Format(dTotal, "$#,##0")

End Sub

```

The following macro does the same job by first assigning the Range values to a Variant and processing the resulting array. The speed increase is very significant, which can be a great advantage if you are handling large ranges:

```

Sub KeeTotal2()
    Dim vSalesData As Variant
    Dim dTotal As Double
    Dim lRow As Long

    'Assign range values to variant
    vSalesData = Range("A2:F54").Value

    'Sum elements of the array
    For lRow = 1 To UBound(vSalesData, 1)
        If vSalesData(lRow, 2) = "Kee" Then dTotal = dTotal + vSalesData(lRow, 6)
    Next lRow

    'Display result
    MsgBox "Kee Total = " & Format(dTotal, "$#,##0")

End Sub

```

You can also assign an array of values directly to a Range. Say you want to place a list of numbers in column G of the RangeObject2.xlsm example, containing a 10% discount on Revenue for customer Kee only. The following macro, once again, assigns the range values to a Variant for processing:

```

Sub KeeDiscount()
    Dim vSalesData As Variant
    Dim vaDiscount() As Variant
    Dim i As Long

    'Assign range values to variant

```

Chapter 4: Using Ranges

```
vSalesData = Range("A2:F54").Value

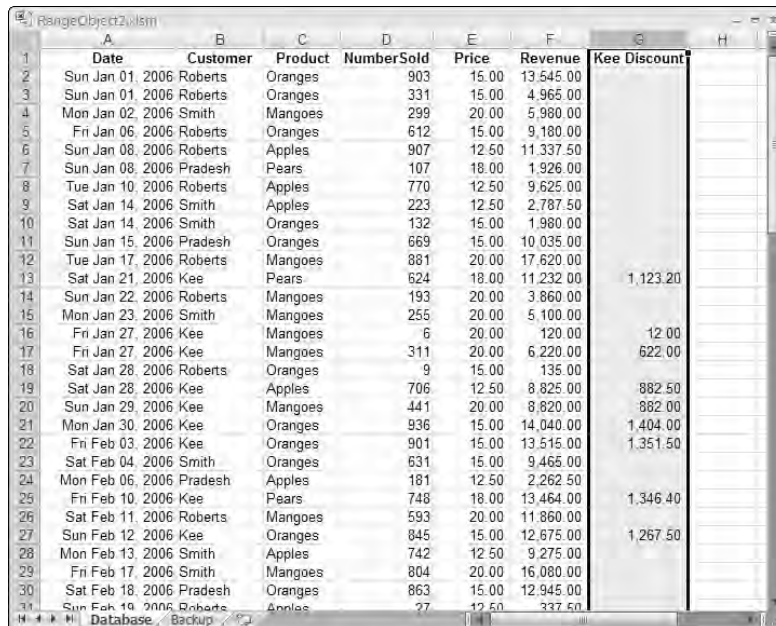
'Match output array row count to input row count
ReDim vaDiscount(1 To UBound(vSalesData, 1), 1 To 1)

'Process data in variant
For i = 1 To UBound(vSalesData, 1)
    If vSalesData(i, 2) = "Kee" Then
        vaDiscount(i, 1) = vSalesData(i, 6) * 0.1
    End If
Next i

'Write array values to worksheet
Range("G2").Resize(UBound(vSalesData, 1), 1).Value = vaDiscount
End Sub
```

The code sets up a dynamic array called `vaDiscount` and uses `ReDim` to give `vaDiscount` the same number of rows in `vSalesData` and one column, so that it retains a two-dimensional structure like a range, even though there is only one column. After the values have been assigned to `vaDiscount`, `vaDiscount` is directly assigned to the range in column G. Note that it is necessary to specify the correct size of the range receiving the values, not just the first cell as in a worksheet copy operation.

The outcome of this operation is shown in Figure 4-15.



	A	B	C	D	E	F	G	H
1	Date	Customer	Product	Number Sold	Price	Revenue	Keel Discount	
2	Sun Jan 01, 2006	Roberts	Oranges	903	15.00	13,545.00		
3	Sun Jan 01, 2006	Roberts	Oranges	331	15.00	4,965.00		
4	Mon Jan 02, 2006	Smith	Mangoes	299	20.00	5,980.00		
5	Fri Jan 06, 2006	Roberts	Oranges	612	15.00	9,180.00		
6	Sun Jan 08, 2006	Roberts	Apples	907	12.50	11,337.50		
7	Sun Jan 08, 2006	Pradesh	Pears	107	18.00	1,926.00		
8	Tue Jan 10, 2006	Roberts	Apples	770	12.50	9,625.00		
9	Sat Jan 14, 2006	Smith	Apples	223	12.50	2,787.50		
10	Sat Jan 14, 2006	Smith	Oranges	132	15.00	1,980.00		
11	Sun Jan 15, 2006	Pradesh	Oranges	669	15.00	10,035.00		
12	Tue Jan 17, 2006	Roberts	Mangoes	881	20.00	17,620.00		
13	Sat Jan 21, 2006	Kee	Pears	624	18.00	11,232.00	1,123.20	
14	Sun Jan 22, 2006	Roberts	Mangoes	193	20.00	3,860.00		
15	Mon Jan 23, 2006	Smith	Mangoes	255	20.00	5,100.00		
16	Fri Jan 27, 2006	Kee	Mangoes	6	20.00	120.00	12.00	
17	Fri Jan 27, 2006	Kee	Mangoes	311	20.00	6,220.00	622.00	
18	Sat Jan 28, 2006	Roberts	Oranges	9	15.00	135.00		
19	Sat Jan 28, 2006	Kee	Apples	706	12.50	8,825.00	882.50	
20	Sun Jan 29, 2006	Kee	Mangoes	441	20.00	8,820.00	882.00	
21	Mon Jan 30, 2006	Kee	Oranges	936	15.00	14,040.00	1,404.00	
22	Fri Feb 03, 2006	Kee	Oranges	901	15.00	13,515.00	1,351.50	
23	Sat Feb 04, 2006	Smith	Oranges	631	15.00	9,465.00		
24	Mon Feb 06, 2006	Pradesh	Apples	181	12.50	2,262.50		
25	Fri Feb 10, 2006	Kee	Pears	748	18.00	13,464.00	1,346.40	
26	Sat Feb 11, 2006	Roberts	Mangoes	593	20.00	11,860.00		
27	Sun Feb 12, 2006	Kee	Oranges	845	15.00	12,675.00	1,267.50	
28	Mon Feb 13, 2006	Smith	Apples	742	12.50	9,275.00		
29	Fri Feb 17, 2006	Smith	Mangoes	804	20.00	16,080.00		
30	Sat Feb 18, 2006	Pradesh	Oranges	863	15.00	12,945.00		
31	Sun Feb 19, 2006	Roberts	Apples	27	12.50	337.50		

Figure 4-15

It is possible to use a one-dimensional array for `vaDiscount`. However, if you assign the one-dimensional array to a range, it will be assumed to contain a row of data, not a column. It is possible to get around

this by using the worksheet `Transpose` function when assigning the array to the range. Say you have changed the dimensions of `vaDiscount` as follows:

```
ReDim vaDiscount(1 To Ubound(vSalesData,1))
```

You could assign this version of `vaDiscount` to a column with:

```
Range("G2").Resize(Ubound(vSalesData, 1), 1).Value = _
    WorksheetFunction.Transpose(vaDiscount)
```

Deleting Rows

A commonly asked question is, “What is the best way to delete unneeded rows from a spreadsheet?” Generally, the requirement is to find the rows that have certain text in a given column and remove those rows. The best solution depends on how large the spreadsheet is and how many items are likely to be removed.

Say that you want to remove all the rows that contain the text *Mangoes* in column C. One way to do this is to loop through all the rows and test every cell in column C. If you do this, it is better to test the last row first and work up the worksheet row by row. This is more efficient because Excel does not have to move any rows up that would later be deleted, which would not be the case if you worked from the top down. Also, if you work from the top down, you can’t use a simple `For . . . Next` loop counter to keep track of the row you are on, because as you delete rows, the counter and the row numbers no longer correspond:

```
Sub DeleteRows()
    Dim lRow As Long

    'Freeze screen
    Application.ScreenUpdating = False

    'Process rows from last data row up to row 1
    For lRow = Cells(Rows.Count, "C").End(xlUp).Row To 1 Step -1

        'Delete rows with Mangoes in C column
        If Cells(lRow, "C").Value = "Mangoes" Then
            Cells(lRow, "C").EntireRow.Delete
        End If

    Next lRow
End Sub
```

A good programming principle to follow is this: If there is an Excel spreadsheet technique you can utilize, it is likely to be more efficient than a VBA emulation of the same technique, such as the `For . . . Next` loop used here.

Excel VBA programmers, especially when they do not have a strong background in the user interface features of Excel, often fall into the trap of writing VBA code to perform tasks that Excel can handle already. For example, you can write a VBA procedure to work through a sorted list of items, inserting rows with subtotals. You can also use VBA to execute the `Subtotal` method of the `Range` object. The second method is much easier to code, and it executes in a fraction of the time taken by the looping procedure.

Chapter 4: Using Ranges

It is much better to use VBA to harness the power built into Excel than to reinvent existing Excel functionality.

However, it isn't always obvious which Excel technique is the best one to employ. A fairly obvious Excel contender to locate the cells to be deleted, without having to examine every row using VBA code, is the Edit ⇨ Find command. The following code uses the Find method to reduce the number of cycles spent in VBA loops:

```
Sub DeleteRows2()  
    Dim rngFoundCell As Range  
  
    'Freeze screen  
    Application.ScreenUpdating = False  
  
    'Find a cell containing Mangoes  
    Set rngFoundCell = Range("C:C").Find(What:="Mangoes")  
  
    'Keep looping until no more cells found  
    Do Until rngFoundCell Is Nothing  
  
        'Delete found cell row  
        rngFoundCell.EntireRow.Delete  
  
        'Find next  
        Set rngFoundCell = Range("C:C").FindNext  
  
    Loop  
End Sub
```

This code is faster than the first procedure when there are not many rows to be deleted. As the percentage increases, the code becomes less efficient. Perhaps you need to look for a better Excel technique.

The fastest way to delete rows that I am aware of is provided by Excel's AutoFilter feature:

```
Sub DeleteRows3()  
    Dim lLastRow As Long          'Last row  
    Dim rng As Range  
    Dim rngDelete As Range  
  
    'Freeze screen  
    Application.ScreenUpdating = False  
  
    'Insert dummy row for dummy field name  
    Rows(1).Insert  
  
    'Insert dummy field name  
    Range("C1").Value = "Temp"  
  
    With ActiveSheet  
        'Reset Last Cell
```

```

        .UsedRange

        'Determine last row
        lLastRow = .Cells.SpecialCells(xlCellTypeLastCell).Row

        'Set rng to the C column data rows
        Set rng = Range("C1", Cells(lLastRow, "C"))

        'Filter the C column to show only the data to be deleted
        rng.AutoFilter Field:=1, Criteria1:="Mangoes"

        'Get reference to the visible cells, including dummy field name
        Set rngDelete = rng.SpecialCells(xlCellTypeVisible)

        'Turn off AutoFilter
        rng.AutoFilter

        'Delete rows
        rngDelete.EntireRow.Delete

        'Reset the last cell
        .UsedRange

    End With
End Sub

```

This is a bit more difficult to code, but it is significantly faster than the other methods, no matter how many rows are to be deleted. To use `AutoFilter`, you need to have field names at the top of your data. A dummy row is first inserted above the data, and a dummy field name is supplied for column C. The `AutoFilter` is only carried out on column C, which hides all the rows except those that have the text *Mangoes*.

The `SpecialCells` method is used to select only the visible cells in column C, which includes the dummy field name row. A reference to these rows is assigned to `rngDelete`. The `AutoFilter` is turned off and the rows in `rngDelete` are deleted.

Summary

This chapter has shown you the most important properties and methods that can be used to manage ranges of cells in a worksheet. The emphasis was on techniques that are difficult or impossible to discover using the macro recorder. The properties and methods discussed include the following:

- `Activate` method
- `Cells` property
- `Columns` and `Rows` properties
- `CurrentRegion` property
- `End` property

Chapter 4: Using Ranges

- `Offset` property
- `Range` property
- `Resize` property
- `Select` method
- `SpecialCells` method
- `Union` and `Intersect` methods

You also saw how to assign a worksheet range of values to a VBA array for efficient processing, and how to assign a VBA array of data to a worksheet range.

This chapter also emphasized that it is very rarely necessary to select cells or activate worksheets, which the macro recorder invariably does because it can only record what you do manually. Activating cells and worksheets is a very time-consuming process and should be avoided if you want your code to run at maximum speed.

The final examples showed that it is usually best to utilize Excel's existing capabilities, tapping into the Excel object model, rather than to write a VBA-coded equivalent. And bear in mind that some Excel techniques are better than others. Experimentation might be necessary to get the best code when speed is important.

Using Names

One of the most useful features in Excel is the ability to create names. You can create a name by selecting the Formulas tab on the Ribbon and clicking the Name Manager button to display the Name Manager dialog box, shown in Figure 5-1. If the name refers to a range, you can create it by selecting the range, typing the name into the Name box at the left side of the Formula bar, and pressing Enter. However, in Excel, names can refer to more than just ranges.

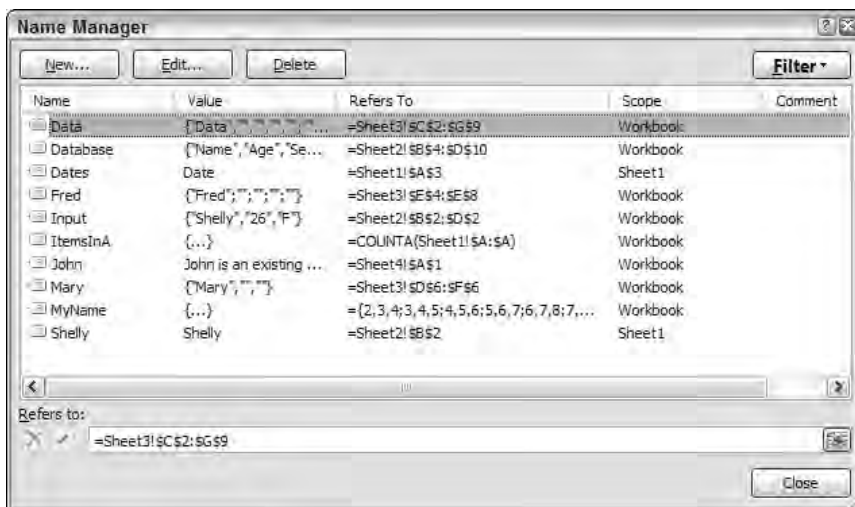


Figure 5-1

A name can contain a number, text, or a formula. Such a name has no visible location on the worksheet and can only be viewed in the Name Manager dialog box. Therefore, you can use names to store information in a workbook without having to place the data in a worksheet cell. Names can be declared hidden so they don't appear in the Name Manager dialog box. This can be a useful way to keep the stored information from being seen by users.

Chapter 5: Using Names

The normal use of names is to keep track of worksheet ranges. This is particularly useful for tables of data that vary in size. If you know that a certain name is used to define the range containing the data you want to work on, your VBA code can be much simpler than it might otherwise be. It is also relatively simple, given a few basic techniques, to change the definition of a name to allow for changes that you make to the tables in your code.

The Excel object model includes a `Names` collection and a `Name` object that can be used in VBA code. Names can be defined globally, at the workbook level, or they can be local, or worksheet-specific. The Name Manager dialog box indicates the level of a name under `Scope`. If you create local names, you can repeat the same name on more than one worksheet in the workbook. To make a `Name` object worksheet-specific, if you are entering it in the Name box, you precede its `Name` property with the name of the active worksheet and an exclamation mark. For example, you can type `Sheet1!Costs` to define a name `Costs` that is local to `Sheet1`, as shown in Figure 5-2.

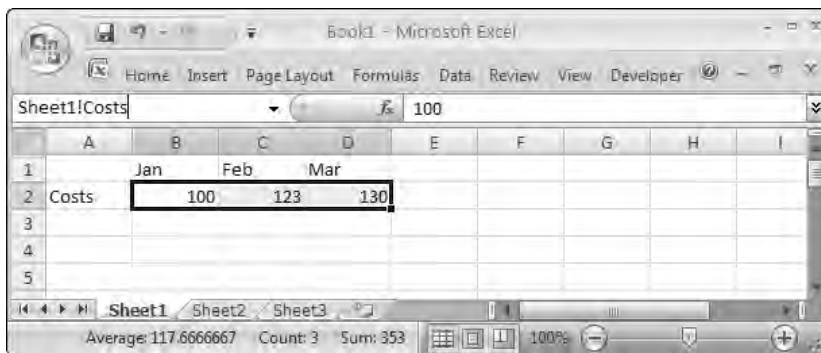


Figure 5-2

If you create the name using the `New` button in the Name Manager dialog box, you can select the scope of the name in the drop-down shown in Figure 5-3.

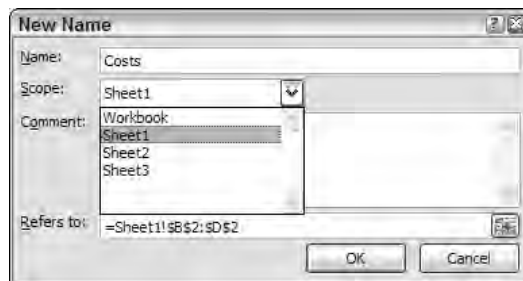


Figure 5-3

When you select a name using the Name box, you see the global names and those that are local to the active sheet. When you display the Name Manager dialog box, you see all the names in the workbook. The local names are identified by the worksheet name under `Scope`.

A great source of confusion with names is that they also have names. You need to distinguish between a Name object and the Name property of that object. The following code returns a reference to a Name object in the Names collection:

```
Names("Data")
```

If you want to change the Name property of a Name object, you use code like the following:

```
Names("Data").Name = "NewData"
```

Having changed its Name property, you would now refer to this Name object as follows:

```
Names("NewData")
```

Global names and local names belong to the Names collection associated with the Workbook object. If you use a reference such as Application.Names or Names, you are referring to the Names collection for the active workbook. If you use a reference such as Workbooks("Data.xls").Names, you are referring to the Names collection for that specific workbook.

Local names, but not global names, also belong to the Names collection associated with the Worksheet object to which they are local. If you use a reference such as Worksheets("Sheet1").Names or ActiveSheet.Names, you are referring to the local Names collection for that worksheet.

There is also another way to refer to names that refer to ranges. You can use the Name property of the Range object. More on this later.

Naming Ranges

You can create a global name that refers to a range using the Add method of the Workbook object's Names collection:

```
Names.Add Name:="Data", RefersTo:="=Sheet1!$D$10:$D$12"
```

It is important to include the equals sign in front of the definition and to make the cell references absolute, using the dollar sign (\$). Otherwise, the name will refer to an address relative to the cell address that was active when the name was defined. You can omit the worksheet reference if you want the name to refer to the active worksheet:

```
Names.Add Name:="Data", RefersTo:="=$D$10:$D$12"
```

If the name already exists, it will be replaced by the new definition.

If you want to create a local name, you can use the following:

```
Names.Add Name:="Sheet1!Sales", RefersTo:="=Sheet1!$E$10:$E$12"
```

Chapter 5: Using Names

Alternatively, you can add the name to the `Names` collection associated with the worksheet, which only includes the names that are local to that worksheet:

```
Worksheets("Sheet1").Names.Add Name:="Costs", RefersTo:="=Sheet1!$F$10:$F$12"
```

Using the Name Property of the Range Object

There is a much simpler way to create a name that refers to a `Range`. You can directly define the `Name` property of the `Range` object:

```
Range("A1:D10").Name = "SalesData"
```

If you want the name to be local, you can include a worksheet name:

```
Range("F1:F10").Name = "Sheet1!Staff"
```

It is generally easier, in code, to work with `Range` objects in this way than to have to generate the string address of a range, preceded by the equals sign that is required by the `RefersTo` parameter of the `Add` method of the `Names` collection. For example, if you created an object variable `rng` and want to apply the name `Data` to it, you need to get the `Address` property of `rng` and append it to an `=`:

```
Names.Add Name:="Data", RefersTo:="=" & rng.Address
```

The alternative method is:

```
rng.Name = "Data"
```

You cannot completely forget about the `Add` method, however, because it is the only way to create names that refer to numbers, formulas, and strings.

Special Names

Excel uses some names internally to track certain features. When you apply a print range to a worksheet, Excel gives that range the name `Print_Area` as a local name. If you set print titles, Excel creates the local name `Print_Titles`. If you select the `Data` tab on the Ribbon and click the `Advanced` button in the `Sort & Filter` chunk to extract data from a list to a new range, Excel creates the local names `Criteria` and `Extract`.

In older versions of Excel, the name `Database` was used to name the range containing your data list (or database). Although it is no longer mandatory to use this name, `Database` is still recognized by some Excel features such as `Advanced Filter`.

If you create a macro that uses the `ActiveSheet.ShowDataForm` method to edit your data list, you will find that the macro does not work if the data list does not start in `A1`. You can rectify this by applying the name `Database` to your data list.

You need to be aware that Excel uses these names, and in general, you should avoid using them unless you want the side effects they can produce. For example, you can remove the print area by deleting the name `Print_Area`. The following two lines of code have the same effect if you have defined a print area:

```
ActiveSheet.PageSetup.PrintArea = ""
ActiveSheet.Names("Print_Area").Delete
```

Excel 2007 also generates special names if you use the Table feature to manage a list of data. By default, Excel calls the tables `Table1`, `Table2`, and so on. These names appear in the Name Manager dialog box but are not included in the `Names` collection. They can't be deleted manually in the Name Manager or in code that references the `Names` collection. For more information on the Table feature, see Chapter 6.

To summarize, you need to take care when using the following names:

- `Criteria`
- `Database`
- `Extract`
- `Print_Area`
- `Print_Titles`
- `Tablen`

Storing Values in Names

The use of names to store data items has already been mentioned in Chapter 2, specifically under the `Evaluate` method topic. Now it's time to look at it in a bit more detail.

When you use a name to store numeric or string data, you should *not* precede the value of the `RefersTo` parameter with an equals sign (=). If you do, it will be taken as a formula. The following code stores a number and a string into `StoreNumber` and `StoreString`, respectively:

```
Dim v As Variant
v = 3.14159
Names.Add Name:="StoreNumber", RefersTo:=v
v = "Sales"
Names.Add Name:="StoreString", RefersTo:=v
```

This provides you with a convenient way to store the data you need in your VBA code from one Excel session to another, so that it does not disappear when you close Excel. When storing strings, you can store up to 255 characters.

You can retrieve the value in a name using the `Evaluate` method equivalent, as follows:

```
v = [StoreNumber]
```

Chapter 5: Using Names

You can also store formulas into names. The formula must start with an equals sign (=). The following places the `COUNTA` function into a name:

```
Names.Add Name:="ItemsInA", RefersTo:="=COUNTA($A:$A)"
```

This name can be used in worksheet cell formulas to return a count of the number of items in column A, as shown in Figure 5-4.

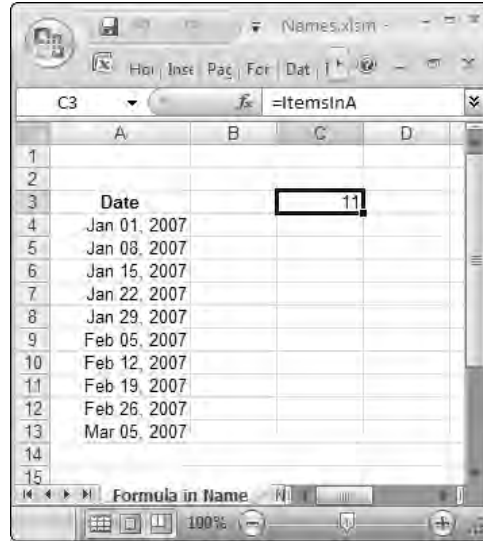


Figure 5-4

Once again, you can use the `Evaluate` method equivalent to evaluate the name in VBA:

```
MsgBox [ItemsInA]
```

Storing Arrays

You can store the values in an array variable in a name just as easily as you can store a number or a label. The following code creates an array of numbers in `aiArray` and stores the array values in `MyName`:

```
Sub ArrayToName()  
    Dim aiArray(1 To 200, 1 To 3) As Integer  
    Dim iRow As Integer  
    Dim iColumn As Integer  
  
    'Create array and store in name  
  
    'Create array
```

```

For iRow = 1 To 200

    For iColumn = 1 To 3

        aiArray(iRow, iColumn) = iRow + iColumn

    Next iColumn

Next iRow

'Store in name
Names.Add Name:="MyName", RefersTo:=aiArray

End Sub

```

There is a limit to the size of an array that can be assigned to a name in Excel 97 and Excel 2000. The maximum number of columns is 256 and the total number of elements in the array cannot exceed 5,461. In Excel 2002, 2003, and 2007, the size is only limited by memory.

The `Evaluate` method can be used to assign the values in a name that holds an array to a `Variant` variable. The following code assigns the contents of `MyName`, created in `ArrayToName`, to `vArray` and displays the last element in the array:

```

Sub NameToArray()
    Dim vArray As Variant

    'Assign contents of name to variant
    vArray = [MyName]

    'Display element of array
    MsgBox vArray(200, 3)

End Sub

```

The array created by assigning a name containing an array to a variant is always one-based, even if you have an `Option Base 0` statement in the declarations section of your module.

Hiding Names

You can hide a name by setting its `Visible` property to `False`. You can do this when you create the name:

```
Names.Add Name:="StoreNumber", RefersTo:=v, Visible:=False
```

You can also hide the name after it has been created:

```
Names("StoreNumber").Visible = False
```

Chapter 5: Using Names

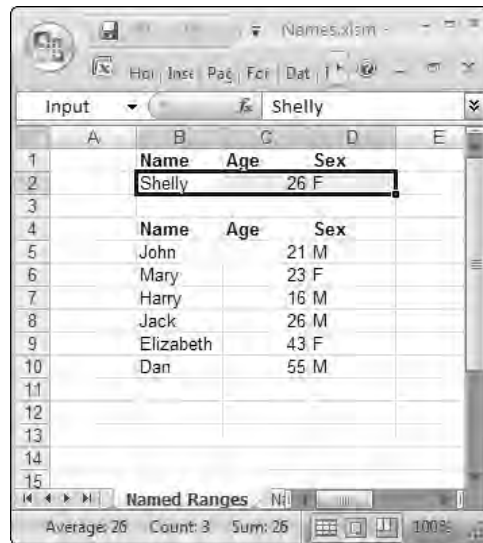
Now the name cannot be seen by users in the Name Manager dialog box. This is not a highly secure way to conceal information, because anyone with VBA skills can detect the name. But it is an effective way to ensure that users are not confused by the presence of strange names.

You should also be aware that if, through the Excel user interface, a user creates a Name object with a Name property corresponding to your hidden name, the hidden name is destroyed. You can prevent this by protecting the worksheet.

Despite some limitations, hidden names do provide a nice way to store information in a workbook.

Working with Named Ranges

The spreadsheet in Figure 5-5 contains a data list in B4:D10 that has been given the name `Database`. There is also a data input area in B2:D2 that has been given the name `Input`.



The screenshot shows an Excel spreadsheet with a named range 'Database' and an input area 'Input'. The 'Input' area is located in cells B2:D2 and contains the text 'Shelly'. The 'Database' area is located in cells B4:D10 and contains a list of names, ages, and sexes. The spreadsheet also shows a 'Named Ranges' task pane at the bottom with the following statistics: Average: 26, Count: 3, Sum: 26.

	A	B	C	D	E
1		Name	Age	Sex	
2		Shelly	26	F	
3					
4		Name	Age	Sex	
5		John	21	M	
6		Mary	23	F	
7		Harry	16	M	
8		Jack	26	M	
9		Elizabeth	43	F	
10		Dan	55	M	
11					
12					
13					
14					
15					

Figure 5-5

If you want to copy the `Input` data to the bottom of the data list and increase the range referred to by the name `Database` to include the new row, you can use the following code:

```
Sub AddNewData()  
    Dim lRows As Long  
  
    'Copy data & Extend range of Database by one row  
  
    With Range("Database")  
        lRows = .Rows.Count + 1
```

```

    Range("Input").Copy Destination:=.Cells(1Rows, 1)
    .Resize(1Rows).Name = "Database"
End With

End Sub

```

The output resulting from this code will be the same as Figure 5-5, but with the data Shelley 26 F in cells B11:D11. The range `Database` will now refer to B4:D11.

The variable `1Rows` is assigned the count of the number of rows in `Database` plus 1, to allow for the new record of data. `Input` is then copied. The destination of the copy is the B11 cell, which is defined by the `Cells` property of `Database`, being `1Rows` down from the top of `Database` in column 1 of `Database`. The `Resize` property is applied to `Database` to generate a reference to a `Range` object with one more row than `Database`, and the `Name` property of the new `Range` object is assigned the name `Database`.

The nice thing about this code is that it is quite independent of the size or location of `Database` in the active workbook and the location of `Input` in the active workbook. `Database` can have seven rows or 7,000 rows. You can add more columns to `Input` and `Database` and the code still works without change. `Input` and `Database` can even be on different worksheets and the code will still work.

Searching for a Name

If you want to test to see if a name exists in a workbook, you can use the following function. It has been designed to work both as a worksheet function and as a VBA callable function, which makes it a little more complex than if it were designed for either job alone:

```

Function IsNameInWorkbook(sName As String) As Boolean
    Dim s As String
    Dim rng As Range

    'See if name exists in workbook

    'Force recalculation if used as worksheet function
    Application.Volatile

    'Ignore errors
    On Error Resume Next

    'Try to get reference to cell using function
    Set rng = Application.Caller
    Err.Clear

    If rng Is Nothing Then
        'Function was called by VBA code
        s = ActiveWorkbook.Names(sName).Name
    Else
        'Function was called by cell
        s = rng.Parent.Parent.Names(sName).Name
    End If

    IsNameInWorkbook = (s <> "")
End Function

```

Chapter 5: Using Names

```
End If

'If no error, name exists
If Err.Number = 0 Then IsNameInWorkbook = True
End Function
```

`IsNameInWorkbook` has an input parameter `sName`, which is the required name as a string. The function has been declared volatile, so it recalculates when it is used as a worksheet function and the referenced name is added or deleted. The function first determines if it has been called from a worksheet cell by assigning the `Application.Caller` property to `rng`.

If it has been called from a cell, `Application.Caller` returns a `Range` object that refers to the cell containing the function. If the function has not been called from a cell, the `Set` statement causes an error, which is suppressed by the preceding `On Error Resume Next` statement. That error, should it have occurred, is cleared because the function anticipates further errors that should not be masked by the first error.

Next, the function uses an `If` test to see if `rng` is undefined. If so, the call was made from another VBA routine. In this case, the function attempts to assign the `Name` property of the `Name` object in the active workbook to the dummy variable `s`. If the name exists, this attempt succeeds and no error is generated. Otherwise an error does occur, but is once again suppressed by the `On Error Resume Next` statement.

If the function has been called from a worksheet cell, the `Else` clause of the `If` test identifies the workbook containing `rng` and attempts to assign the `Name` property of the required `Name` object to `s`. The parent of `rng` is the worksheet containing `rng`, and the parent of that worksheet is the workbook containing `rng`. Once again, an error will be generated if the name does not exist in the workbook.

Finally, `IsNameInWorkbook` checks the `Number` property of the `Err` object to see if it is zero. If it is, the return value of the function is set to `True` because the name does exist. If there is a non-zero error number, the function is left to return its default value of `False` because the name does not exist.

You could use `IsNameInWorkbook` in a spreadsheet cell as follows:

```
=IF(IsNameInWorkbook("John"),"John is ","John is not ")&"an existing name"
```

You could use the following procedure to ask the user to enter a name and determine its existence:

```
Sub TestName()

    If IsNameInWorkbook(InputBox("What Name")) Then
        MsgBox "Name exists"
    Else
        MsgBox "Name does not exist"
    End If

End Sub
```

Note that if you are searching for a local name, you must include its sheet name, in the form `Sheet1!Name`, in the previous examples.

If you invoke `IsNameInWorkbook` as a worksheet function *and* if the name John is present, you will get output like that shown in Figure 5-6.

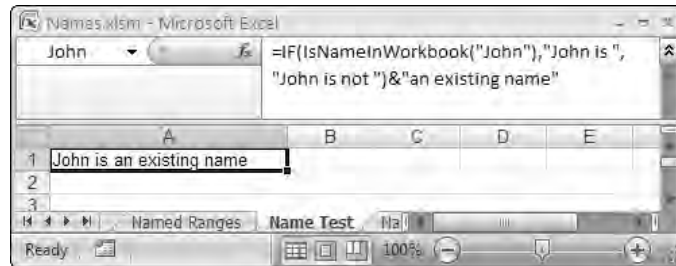


Figure 5-6

Searching for the Name of a Range

The `Name` property of the `Range` object returns the name of the range, if the range has a name and the `RefersTo` property of the `Name` object corresponds exactly to the range.

You might be tempted to display the name of a range `rng` with the following code:

```
MsgBox rng.Name
```

This code fails because the `Name` property of a `Range` object returns a `Name` object. The code will display the default property value of the `Name` object, which is its `RefersTo` property. What you want is the `Name` property of the `Name` object, so you must use:

```
MsgBox rng.Name.Name
```

This code only works if `rng` has a name. It will return a run-time error if `rng` does not have one. You can use the following code to display the names of the selected cells in the active sheet:

```
Sub TestNameOfRange()
    Dim nmName As Name

    'See if range has a name

    'Ignore errors
    On Error Resume Next

    'Try to get name
    Set nmName = Selection.Name

    'Display result
    If nmName Is Nothing Then
        MsgBox " Selection has no name"
    Else
        MsgBox nmName.Name
    End If
End Sub
```

Chapter 5: Using Names

If a range has more than one name, the first of the names, in alphabetical order, will be returned.

When this macro is run, the output will look something like Figure 5-7.

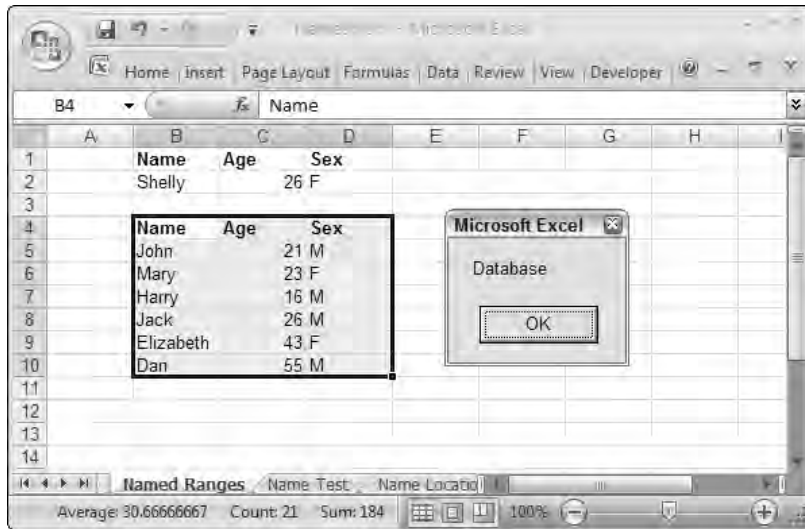


Figure 5-7

Determining which Names Overlap a Range

When you want to check the names that have been applied to ranges in a worksheet, it can be handy to get a list of all the names that are associated with the currently selected cells. You might be interested in names that completely overlap the selected cells, or names that partly overlap the selected cells. The following code lists all the names that completely overlap the selected cells of the active worksheet:

```
Sub SelectionEntirelyInNames()  
    Dim sMessage As String  
    Dim nmName As Name  
    Dim rngNameRange As Range  
    Dim rng As Range  
  
    'List all names that entirely contain  
    'the selected cells  
  
    'Ignore errors  
    On Error Resume Next  
  
    'Look at all names in workbook  
    For Each nmName In Names  
  
        'Start with nothing & try to assign range  
        Set rngNameRange = Nothing
```

```

Set rngNameRange = nmName.RefersToRange

'If successful, we have a range reference
If Not rngNameRange Is Nothing Then

    'See if range is in active sheet
    If rngNameRange.Parent.Name = ActiveSheet.Name Then

        'See if selection is in range
        Set rng = Intersect(Selection, rngNameRange)
        If Not rng Is Nothing Then

            'See if range is entirely in selection
            If Selection.Address = rng.Address Then
                sMessage = sMessage & nmName.Name & vbCr
            End If

        End If

    End If

End If

End If

Next nmName

'Displaymessage
If sMessage = "" Then
    MsgBox "The selection is not entirely in any name"
Else
    MsgBox sMessage
End If
End Sub

```

`SelectionEntirelyInNames` starts by suppressing errors with `On Error Resume Next`. It then goes into a `For Each...Next` loop that processes all the names in the workbook. It sets `rngNameRange` to `Nothing` to get rid of any range reference left in it from one iteration of the loop to the next. It then tries to use the `Name` property of the current `Name` object as the name of a `Range` object and assign a reference to the `Range` object to `rngNameRange`. This will fail if the name does not refer to a range, so the rest of the loop is only carried out if a valid `Range` object has been assigned to `rngNameRange`.

The next `If` test checks that the range that `rngNameRange` refers to is on the active worksheet. The parent of `rngNameRange` is the worksheet containing `rngNameRange`. The inner code is only executed if the name of the parent worksheet is the same as the name of the active sheet. `rng` is then assigned to the intersection (the overlapping cells) of the selected cells and `rngNameRange`. If there is an overlap and `rng` is not `Nothing`, the innermost `If` is executed.

This final `If` checks that the overlapping range in `rng` is identical to the selected range. If this is the case, then the selected cells are contained entirely in `rngNameRange` and the `Name` property of the current `Name` object is added to any names already in `sMessage`. In addition, a carriage return character is appended, using the VBA intrinsic constant `vbCr`, so each name is on a new line in `sMessage`.

Chapter 5: Using Names

When the `For Each . . . Next` loop terminates, the following `If` tests to see if there is anything in `sMessage`. If `sMessage` is a zero-length string, `MsgBox` displays an appropriate message to say that no names were found. Otherwise, the list of found names in `sMessage` is displayed.

When this code is run in a spreadsheet with three named ranges — `Data`, `Fred`, and `Mary` — you get the result shown in Figure 5-8 upon running `SelectionEntirelyInNames`.

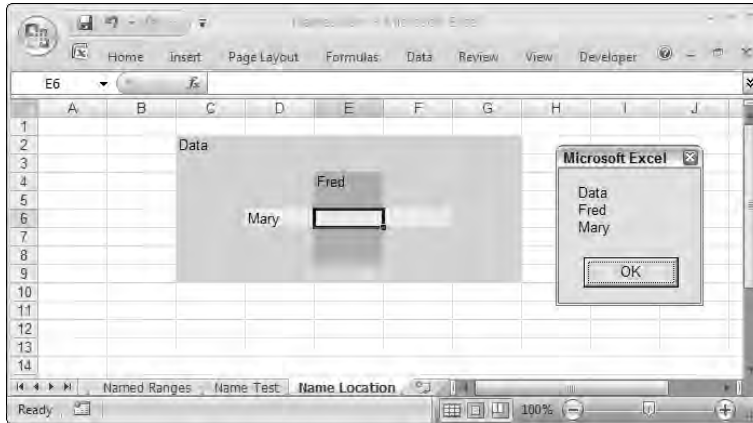


Figure 5-8

If you want to find out which names are overlapping the selected cells, regardless of whether they entirely contain the selected cells, you can remove the second innermost `If` test, as in the following code:

```
Sub NamesOverlappingSelection()  
    Dim sMessage As String  
    Dim nmName As Name  
    Dim rngNameRange As Range  
    Dim rng As Range  
  
    'List all names that overlap  
    'the selected cells  
  
    'Ignore errors  
    On Error Resume Next  
  
    'Look at all names in workbook  
    For Each nmName In Names  
  
        'Start with nothing & try to assign range  
        Set rngNameRange = Nothing  
        Set rngNameRange = Range(nmName.Name)  
  
        'If successful, we have a range reference  
        If Not rngNameRange Is Nothing Then  
  
            'See if range is in active sheet
```

```

    If rngNameRange.Parent.Name = ActiveSheet.Name Then

        'See if selection overlaps range
        Set rng = Intersect(Selection, rngNameRange)

        If Not rng Is Nothing Then

            sMessage = sMessage & nmName.Name & vbCr

        End If

    End If

End If

Next nmName

'Displaymessage
If sMessage = "" Then
    MsgBox "The selection is not entirely in any name"
Else
    MsgBox sMessage
End If

End Sub

```

Note that `SelectionEntirelyInNames` and `NamesOverlappingSelection` use different techniques to assign the range referred to by the name to the object variable `rngNameRange`. The following statements are equivalent:

```

Set rngNameRange = nmName.RefersToRange
Set rngNameRange = Range(nmName.Name)

```

Summary

This chapter has presented an in-depth discussion of using names in Excel VBA. You have seen how to do the following:

- ❑ Use names to keep track of worksheet ranges
- ❑ Use names to store numeric and string data in a worksheet
- ❑ Hide names from the user if necessary
- ❑ Check for the presence of names in workbooks and in ranges
- ❑ Determine which names completely or partially overlap a selected range

Data Lists

This chapter shows you how to set up VBA code to manage data in lists, and code to filter and sort information in lists. The features examined are:

- Sorting
- Tables (called Lists in Excel 2003)
- AutoFilter
- Advanced Filter
- Data Forms

As always, you can use the macro recorder to generate some basic code for these operations. However, the recorded code needs modification to make it useful, and the recorder can even generate erroneous code in some cases. You will see that dates can be a problem, if not handled properly, especially in an international setting.

You will also see that there is more than one way to perform some tasks. Because Excel has introduced new objects that manage, filter, and sort data, there has been some duplication of the features of older objects. This can be confusing, but it does give you a wide choice of options that you can tailor to fit your needs.

Structuring the Data

Before you can apply Excel's list management tools, your data must be set up in a very specific way. The data must be structured like a database table, with headings at the top of each column, which are the field names, and the data itself must consist of single rows of information, which are the equivalent of database records. The top row holding the field names is called the header record. Figure 6-1 shows a list that holds information on students.



The screenshot shows an Excel spreadsheet with a data list. The columns are labeled A through F, and the rows are numbered 1 through 11. The data is as follows:

	A	B	C	D	E	F
1						
2						
3		Name	Age	Sex		
4		Glenda	18	F		
5		John	21	M		
6		Helena	22	F		
7		Henri	19	M		
8		Jack	18	M		
9		Mary	16	F		
10						
11						

Figure 6-1

Excel should never be considered a fully equipped database application. It is limited in the amount of data it can handle, and it cannot efficiently handle multiple related database tables. However, Excel can work with other database applications to provide you with the data you need, and it has some powerful tools, such as Data Form, AutoFilter, Advanced Filter, SubTotal, and PivotTables, for analyzing, manipulating, and presenting that data. Learn about PivotTables in Chapter 7. See how to connect to external data sources in Chapters 20 and 21.

Sorting a Range

To sort the data displayed in Figure 6-1 by Sex, turn on the macro recorder and select D3, as shown in Figure 6-2. Select the Data tab in the Ribbon and click the AZ button in the top-left corner of the Sort & Filter group.

You will record code similar to the following:

```
Range("D3").Select
ActiveWorkbook.Worksheets("Sheet1").Sort.SortFields.Clear
ActiveWorkbook.Worksheets("Sheet1").Sort.SortFields.Add Key:=Range("D3"), _
    SortOn:=SortOnValues, Order:=xlAscending, DataOption:=xlSortNormal
With ActiveWorkbook.Worksheets("Sheet1").Sort
    .SetRange Range("B4:D9")
    .Header = xlNo
    .MatchCase = False
    .Orientation = xlTopToBottom
    .SortMethod = xlPinYin
    .Apply
End With
```

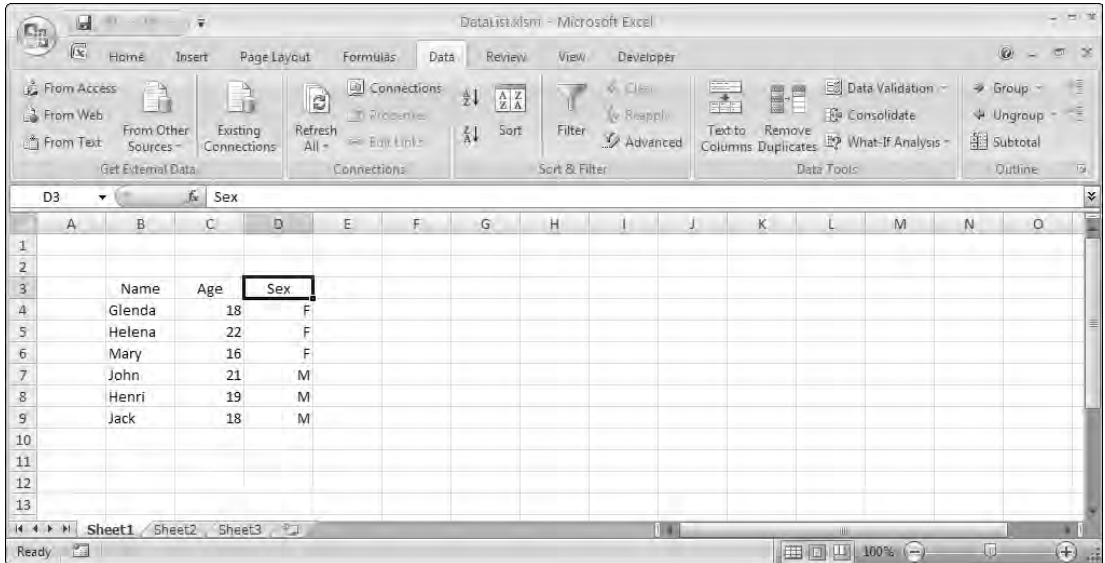



Figure 6-2

This code uses the `Sort` object introduced in Excel 2007. The `Sort` object's parent can be a `Worksheet` object, as here, or it can be the `AutoFilter` object or the `ListObject` object. You will take a look at the last two shortly. The `QueryTable` object can also be the parent of the `Sort` object. See Chapter 21 for more details.

The `Sort` object contains a `SortFields` collection, where `SortField` objects can be added to define as many fields as you need to carry out your sort. In the recorded code, the `SortFields` collection is cleared before the single new `SortField` is added that specifies the D column as the column to be sorted. The `SetRange` method is used to specify the data range, which does not include the header fields, and the `Apply` method executes the sort. The `SortMethod` property is only applicable to Asian languages and can be omitted if you are not using an Asian language. The code can be used pretty much as it is, with the exception of the selection of D3, which is unnecessary. You could tidy it up as follows:

```
With ActiveWorkbook.Worksheets("Sheet1").Sort
    .SortFields.Clear
    .SortFields.Add Key:=Range("D3"), _
                   SortOn:=SortOnValues, _
                   Order:=xlAscending, _
                   DataOption:=xlSortNormal
    .SetRange Range("B4:D9")
    .Header = xlNo
    .MatchCase = False
    .Orientation = xlTopToBottom
    .SortMethod = xlPinYin
    .Apply
End With
```

To specify more sort keys, you use the `Add` method of the `SortFields` collection as many times as necessary. The keys need to be added in the order of their significance.

Older Excel Versions

If you record the same sort in earlier versions of Excel, you get code like the following:

```
Range("B3:D9").Sort Key1:=Range("D3"), Order1:=xlAscending, Header:= _  
xlGuess, OrderCustom:=1, MatchCase:=False, Orientation:=xlTopToBottom, _  
DataOption1:=xlSortNormal
```

This code uses the `Sort` method of the `Range` object. The code is much simpler than the code generated by the `Sort` object but is also more limited.

You can define as many sort keys as you need by adding `DataField` objects to the `DataFields` collection of the `Sort` object. The `Range` object `Sort` method is limited to three keys, all specified in a single execution of the `Sort` method. This limitation can be overcome by performing a series of sorts in the reverse order of the significance of the keys.

Naturally, this code is still supported in Excel 2007 and will be in the future. If you only need to sort on three or fewer keys, it gives you a simpler alternative to the `Sort` object.

Creating a Table

The data in Figure 6-1 can be easily converted to a Table. Select a cell in the data and select the `Insert` tab of the Ribbon. Click the `Table` button in the `Tables` group to show the dialog box in Figure 6-3.

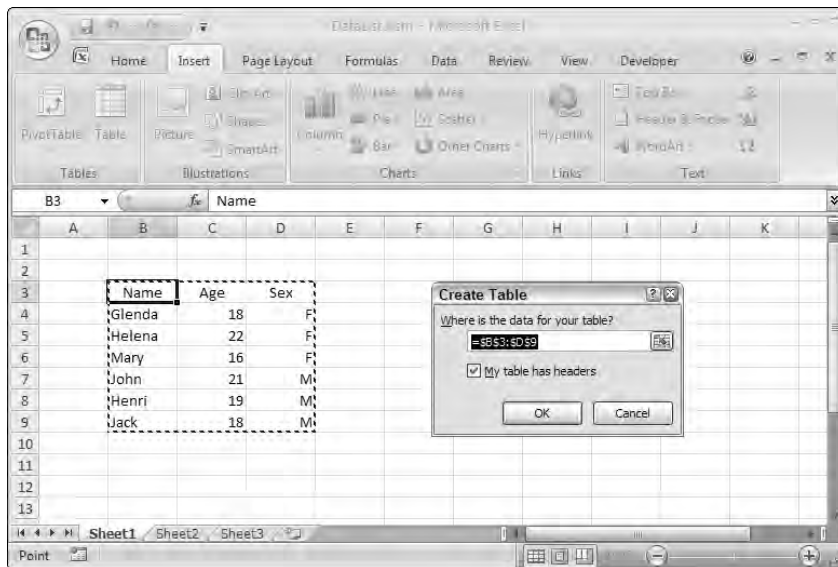


Figure 6-3

The resulting table is shown in Figure 6-4. It has been assigned the name `Table1` by default. You can change this to something more meaningful in the `Properties` group on the `Table Tools Design` tab of the Ribbon.

If you record the creation of the table, you will get code like the following:

```
ActiveSheet.ListObjects.Add(xlSrcRange, Range("$B$3:$D$9"), , xlYes).Name = _
    "Table1"
```

Converting a range to a table creates a `ListObject` object. This is the same object introduced in Excel 2003, where the table is called a list. Excel 2007 supports a number of new properties for the `ListObject` object, mainly concerned with the table's appearance.

Tables provide you with a way of formally identifying a data structure, and Excel provides tools and intelligence to help manage the data and its formatting. Tables also allow you to link to external data. See Chapter 21 for information on using `ListObject` objects to link to external data.

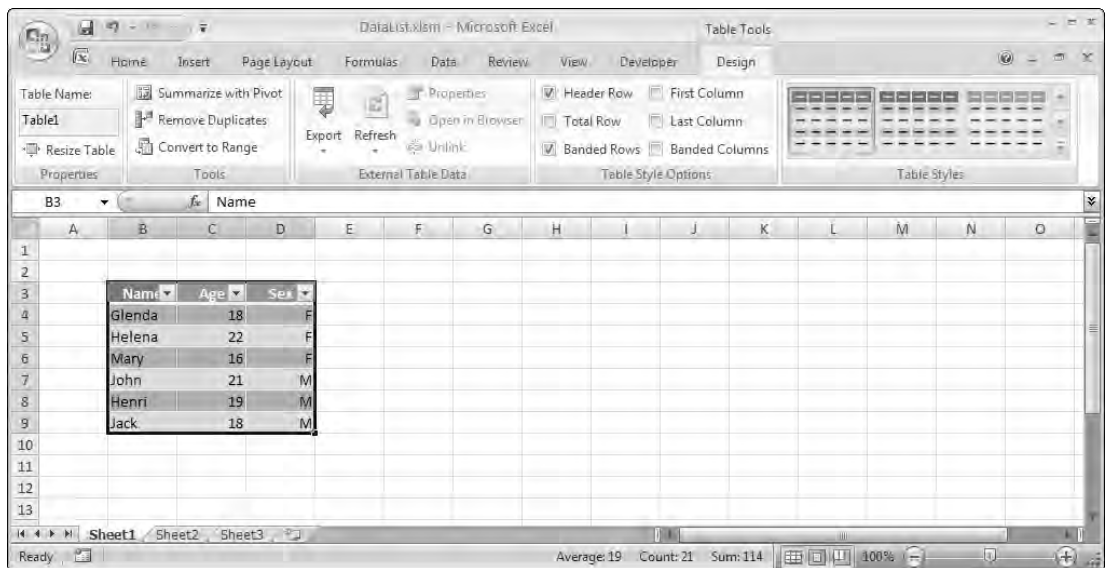


Figure 6-4

Sorting a Table

You can sort the data in the table in the same way as you sort a range, or by clicking one of the drop-downs beside the field names and selecting one of the sort options. If you record a sort based on the Name field using the drop-down beside the field name, you will get code like the following:

```
ActiveWorkbook.Worksheets("Sheet1").ListObjects("Table1").Sort.SortFields.Clear
ActiveWorkbook.Worksheets("Sheet1").ListObjects("Table1").Sort.SortFields.Add _
    Key:=Range("Table1[ [#All], [Name]]"), SortOn:=SortOnValues, Order:= _
    xlAscending, DataOption:=xlSortNormal
With ActiveWorkbook.Worksheets("Sheet1").ListObjects("Table1").Sort
    .SetRange Range("Table1 [#All]")
    .Header = xlYes
```

Chapter 6: Data Lists

```
.MatchCase = False  
.Orientation = xlTopToBottom  
.SortMethod = xlPinYin  
.Apply  
End With
```

The code for sorting a table is similar to the code for sorting a range. Instead of the `Worksheet` object being the parent of the `Sort` object, the `ListObject` object is the parent. Notice also that there are new ways to reference data ranges. The `SetRange` method uses `Range("Table1[#All]")` to specify all of the data in the table. The `SortField` object key is specified using `Range("Table1[#All],[Name]")`.

AutoFilter

The AutoFilter feature is a very easy way to select data from a list. As you might expect, AutoFilter works with tables or with any list of data. You can activate AutoFilter by selecting a cell in your data, selecting the Data tab on the Ribbon, and clicking the Filter button in the Sort & Filter group. Drop-down menu buttons will appear beside each field name, as shown in Figure 6-5. If you want an exact match on a field such as `Customer`, all you need to do is click the drop-down beside the field and check the required match, as shown in Figure 6-5.

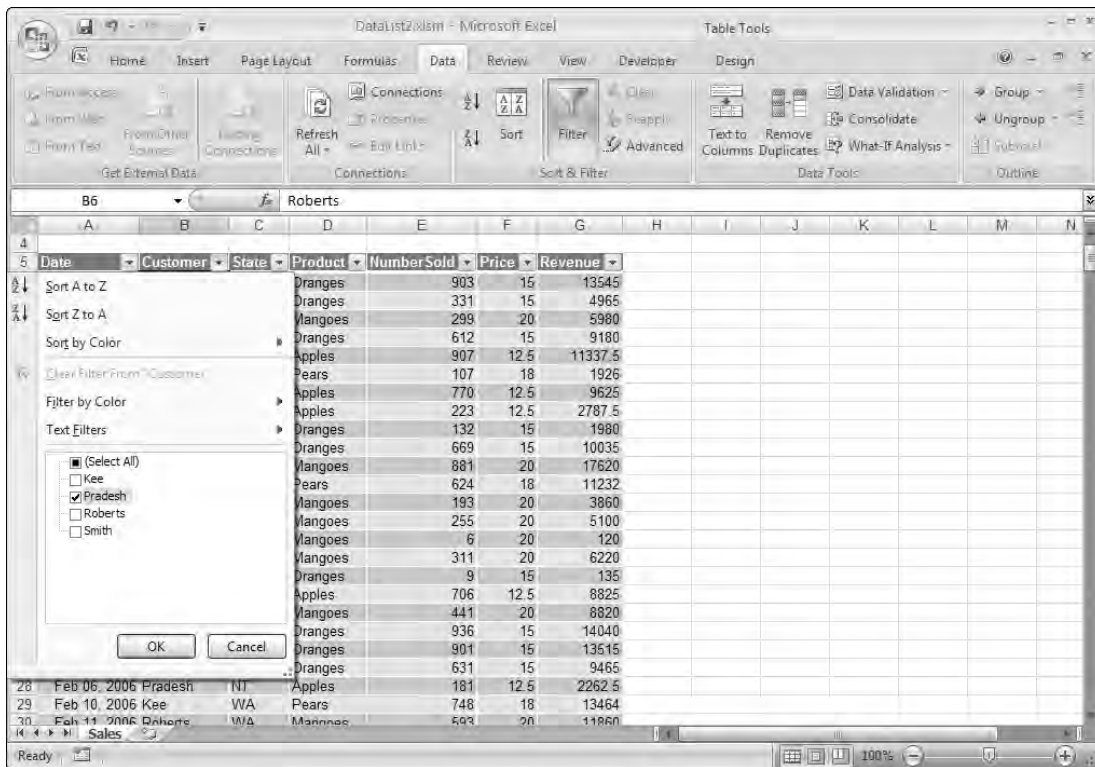


Figure 6-5

If you record this process with a normal range of data, you will get code like the following:

```
ActiveSheet.Range("$A$5:$G$433").AutoFilter Field:=2, Criteria1:="Pradesh"
```

If you record the same process in a table, you will get code like the following:

```
ActiveSheet.ListObjects("Table2").Range.AutoFilter Field:=2, Criteria1:= _
"Pradesh"
```

If you select two items, you will get code like the following:

```
ActiveSheet.ListObjects("Table2").Range.AutoFilter Field:=2, Criteria1:= _
"=Kee", Operator:=xlOr, Criteria2:="=Pradesh"
```

If you select more than two items, you will get code like the following:

```
ActiveSheet.ListObjects("Table2").Range.AutoFilter Field:=2, Criteria1:= _
Array("Kee", "Pradesh", "Roberts"), Operator:=xlFilterValues
```

The following code clears the filter and displays all the data for the field:

```
ActiveSheet.ListObjects("Table2").Range.AutoFilter Field:=2
```

AutoFilter Object

When you AutoFilter a range that is not in a table, Excel uses an `AutoFilter` object whose parent is the `Worksheet` object. There can only be one `AutoFilter` object for each worksheet. If you AutoFilter a second range in a worksheet, all the settings for the first `AutoFilter` are lost.

When you AutoFilter a table, the parent of the `AutoFilter` object is the `ListObject` object. Because you can have multiple tables in a worksheet, it is possible to have an `AutoFilter` operating simultaneously in each table.

You don't create or manipulate an `AutoFilter` object directly. You use the `AutoFilter` method of the `Range` object. If you examine the previous code, you will see that with a table, the `Range` property of the `ListObject` object is used to reference the range associated with the table and the `AutoFilter` method of that `Range` object is executed.

You can use the `AutoFilter` object to obtain information about an existing `AutoFilter`. However, you will get an error if you reference the `AutoFilter` object if it is not in use. The `AutoFilter` object only exists when the `AutoFilter` feature is turned on. You can determine whether the `Worksheet` `AutoFilter` is active by using the value of the `AutoFilterMode` property, which returns a Boolean value:

```
If ActiveSheet.AutoFilterMode Then
```

This is a read-only property that can't be used to switch on the `AutoFilter`. As you have seen, you do that with the `AutoFilter` method of the `Range` object. To switch off a range `AutoFilter`, you use the `AutoFilter` method of the `Range` object with no parameters:

```
Range("B3:D9").AutoFilter
```

Chapter 6: Data Lists

This code acts as a toggle. It switches `AutoFilter` on if it is off, and off if it is on. If you want to ensure that the worksheet `AutoFilter` is turned off, you can use code like the following:

```
If ActiveSheet.AutoFilterMode Then
    ActiveSheet.AutoFilter.Range.AutoFilter
End If
```

You can determine whether a `ListObject` object `AutoFilter` is turned on or off by testing its `ShowAutoFilter` property. This is a read/write property that can also be used to explicitly turn the table's `AutoFilter` on or off. You can ensure that the `AutoFilter` is switched off with the following code:

```
ActiveSheet.ListObjects("Table1").ShowAutoFilter = False
```

Filter Object

There is a `Filters` collection associated with the `AutoFilter` object that holds a `Filter` object for each field in the `AutoFilter`. The `On` property of the `Filter` object indicates whether it is active. If it is active, you can discover the values of its properties. The following code returns the value `"=Pradesh"` after you have set the filter as shown in Figure 6-5, after converting the data to a table:

```
With ActiveSheet.ListObjects(1)
    If .ShowAutoFilter Then
        With .AutoFilter.Filters(2)
            If .On Then
                MsgBox .Criteria1
            End If
        End With
    End If
End With
```

The `Filter` object can only return the properties of a filter. You can't assign values to the properties. That can only be done using the `AutoFilter` method of the `Range` object.

Date Custom Filter

If you want something a bit more complex, such as a range of dates, you need to do a bit more work. The following screen shows how you can manually filter the data in a table to show a particular month. Click the drop-down button beside `Date` and choose `Date Filters`. Then select either `Custom Filter` or `Between`. You can then fill in the dialog box as shown in Figure 6-6.

The format you use when you type in dates in the Custom `AutoFilter` dialog box depends on your regional settings. You can use a `dd/mm/yy` format if you work with UK settings, or a `mm/dd/yy` format if you work with U.S. settings. Some formats that are more international, such `yyyy-m-d`, are also recognized. You can use the calendar controls in the `AutoFilter` dialog box to insert your dates in your regional date format.

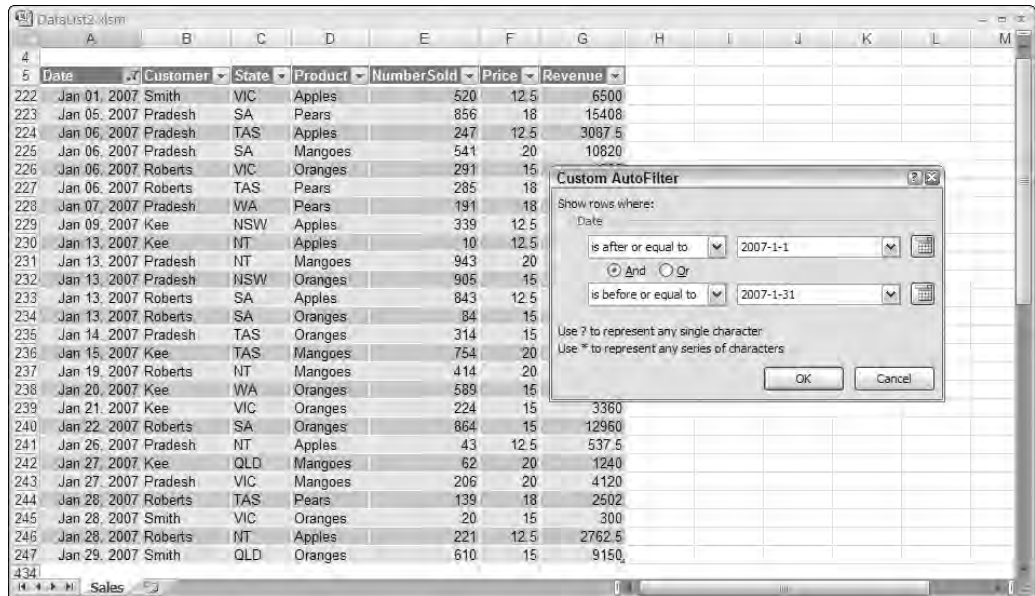


Figure 6-6

Adding Combo Boxes

You can make filtering even easier for a user by placing controls in the worksheet to run AutoFilter. This also gives you the opportunity to do far more with the data than filter it. You could copy the filtered data to another worksheet and generate a report, you could chart the data, or you could delete it. Figure 6-7 shows two ActiveX combo box controls that allow the user to select the month and year required.

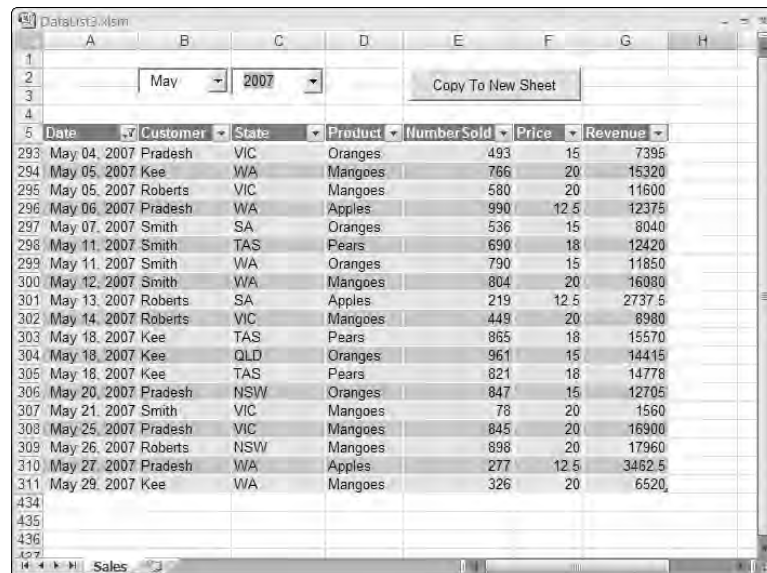


Figure 6-7

Chapter 6: Data Lists

The combo boxes have the default names of `ComboBox1` and `ComboBox2`. To place list values into the combo boxes, you can enter the list values into a worksheet column and define the `ListFillRange` property of the `ComboBox` object as something like `"=Sheet2!A1:A12"`. Alternatively, you can use the following `Workbook_Open` event procedure in the `ThisWorkbook` module of the workbook to populate the combo boxes when the workbook is opened:

```
Private Sub Workbook_Open()  
    Dim vMonths As Variant  
    Dim vYears As Variant  
    Dim i As Integer  
  
    'Create date arrays  
    vMonths = Array("Jan", "Feb", "Mar", "Apr", "May", "Jun", _  
                   "Jul", "Aug", "Sep", "Oct", "Nov", "Dec")  
    vYears = Array(2006, 2007)  
  
    'Populate months using AddItem method  
    For i = LBound(vMonths) To UBound(vMonths)  
        Sheet1.ComboBox1.AddItem vMonths(i)  
    Next i  
  
    'Populate years using List property  
    Sheet1.ComboBox2.List = WorksheetFunction.Transpose(vYears)  
End Sub
```

The `AddItem` method of the `ComboBox` object adds the `Months` array values to the `ComboBox1` list. To show an alternative technique, the worksheet `Transpose` function is used to convert the `Years` array from a row to a column, and the values are assigned to the `List` property of `ComboBox2`.

Note that the programmatic name of `Sheet1`, which you can see in the Project Explorer window or the Properties window of the VBE, has been used to define the location of the combo boxes. Even though the name of the worksheet is `Sales`, the programmatic name is still `Sheet1`, unless you change it at the top of the Properties window where it is identified by (Name), rather than `Name`, as shown in Figure 6-8.

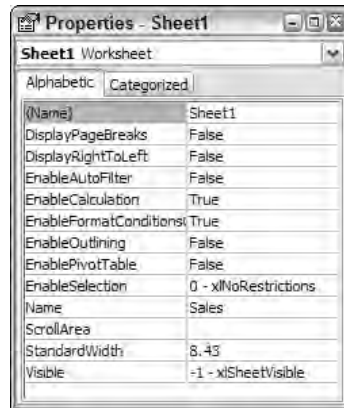


Figure 6-8

In the code module behind the worksheet, the following code is entered:

```
Private Sub ComboBox1_Click()
    If ComboBox2.Value="" Then Exit Sub
    Call FilterDates
End Sub

Private Sub ComboBox2_Click()
    If ComboBox1.Value="" Then Exit Sub
    Call FilterDates
End Sub
```

As long as the other combo box has been assigned a value, when you click an entry in their drop-down lists, each combo box executes the `FilterDates` procedure, which is described next. `FilterDates` can be in the same module and declared `Private` if you do not want any other modules to be able to use it, or it can be in a standard code module if you want to use it as a general utility procedure.

So, how do you construct the `FilterDates` procedure? As shown in previous chapters, you can use the macro recorder to get something to start with, and then refine the code to make it more flexible and efficient. If you use the macro recorder to record the process of filtering the dates, you will get code like this:

```
ActiveSheet.ListObjects("Table1").Range.AutoFilter Field:=1, Criteria1:= _
">=1/01/2007", Operator:=xlAnd, Criteria2:="<=31/01/2007"
```

You might notice that the dates have been translated to the format of the regional settings — in this case that of Australia, which uses the same format as the UK. The format generated by the recorder is `d/mm/yyyy`. Also note that the dates are formatted as text, rather than dates, because the criteria must include the logical operators.

Date Format Problems

Unfortunately, the previous code does not perform as expected. When you run the recorded macro, the dates are interpreted by VBA as U.S. dates in the format `mm/dd/yyyy`. `Criteria2` is not understood by VBA. To make this macro perform properly, you need to convert the dates to a U.S. format. Of course, you will not have this problem with your recorded code if you work with U.S. date formats in your regional settings in the first place.

Trying to make your VBA code compatible with dates in all language versions of Excel is very difficult. See Chapter 25 for more details.

The following `FilterDates` procedure is executed from the `Click` event procedures of the combo boxes, and it computes the start and end dates required for the criteria of the `AutoFilter` method. `FilterDates` has been placed in the same module as the combo box event procedures and declared as `Private`, so it does not appear in the Macro dialog box:

```
Private Sub FilterDates()
    Dim iStartMonth As Integer
    Dim iStartYear As Integer
    Dim dteStartDate As Date
```

```
Dim dteEndDate As Date
Dim sStartCriterion As String
Dim sEndCriterion As String

'Get Date values
iStartMonth = Me.ComboBox1.ListIndex + 1
iStartYear = Me.ComboBox2.Value

'Calculate date values and format as US Dates
dteStartDate = DateSerial(iStartYear, iStartMonth, 1)
dteEndDate = DateSerial(iStartYear, iStartMonth + 1, 1)
sStartCriterion = ">=" & Format(dteStartDate, "mm/dd/yyyy")
sEndCriterion = "<" & Format(dteEndDate, "mm/dd/yyyy")

'Apply AutoFilter
Me.ListObjects("Table1").Range.AutoFilter _
    Field:=1, _
    Criteria1:=sStartCriterion, _
    Operator:=xlAnd, _
    Criteria2:=sEndCriterion

End Sub
```

`FilterDates` assigns the values selected in the combo boxes to `iStartMonth` and `iStartYear`. The `Me` keyword has been used to refer to the sheet containing the code, rather than the object name `Sheet1`. This makes the code portable, which means it can be used in other sheet modules without worrying about the name of the sheet.

`iStartMonth` uses the `ListIndex` property of `ComboBox1` to obtain the month as a number. Because the `ListIndex` is zero-based, 1 is added to give the correct month number. The `DateSerial` function translates the year and month numbers into a date and assigns the date to `dteStartDate`. The second `DateSerial` function calculates a date that is one month ahead of `dteStartDate` and assigns it to `dteEndDate`.

The `Format` function is used to turn `dteStartDate` and `dteEndDate` back into strings in the U.S. date format of `mm/dd/yyyy`. The appropriate logical operators are placed in front, and the resulting strings are assigned to `sStartCriterion` and `sEndCriterion`, respectively. `FilterDates` finally executes the `AutoFilter` method on the table `Table1`, using the computed criteria.

Getting the Exact Date

Another tricky problem with `AutoFilter` occurs with dates in all language versions of Excel. The problem arises when you want to get an exact date, rather than a date within a range of dates. In this case, `AutoFilter` matches your date with the formatted appearance of the dates in the worksheet, not the underlying date values.

Excel holds dates as numeric values equal to the number of days since Jan 1, 1900. For example, Jan 1, 2007 is held as 39,083. When you ask for dates greater than or equal to Jan 1, 2007, Excel looks for date serial numbers greater than or equal to 39,083. However, when you ask for dates equal to Jan 1, 2007, Excel does not look for the numeric value of the date. Excel checks for the string value "Jan 1, 2007" as it appears formatted in the worksheet and as it is returned by the `Text` property of the `Range` object.

The following adaptation of `FilterDates` will handle an exact date match in your list, because `sExactCriterion` is assigned the date value as a string, in the format "mmm dd, yyyy". It obtains the format from the worksheet using the `NumberFormat` property of the first cell in the body of the table, where the date is formatted as mmm dd, yyyy:

```
Sub FilterExactDate()
    Dim iExactMonth As Integer
    Dim iExactYear As Integer
    Dim dteExactDate As Date
    Dim sExactCriterion As String
    Dim sDateFormat As String
    Dim loMyData as ListObject

    'Get Date values
    iExactMonth = Sheet1.ComboBox1.ListIndex + 1
    iExactYear = Sheet1.ComboBox2.Value

    'Get Format from Table
    Set loMyData = Sheet1.ListObjects("Table1")
    sDateFormat = loMyData.DataBodyRange(1).NumberFormat
    'Calculate as a date and format as in worksheet
    dteExactDate = DateSerial(iExactYear, iExactMonth, 1)

    sExactCriterion = Format(dteExactDate, sDateFormat)

    'Filter Table1

    loMyData.Range.AutoFilter _
        Field:=1, Criteria1:=sExactCriterion
End Sub
```

The previous code will give all the entries for the first of the month, because 1 is specified as the third parameter in the `DateSerial` function. To select any day of the month, a third combo box could be added to cell A2 and some code added to the `ComboBox1_Click` event procedure to list the correct number of days for the month specified in `ComboBox1`.

Copying the Visible Rows

If you want to make it easy to create a new worksheet containing a copy of the filtered data, you can place an ActiveX command button at the top of the worksheet and enter the following `Click` event procedure in the worksheet module. This procedure copies the visible cells in `Table1`:

```
Private Sub CommandButton1_Click()
    Dim wksNew As Worksheet
    Dim sWksName As String
    Dim sMonth As String
    Dim sYear As String
    Dim wksDummyWks As Worksheet

    'Get Date values
    sMonth = Me.ComboBox1.Value
    sYear = Me.ComboBox2.Value

    'Check that month has not been copied
```

```
On Error Resume Next
sWksName = Format(DateValue(sYear & "-" & sMonth & "-1"), "mmm yyyy")
Set wksDummyWks = Worksheets(sWksName)
If Err.Number = 0 Then
    MsgBox "This data has already been copied"
    Exit Sub
End If
On Error GoTo 0

'Add new worksheet and copy visible cells from Table1
Set wksNew = Worksheets.Add
Me.ListObjects(1).Range.SpecialCells(xlCellTypeVisible).Copy _
    Destination:=wksNew.Range("A1")
wksNew.Columns("A:G").AutoFit

'Name worksheet
wksNew.Name = sWksName
End Sub
```

The `Click` event procedure first calculates a name for the new worksheet in the format `mmm yyyy`. It then checks to see if this worksheet already exists by setting a dummy object variable to refer to a worksheet with the new name. If this does not cause an error, the worksheet already exists and the procedure issues a message and exits.

If there is no worksheet with the new name, the event procedure adds a new worksheet at the beginning of the existing worksheets. It copies the visible cells in `Table1` to the new sheet and `AutoFits` the column widths to accommodate the copied data. The procedure then names the new worksheet.

Finding the Visible Rows

When you use `AutoFilter`, Excel simply hides the rows that do not match the current filters. If you want to process just the rows that are visible in your code, you need to look at each row in the list and decide if it is hidden or not. There is a trick to this. When referring to the `Hidden` property of a `Range` object, the `Range` object must be an entire row, extending from column A to column XFD, or an entire column, extending from row 1 to row 1048576. You can't use the `Hidden` property with a single cell or a seven-column row from the list shown in Figure 6-9.

The following code checks each row that is visible on the screen and shades the background of any row that has an invalid `Revenue` calculation:

```
Private Sub CommandButton1_Click()
    Dim rngData As Range
    Dim rngRow As Range
    Dim dNumberSold As Double
    Dim dPrice As Double
    Dim dRevenue As Double

    'Locate datarows
    Set rngData = Sheet1.ListObjects("Table1").DataBodyRange

    'Loop through all data rows
```

```

For Each rngRow In rngData.Rows

    'Only process visible rows
    If rngRow.EntireRow.Hidden = False Then

        'Check calculation
        dNumberSold = rngRow.Cells(5).Value
        dPrice = rngRow.Cells(6).Value
        dRevenue = rngRow.Cells(7).Value

        'If wrong, display error
        If Abs(dNumberSold * dPrice - dRevenue) > 0.000001 Then

            rngRow.Select
            rngRow.Interior.ColorIndex = 3
            MsgBox "Error in selected row"

        End If

    End If

Next rngRow
End Sub

```

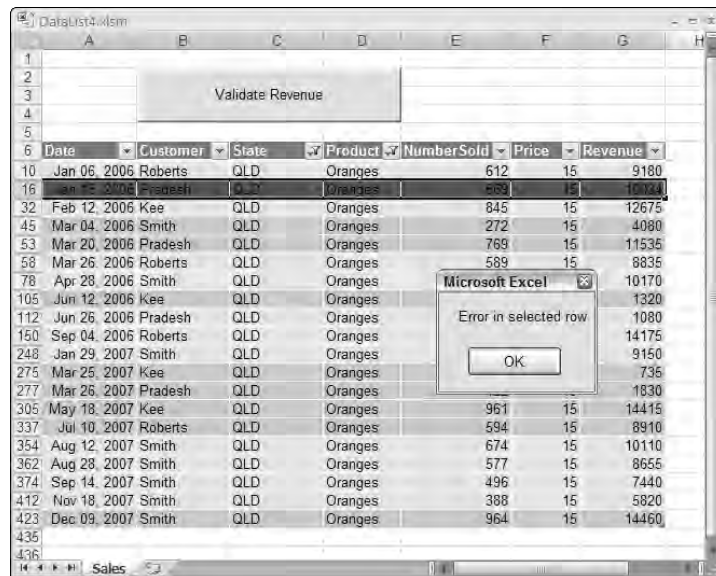


Figure 6-9

The Click event procedure for the command button first defines an object variable `rngData` referring to the rows of data in `Table1`, excluding the Header Row, which is returned by the `DataBodyRange` property of the `ListObject` object. It then uses a `For Each...Next` loop to process all the rows in `rngData`.

Chapter 6: Data Lists

The first `If` test ensures that only rows that are not hidden are processed. The `dNumberSold`, `dPrice`, and `dRevenue` values for the current row are assigned to variables, and the second `If` tests that the `dRevenue` figure is within a reasonable tolerance of the product of `dNumberSold` and `dPrice`.

Because worksheet computations are done with binary representations of numbers to an accuracy of about 15 significant figures, it is not always appropriate to check that two numbers are equal to the last decimal point, especially if the input figures have come from other worksheet calculations. It is better to see if they differ by an acceptably small amount. Because the difference can be positive or negative, the `Abs` function is used to convert both positive and negative differences to a positive difference before comparison.

If the test shows an unacceptable difference, the row is selected and a message is displayed. The row is also given a background color of red.

Advanced Filter

A powerful way to filter data from a list is to use Advanced Filter. You can filter the list in place, like `AutoFilter`, or you can extract it to a different location. The extract location can be in the same worksheet, in another worksheet in the same workbook, or in another open workbook. In the following example, the data for NSW and VIC has been extracted for the first quarter of 2007. The data has been copied from the workbook containing the data list to a new workbook.

The source data can be in a `Table` or can be in a normal range. In the following examples, the data is in a normal range named `Database`.

When you use Advanced Filter, you specify your criteria in a worksheet range. An example of a `Criteria` range is shown in `A1:C3` of the screen in `Figure 6-10`. This worksheet is in a workbook called `DataList6.xlsx`. The data list is in `DataList5.xlsx`, which contains the same data used in the `AutoFilter` examples.

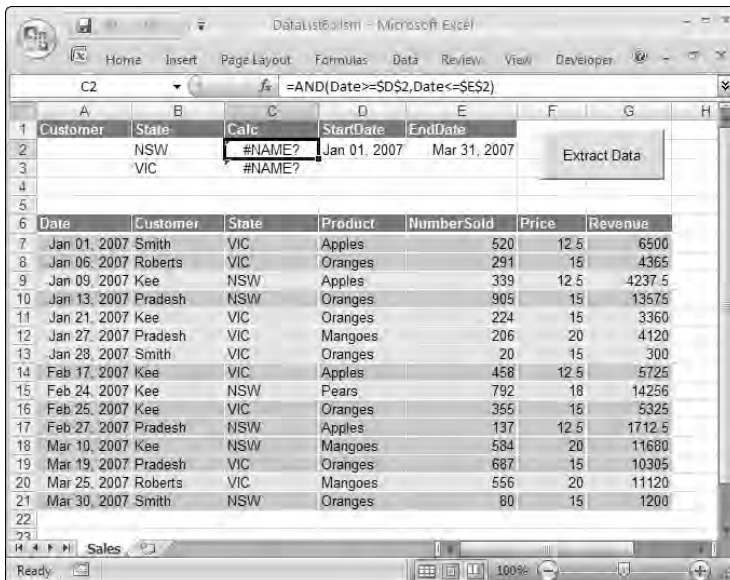


Figure 6-10

The top row of the `Criteria` range contains the field names from the list that you want to filter on. You can have as many rows under the field names as you need. Criteria on different rows are combined using the `OR` operator. Criteria across a row are combined using the `AND` operator. You can also use computed criteria in the form of logical statements that evaluate to `True` or `False`. In the case of computed criteria, the top row of the `Criteria` range must be empty or contain a label that is not a field name in the list, such as `Calc` in this case.

When you create computed criteria, you can refer to the data list field names in your formulas, as you can see in the Formula bar above the worksheet. The Formula bar shows the contents of C2, which is as follows:

```
=AND(Date>=$D$2,Date>=$E$2)
```

The formula in C3 is identical to the formula in C2.

The criteria shown can be thought of as applying this filter:

```
(State=NSW AND Date>=Jan 1, 2007 AND Date<=Mar 31, 2007) OR _
(State=VIC AND Date>=Jan 1, 2007 AND Date<=Mar 31, 2007)
```

Because the field names are not workbook names, the formulas evaluate to a `#NAME?` error.

To facilitate the Advanced Filter, the data list in the `DataList5.xlsx` workbook has been named `Database`. In the `DataList6.xlsx` workbook, A1:C3 has been named `Criteria` and A6:G6 has been named `Extract`. If you carry out the Advanced Filter manually, selecting the `Data` tab of the Ribbon and clicking the `Advanced` button in the `Sort & Filter` group, you see the dialog box in Figure 6-11, where you can enter the names as shown. The `List Range` entry, which is obscured, is `DataList5.xlsx!Database`.

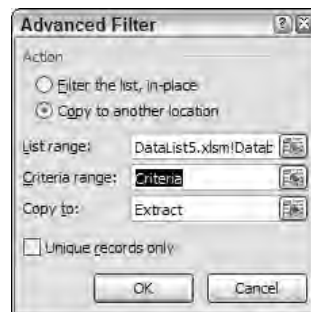


Figure 6-11

To automate this process, the command button with the `Extract Data` caption runs the following `Click` event procedure:

```
Private Sub CommandButton1_Click()
    Dim rngData As Range
    Dim rngCriteria As Range
    Dim rngExtract As Range

    'Define Database, Criteria & Extract Ranges
```

Chapter 6: Data Lists

```
Set rngData = Workbooks("DataList5.xlsm").Worksheets("Sales").Range("Database")
Set rngCriteria = ThisWorkbook.Worksheets("Sales").Range("Criteria")
Set rngExtract = ThisWorkbook.Worksheets("Sales").Range("Extract")

'Extract data with Advanced Filter
rngData.AdvancedFilter Action:=xlFilterCopy, _
                      CriteriaRange:=rngCriteria, _
                      CopyToRange:=rngExtract, _
                      Unique:=False

End Sub
```

The event procedure defines three object variables referring to the Database, Criteria, and Extract ranges. It then runs the `AdvancedFilter` method of the Database Range object.

Data Form

Excel has a built-in form that you can use to view, find, and edit data in a list. The feature is not available on the Ribbon, so you need to add it to the Quick Access menu if you want to use it through the user interface. Right-click the Quick Access menu and choose `Customize Quick Access Toolbar` to open the dialog box shown in Figure 6-12. Select the `Customization` button, if necessary, and from the drop-down above the left list box, select `Commands Not in the Ribbon`. Find the `Form` command and add it to the Quick Access menu.

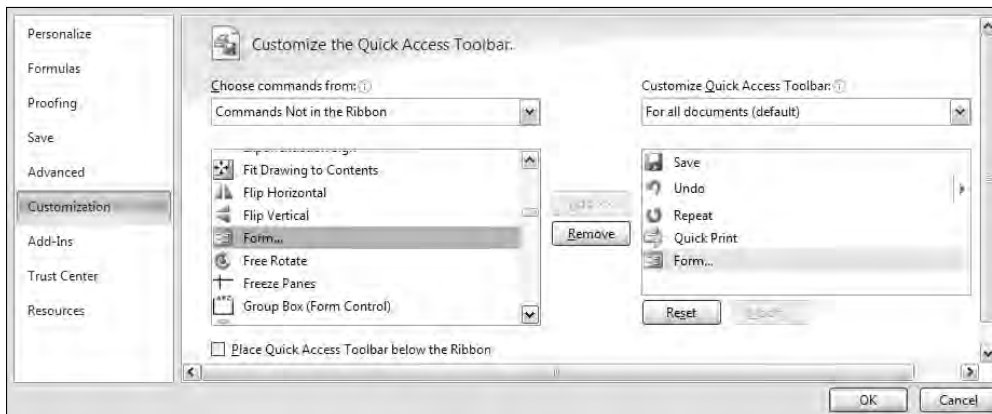


Figure 6-12

This feature can be used with a normal range of data or a Table. If you select a single cell in the data, or select the entire list, and click the `Form` button, you will see a form like the one in Figure 6-13.

If you record this process, you will get code like the following:

```
Range("B2").Select
ActiveSheet.ShowDataForm
```

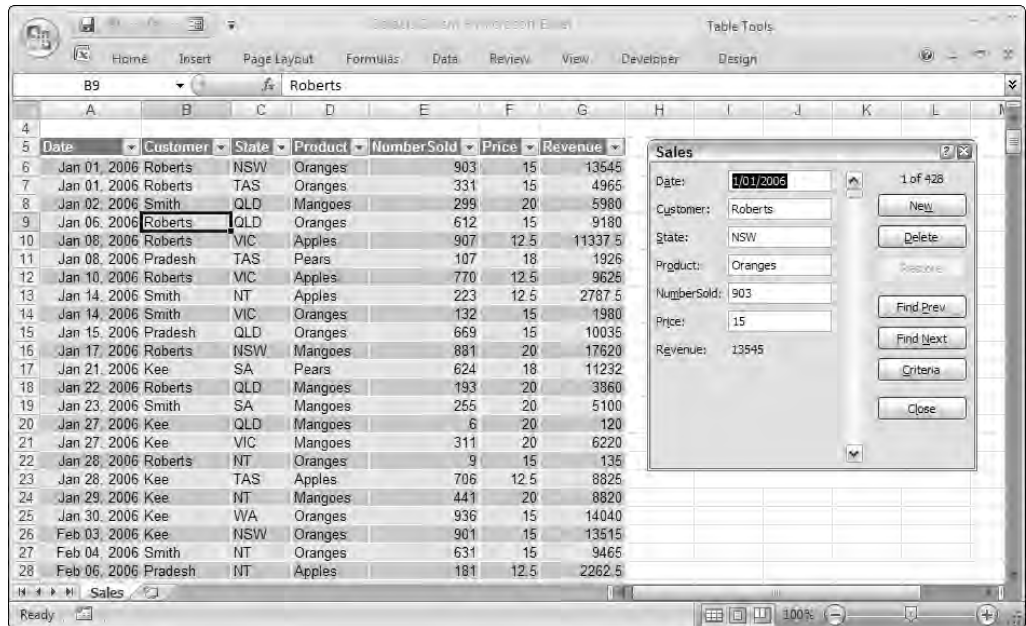



Figure 6-13

If your list starts in A1, and you record selecting the first cell and showing the Data Form, then the recorded macro works. If your list starts in any cell other than A1, and you record while selecting the top-left corner and showing the Data Form, the recorded macro will give an error message when you try to run it. You can overcome this problem by applying the name `Database` to your list.

If you don't work entirely with U.S. date and number formats, the Data Form feature is quite dangerous when displayed by VBA code using the `ShowDataForm` method. The Data Form, when invoked by VBA, displays dates and numbers only in U.S. format. On the other hand, any dates or numbers typed in by a user are interpreted according to the regional settings in the Windows Control Panel. Therefore, if you set the date in the British format (dd/mm/yyyy), when you use the Data Form, the dates become corrupted. See Chapter 25 for more details.

Summary

As you have seen, the AutoFilter and Advanced Filter features can be combined with VBA code to provide flexible ways for users to extract information from data lists. By combining these features with ActiveX controls, such as combo boxes and command buttons, you can make them readily accessible to all levels of users. You can use the macro recorder to get an indication of the required methods and adapt the recorded code to accept input from the ActiveX controls.

Chapter 6: Data Lists

Tables provide a way to formalize data structures and tools to maintain data lists. Sorting and filtering is facilitated when the data is in a table. The `Tables ListObject` object makes it easier to generate VBA references to your data and to manipulate it programmatically.

However, you need to take care if you work with non-U.S. date formats. You need to bear in mind that VBA requires you to use U.S. date formats when you compare ranges of dates using `AutoFilter`. If this interests you, you should check out Chapter 21, which deals with international programming issues.

Also, when you want to detect which rows have been hidden by `AutoFilter`, you need to be aware that the `Hidden` property of the `Range` object can only be applied to entire worksheet rows.

`Advanced Filter` provides the VBA programmer with very powerful filtering in Excel. You can set up much more complex criteria with `Advanced Filter` than you can with `AutoFilter`, and you can copy filtered data to a specified range. You can also use `Advanced Filter` to copy filtered data from one workbook to another.

The `Data Form` feature makes it very easy to set up a data maintenance macro. However, you should apply the name `Database` to your data list if the top-left corner of the list is not in the A1 cell.

7

PivotTables

PivotTables are an extension of the cross tabulation tables used in presenting statistics, and can be used to summarize complex data in a table format. An example of cross tabulation is a table showing the number of employees in an organization, broken down by age and sex. PivotTables are more powerful and can show more than two variables, so they could be used to show employees broken down by age, sex, and alcohol, to quote an old statistician's joke.

The input data for a can come from an Excel worksheet, a text file, an Access database, or a wide range of external database applications. PivotTables can handle up to 256 column or row fields, if you can interpret the results. They can perform many types of standard calculations, such as summing, counting, and averaging. They can produce subtotals and grand totals.

Data can be grouped as in Excel's outline feature, and you can hide unwanted rows and columns. You can also define calculations within the body of the table. PivotTables are also very flexible if you want to change the layout of the data and add or delete variables. You can create PivotCharts that are linked to your PivotTable results in such a way that you can manipulate the data layout from within the chart.

PivotTables are designed so that you can easily generate and manipulate them manually. If you want to create many of them, or provide a higher level of automation to users, you can tap into the Excel object model. This chapter examines the following objects:

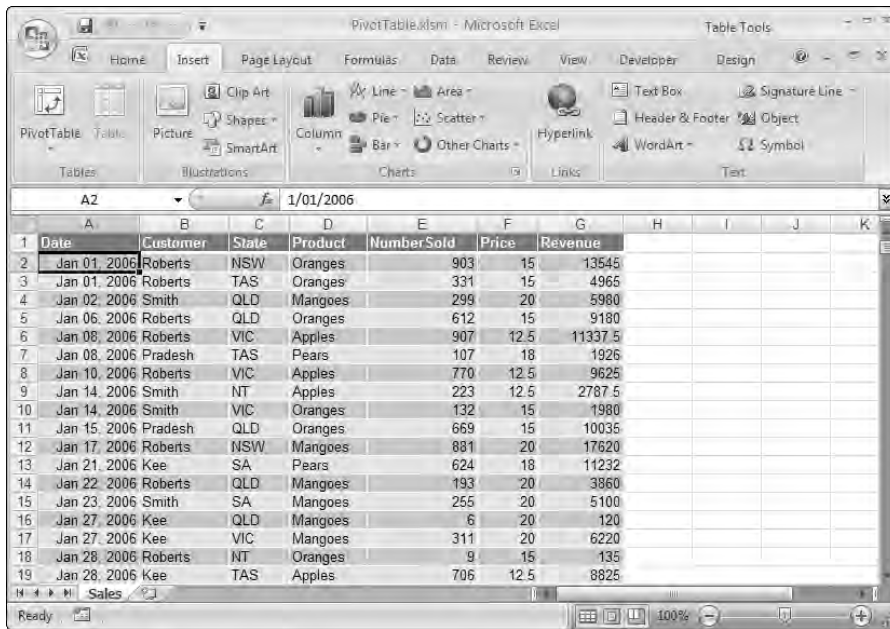
- PivotTables
- PivotCaches
- PivotFields
- PivotItems
- PivotCharts

The PivotTable feature has evolved more than most other established Excel features. With each new version of Excel, PivotTables have been made easier to use and provided with new features. Some of these capabilities and the code covered in this chapter might not work in older versions of Excel.

Creating a PivotTable Report

PivotTables can accept input data from a spreadsheet, or from an external source such as an Access database. When using Excel data, the data should be structured as a data list, as explained at the beginning of Chapter 6, although it is also possible to use data from another PivotTable or from multiple consolidation ranges. The columns of the list are fields and the rows are records, apart from the top row that defines the names of the fields.

Take the Table in Figure 7-1, containing data for 2006 and 2007, as your input data. It is not necessary to have your data in a Table, but the Table provides tools that help maintain and identify the data.



	A	B	C	D	E	F	G	H	I	J	K
1	Date	Customer	State	Product	NumberSold	Price	Revenue				
2	Jan 01, 2006	Roberts	NSW	Oranges	903	15	13545				
3	Jan 01, 2006	Roberts	TAS	Oranges	331	15	4965				
4	Jan 02, 2006	Smith	QLD	Mangoes	299	20	5980				
5	Jan 06, 2006	Roberts	QLD	Oranges	612	15	9180				
6	Jan 08, 2006	Roberts	VIC	Apples	907	12.5	11337.5				
7	Jan 08, 2006	Pradesh	TAS	Pears	107	18	1926				
8	Jan 10, 2006	Roberts	VIC	Apples	770	12.5	9625				
9	Jan 14, 2006	Smith	NT	Apples	223	12.5	2787.5				
10	Jan 14, 2006	Smith	VIC	Oranges	132	15	1980				
11	Jan 15, 2006	Pradesh	QLD	Oranges	669	15	10035				
12	Jan 17, 2006	Roberts	NSW	Mangoes	881	20	17620				
13	Jan 21, 2006	Kee	SA	Pears	624	18	11232				
14	Jan 22, 2006	Roberts	QLD	Mangoes	193	20	3860				
15	Jan 23, 2006	Smith	SA	Mangoes	255	20	5100				
16	Jan 27, 2006	Kee	QLD	Mangoes	6	20	120				
17	Jan 27, 2006	Kee	VIC	Mangoes	311	20	6220				
18	Jan 28, 2006	Roberts	NT	Oranges	9	15	135				
19	Jan 28, 2006	Kee	TAS	Apples	706	12.5	8825				

Figure 7-1

As usual in Excel, it is a good idea to have an identifier for your data range that you can use as a reference in your code. You could define a name for the data range. Alternatively, you can create a Table to contain the data. In this case the data is placed in Table1. You can then refer to the Table in your code.

You want to summarize NumberSold within the entire time period by Customer and Product. With the cell pointer in the data list, select the Insert tab of the Ribbon and click the PivotTable button in the Tables group. You will see the dialog box shown in Figure 7-2, which will show the name of the Table as the data source.

When you click OK, you will see a screen like that in Figure 7-3. Drag the Customer field to the Row Labels area, the Product field to the Column Labels area, and the NumberSold field to the Values area, as shown in Figure 7-3. As you drag the fields, the PivotTable Report will be constructed in the worksheet.

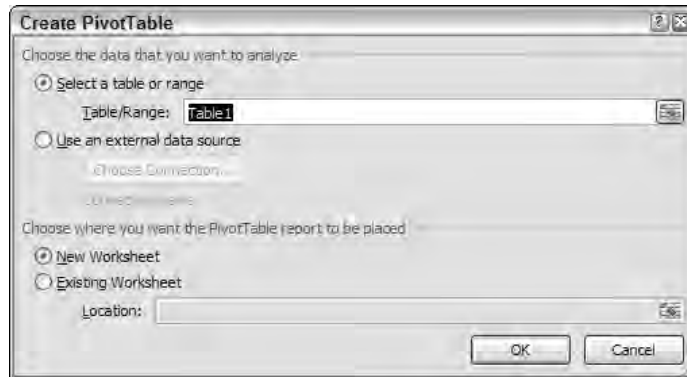


Figure 7-2

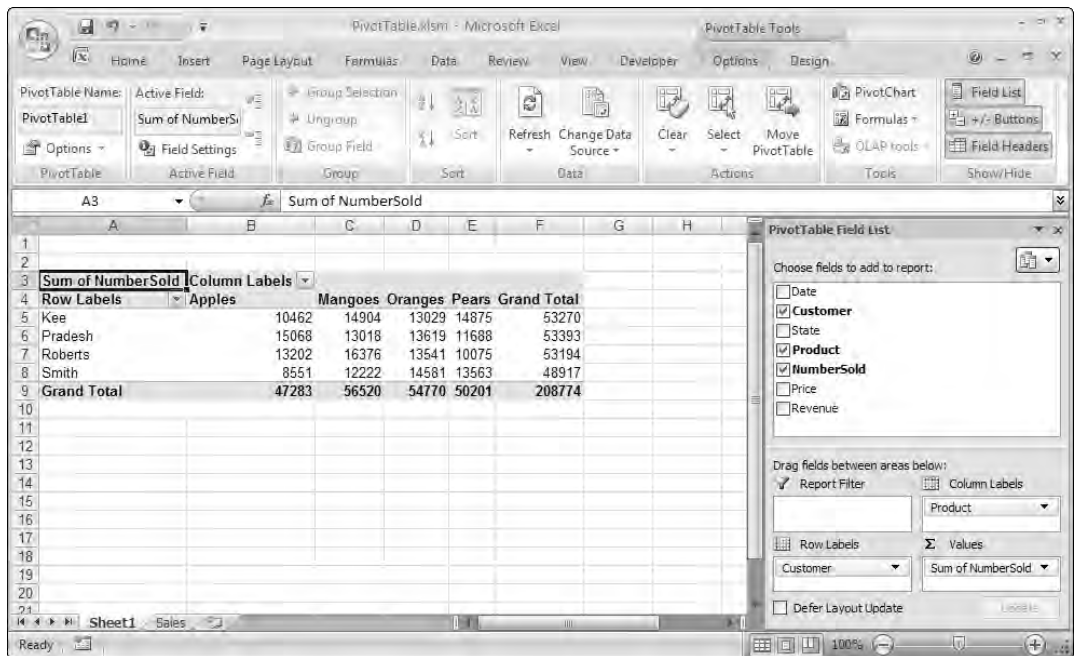


Figure 7-3

If you use the macro recorder to create a macro to carry out this operation, it will look similar to the following code:

```
Sheets.Add
ActiveWorkbook.PivotCaches.Create(SourceType:=xlDatabase, SourceData:= _
    "Table1", Version:=xlPivotTableVersion12).CreatePivotTable _
    TableDestination:="Sheet1!R3C1", TableName:="PivotTable1", _
    DefaultVersion:= xlPivotTableVersion12
```

Chapter 7: PivotTables

```
Sheets("Sheet1").Select
With ActiveSheet.PivotTables("PivotTable1").PivotFields("Customer")
    .Orientation = xlRowField
    .Position = 1
End With
With ActiveSheet.PivotTables("PivotTable1").PivotFields("Product")
    .Orientation = xlColumnField
    .Position = 1
End With
ActiveSheet.PivotTables("PivotTable1").AddDataField ActiveSheet.PivotTables( _
    "PivotTable1").PivotFields("NumberSold"), "Sum of NumberSold", xlSum
```

The code creates a new worksheet, which becomes the active sheet. It then uses the `Add` method of the `PivotCaches` collection to create a new `PivotCache`. `PivotCaches` are discussed later. It next uses the `CreatePivotTable` method of the `PivotCache` object to create an empty `PivotTable` in the new worksheet, starting in the R3C1 (or A3) cell, and names the `PivotTable` `PivotTable1`. The code also uses a parameter declaring the `DefaultVersion`. The source range is specified as the `Table` range.

The `PivotTable` is set up to start in A3 to leave room for page fields, which are also called Report Filters in Excel 2007, above the `Table`. Page fields are discussed later.

The `Customer` field is defined to be a row field and the `Product` field to be a column field. The `Position` property ranks multiple row fields or column fields, which is not necessary here. Finally, `NumberSold` is added as a data field, which means that it appears in the body of the `Table` where it is to be summed.

As it stands, the recorded code is very inflexible. It adds a new worksheet and assumes that it is named `Sheet1`. It applies the name `PivotTable1` and then depends on that name in the subsequent code. The code can be generalized and tidied up as shown:

```
Sub CreatePivotTable()

    Dim wks As Worksheet
    Dim pvc As PivotCache
    Dim pvt As PivotTable

    'Add new worksheet
    Set wks = Worksheets.Add

    'Create PivotCache
    Set pvc = ActiveWorkbook.PivotCaches.Create( _
        SourceType:=xlDatabase, _
        SourceData:=Sheet1.ListObjects("Table1").Range)

    'Create PivotTable
    Set pvt = pvc.CreatePivotTable(TableDestination:=wks.Range("A3"), _
        DefaultVersion:=xlPivotTableVersion12)

    'Define fields in PivotTable
    With pvt
        With .PivotFields("Customer")
            .Orientation = xlRowField
            .Position = 1
        End With
    End With
```

```

    With .PivotFields("Product")
        .Orientation = xlColumnField
        .Position = 1
    End With
    .AddDataField .PivotFields("NumberSold"), "Sum of NumberSold", xlSum
End With

End Sub

```

The `SourceData` parameter of the `Create` method of the `PivotCaches` collection is very flexible. It accepts a `Range` object or a text address as well as a text reference to a `Table`. You have specified the `Range` property of `Table1`. If you had assigned the name `Database` to the data, you could use the following:

```
SourceData:="Database")
```

The `TableDestination` parameter of the `CreatePivotTable` method also accepts the `Range` object reference you used. If you are not concerned about the name of the `PivotTable`, you can leave out the `TableName` parameter and accept the default name.

PivotCaches

A `PivotCache` is a buffer, or holding area, where data is stored and accessed as required from a data source. It acts as a communication channel between the data source and the `PivotTable`.

In Excel 2007, you can create a `PivotCache` using the `Create` method of the `PivotCaches` collection, as seen in the recorded code. You have extensive control over what data you draw from the source when you create a `PivotCache`. Particularly in conjunction with ADO (ActiveX Data Objects), which is demonstrated at the end of this chapter, you can achieve high levels of programmatic control over external data sources. Chapters 20 and 21 show you the great flexibility of ADO and other ways to handle external data sources. You can use the techniques from those chapters to construct data sources for `PivotTables`.

You can also use a `PivotCache` to generate multiple `PivotTables` from the same data source. This is more efficient than forcing each `PivotTable` to maintain its own data source.

When you have created a `PivotCache`, you can create any number of `PivotTables` from it using the `CreatePivotTable` method of the `PivotCache` object.

PivotTables Collection

You can use another method to create a `PivotTable` from a `PivotCache`, using the `Add` method of the `PivotTables` collection. If you have already created a `PivotCache` in your workbook and you want to create a second `PivotTable`, you can use the following code:

```

Sub AddTable()
    Dim pvc As PivotCache
    Dim pvt As PivotTable

    'Access existing PivotCache
    Set pvc = ActiveWorkbook.PivotCaches(1)

    'Add new PivotTable to PivotTables collection

```

```
Set pvt = ActiveSheet.PivotTables.Add(PivotCache:=pvc, _  
                                     TableDestination:=Range("A3"))
```

```
End Sub
```

There is no particular advantage to using this method compared with the `CreatePivotTable` method. It's just another thread in the rich tapestry of Excel.

PivotFields

The columns in the data source are referred to as fields. When the fields are used in a PivotTable, they become `PivotField` objects and belong to the `PivotFields` collection of the `PivotTable` object. The `PivotFields` collection contains all the fields in the data source and any calculated fields you have added, not just the fields that are visible in the PivotTable report. Calculated fields are discussed later in this section.

You can add `PivotFields` to a report using two different techniques. You can use the `AddFields` method of the `PivotTable` object, or you can assign a value to the `Orientation` property of the `PivotField` object, as shown here:

```
Sub AddFieldsToTable()  
    'Adds new fields to an existing PivotTable  
  
    'Access existing PivotTable  
    With ActiveSheet.PivotTables(1)  
  
        'Add new State field to rows  
        .AddFields RowFields:="State", AddToTable:=True  
  
        'Add Date as new page field  
        .PivotFields("Date").Orientation = xlPageField  
  
    End With  
  
End Sub
```

If you run this code on the example PivotTable, you will get the result shown in Figure 7-4.

The `AddFields` method can add multiple row, column, and page fields. These fields replace any existing fields, unless you set the `AddToTable` parameter to `True`, as in the previous example. However, `AddFields` can't be used to add or replace data fields. The following code redefines the layout of the fields in the existing Table, apart from the data field:

```
Sub RedefinePivotTable()  
    'Reorganize an existing PivotTable  
    Dim pvt As PivotTable  
  
    'Access existing PivotTable
```



```

Set pvt = ActiveSheet.PivotTables(1)

'Specify arrangement of row, column and page fields
pvt.AddFields RowFields:=Array("Product", "Customer"), _
              ColumnFields:="State", _
              PageFields:="Date"

End Sub

```

	Date	(All)					
			Sum of NumberSold	Column Labels			
	Row Labels	Apples	Mangoes	Oranges	Pears	Grand Total	
	Kee	10462	14904	13029	14875	53270	
	NSW	2059	2452	2779	792	8082	
	NT	1353	2364	1722	1283	6722	
	QLD	526	2257	1943	1259	5985	
	SA	2559	2076	1612	2919	9166	
	TAS	1562	994	499	3976	7031	
	VIC	1837	2976	856	1917	7586	
	WA	566	1785	3618	2729	8698	
	Pradesh	15068	13018	13619	11688	53393	
	NSW	2188	1112	2318	1929	7547	
	NT	2285	2369	1681	4563	10898	
	QLD	2176	1352	1632		5160	
	SA	2795	1329	1589	1734	7447	
	TAS	2065	1272	1433	724	5494	
	VIC	1593	3807	2618	1911	9929	
	WA	1966	1777	2348	827	6918	
	Roberts	13202	16376	13541	10075	53194	
	NSW	274	2807	1738	3127	7946	
	NT	2791	1483	1056	1460	6790	
	QLD	2053	1045	2740	1852	7690	
	SA	3572	1846	2942		8360	
	TAS	876	3407	1599	781	6663	
	VIC	3015	2428	291	2317	8051	
	WA	621	3360	3175	538	7694	
	Smith	8551	12222	14581	13563	48917	
	NSW	1182	1886	2711	1069	6848	

Figure 7-4

Note that you can use the Array function to include more than one field in a field location. The result is shown in Figure 7-5.

You can use the Orientation and Position properties of the PivotField object to reorganize the Table. Position defines the hierarchy of fields within the Table, counting from the top level down. The following code, added to the end of the RedefinePivotTable code, would move the Customer fields above the Product fields as shown in Figure 7-6, for example:

```
pvt.PivotFields("Customer").Position = 1
```

The screenshot shows a PivotTable with the following data:

Row Labels	NSW	NT	QLD	SA	TAS	VIC	WA	Grand Total
Apples	5703	7660	5538	10141	5587	9206	3448	47283
Kee	2059	1353	526	2559	1562	1837	566	10462
Pradesh	2188	2285	2176	2795	2065	1593	1966	15068
Roberts	274	2791	2053	3572	876	3015	621	13202
Smith	1182	1231	783	1215	1084	2761	295	8551
Mangoes	8257	8663	6490	7112	5970	11171	8857	56520
Kee	2452	2364	2257	2076	994	2976	1785	14904
Pradesh	1112	2369	1352	1329	1272	3807	1777	13018
Roberts	2807	1483	1045	1846	3407	2428	3360	16376
Smith	1886	2447	1836	1861	297	1960	1935	12222
Oranges	9546	7585	10974	6733	4891	5110	9931	54770
Kee	2779	1722	1943	1612	499	856	3618	13029
Pradesh	2318	1681	1632	1589	1433	2618	2348	13619
Roberts	1738	1056	2740	2942	1599	291	3175	13541
Smith	2711	3126	4659	590	1360	1345	790	14581
Pears	6917	11682	4253	8287	6607	7423	5032	50201
Kee	792	1283	1259	2919	3976	1917	2729	14875
Pradesh	1929	4563		1734	724	1911	827	11688
Roberts	3127	1460	1852		781	2317	538	10075
Smith	1069	4376	1142	3634	1126	1278	938	13563
Grand Total	30423	35590	27255	32273	23055	32910	27268	208774

Figure 7-5

The screenshot shows a PivotTable with the following data:

Row Labels	NSW	NT	QLD	SA	TAS	VIC	WA	Grand Total
Kee	8082	6722	5985	9166	7031	7586	8698	53270
Apples	2059	1353	526	2559	1562	1837	566	10462
Mangoes	2452	2364	2257	2076	994	2976	1785	14904
Oranges	2779	1722	1943	1612	499	856	3618	13029
Pears	792	1283	1259	2919	3976	1917	2729	14875
Pradesh	7547	10898	5160	7447	5494	9929	6918	53393
Apples	2188	2285	2176	2795	2065	1593	1966	15068
Mangoes	1112	2369	1352	1329	1272	3807	1777	13018
Oranges	2318	1681	1632	1589	1433	2618	2348	13619
Pears	1929	4563		1734	724	1911	827	11688
Roberts	7946	6790	7690	8360	6663	8051	7694	53194
Apples	274	2791	2053	3572	876	3015	621	13202
Mangoes	2807	1483	1045	1846	3407	2428	3360	16376
Oranges	1738	1056	2740	2942	1599	291	3175	13541
Pears	3127	1460	1852		781	2317	538	10075
Smith	6848	11180	8420	7300	3867	7344	3958	48917
Apples	1182	1231	783	1215	1084	2761	295	8551
Mangoes	1886	2447	1836	1861	297	1960	1935	12222
Oranges	2711	3126	4659	590	1360	1345	790	14581
Pears	1069	4376	1142	3634	1126	1278	938	13563
Grand Total	30423	35590	27255	32273	23055	32910	27268	208774

Figure 7-6

You can use the `Function` property of the `PivotField` object to change the way a data field is summarized, and the `NumberFormat` property to set the appearance of the numbers. The following code, added to the end of `RedefinePivotTable`, adds `Revenue` to the data area, summing it and placing it second to

NumberSold by default. The next lines of code change the position of NumberSold to the second position, and change "Sum of NumberSold" to "Count of NumberSold", which tells you how many sales transactions occurred:

```
Sub AddDataField()
    Dim pvt As PivotTable

    Set pvt = ActiveSheet.PivotTables(1)

    'Add and format new Data field
    With pvt.PivotFields("Revenue")
        .Orientation = xlDataField
        .NumberFormat = "0"
    End With

    'Edit existing Data field
    With pvt.DataFields("Sum of NumberSold")
        .Position = 2
        .Function = xlCount
        .NumberFormat = "0"
    End With

End Sub
```

Note that you need to refer to the name of the data field in the same way as it is presented in the Table — "Sum of NumberSold". If any further code followed, it would need to now refer to "Count of NumberSold". Alternatively, you could refer to the data field by its index number or assign it a name of your own choosing.

The result of all these code changes is shown in Figure 7-7.

1	Date	(All)						
2								
3		Column Labels						
4		NSW	NT	QLD	SA			
5	Row Labels	Sum of Revenue	Count of NumberSold	Sum of Revenue	Count of NumberSold	Sum of Revenue	Count of NumberSold	Sum of Revenue
6	Keel	130719	14	113117	13	103522		16
7	Apples	25738	6	16913	3	6575		1
8	Mangoes	49040	3	47280	5	45140		7
9	Oranges	41685	4	25830	3	29145		4
10	Pears	14256	1	23094	2	22662		4
11	Pradesh	119082	16	183292	21	78720		10
12	Apples	27350	4	28563	6	27200		3
13	Mangoes	22240	2	47380	4	27040		3
14	Oranges	34770	4	25215	3	24480		4
15	Pears	34722	6	82134	8			
16	Roberts	141921	13	106668	17	120999		14
17	Apples	3425	2	34888	6	25663		3
18	Mangoes	56140	1	29660	4	20900		3
19	Oranges	26070	3	15840	4	41100		4
20	Pears	56286	4	26280	3	33336		4
21	Smith	112402	15	189986	21	136949		15
22	Apples	14775	3	15388	3	9788		2
23	Mangoes	37720	4	48940	5	36720		3
24	Oranges	40665	5	46890	6	69885		8
25	Pears	19242	3	78768	7	20556		2
26	Grand Total	504124	58	593061	72	440189		55
27								

Figure 7-7

CalculatedFields

You can create new fields in a PivotTable by performing calculations on existing fields. For example, you might want to calculate the weighted average price of each product. You could create a new field called `AveragePrice` and define it to be `Revenue` divided by `NumberSold`, as in the following code:

```
Sub CalculateAveragePrice()  
    Dim pvt As PivotTable  
  
    'Add new Worksheet and PivotTable  
    Worksheets.Add  
    Set pvt = ActiveWorkbook.PivotCaches(1).CreatePivotTable( _  
        TableDestination:=ActiveCell, TableName:="AveragePrice")  
    With pvt  
  
        'Remove AveragePrice if it exists  
        On Error Resume Next  
        .PivotFields("AveragePrice").Delete  
        On Error GoTo 0  
  
        'Create new AveragePrice  
        .CalculatedFields.Add Name:="AveragePrice", _  
            Formula:="=Revenue/NumberSold"  
  
        'Add Row and Column fields  
        .AddFields RowFields:="Customer", ColumnFields:="Product"  
  
        'Add AveragePrice as Data field  
        With .PivotFields("AveragePrice")  
            .Orientation = xlDataField  
            .NumberFormat = "0.00"  
        End With  
  
        'Remove grand totals  
        .ColumnGrand = False  
        .RowGrand = False  
  
    End With  
  
End Sub
```

`CalculateAveragePrice` adds a new worksheet and uses the `CreatePivotTable` method of the previously created `PivotCache` to create a new `PivotTable` in the new worksheet. So you can run this code repeatedly, it deletes any existing `PivotField` objects called `AveragePrice`. The `On Error` statements ensure that the code keeps running if `AveragePrice` does not exist.

The `CalculatedFields` collection is accessed using the `CalculatedFields` method of the `PivotTable`. The `Add` method of the `CalculatedFields` collection is used to add the new field. Note that the new field is really added to the `PivotCache`, even though it appears to have been added to the `PivotTable`. It is now also available to your first `PivotTable`, and deleting the new `PivotTable` would not delete `AveragePrice` from the `PivotCache`. Once the new field exists, you treat it like any other member of the `PivotFields` collection. The final lines of code remove the grand totals that appear by default.

The Table in Figure 7-8 results from the changes outlined in the previous paragraph. Because the prices do not vary in the source data, it is not surprising that the weighted average prices for each product in the PivotTable do not vary either.

1	Sum of AveragePrice	Column Labels			
2	Row Labels	Apples	Mangoes	Oranges	Pears
3	Kee	12.50	20.00	15.00	18.00
4	Pradesh	12.50	20.00	15.00	18.00
5	Roberts	12.50	20.00	15.00	18.00
6	Smith	12.50	20.00	15.00	18.00
7					
8					

Figure 7-8

Take care when creating `CalculatedFields`. You need to appreciate that the calculations are performed *after* the source data has been summed. In this example, `Revenue` and `NumberSold` were summed and one sum was divided by the other sum. This works fine for calculating a weighted average price and is also suitable for simple addition or subtraction. Other calculations might not work as you expect.

For example, say you don't have `Revenue` in the source data, and you decide to calculate it by defining a `CalculatedField` equal to `Price` multiplied by `NumberSold`. This would not give the correct result. You can't get `Revenue` by multiplying the sum of `Price` by the sum of `NumberSold`, except in the special case where only one record from the source data is represented in each cell of the PivotTable.

PivotItems

Each `PivotField` object has a `PivotItems` collection associated with it. You can access the `PivotItems` using the `PivotItems` method of the `PivotField` object. It is a bit peculiar that this is a method and not a property, and is in contrast to the `HiddenItems` property and `VisibleItems` property of the `PivotField` object that return subsets of the `PivotItems` collection.

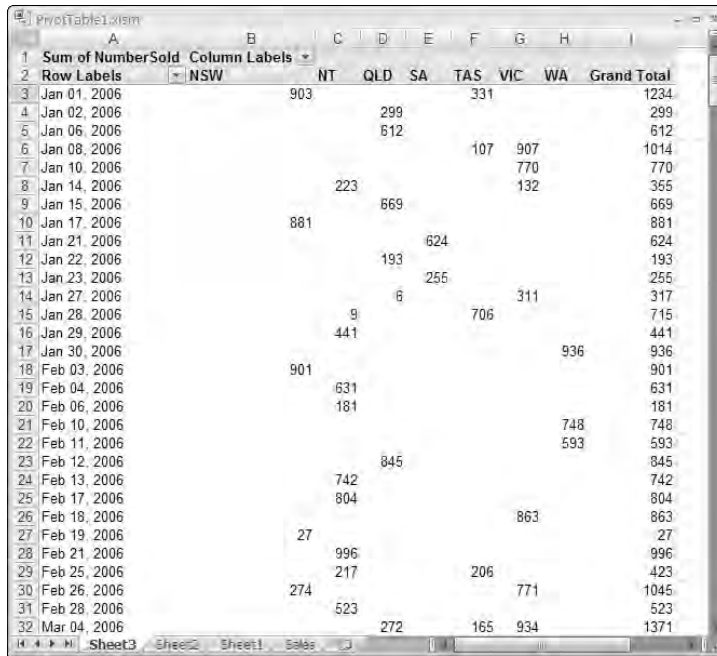
The `PivotItems` collection contains the unique values in a field. For example, the `Product` field in the source data has four unique values— "Apples", "Mangoes", "Oranges", and "Pears", which constitute the `PivotItems` collection for that field.

Grouping

You can group the items in a field in any way you like. For example, `NSW`, `QLD`, and `VIC` (New South Wales, Queensland, and Victoria, respectively) could be grouped as `EasternStates`. This can be very useful when you have many items in a field. You can also group dates, which have a predefined group structure including years, quarters, and months.

Chapter 7: PivotTables

If you bring the `Date` field from the source data into the PivotTable as a row field, as shown in Figure 7-9, you will have nearly 350 rows in the Table because there are that many unique dates.



The screenshot shows a PivotTable with the following structure:

1	Sum of NumberSold	Column Labels							
2	Row Labels	NSW	NT	QLD	SA	TAS	VIC	WA	Grand Total
3	Jan 01, 2006	903					331		1234
4	Jan 02, 2006			299					299
5	Jan 06, 2006			612					612
6	Jan 08, 2006						107	907	1014
7	Jan 10, 2006						770		770
8	Jan 14, 2006		223				132		355
9	Jan 15, 2006			669					669
10	Jan 17, 2006	881							881
11	Jan 21, 2006				624				624
12	Jan 22, 2006			193					193
13	Jan 23, 2006				255				255
14	Jan 27, 2006				6			311	317
15	Jan 28, 2006		9			706			715
16	Jan 29, 2006		441						441
17	Jan 30, 2006							936	936
18	Feb 03, 2006	901							901
19	Feb 04, 2006			631					631
20	Feb 06, 2006			181					181
21	Feb 10, 2006							748	748
22	Feb 11, 2006							593	593
23	Feb 12, 2006				845				845
24	Feb 13, 2006			742					742
25	Feb 17, 2006			804					804
26	Feb 18, 2006							863	863
27	Feb 19, 2006		27						27
28	Feb 21, 2006			996					996
29	Feb 25, 2006			217		206			423
30	Feb 26, 2006		274				771		1046
31	Feb 28, 2006			523					523
32	Mar 04, 2006				272		165	934	1371

Figure 7-9

You can group the `Date` items to get a more meaningful summary. You can do this manually by selecting a cell in the PivotTable containing a date item, right-clicking the cell, and clicking `Group`. The dialog box shown in Figure 7-10 appears, where you can select both `Months` and `Years`.



Figure 7-10

When you click OK, you will see the result shown in Figure 7-11.

Sum of NumberSold		Column Labels						Grand Total
Row Labels	NSW	NT	QLD	SA	TAS	VIC	WA	Grand Total
2006								
Jan	1784	673	1779	879	1144	2120	936	9315
Feb	1202	4094	845		206	1634	1341	9322
Mar	565	869	2204	567	456	1357	295	6313
Apr	1614	1085	678	1376	223	2326	1476	8778
May	2776	3133		2887	755	286	325	10162
Jun	1726	1477	506	625	2442		263	7039
Jul	1020	1681	776	448	695	1120	1618	7358
Aug	2015	1238	1790	843	553	2371		8810
Sep	2351	1265	2212	347	1680	405	1397	9657
Oct	1322	1196	88	2038	1375	1215	97	7331
Nov	744	2252	641	2595	864	575	423	8094
Dec	1662	2454	2431	2519	528	1699	3533	14826
2007								
Jan	1244	1631	672	3188	1739	1261	780	10515
Feb	929	2219	673	201	1020	813	42	5897
Mar	664	1717	1337	1442	729	1243	664	7796
Apr	2383	584		548	1006	969		5490
May	1745		961	755	2376	2445	3953	12235
Jun	930	560	866	220	1202	831	3051	7660
Jul	73		1762	3664	1997	2064	1130	10690
Aug	751	885	1695	604	481	1556	107	6079
Sep	119	3402	496	3368	443	1444	1801	11073
Oct	364	1282	1165	1505	642	1355	996	7309
Nov	936	1009	2065	122	499	1596	920	7147
Dec	1504	884	1613	1532		2225	2120	9878
Grand Total	30423	35590	27255	32273	23055	32910	27268	208774

Figure 7-11

The following code can be used to perform the same grouping operation:

```

Sub GroupDates ()
    Dim pvc As PivotCache
    Dim pvt As PivotTable
    Dim rng As Range

    'Add New Worksheet
    Worksheets.Add Before:=Sheets(1)

    'Get reference to existing PivotCache
    Set pvc = ActiveWorkbook.PivotCaches(1)

    'Add PivotTable
    Set pvt = ActiveSheet.PivotTables.Add(PivotCache:=pvc, _
        TableDestination:=Range("A3"))

    'Define fields in PivotTable
    With pvt
        .PivotFields("Date").Orientation = xlRowField
        .PivotFields("State").Orientation = xlColumnField
        .AddDataField .PivotFields("NumberSold"), "Sum of NumberSold", xlSum
    End With

    'Locate the first Date

```

Chapter 7: PivotTables

```
Set rng = .PivotFields("Date").DataRange.Cells(1,1)

'Group all Dates by Month & Year
rng.Group Start:=True, End:=True, _
    Periods:=Array(False, False, False, False, True, False, True)

End With

End Sub
```

The grouping is carried out on the `Range` object underneath the labels for the field or its items. You need to select one of the labels containing an item name. If you choose a number of item names, you will group just those selected items.

`GroupDates` creates an object variable, `rng`, referring to the cell containing the first `Date` item. The `Group` method is applied to this cell, using the parameters that apply to dates. The `Start` and `End` parameters define the start date and end date to be included. When they are set to `True`, all dates are included. The `Periods` parameter array corresponds to the choices in the Grouping dialog box, selecting Months and Years.

The following code ungroups the dates:

```
Sub UnGroupDates()
    Dim rng As Range

    'Get reference to data
    Set rng = ActiveSheet.PivotTables(1).PivotFields("Date").DataRange

    'Ungroup
    rng.Ungroup

End Sub
```

Note that you have used the `DataRange` property of the `PivotField` object to locate the dates. The `DataLabel` property, which you could use to locate the dates when grouping them in previous versions of Excel, does not work in Excel 2007.

You can regroup them with the following code:

```
Sub ReGroupDates()
    Dim rng As Range

    'Get reference to single cell in data
    Set rng = ActiveSheet.PivotTables(1).PivotFields("Date").DataRange.Cells(1, 1)

    'Group all dates by Month & Year
    rng.Group Start:=True, End:=True, _
        Periods:=Array(False, False, False, False, True, False, True)

End Sub
```


The `DataRange` property has been used to refer to the items to be grouped. You can't refer to all the cells in the range, as mentioned previously, so the `Cells` property is used to refer to the first cell in the range.

Visible Property

You can hide items by setting their `Visible` property to `False`. Say you are working with the grouped dates from the last exercise, and you want to see only Jan 2006 and Jan 2007, as shown in Figure 7-12.

Row Labels	NSW	NT	QLD	SA	TAS	VIC	WA	Grand Total
2006								
Jan	1784	673	1779	879	1144	2120	936	9315
2007								
Jan	1244	1631	672	3188	1739	1261	780	10515
Grand Total	3028	2304	2451	4067	2883	3381	1716	19830

Figure 7-12

You could use the following code:

```
Sub CompareMonths()
    Dim pvt As PivotTable
    Dim pvi As PivotItem
    Dim sMonth As String

    'Specify month to be visible
    sMonth = "Jan"

    'Get reference to PivotTable
    Set pvt = ActiveSheet.PivotTables(1)

    'Hide all years except 2006 & 2007
    For Each pvi In pvt.PivotFields("Years").PivotItems
        If pvi.Name <> "2006" And pvi.Name <> "2007" Then
            pvi.Visible = False
        End If
    Next pvi

    'Make sure specified month is visible - can't hide all data
    pvt.PivotFields("Date").PivotItems(sMonth).Visible = True

    'Hide all months in Date except specified month
    For Each pvi In pvt.PivotFields("Date").PivotItems
        If pvi.Name <> sMonth Then pvi.Visible = False
    Next pvi

End Sub
```

Chapter 7: PivotTables

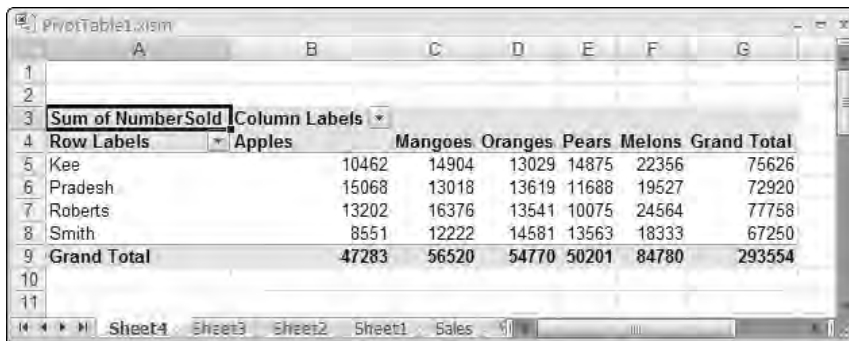
CompareMonths loops through all the items in the Years and Date fields, setting the Visible property to False if the item is not one of the required items. The code has been designed to be reusable for comparing other months by assigning new values to sMonth. Note that the required month is made visible before processing the items in the Date field. This is necessary to ensure that the required month is visible, and also so you don't try to make all the items hidden at once, which would cause a run-time error.

CalculatedItems

You can add calculated items to a field using the Add method of the CalculatedItems collection. Say you wanted to add a new product—melons. You estimate that you would sell 50% more melons than mangoes. This could be added to the Table created by CreatePivotTable using the following code:

```
Sub AddCalculatedItem()  
'Add a new calculated item to the  
'PivotTable produced by CreatePivotTable  
  
With ActiveSheet.PivotTables(1).PivotFields("Product")  
    .CalculatedItems.Add Name:="Melons", Formula:="=Mangoes*1.5"  
End With  
  
End Sub
```

This would give the result in Figure 7-13.



Row Labels	Apples	Mangoes	Oranges	Pears	Melons	Grand Total
Kee	10462	14904	13029	14875	22356	75626
Pradesh	15068	13018	13619	11688	19527	72920
Roberts	13202	16376	13541	10075	24564	77758
Smith	8551	12222	14581	13563	18333	67250
Grand Total	47283	56520	54770	50201	84780	293554

Figure 7-13

You can remove the CalculatedItem by deleting it from either the CalculatedItems collection or the PivotItems collection of the PivotField:

```
Sub DeleteCalculatedItem()  
'Delete an item from the  
'PivotTable produced by CreatePivotTable  
'and AddCalculatedItem  
  
With ActiveSheet.PivotTables(1).PivotFields("Product")  
    .PivotItems("Melons").Delete  
End With  
  
End Sub
```

PivotCharts

PivotCharts were introduced in Excel 2000. They follow all the rules associated with Chart objects, except that they are linked to a PivotTable object. If you change the layout of a PivotChart, Excel automatically changes the layout of the linked PivotTable. Conversely, if you change the layout of a PivotTable that is linked to a PivotChart, Excel automatically changes the layout of the chart.

The following code creates a new PivotChart, based on the PivotTable set up by CreatePivotTable in the active worksheet:

```
Sub CreatePivotChart()
'Add a PivotChart to a PivotTable in the active worksheet

    Dim shp As Shape

    'Create the chart in a Shape object
    Set shp = ActiveSheet.Shapes.AddChart(xlColumnStacked)

    'Assign the PivotTable data to the chart source
    shp.Chart.SetSourceData Source:=ActiveSheet.PivotTables(1).TableRange1, _
        PlotBy:=xlColumns

    'Fit the Chart to a range of cells
    With Range("A11:F28")
        shp.Left = .Left
        shp.Top = .Top
        shp.Width = .Width
        shp.Height = .Height
    End With

    'Change the layout of the PivotTable & the PivotChart
    With shp.Chart.PivotLayout.PivotTable
        .PivotFields("Customer").Orientation = xlColumnField
        .PivotFields("Product").Orientation = xlRowField
    End With

    'Change PivotChart format
    shp.Chart.ChartType = xlCylinderColStacked

End Sub
```

The code produces the chart in Figure 7-14.

The code creates a new Shape object containing a Chart object using the AddChart method of the Shapes collection. This Shape object is also a member of the ChartObjects collection, which is explained in more detail in Chapter 8. It then assigns the PivotTable range to be the data source for the Chart object using the SetSourceData method of the Chart object.

To change the position of the chart, the Shape object is aligned with a range of cells. To change the layout of the PivotTable and the PivotChart that is now linked to the PivotTable, the code uses the PivotLayout property of the Chart object to reference the Chart object's PivotLayout object that gives access to the PivotTable object.

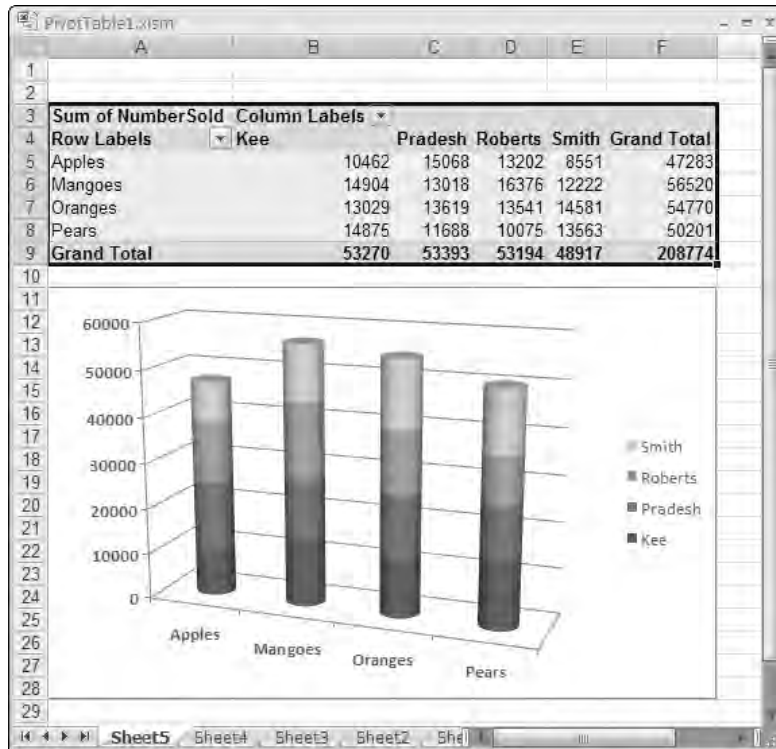


Figure 7-14

Formatting changes to the chart can be made through the properties and methods of the `Chart` object embedded in the `Shape` object. See Chapter 8 for more information on manipulating charts.

External Data Sources

Excel is ultimately limited in the quantity of data it can store, and it is very poor at handling multiple related Tables of data. Therefore, you might want to store your data in an external database application and draw out the data you need as required. A powerful way to do this is to use ADO (ActiveX Data Objects), a topic covered in greater depth in Chapter 20.

The following example shows how to connect to an Access database called `SalesDB.accdb` containing data similar to that you have been using, but potentially much more comprehensive and complex. To run the following code, you must create a reference to ADO. To do this, go to the VBE window and click `Tools` → `References`. From the list, find `Microsoft ActiveX Data Objects` and click in the checkbox beside it. If you find multiple versions of this library, choose the one with the highest version number.

When you run the code, it creates a PivotCache, creates a new worksheet at the front of the workbook, and adds a PivotTable that is similar to those you have already created, but the data source will be the Access database:

```
Sub PivotTableDataViaADO()
    Dim con As ADODB.Connection
    Dim rs As ADODB.Recordset
    Dim sSQL As String
    Dim pvc As PivotCache
    Dim pvt As PivotTable

    'Set up connection
    Set con = New ADODB.Connection
    con.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=C:\Data\SalesDB.accdb;"

    'Define SQL statement
    sSQL = "Select * From SalesData"

    'Open the recordset
    Set rs = New ADODB.Recordset
    Set rs.ActiveConnection = con
    rs.Open sSQL

    'Create the PivotTable cache
    Set pvc = ActiveWorkbook.PivotCaches.Add(SourceType:=xlExternal)
    Set pvc.Recordset = rs

    'Create the PivotTable
    Worksheets.Add Before:=Sheets(1)
    Set pvt = ActiveSheet.PivotTables.Add(PivotCache:=pvc, _
        TableDestination:=Range("A1"))

    With pvt
        .NullString = "0"
        .SmallGrid = False
        .AddFields RowFields:="State", ColumnFields:="Product"
        .PivotFields("NumberSold").Orientation = xlDataField
    End With

End Sub
```

First you create a Connection object linking you to the Access database using the Open method of the ADO Connection object. You then define a SQL (Structured Query Language) statement that says you want to select all the data in a Table called SalesData in the Access database. The Table is almost identical to the one you have been using in Excel, having the same fields and data. See Chapter 20 to get more information on SQL and the terminology that is used in ADO.

You then assign a reference to a new ADO Recordset object to the object variable rs. The ActiveConnection property of rs is assigned a reference to the Connection object. The Open method then populates the recordset with the data in the Access SalesData Table, following the instruction in the SQL statement.

Chapter 7: PivotTables

You then open a new `PivotCache`, declaring its data source as external by setting the `SourceType` parameter to `xlExternal`, and set its `Recordset` property equal to the ADO recordset `rs`. The rest of the code uses techniques you have already seen to create the `PivotTable` using the `PivotCache`.

Chapter 20 goes into much more detail about creating recordsets, and with a much greater explanation of the techniques used. Armed with the knowledge in that chapter, and knowing how to connect a recordset to a `PivotCache` from the previous example, you will be in a position to utilize an enormous range of data sources.

Summary

You use `PivotTables` to summarize complex data. This chapter examined various techniques that you can use to create `PivotTables` from a data source such as an Excel Table using VBA.

The chapter covered setting up `PivotCaches` and showed how they relate to `PivotTables`. You can add fields to `PivotTables` as row, column, or data fields. You can calculate fields from other fields, and items in fields. You can group items. You might do this to summarize dates by years and months, for example. You can hide items, so you see only the data required.

You can link a `PivotChart` to a `PivotTable` so changes in either are synchronized. A `PivotLayout` object connects them.

Using ADO, you can link your `PivotTables` to external data sources.

Charts

In this chapter, you see how you can use the macro recorder to discover what objects, methods, and properties are required to manipulate charts. You will then improve and extend that code to make it more flexible and efficient. This chapter is designed to show you how to gain access to `Chart` objects in VBA code so that you can start to program the vast number of objects that Excel charts contain. You can find more information on these objects in Appendix A. Specifically, this chapter examines:

- ❑ Creating `Chart` objects on separate sheets
- ❑ Creating `Chart` objects embedded in a worksheet
- ❑ Editing data series in charts
- ❑ Defining series with arrays
- ❑ Defining chart labels

You can create two types of charts in Excel: charts that occupy their own chart sheets and charts that are embedded in a worksheet. They can be manipulated in code in much the same way. The only difference is that, whereas the chart sheet is a `Chart` object in its own right, the chart embedded in a worksheet is contained by a `ChartObject` object. Each `ChartObject` on a worksheet is a member of the worksheet's `ChartObjects` collection. Chart sheets are members of the workbook's `Charts` collection.

Each `ChartObject` is a member of the `Shapes` collection, as well as a member of the `ChartObjects` collection. The `Shapes` collection provides you with an alternative way to refer to embedded charts. The macro recorder generates code that uses the `Shapes` collection rather than the `ChartObjects` collection.

Chart Sheets

Select the data in cells A3:D7, as shown in Figure 8-1, and turn on the macro recorder. Right-click the sheet name tab, which contains the sheet name Sales, and select Insert. In the Insert dialog box, select Chart and click OK.

Amalgamated Fruit Inc.				
3	Mangoes	Jan	Feb	Mar
4	South	7777	7345	5364
5	North	1284	3238	4212
6	East	841	733	7858
7	West	5569	4311	5280
8				
9	Bananas	Jan	Feb	Mar
10	South	436	6539	8166
11	North	4175	6362	4026
12	East	2568	9731	9735
13	West	3135	1312	661
14				
15	Lychees	Jan	Feb	Mar
16	South	7052	905	6428
17	North	3964	116	6013
18	East	9256	1434	6409
19	West	1684	8708	4412
20				
21	Rambutan	Jan	Feb	Mar
22	South	8068	8683	4976
23	North	7480	565	21
24	East	8190	7642	4888
25	West	1659	4444	5905
26				
27		73138	72068	84354

Figure 8-1

Figure 8-2 shows the chart that's created. Don't turn off the recorder. You will record some adjustments to the chart.

From the Design tab of the Ribbon, click the Switch Row/Column button in the Data group and then the Quick Layout Button in the Chart Layouts group, and choose the top-left layout. A Chart title will appear that you can change to Mangoes. Click in the chart area to force the recorder to add the title change to its code, and turn off the recorder. The final chart should look like the chart in Figure 8-3.

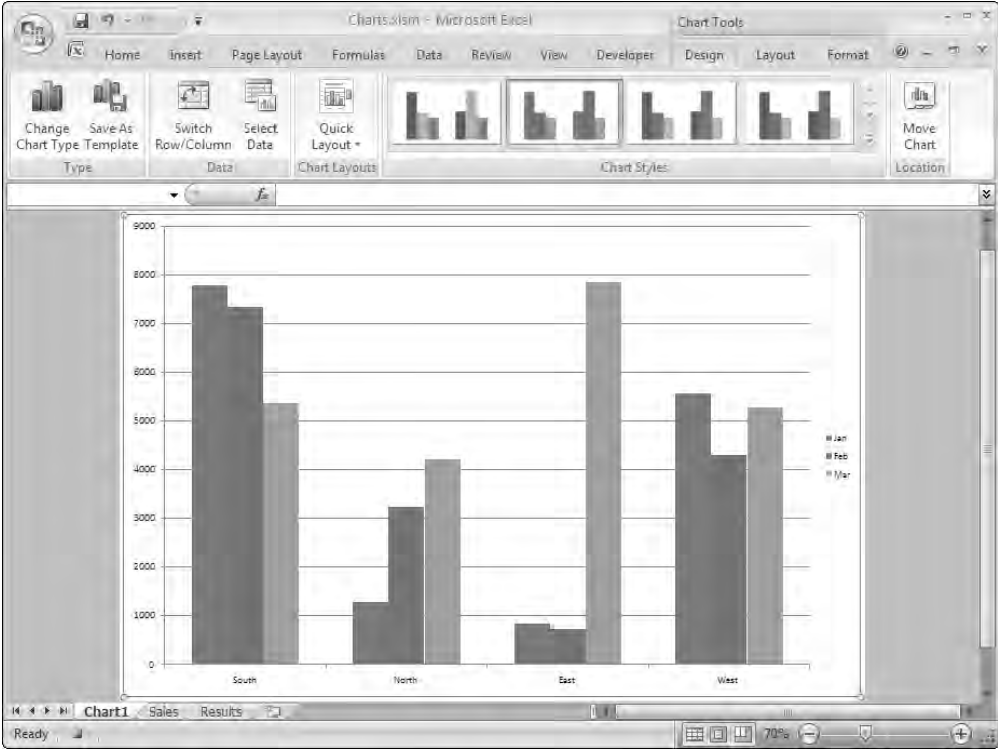


Figure 8-2

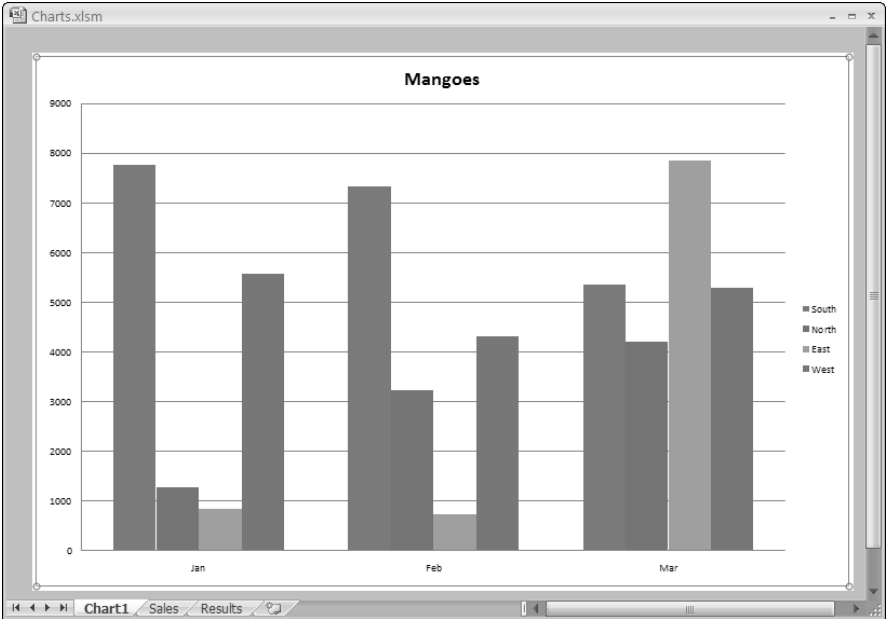


Figure 8-3

The Recorded Macro

The recorded macro should look like the following:

```
Sheets("Sales").Select
ActiveSheet.Shapes.AddChart.Select
ActiveChart.SetSourceData Source:=Range("Sales!$A$3:$D$7"), PlotBy:= _
    xlColumns
ActiveChart.ChartType = xlColumnClustered
Charts.Add
ActiveChart.ChartArea.Select
ActiveChart.PlotBy = xlRows
ActiveChart.ApplyLayout (1)
ActiveChart.ChartTitle.Select
ActiveChart.ChartTitle.Text = "Mangoes"
ActiveChart.ChartArea.Select
```

Although you inserted a new chart sheet, the recorded macro uses the `AddChart` method of the `Shapes` object to create an embedded chart in the Sales worksheet. (Note that the recorder prefers to refer to a `ChartObject` as a `Shape` object, which is an alternative pointed out at the beginning of this chapter.) It uses the `SetSourceData` method to define the ranges plotted by the active chart and sets the chart's `ChartType` property.

The macro then uses the `Add` method of the `Charts` collection to create a new chart sheet. At this point, the embedded chart is used to create the new chart sheet. The `PlotBy` property changes the orientation of the chart, and the `ApplyLayout` method applies the selected format so that a title appears. The `Text` property of the `ChartTitle` object is assigned the string "Mangoes". Finally, the macro records that you selected the chart area.

Adding a Chart Sheet Using VBA Code

The recorded code is a bit odd. There is no need to create an embedded chart. You can simply add a chart sheet and set its properties directly. You can also create an object variable, so that you have a simple and efficient way of referring to the chart in subsequent code. Rather than limit yourself to the preset layouts, you can select the chart features you want, such as a title. There is no need to plot by columns and then plot by rows. The following code incorporates these changes:

```
Sub AddChartSheet()
    Dim cht As Chart

    'Create new chart sheet
    Set cht = Charts.Add

    With cht

        'Specify source data and orientation
        .SetSourceData Source:=Sheets("Sales").Range("A3:D7"), _
            PlotBy:=xlRows
        .ChartType = xlColumnClustered

        'Add a title and assign it a value
```

```

.HasTitle = True
.ChartTitle.Text = "Mangoes"

End With

End Sub

```

Embedded Charts

When you create a chart embedded as a `ChartObject`, it is a good idea to name the `ChartObject` so that it can be easily referenced in later code. When you select the chart, you will see its name to the left of the Formula bar at the top of the screen in the name box.

You can select and change the name of the `ChartObject` in the name box and press Enter to update it. The embedded chart in Figure 8-4 was created, dragged to its new location, and had its name changed to `MangoesChart`. The name can also be changed in the Layout tab of the Ribbon by clicking the Properties button.

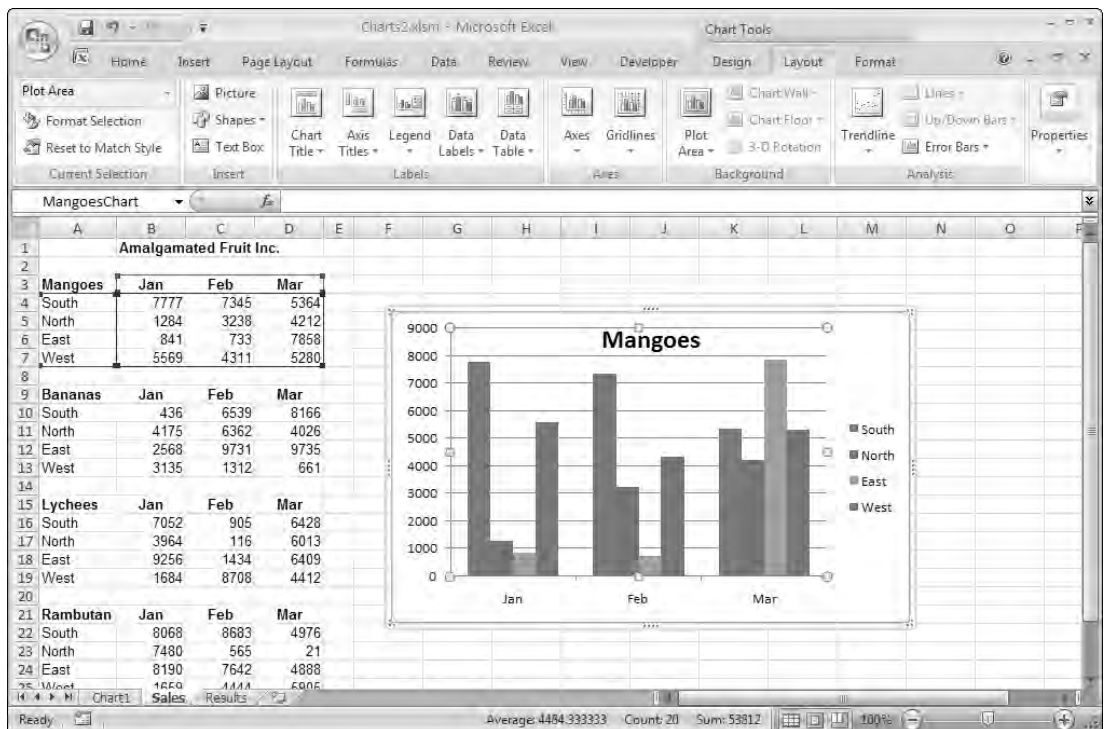


Figure 8-4

Chapter 8: Charts

The chart was created, using the Insert tab of the Ribbon, by clicking the Column button and selecting the top-left chart type from the 2-D Column section of the resulting dialog box. The name was changed in the name box. The row/column orientation was switched in the Design tab as before. The title was added by activating the Layout tab of the Ribbon and clicking the Chart Title button in the Labels group and choosing Centered Overlay Title.

Using the Macro Recorder

If you select cells A3:D7 and turn on the macro recorder before creating the chart in Figure 8-4, you will get code like the following:

```
ActiveSheet.Shapes.AddChart.Select
ActiveChart.SetSourceData Source:=Range("Sales!$A$3:$D$7"), PlotBy:= _
    xlColumns
Selection.Name = "MangoesChart"
ActiveChart.ChartType = xlColumnClustered
ActiveChart.PlotBy = xlRows
ActiveChart.SetElement (msoElementChartTitleCenteredOverlay)
ActiveChart.ChartTitle.Text = "Mangoes"
ActiveChart.ChartArea.Select
```

The recorded macro is similar to the one that created a chart sheet, with some interesting differences. Whereas most lines refer to the `ActiveChart` object, the code that names the `ChartObject` refers to `Selection`, which is the `Shape` or `ChartObject` containing the `Chart` object.

The title of the chart is created using the `SetElement` method. This method has more than 100 constants with self-explanatory names that you can use to change the appearance of the chart. You can find these constants listed in the Object Browser.

Adding an Embedded Chart Using VBA Code

The following code creates a `Shape` object containing the required chart:

```
Sub AddChart()
    Dim shp As Shape

    'Delete any existing ChartObjects
    On Error Resume Next
    ActiveSheet.ChartObjects.Delete
    On Error GoTo 0

    'Create new embedded chart
    Set shp = ActiveSheet.Shapes.AddChart

    'Position Shape over range
    With Range("F3:M19")
        shp.Top = .Top
        shp.Left = .Left
    End With
End Sub
```

```

        shp.Height = .Height
        shp.Width = .Width
    End With

    With shp

        'Assign name to Shape containing chart
        .Name = "MangoesChart"

        With .Chart

            'Specify source data and orientation
            .SetSourceData Source:=Sheets("Sales").Range("A3:D7"), _
                PlotBy:=xlRows
            .ChartType = xlColumnClustered

            'Add a title and assign it a value
            .SetElement msoElementChartTitleCenteredOverlay
            .ChartTitle.Text = "Mangoes"

        End With

    End With

End Sub

```

`AddChart` first deletes any existing `ChartObjects`. It then sets the object variable `shp` to refer to the Shape object added using the `AddChart` method. Although the Shape is also a `ChartObject`, the `AddChart` method returns a Shape object. There is no `AddChart` method for the `ChartObjects` collection.

`AddChart` aligns the Shape with F3:M19 by assigning the `Top`, `Left`, `Width`, and `Height` property values of the range to the same properties of the Shape, and then applies the name "MangoesChart" to the Shape. The `Chart` property of the Shape object is then used to return a reference to the embedded chart and the properties set, as you have seen previously.

Editing Data Series

The `SetSourceData` method of the `Chart` object is the quickest way to define a completely new set of data for a chart. You can also manipulate individual series using the `Series` object, which is a member of the chart's `SeriesCollection` object. The following example is designed to show you how to access individual series.

The code will take the `MangoesChart` and delete all the series from it, and then replace them with four new series, one at a time. The new chart will contain product information for a region nominated by the user. To make it easier to locate each set of product data, names have been assigned to each product range in the worksheet. For example, A3 has been given the name `Mangoes`, corresponding to the label in A3. The final chart will be similar to the chart in Figure 8-5.

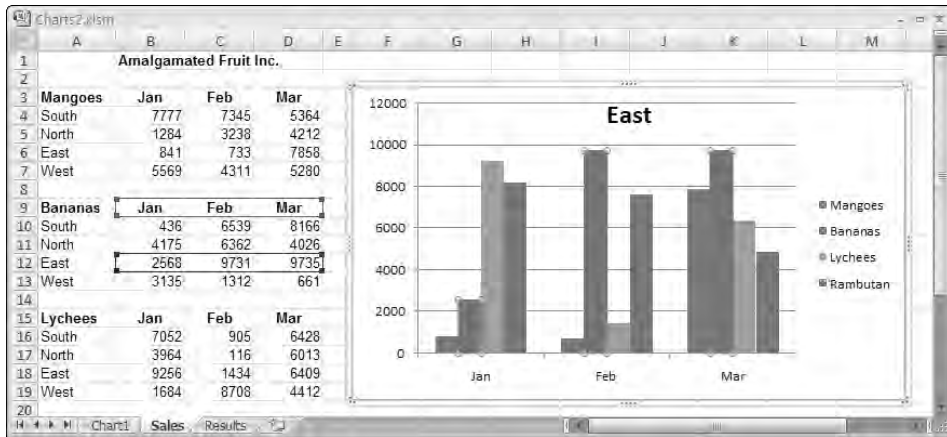


Figure 8-5

The following code converts MangoesChart to include the new data (note that the original chart must still be on the spreadsheet for this to work). Because `MangoesToRegion` is a fairly long procedure, it is examined in sections:

```
Sub MangoesToRegion()
    Dim cbo As ChartObject
    Dim cht As Chart
    Dim scSeries As SeriesCollection
    Dim iCount As Integer
    Dim rngYAxis As Range
    Dim rngXAxis As Range
    Dim vProducts As Variant
    Dim vRegions As Variant
    Dim iRegion As Integer
    Dim vAnswer As Variant

    'Set up arrays for Product & Region names
    vProducts = Array("Mangoes", "Bananas", "Lychees", "Rambutan")
    vRegions = Array("South", "North", "East", "West")

    'Determine that MangoesChart exists
    On Error Resume Next
    Set cbo = Worksheets("Sales").ChartObjects("MangoesChart")
    If cbo Is Nothing Then
        MsgBox "MangoesChart was not found - procedue aborted", vbCritical
        Exit Sub
    End If
    On Error GoTo 0
End Sub
```

`MangoesToRegion` first assigns the product names to `vProducts` and the region names to `vRegions`. It then tries to set the `cbo` object variable by assigning the variable a reference to the `ChartObject` named `MangoesChart`. If this fails, the procedure is aborted. Because it is not the main point of the exercise, this section of code has been kept very simple:

```

'Get Region number
Do 'While vAnswer < 1 Or vAnswer > 4
    vAnswer = InputBox("Enter Region number (1 to 4)")
    If vAnswer = "" Then Exit Sub
    If vAnswer >= 1 And vAnswer <= 4 Then
        Exit Do
    Else
        MsgBox "Region must be 1, 2, 3 or 4", vbCritical
    End If
Loop
iRegion = CInt(vAnswer)

```

The user is then asked to enter the region number. The `Do . . . Loop` will continue until the user clicks Cancel, clicks OK without entering anything, or enters a number between 1 and 4. If a number between 1 and 4 is entered, the value is converted to an integer value, using the `CInt` function, and assigned to `iRegion`:

```

'Set up new chart
Set cht = cbo.Chart
Set scSeries = cht.SeriesCollection
'Delete all existing chart series
For iCount = scSeries.Count To 1 Step -1
    scSeries(iCount).Delete
Next iCount

```

Next, `cht` is assigned a reference to the chart in the `ChartObject`. Then `scSeries` is assigned a reference to the `SeriesCollection` in the chart. The following `For . . . Next` loop deletes all the members of the collection. This is done backwards because deleting the lower number series first automatically decreases the item numbers of the higher series. In this case there will be no series 3 when you try to delete it, which will cause a run-time error. Alternatively, you could have deleted series 1 each time around the loop and the direction of the loop would not have mattered:

```

'Add Products for Region
For iCount = LBound(vProducts) To UBound(vProducts)
    'Define chart ranges
    Set rngYAxis = Range(vProducts(iCount)).Offset(iRegion, 1).Resize(1, 3)
    Set rngXAxis = Range(vProducts(iCount)).Offset(0, 1).Resize(1, 3)

    'Add new series & assign data
    With scSeries.NewSeries
        .Name = vProducts(iCount)
        .Values = rngYAxis
        .XValues = "=" & rngXAxis.Address _
            (RowAbsolute:=True, _
             ColumnAbsolute:=True, _
             ReferenceStyle:=xlR1C1, _
             External:=True)
    End With
Next iCount

```

Chapter 8: Charts

The `For . . . Next` loop adds a new series to the chart for each product. The loop uses the `UBound` and `LBound` functions to avoid having to know the `Option Base` setting for the module. The range object `rngYAxis` is assigned a reference to the chosen region data within the current product data.

`Range(vProducts(i))` refers to the ranges containing the product tables. Each range has been assigned a name corresponding to the text entries in `vProducts(i)`. `iRegion` is used as the row offset into the product data to refer to the correct region data. The column offset is 1 so that the name of the region is excluded from the data. `Resize` ensures that the data range has one row and three columns. The range object `rngXAxis` is assigned a reference to the month names at the top of the product data table.

Following the `With` statement, `MangoesToRegion` uses the `NewSeries` method to add a new empty series to the chart. The `NewSeries` method returns a reference to the new series, which supplies the `With . . . End With` reference that is used by the lines between `With` and `End With`. The `Name` property of the series, which appears in the legend, is assigned the current product name.

The `Values` property of the new series is assigned a reference to `rngYAxis`. The `XValues` property could have been assigned a direct reference to `rngXAxis` in the same way. However, both properties can also be defined by a formula reference as an external reference in the A1 or R1C1 style. The string value generated and assigned to the `Mangoes` series `XValues` property is:

```
=[Charts2.xlsm]Sales!R3C2:R3C4
```

The final section of code is as follows:

```
'Define chart title
cht.ChartTitle.Text = vRegions(iRegion + LBound(vRegions) - 1)

'Give name to chartobject
cbo.Name = "RegionChart"

End Sub
```

The `ChartTitle.Text` property is assigned the appropriate string value in the `vRegions` array, using the value of `iRegion` as an index to the array. To avoid having to know the `Option Base` setting for the module, `LBound(vRegions) - 1` has been used to adjust the index value in `iRegion`, which ranges from 1 to 4. If the `Option Base` setting is 0, this expression returns a value of 1, which adjusts the value of `iRegion` such that it ranges from 0 to 3. If the `Option Base` setting is 1, the expression returns 0, which does not change the `iRegion` value so it still has the range of 1 to 4. Another way to handle the `Option Base` is to use the following code:

```
cht.ChartTitle.Text = vaRegions(iRegion - Array(0,1)(1))
```

The code finally changes the name of the `ChartObject` to `RegionChart`.

Defining Chart Series with Arrays

A chart series can be defined by assigning a VBA array to its `Values` property. This can come in handy if you want to generate a chart that is not linked to the original data. The chart can be distributed in a separate workbook that is independent of the source data.

Figure 8-6 shows a chart of the Mangoes data. You can see the definition of the first data series in the SERIES function in the Formula bar above the worksheet. The month names and the values on the vertical axis are defined by arrays. The region names have been assigned as text to the series names.

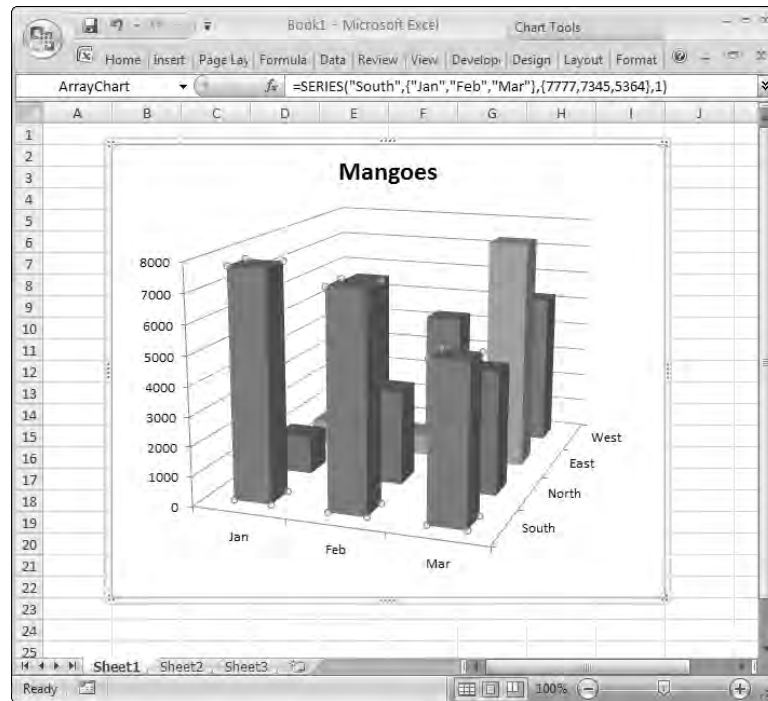


Figure 8-6

The 3D chart can be created using the following code:

```
Sub MakeArrayChart()
    Dim wksSource As Worksheet
    Dim rngSource As Range
    Dim wkb As Workbook
    Dim wks As Worksheet
    Dim cht As Chart
    Dim seNewSeries As Series
    Dim iCount As Integer
    Dim vSalesArray As Variant
    Dim vMonthArray As Variant

    'Create month array
    vMonthArray = Array("Jan", "Feb", "Mar")

    'Define the data source
    Set wksSource = ThisWorkbook.Worksheets("Sales")
    Set rngSource = wksSource.Range("Mangoes")

    'Create a new workbook
```

Chapter 8: Charts

```
Set wkb = Workbooks.Add
Set wks = wkb.Worksheets(1)

'Add a new chart object and embed it in the worksheet
Set cht = wks.Shapes.AddChart.Chart
```

`MakeArrayChart` assigns the month names to `vMonthArray`. This data could have come from the worksheet, if required, like the sales data. A reference to the worksheet that is the source of the data is assigned to `wksSource`. The Mangoes range is assigned to `rngSource`. A new workbook is created for the chart and a reference to it is assigned to `wkb`. A reference to the first worksheet in the new workbook is assigned to `wks`. A new chart is embedded in a `Shape` belonging to the `Shapes` collection in `wks`, and a reference to the embedded chart is assigned to `cht`:

```
With cht

    'Define the chart type
    .ChartType = xl3DColumn

    For iCount = 1 To 4

        'Create a new series
        Set seNewSeries = .SeriesCollection.NewSeries

        'Assign the data as arrays
        vSalesArray = WorksheetFunction.Transpose( _
            rngSource.Offset(iCount, 1).Resize(1, 3).Value)
        seNewSeries.Values = vSalesArray
        seNewSeries.XValues = WorksheetFunction.Transpose(vMonthArray)
        seNewSeries.Name = "=" & rngSource.Cells(iCount + 1, 1).Value & ""

    Next iCount

    'Adjust format
    .HasLegend = False
    .HasTitle = True
    .ChartTitle.Text = "Mangoes"

    'Position the ChartObject in B2:I22 and name it
    With .Parent
        .Top = wks.Range("B2").Top
        .Left = wks.Range("B2").Left
        .Width = wks.Range("B2:I22").Width
        .Height = wks.Range("B2:I22").Height
        .Name = "ArrayChart"
    End With

End With
End Sub
```

In the `With...End With` structure, the `ChartType` property of `cht` is changed to a 3D column type. The `For...Next` loop creates the four new series. Each time around the loop, a new series is created with the `NewSeries` method. The region data from the appropriate row is directly assigned to the variant `vSalesArray`, and `vSalesArray` is immediately assigned to the `Values` property of the new series.

`vMonthArray` is assigned to the `XValues` property of the new series. The text in column A of the `Mangoes` range is assigned to the `Name` property of the new series.

The code then removes the chart legend, which is added by default, and sets the chart title. The final code operates on the `Shape`, which is the chart's parent, to place the chart exactly over `B2:I22`, and names the chart `ArrayChart`.

The result is a chart in a new workbook that is quite independent of the original workbook and its data. If the chart had been copied and pasted into the new workbook, it would still be linked to the original data.

Converting a Chart to Use Arrays

You can easily convert an existing chart to use arrays instead of cell references and make it independent of the original data it was based on. The following code shows how:

```
Sub ConvertSeriesValuesToArrays()
    Dim seSeries As Series
    Dim cht As Chart

    On Error GoTo Failure

    'Get reference to embedded chart
    Set cht = ActiveSheet.ChartObjects(1).Chart

    'Process each series in charts series collection
    For Each seSeries In cht.SeriesCollection

        'Convert range references to numeric/string values
        seSeries.Values = seSeries.Values
        seSeries.XValues = seSeries.XValues
        seSeries.Name = seSeries.Name

    Next seSeries

    Exit Sub

Failure:
    MsgBox "Sorry, the data exceeds the array limits"

End Sub
```

For each series in the chart, the `Values`, `XValues`, and `Name` properties are set equal to themselves. Although these properties can be assigned range references, they always return an array of values when they are interrogated. This behavior can be exploited to convert the cell references to arrays.

In previous versions of Excel, the number of characters that can be contained in the `SERIES` function arrays is limited to 250 characters, or thereabouts. This limit does not apply to Excel 2007, but the code sets up an error trap to cover this possibility, should it be used in a previous version.

Determining the Ranges Used in a Chart

The behavior that is beneficial when converting a chart to use arrays is a problem when you need to programmatically determine the ranges that a chart is based on. If the `Values` and `XValues` properties returned the strings or range objects that you used to define them, the task would be easy.

The only property that contains information on the ranges is the `Formula` property that returns the formula containing the `SERIES` function as a string. The formula would be like the following:

```
=SERIES("Mangoes", Sales!$B$3:$D$3, Sales!$B$5:$D$5, 1)
```

The `XValues` are defined by the second parameter and the `Values` by the third parameter. You need to locate the commas and extract the text between them as shown in the following code, designed to work with a chart embedded in the active sheet:

```
Sub GetRangesFromChart()  
    Dim seSeries As Series  
    Dim sSeriesFunction As String  
    Dim iFirstComma As Integer, iSecondComma As Integer, iThirdComma As Integer  
    Dim sValueRange As String, sXValueRange As String  
    Dim rngValueRange As Range, rngXValueRange As Range  
  
    On Error GoTo Oops  
  
    'Get the SERIES function from the first series in the chart  
    Set seSeries = ActiveSheet.ChartObjects(1).Chart.SeriesCollection(1)  
    sSeriesFunction = seSeries.Formula  
  
    'Locate the commas  
    iFirstComma = InStr(1, sSeriesFunction, ",")  
    iSecondComma = InStr(iFirstComma + 1, sSeriesFunction, ",")  
    iThirdComma = InStr(iSecondComma + 1, sSeriesFunction, ",")  
  
    'Extract the range references as strings  
    sXValueRange = Mid(sSeriesFunction, iFirstComma + 1, _  
        iSecondComma - iFirstComma - 1)  
    sValueRange = Mid(sSeriesFunction, iSecondComma + 1, _  
        iThirdComma - iSecondComma - 1)  
  
    'Convert the strings to range objects  
    Set rngXValueRange = Range(sXValueRange)  
    Set rngValueRange = Range(sValueRange)  
  
    'Color the ranges  
    rngXValueRange.Interior.ColorIndex = 3  
    rngValueRange.Interior.ColorIndex = 4  
  
    Exit Sub  
  
Oops:  
    MsgBox "Sorry, an error has occurred" & vbCrLf & _  
        "This chart might not contain range references"  
  
End Sub
```

sSeriesFunction is assigned the formula of the series, which contains the SERIES function as a string. The positions of the first, second, and third commas are found using the InStr function. The Mid function is used to extract the range references as strings, and they are converted to Range objects using the Range property.

The conversion of the strings to Range objects works even when the range references are not on the same sheet or in the same workbook as the embedded chart, as long as the source data is in an open workbook.

You could then proceed to manipulate the Range objects. You can change cell values in the ranges, for example, or extend or contract the ranges, once you have programmatic control over them. For illustration purposes, the code changes the color of the ranges in the worksheet by changing the ColorIndex property.

Chart Labels

In Excel, it is easy to add data labels to a chart as long as the labels are based on the data series values or X-axis values. These options are available in the Layout tab of the Ribbon under the Data Labels button in the Labels group.

You can also enter your own text as labels, but this involves a lot of manual work. You would need to add standard labels to the series, and then individually select each one and replace it with your own text. Alternatively, you can write a macro to do it for you.

Figure 8-7 shows a chart of sales figures for each month, with the name of the top salesperson for each month. The labels have been given the text entries from row 4 of the worksheet.

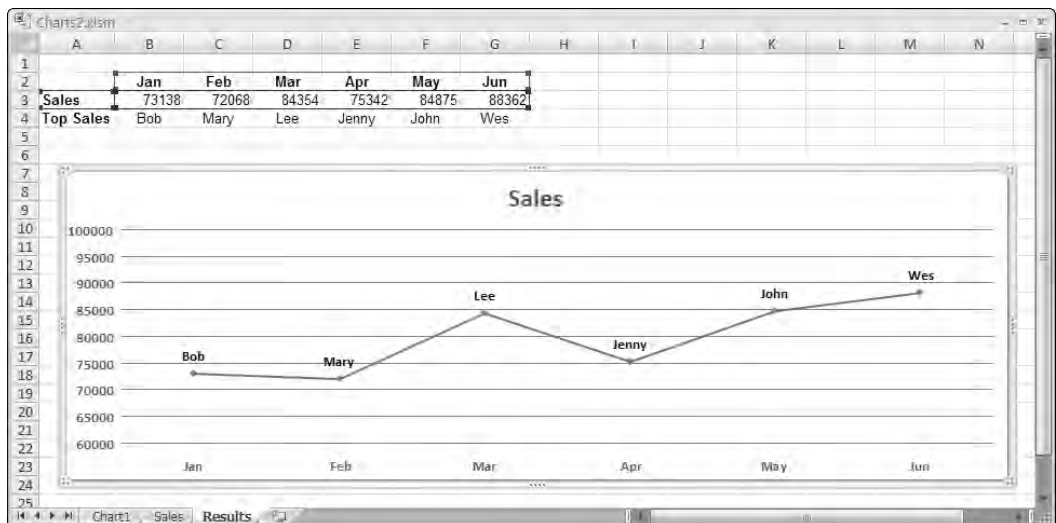


Figure 8-7

Chapter 8: Charts

Say you have set up a line chart like the one in Figure 8-7, but without the data labels. You can add the data labels using the following code:

```
Sub AddDataLabels()  
    Dim seSales As Series  
    Dim pts As Points  
    Dim pt As Point  
    Dim rngLabels As Range  
    Dim iPointIndex As Integer  
  
    'Specify cells containing labels  
    Set rngLabels = Range("B4:G4")  
  
    'Get first series from chart  
    Set seSales = ActiveSheet.ChartObjects(1).Chart.SeriesCollection(1)  
  
    'Enable labels  
    seSales.HasDataLabels = True  
  
    'Process each point in Points collection  
    Set pts = seSales.Points  
    For Each pt In pts  
  
        iPointIndex = iPointIndex + 1  
  
        pt.DataLabel.Text = rngLabels.Cells(iPointIndex).Text  
        pt.DataLabel.Font.Bold = True  
        pt.DataLabel.Position = xlLabelPositionAbove  
  
    Next pt  
  
End Sub
```

The object variable `rngLabels` is assigned a reference to B4:G4. `seSales` is assigned a reference to the first, and only, series in the embedded chart, and the `HasDataLabels` property of the series is set to `True`. The `For Each...Next` loop processes each `Point` object in the `Points` collection in the data series. For each point, the code assigns the `Text` property of the corresponding cell to the `Text` property of the point's data label. The data label is also made bold and the label is positioned above the data point.

Summary

It is easy to create a programmatic reference to a chart on a chart sheet. The `Chart` object is a member of the `Charts` collection of the workbook. To reference a chart embedded in a worksheet, you need to be aware that the `Chart` object is contained in a `ChartObject` object that belongs to the `ChartObjects` collection of the worksheet. `ChartObject` objects also belong to the `Shapes` collection of the worksheet.

You can move or resize an embedded chart by changing the `Top`, `Left`, `Width`, and `Height` properties of the `ChartObject`. If you already have a reference to the `Chart` object, you can get a reference to the `ChartObject` object through the `Parent` property of the `Chart` object.

Individual series in a chart are `Series` objects, and they belong to the `SeriesCollection` object of the chart. The `Delete` method of the `Series` object is used to delete a series from a chart. You use the `NewSeries` method of the `SeriesCollection` object to add a new series to a chart.

You can assign a VBA array, rather than the more commonly used `Range` object, to the `Values` property of a `Series` object. This creates a chart that is independent of worksheet data and can be distributed without a supporting worksheet.

The `Values` and `xValues` properties return data values, not the range references used in a chart. You can determine the ranges referenced by a chart by examining the `SERIES` function in the `Formula` property of each series.

The data points in a chart are `Point` objects and belong to the `Points` collection of the `Series` object. Excel does not provide an easy way to specify cell values as labels on series data points through the user interface. However, this can be easily done using VBA code.

Event Procedures

Excel makes it very easy for you to write code that runs when a range of worksheet, chart sheet, and workbook events occur. Previous chapters have shown you how to highlight the active row and column of a worksheet by placing code in the `Worksheet_SelectionChange` event procedure (see Chapter 1). This runs every time the user selects a new range of cells. You have also seen how to synchronize the worksheets in a workbook using the `Worksheet_Deactivate` and `Worksheet_Activate` events (see Chapter 3).

It is easy to create workbook, chart sheet, and worksheet events, because Excel automatically provides you with code modules for these objects. However, note that the chart events that are supplied automatically in a chart module apply only to chart sheets, not to embedded charts. If you want to write event procedures for embedded charts, you can do so, but it takes a bit more knowledge and effort.

Many other high-level events also can be accessed, for the `Application` object, for example. These events are covered later on in Chapters 16 and 26. Events associated with controls and forms are also discussed in their own chapters. This chapter examines in more detail worksheet, chart, and workbook events and related issues.

Event procedures are always associated with a particular object and are contained in the class module that is associated with that object, such as the `ThisWorkbook` module or the code module behind a worksheet or a UserForm. Don't try to place an event procedure in a standard module.

Worksheet Events

The following worksheet event procedures are available in the code module behind each worksheet:

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

Chapter 9: Event Procedures

- Private Sub Worksheet_BeforeRightClick(ByVal Target As Range, Cancel As Boolean)
- Private Sub Worksheet_Calculate()
- Private Sub Worksheet_Change (ByVal Target As Range)
- Private Sub Worksheet_Deactivate()
- Private Sub Worksheet_FollowHyperlink(ByVal Target As Hyperlink)
- Private Sub Worksheet_PivotTableUpdate (ByVal Target As PivotTable)
- Private Sub Worksheet_SelectionChange (ByVal Target As Range)

You should use the drop-down menus at the top of the code module to create the first and last lines of any procedure you want to use. For example, in a worksheet code module, you can select the `Worksheet` object from the left-hand drop-down list. This will generate the following lines of code:

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
...
End Sub
```

The `SelectionChange` event is the default event for the `Worksheet` object. If you want a different event, select the event from the right-hand drop-down list, and delete the preceding lines of code.

As an alternative to using the drop-downs, you can type the first line of the procedure yourself, but by doing so, it's easy to make mistakes. The arguments must correspond in number, order, and type with the arguments specified for each event procedure. You are permitted to use different parameter names if you wish, but it is better to stick with the standard names to avoid confusion.

Most parameters must be declared with the `ByVal` keyword, which prevents your code from passing back changes to the object or item referenced by assigning a new value to the parameter. If the parameter represents an object, you can change the object's properties and execute its methods, but you cannot pass back a change in the object definition by assigning a new object definition to the parameter.

Some event procedures are executed before the associated event occurs and have a `Cancel` parameter that is passed by reference. You can assign a value of `True` to the `Cancel` parameter to cancel the associated event. For example, you could prevent a user accessing the worksheet shortcut menu by canceling the `RightClick` event in the `Worksheet_BeforeRightClick` event procedure:

```
Private Sub Worksheet_BeforeRightClick(ByVal Target As Range, _
Cancel As Boolean)
    Cancel = True
End Sub
```

Enable Events

It is important to turn off event handling in some event procedures to prevent unwanted infinite recursion. For example, if a worksheet `Change` event procedure changes the worksheet, it will trigger the `Change` event and run itself again. The event procedure will change the worksheet again and trigger the `Change` event again, and so on.

If only one event procedure is involved, Excel 2007 will usually detect the recursion and terminate it after about 100 cycles. If more than one event procedure is involved, the process can continue indefinitely or until you press Esc or Ctrl+Break enough times to stop each process. It is even possible that Excel 2007 will crash, depending on how much RAM is available and what other code is doing.

For example, there could be a Calculation event procedure active as well as a Change event procedure. If both procedures change a cell that is referenced in a calculation, both events are triggered into an interactive chain reaction. That is, the first event triggers the second event, which triggers the first event again, and so on. The following Change event procedure makes sure that it does not cause a chain reaction by turning off event handling while it changes the worksheet. It is important to turn event handling back on again before the procedure ends:

```
Private Sub Worksheet_Change(ByVal Target As Range)
    Application.EnableEvents = False
    Range("A1").Value = 100
    Application.EnableEvents = True
End Sub
```

Application.EnableEvents = False does not affect events outside the Excel object model. Events associated with ActiveX controls and user forms, for example, will continue to occur.

Worksheet Calculate

The Worksheet_Calculate event occurs whenever the worksheet is recalculated. It is usually triggered when you enter new data into cells that are referenced in formulas in the worksheet. You could use the Worksheet_Calculate event to warn you, as you enter new data assumptions into a forecast, when key results go outside their expected range of values. In the Figure 9-1 worksheet, you want to know when the profit figure in cell N9 exceeds 600 or is lower than 500.

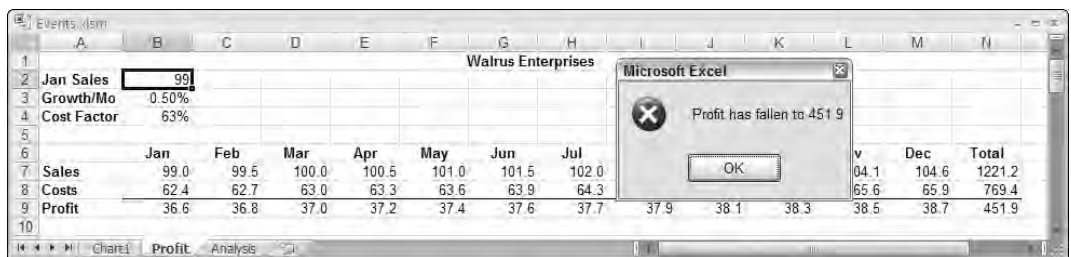


Figure 9-1

The following event procedure runs every time the worksheet recalculates, checks cell N9, which has been named FinalProfit, and generates messages if the figure goes outside the required band of values:

```
Private Sub Worksheet_Calculate()
    Dim dProfit As Double

    'After recalc access value in FinalProfit cell
```

```
dProfit = Me.Range("FinalProfit").Value

'Display value if outside range 500 to 600
If dProfit > 600 Then
    MsgBox "Profit has risen to " & Format(dProfit, "#,##0.0"), vbExclamation
ElseIf dProfit < 500 Then
    MsgBox "Profit has fallen to " & Format(dProfit, "#,##0.0"), vbCritical

End If

End Sub
```

Chart Events

The following chart event procedures are available in the code module for each chart object:

- Private Sub Chart_Activate()
- Private Sub Chart_BeforeDoubleClick (ByVal ElementID As Long, ByVal Arg1 As Long, ByVal Arg2 As Long, Cancel As Boolean)
- Private Sub Chart_BeforeRightClick (Cancel As Boolean)
- Private Sub Chart_Calculate()
- Private Sub Chart_Deactivate()
- Private Sub Chart_MouseDown (ByVal Button As XlMouseButton, ByVal Shift As Long, ByVal x As Long, ByVal y As Long)
- Private Sub Chart_MouseMove (ByVal Button As XlMouseButton, ByVal Shift As Long, ByVal x As Long, ByVal y As Long)
- Private Sub Chart_MouseUp (ByVal Button As XlMouseButton, ByVal Shift As Long, ByVal x As Long, ByVal y As Long)
- Private Sub Chart_Resize()
- Private Sub Chart_Select (ByVal ElementID As XlChartItem, ByVal Arg1 As Long, ByVal Arg2 As Long)
- Private Sub Chart_SeriesChange (ByVal SeriesIndex As Long, ByVal PointIndex As Long)

Before Double Click

Say you wanted to provide users with some shortcuts for formatting a chart. You could provide those shortcuts by trapping the double-click event and writing your own code to respond to the event.

The following event procedure formats three chart elements when they are double-clicked. If, in the chart shown in Figure 9-2, you double-click the legend, it is removed.

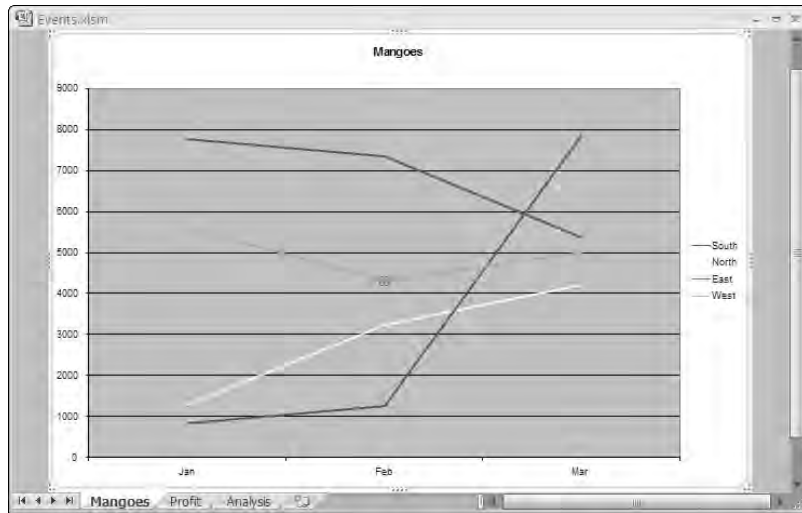


Figure 9-2

If you double-click the plot area (the area containing the plotted lines), the legend is displayed. If you double-click a series line with all points selected, it changes the color of the line. If a single point in the series is selected, the data label at the point is toggled on and off:

```
Private Sub Chart_BeforeDoubleClick(ByVal ElementID As Long, _
    ByVal Arg1 As Long, ByVal Arg2 As Long, Cancel As Boolean)

    Dim seSeries As Series

    'Determine which chart element has been double clicked
    Select Case ElementID

        'If Legend double clicked - remove it
        Case xlLegend
            Me.HasLegend = False
            Cancel = True

        'If plot area - add legend
        Case xlPlotArea
            Me.HasLegend = True
            Cancel = True

        'If a data series
        Case xlSeries
            'Arg1 is the Series index
            'Arg2 is the Point index (-1 if the entire series is selected)
            Set seSeries = Me.SeriesCollection(Arg1)

            'If series selected - change line color
```

```
    If Arg2 = -1 Then
        With seSeries.Border
            If .ColorIndex = xlColorIndexAutomatic Then
                .ColorIndex = 1
            Else
                .ColorIndex = (.ColorIndex Mod 56) + 1
            End If
        End With
    Else
        'If point selected - toggle data label
        With seSeries.Points(Arg2)
            .HasDataLabel = Not .HasDataLabel
        End With
    End If
    Cancel = True

End Select

End Sub
```

The `ElementID` parameter passes an identifying number to indicate the element that was double-clicked. You can use intrinsic constants, such as `xlLegend`, to determine the element. At the end of each case, `Cancel` is assigned `True` so any default double-click event is canceled.

When restoring the legend, it would have been more intuitive to double-click the chart area where the legend is placed. Unfortunately, the initial release of Excel 2007 does not respond to most chart area events. Presumably this will be fixed in a service release before too long.

Note the use of the keyword `Me` to refer to the object associated with the code module. Using `Me` instead of `Chart1` makes the code portable to other charts. In fact, you can omit the object reference "`Me.` " and use "`HasLegend =` ". In a class module for an object, you can refer to properties of the object without qualification. However, qualifying the property makes it clear that you have created a property and not a variable.

If the chart element is a series, `Arg1` contains the series index in the `SeriesCollection`, and if a single point in the series has been selected, `Arg2` contains the point index. `Arg2` is `-1` if the whole series is selected.

If the whole series is selected, the event procedure assigns 1 to the color index of the series border, if the color index is automatic. If the color index is not automatic, it increases the color index by 1. To limit the choice to 56 colors, the procedure uses the `Mod` operator, which divides the color index by 56 and gives the remainder, before adding 1. The only color index value that is affected by this is 56. `56 Mod 56` returns 0, which means that the next color index after 56 is 1.

If a single point is selected in the series, the procedure toggles the data label for the point. If the `HasDataLabel` property of the point is `True`, `Not` converts it to `False`. If the `HasDataLabel` property of the point is `False`, `Not` converts it to `True`.

Workbook Events

The following workbook event procedures are available:

- `Private Sub Workbook_Activate()`
- `Private Sub Workbook_AddinInstall()`
- `Private Sub Workbook_AddinUninstall()`
- `Private Sub Workbook_AfterXmlExport (ByVal Map As XmlMap, ByVal Url As String, ByVal Result As XmlExportResult)`
- `Private Sub Workbook_AfterXmlImport (ByVal Map As XmlMap, ByVal IsRefresh As Boolean, ByVal Result As XmlImportResult)`
- `Private Sub Workbook_BeforeClose (Cancel As Boolean)`
- `Private Sub Workbook_BeforePrint (Cancel As Boolean)`
- `Private Sub Workbook_BeforeSave (ByVal SaveAsUI As Boolean, Cancel As Boolean)`
- `Private Sub Workbook_BeforeXmlExport (ByVal Map As XmlMap, ByVal Url As String, Cancel As Boolean)`
- `Private Sub Workbook_BeforeXmlImport (ByVal Map As XmlMap, ByVal Url As String, ByVal IsRefresh As Boolean, Cancel As Boolean)`
- `Private Sub Workbook_Deactivate()`
- `Private Sub Workbook_NewSheet (ByVal Sh As Object)`
- `Private Sub Workbook_Open()`
- `Private Sub Workbook_PivotTableCloseConnection (ByVal Target As PivotTable)`
- `Private Sub Workbook_PivotTableOpenConnection (ByVal Target As PivotTable)`
- `Private Sub Workbook_RowsetComplete (ByVal Description As String, ByVal Sheet As String, ByVal Success As Boolean)`
- `Private Sub Workbook_SheetActivate (ByVal Sh As Object)`
- `Private Sub Workbook_SheetBeforeDoubleClick (ByVal Sh As Object, ByVal Target As Range, Cancel As Boolean)`
- `Private Sub Workbook_SheetBeforeRightClick (ByVal Sh As Object, ByVal Target As Range, Cancel As Boolean)`
- `Private Sub Workbook_SheetCalculate (ByVal Sh As Object)`
- `Private Sub Workbook_SheetChange (ByVal Sh As Object, ByVal Target As Range)`
- `Private Sub Workbook_SheetDeactivate (ByVal Sh As Object)`
- `Private Sub Workbook_SheetFollowHyperlink (ByVal Sh As Object, ByVal Target As Hyperlink)`
- `Private Sub Workbook_SheetPivotTableUpdate (ByVal Sh As Object, ByVal Target As PivotTable)`

Chapter 9: Event Procedures

- ❑ `Private Sub Workbook_SheetSelectionChange (ByVal Sh As Object, ByVal Target As Range)`
- ❑ `Private Sub Workbook_Sync (ByVal SyncEventType As Office.MsoSyncEventType)`
- ❑ `Private Sub Workbook_WindowActivate (ByVal Wn As Window)`
- ❑ `Private Sub Workbook_WindowDeactivate (ByVal Wn As Window)`
- ❑ `Private Sub Workbook_WindowResize (ByVal Wn As Window)`

Some of the workbook event procedures are the same as the worksheet and chart event procedures. The difference is that when you create these procedures (such as the `Change` event procedure) in a worksheet or chart, it applies to only that sheet. When you create a workbook event procedure (such as the `SheetChange` event procedure), it applies to all the sheets in the workbook.

One of the most commonly used workbook event procedures is the `Open` event procedure. This is used to initialize the workbook when it opens. You can use it to set the calculation mode, establish screen settings, alter the Ribbon, or enter data into combo boxes or list boxes in the worksheets.

Similarly, the `Workbook_BeforeClose` event procedure can be used to tidy up when the workbook is closed. It can restore screen and option settings, for example. It can also be used to prevent a workbook's closure by setting `Cancel` to `True`. The following event procedure will only allow the workbook to close if the figure in the cell named `FinalProfit` is between 500 and 600:

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)
    Dim dProfit As Double

    dProfit = ThisWorkbook.Worksheets(1).Range("FinalProfit").Value
    If dProfit < 500 Or dProfit > 600 Then
        MsgBox "Profit must be in the range 500 to 600"
        Cancel = True
    End If
End Sub
```

Note that if you assign `True` to `Cancel` in the workbook `BeforeClose` event procedure, you also prevent Excel from closing.

Save Changes

If you want to make sure that all changes are saved when the workbook closes, but you don't want the user to be prompted to save changes, you can save the workbook in the `BeforeClose` event procedure. You can check to see if this is really necessary using the `Saved` property of the workbook, which will be `False` if there are unsaved changes:

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)
    If Not ThisWorkbook.Saved Then
        ThisWorkbook.Save
    End If
End Sub
```


If, on the other hand, you want to discard any changes to the workbook, and you don't want users to be prompted to save changes in a workbook when they close it, you can set the `Saved` property of the workbook to `True` in the `BeforeClose` event procedure:

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)
    ThisWorkbook.Saved = True
End Sub
```

This fools Excel into thinking that any changes have been saved.

Headers and Footers

A common need in Excel is to print information in the page header or footer that either comes from the worksheet cells or is not available in the standard header and footer options. You might want to insert a company name that is part of the data in the worksheet and display the full path to the workbook file.

The filename is available as an option in headers and footers in Excel 2007. It can be inserted using the code `&F`, as shown in the following code. Data can be accessed from worksheet cells in the usual way. You can insert this information using the `BeforePrint` event procedure to ensure it is always up-to-date in reports. The following procedure puts the text in cell A1 of the worksheet named `Profit` in the left footer, clears the center footer, and puts the filename in the right footer. It applies the changes to every worksheet in the file:

```
Private Sub Workbook_BeforePrint(Cancel As Boolean)
    Dim wks As Worksheet
    Dim sFileName As String
    Dim sCompanyName As String

    'Set up values & codes for footer
    sCompanyName = Worksheets("Profit").Range("A1").Value
    sFileName = "&F" 'Code generating file name

    For Each wks In ThisWorkbook.Worksheets

        'Define footer
        With wks.PageSetup
            .LeftFooter = sCompanyName
            .CenterFooter = ""
            .RightFooter = sFileName
        End With

    Next wks

End Sub
```

The footer can be seen in Page Layout View, as shown in Figure 9-3.

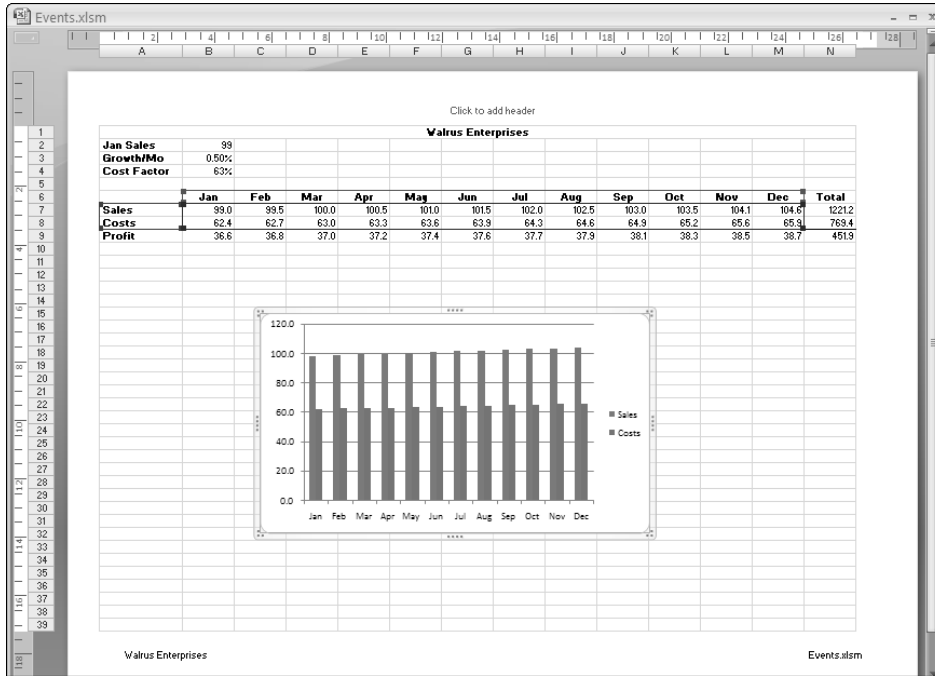


Figure 9-3

Summary

In this chapter, you have seen some useful examples of how to utilize event procedures to respond to user actions.

You have been introduced to worksheet, chart, and workbook events, delving a little deeper into the following events:

- Worksheet_Calculate
- Chart_BeforeDoubleClick
- Workbook_BeforeClose
- Workbook_BeforePrint

VBA is essentially an event-driven language, so a good knowledge of the events at your disposal can open up a whole new world of functionality you never knew existed.

To find out more, have a play with the Object Browser, and consult the object model in Appendix A.

10

Adding Controls

As discussed in Chapter 1, you can add two different types of controls to Excel worksheets: ActiveX controls or Form controls. The Form controls originated in Excel 5 and Excel 95 and provide controls for the dialog sheets used in those versions, as well as controls embedded in a worksheet or chart. Dialog sheets have been superseded by UserForms since the release of Excel 97, and UserForms utilize the ActiveX controls.

You can create controls by activating the Developer tab of the Ribbon and clicking the Insert button in the Controls group. This displays the dialog box shown in Figure 10-1.

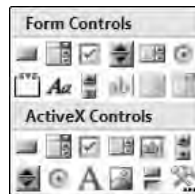


Figure 10-1

Form and ActiveX Controls

The Form controls and dialog sheets are still supported in Excel. Form controls even have some advantages over the ActiveX controls. They are less complex than the ActiveX controls. However, each Form control can only respond to a single event. In most cases, that event is the `Click` event—the edit box is an exception, responding to the `Change` event.

If you want to create controls and define their event procedures in your VBA code, as opposed to creating them manually, the Form controls are easier to work with. A big advantage over an ActiveX control is that the event procedure for a Form control can be placed in a standard module, can have any valid VBA procedure name, and can be created when you write the code for the application, before the control is created.

Chapter 10: Adding Controls

You can create the control programmatically, when it is needed, and assign the procedure name to the `OnAction` property of the control. You can even assign the same procedure to more than one control. On the other hand, ActiveX event procedures must be placed in the class module behind the worksheet or user form in which they are embedded, and must have a procedure name that corresponds with the name of the control and the name of the event. For example, the click event procedure for a control named `OptionButton1` must be as follows:

```
Sub OptionButton1_Click()
```

If you try to create an event procedure for an ActiveX control before the control exists, and you try to reference that control in your code, you will get compiler errors, so you have to create the event procedure programmatically. This is not an easy task, as you will see in later sections.

In addition, see Chapter 26 for an example of adding an event procedure programmatically to a UserForm control.

On the other hand, a procedure that is executed by a Form control does not need to have a special name and can use the `Caller` property of the `Application` object to obtain a reference to the control that executes it. The control name does not need to be included in the name of the procedure or in references to the control, as you will see later in this chapter.

ActiveX Controls

Figure 10-2 shows four types of ActiveX controls embedded in the worksheet. The scrollbar in cells C3:F3 allows you to set the value in cell B3. The spin button in cell C4 increments the growth percentage in cell B4. The checkbox in cell B5 increases the tax rate in cell B16 from 30% to 33%, if it is checked. The option buttons in column I change the cost factor in cell B15, and also change the maximum and minimum values for the scrollbar.

	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Total
Jan Sales	150,000												
Sales Growth/Mo	1.00%												
Tax Increase?	<input checked="" type="checkbox"/>												
Sales	150,000	151,500	153,015	154,545	156,091	157,652	159,228	160,820	162,429	164,053	165,693	167,350	1,902,375
Costs	94,500	95,445	96,399	97,363	98,337	99,320	100,314	101,317	102,330	103,353	104,387	105,431	1,198,497
Profit Before Tax	55,500	56,055	56,616	57,182	57,754	58,331	58,914	59,504	60,099	60,700	61,307	61,920	703,879
Tax	18,315	18,498	18,683	18,870	19,059	19,249	19,442	19,636	19,833	20,031	20,231	20,433	232,280
Profit After Tax	37,185	37,557	37,932	38,312	38,695	39,082	39,473	39,867	40,266	40,669	41,075	41,486	471,599
Cost Factor	63%												
Tax Rate	33%												

Figure 10-2

An ActiveX control can be linked to a worksheet cell, using its `LinkedCell` property, so that the cell always displays the `Value` property of the control. None of the ActiveX controls shown in Figure 10-2 use a link cell, although the scrollbar could have been linked because its value is displayed in B3. Each control uses an event procedure to make its updates. This gives you far more flexibility than a simple cell link and removes the need to dedicate a worksheet cell to the task.

Scrollbar Control

The scrollbar uses the `Change` event and the `Scroll` event to assign the `Value` property of the scrollbar to cell B3. The maximum and minimum values of the scrollbar are set by the option buttons (this is discussed later):

```
Private Sub ScrollBar1_Change()  
  
    'Assign scrollbar value to B3  
    Range("B3").Value = ScrollBar1.Value  
  
End Sub
```

The `Change` event procedure is triggered when the scrollbar value is changed by clicking the scroll arrows, by clicking above or below the scroll box (or to the left or right if it is aligned horizontally), or by dragging the scroll box. However, a small glitch occurs immediately after you change the option buttons. Dragging the scroll box does not trigger the `Change` event on the first attempt. Utilizing the `Scroll` event procedure solves this problem.

The `Scroll` event causes continuous updating as you drag the scrollbar, so you can see what figure you are producing as you drag, rather than after you have released the scrollbar. It might not be practical to use the `Scroll` event procedure in a very large worksheet in auto-recalculation mode because of the large number of recalculations it causes.

Spin Button Control

The spin button control uses the `SpinDown` and `SpinUp` events to decrease and increase the value in cell B4:

```
'Spin down event procedure for spin button  
Private Sub SpinButton1_SpinDown()  
  
    With Range("B4")  
  
        'Decrease value in B4 by .05%. Stop at 0%  
        .Value = WorksheetFunction.Max(0, .Value - 0.0005)  
  
    End With  
  
End Sub  
  
'Spin up event procedure for spin button
```

Chapter 10: Adding Controls

```
Private Sub SpinButton1_SpinUp()  
  
    With Range("B4")  
  
        'Increase value in B4 by .05%. Stop at 1%  
        .Value = WorksheetFunction.Min(0.01, .Value + 0.0005)  
  
    End With  
  
End Sub
```

The `Value` property of the spin button is ignored. It is not suitable to be used directly as a percentage figure because it can only be a long integer value. The events are used as triggers to run the code that operates directly on the value in B4. The growth figure is kept in the range of zero to 1 percent.

Clicking the down side of the spin button runs the `SpinDown` event procedure, which decreases the value in cell B4 by 0.05%. The worksheet `Max` function is used to ensure that the calculated figure does not become less than zero. The `SpinUp` event procedure increases the value in cell B4 by 0.05%. It uses the `Min` function to ensure that the calculated value does not exceed 1%.

CheckBox Control

The `CheckBox` control returns a `True` value when checked, or a `False` value if it is unchecked. The `Click` event procedure that follows uses an `If` structure to set the value in cell B16:

```
Private Sub CheckBox1_Click()  
  
    If CheckBox1.Value Then  
  
        'If check box ticked, tax rate is 33%  
        Range("B16").Value = 0.33  
  
    Else  
  
        'If check box not ticked, tax rate is 30%  
        Range("B16").Value = 0.3  
  
    End If  
  
End Sub
```

Option Button Controls

Each option button has code similar to the following:

```
Private Sub OptionButton1_Click()  
    Call Options  
End Sub
```

The processing for all the buttons is carried out in the following procedure, which is in the class module behind the `Profit` worksheet that holds the previous event procedures:

```

Private Sub Options()
    Dim dCostFactor As Double
    Dim lScrollBarMax As Long
    Dim lScrollBarMin As Long

    'Determine which option button is True
    Select Case True

        'Mangoes
        Case OptionButton1.Value
            dCostFactor = 0.63
            lScrollBarMin = 50000
            lScrollBarMax = 150000

        'Lychees
        Case OptionButton2.Value
            dCostFactor = 0.74
            lScrollBarMin = 25000
            lScrollBarMax = 75000

        'Bananas
        Case OptionButton3.Value
            dCostFactor = 0.57
            lScrollBarMin = 10000
            lScrollBarMax = 30000

        'Rambutan
        Case OptionButton4.Value
            dCostFactor = 0.65
            lScrollBarMin = 15000
            lScrollBarMax = 30000

    End Select

    'Apply factors
    Range("B15").Value = dCostFactor
    ScrollBar1.Min = lScrollBarMin
    ScrollBar1.Max = lScrollBarMax
    ScrollBar1.Value = lScrollBarMax

End Sub

```

The `Select Case` structure is used here in an unusual way. Normally you use a variable reference in the first line of a `Select Case` and use comparison values in the `Case` statements. Here, you used the value `True` in the `Select Case` and referenced the option button `Value` property in the `Case` statements. This provides a nice structure for processing a set of option buttons where you know that only one can have a `True` value.

Only one option button can be selected and have a value of `True` in the preceding worksheet, because they all belong to the same group. As you add option buttons to a worksheet, the `GroupName` property of the button is set to the name of the worksheet—`Profit`, in this case. If you want two sets of unrelated option buttons, you need to assign a different `GroupName` to the second set.

Options uses the `Select Case` structure to carry out any processing that is different for each option button. The code following the `End Select` carries out any processing that is common to all the option buttons. This approach also works very well when the coding is more complex, and also when the code is triggered by another control, such as a command button, rather than the option button events.

Forms Controls

Figure 10-3 shows a Form control that is being used to select a product name to be entered in column D. The control appears over any cell in column D that you double-click. When you select the product, the product name is entered in the cell “behind” the control, the price of the product is entered in column F on the same row, and the control disappears.



The screenshot shows an Excel spreadsheet with a table of sales data. The columns are labeled A through H, and the rows are numbered 1 through 22. The data is as follows:

	A	B	C	D	E	F	G	H
1	Date	Customer	State	Product	NumberSold	Price	Revenue	
2	Jan 01, 2007	Roberts	NSW	Bananas	903	\$15.00	\$13,545.00	
3	Jan 01, 2007	Roberts	TAS	Bananas	331	\$15.00	\$4,965.00	
4	Jan 02, 2007	Smith	QLD	Mangoes	299	\$20.00	\$5,980.00	
5	Jan 06, 2007	Roberts	QLD	Bananas	612	\$15.00	\$9,180.00	
6	Jan 08, 2007	Roberts	VIC	Lychees	907	\$12.50	\$11,337.50	
7	Jan 08, 2007	Pradesh	TAS	Rambutan	107	\$18.00	\$1,926.00	
8	Jan 10, 2007	Roberts	VIC	Lychees	770	\$12.50	\$9,625.00	
9	Jan 14, 2007	Smith	NT	Lychees	223	\$12.50	\$2,787.50	
10	Jan 14, 2007	Smith	VIC	Bananas	132	\$15.00	\$1,980.00	
11	Jan 15, 2007	Pradesh	QLD	Bananas	669	\$15.00	\$10,035.00	
12	Jan 17, 2007	Roberts	NSW	Mangoes	881	\$20.00	\$17,620.00	
13	Jan 21, 2007	Kee	SA	Rambutan	624	\$18.00	\$11,232.00	
14	Jan 22, 2007	Roberts	QLD	Rambutan	193	\$20.00	\$3,860.00	
15	Jan 23, 2007	Smith	SA	Mangoes	255	\$20.00	\$5,100.00	
16	Jan 27, 2007	Kee	QLD	Manqoes	6	\$20.00	\$120.00	
17								
18								
19								
20								
21								
22								

A dropdown menu is open over the 'Product' column in row 17, showing the following options: Bananas, Lychees, Mangoes, and Rambutan. The spreadsheet is titled 'Controls.xlsx' and the active sheet is 'SalesData'.

Figure 10-3

If you hover your cursor over the Form button that creates the control shown in Figure 10-3, the ScreenTip that pops up describes this control as a `ComboBox`. However, in the Excel object model, it is called a `DropDown` object, and it belongs to the `DropDowns` collection.

The `DropDown` object is a hidden member of the Excel object model in Excel 97 and later versions. You will not find any help screens for this object, and it will not normally appear in the Object Browser. You can make it visible in the Object Browser if you right-click in the Object Browser window and select Show Hidden Members from the shortcut menu. You can learn a lot about the Forms toolbar controls by using the macro recorder and the Object Browser, but you will need to have access to Excel 5 or Excel 95 to get full documentation on them.

The DropDown control is created by a procedure called from the following BeforeDoubleClick event procedure in the SalesData sheet, which has the programmatic name Sheet2:

```
Private Sub Worksheet_BeforeDoubleClick(ByVal Target As Range, Cancel As Boolean)
    If Not Intersect(Target, Columns("D")) Is Nothing Then
        Call AddDropDown(Target)
        Cancel = True
    End If
End Sub
```

The event procedure checks that Target (the cell that was double-clicked) is in column D. If so, it then runs the AddDropDown procedure, passing Target as an input argument, and cancels the double-click event.

The following two procedures are in a standard module:

```
Sub AddDropDown(Target As Range)
    Dim ddBox As DropDown
    Dim vProducts As Variant
    Dim i As Integer

    'Create array of products
    vProducts = Array("Bananas", "Lychees", "Mangoes", "Rambutan")

    'Add the drop down control in Target cell
    With Target
        Set ddBox = Sheet2.DropDowns.Add(.Left, .Top, .Width, .Height)
    End With

    'Define macro to run and populate list
    With ddBox
        .OnAction = "EnterProdInfo"
        For i = LBound(vProducts) To UBound(vProducts)
            .AddItem vProducts(i)
        Next i
    End With
End Sub

Private Sub EnterProdInfo()
    Dim vPrices As Variant

    'Create array of prices
    vPrices = Array(15, 12.5, 20, 18)

    'Enter selected item into cell beneath drop down
    With Sheet2.DropDowns(Application.Caller)
        .TopLeftCell.Value = .List(.ListIndex)
        .TopLeftCell.Offset(0, 2).Value = vPrices(.ListIndex + LBound(vPrices) - 1)

        'Delete drop down
        .Delete
    End With
End Sub
```

Chapter 10: Adding Controls

The `AddDropDown` procedure is not declared `Private`, because it would not then be possible to call it from the `Sheet2` code module. This would normally be a problem if you wanted to prevent users from seeing the procedure in the Macro dialog box. However, because it has an input argument, it will not be shown in the dialog box anyway. Also, it does not matter whether `AddDropDown` is placed in the `Sheet2` module or a standard module. It will operate in either location.

`AddDropDown` uses the `Add` method of the `DropDowns` collection to create a new drop-down. It aligns the new control exactly with `Target`, giving it the same `Left`, `Top`, `Width`, and `Height` properties as the cell. In the `With...End With` construction, the procedure defines the `OnAction` property of the drop-down to be the `EnterProdInfo` procedure. This means that `EnterProdInfo` will be run when an item is chosen from the drop-down. The `For...Next` loop uses the `AddItem` method of the drop-down to place the list of items in `vProducts` into the drop-down list.

`EnterProdInfo` has been declared `Private` to prevent its appearance in the Macro dialog box. Although it is private, the drop-down can access it. `EnterProdInfo` could have been placed in the `Sheet2` code module, but the `OnAction` property of the drop-down would have to be assigned `Sheet2.EnterProdInfo`.

`EnterProdInfo` loads `vPrices` with the prices corresponding to the products. It then uses `Application.Caller` to return the name of the drop-down control that called the `OnAction` procedure. It uses this name as an index into the `DropDowns` collection on `Sheet2` to get a reference to the `DropDown` object itself. In the `With...End With` construction, `EnterProdInfo` uses the `ListIndex` property of the `DropDown` object to get the index number of the item chosen in the drop-down list.

You cannot directly access the name of the chosen item in a `DropDown` object, unlike an `ActiveX` `ComboBox` object that returns the name in its `Value` property. The `Value` property of a drop-down is the same as the `ListIndex`, which returns the numeric position of the item in the list. To get the item name from a drop-down, you use the `ListIndex` property as a one-based index to the `List` property of the drop-down. The `List` property returns an array of all the items in the list.

The `TopLeftCell` property of the `DropDown` object returns a reference to the `Range` object under the top-left corner of the `DropDown` object. `EnterProdInfo` assigns the item chosen in the list to the `Value` property of this `Range` object. It then assigns the price of the product to the `Range` object that is offset two columns to the right of the `TopLeftCell` `Range` object.

`EnterProdInfo` also uses the `ListIndex` property of the drop-down as an index into the `Prices` array. The problem with this is that the drop-down list is always one-based, whereas the `Array` function list depends on the `Option Base` statement in the declarations section of the module. `LBound(vPrices) - 1` is used to reduce the `ListIndex` value by 1 if `Option Base 0` is in effect or by 0 if `Option Base 1` is in effect.

You can also use the following code to ensure that the resulting array is zero-based under `Option Base 1`:

```
vPrices = VBA.Array(15, 12.5, 20, 18)
```

Dynamic ActiveX Controls

As previously stated, it is more difficult to program the `ActiveX` controls than the `Form` controls. At the same time, the `ActiveX` controls are more powerful, so it is a good idea to know how to program them.

You will see how to construct a combo box that behaves in a similar way to the drop-down in the previous example. Just to be different, use the `BeforeRightClick` event to trigger the appearance of a combo box in the D column of the `SalesData` worksheet, as follows:

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

    Dim ole As OLEObject
    Dim ctl As MSForms.ComboBox
    Dim lLine As Long
    Dim oCodeModule As Object

    'Turn screen updating off
    Application.ScreenUpdating = False

    'Determine if the combo box should be built
    If Intersect(ActiveCell, Columns("D")) Is Nothing Then Exit Sub
    On Error Resume Next
    Set ole = Me.OLEObjects("Combo")
    If Not ole Is Nothing Then
        Cancel = True
        Exit Sub
    End If
    On Error GoTo 0

    'Add the combo box to the active cell
    With ActiveCell
        Set ole = Me.OLEObjects.Add(ClassType:="Forms.ComboBox.1", _
                                   Link:=False, DisplayAsIcon:=False, Left:=.Left, Top:=.Top, _
                                   Width:=.Width, Height:=.Height)
    End With

    ole.Name = "Combo"
    Set ctl = ole.Object
    ctl.Name = "Combo"

    With ctl
        .AddItem "Bananas"
        .AddItem "Lychees"
        .AddItem "Mangoes"
        .AddItem "Rambutan"
    End With

    'Build the event procedure for the combo box click event
    Set oCodeModule = ThisWorkbook.VBProject.VBComponents(Me.CodeName).CodeModule
    With oCodeModule
        lLine = .CreateEventProc("Click", "Combo")
        .ReplaceLine lLine + 1, " ProcessComboClick"
    End With
    Cancel = True

    'Make sure the Excel window is active
    Application.Visible = False
    Application.Visible = True

End Sub
```

Chapter 10: Adding Controls

First, check to see that the event took place in the D column. Also, check to make sure that there is no existing combo box in the worksheet, which would mean that the user has created a combo box but has not yet selected an item from it. This did not matter in the previous example, where the combo boxes were independent even though they used the same `OnAction` code. Your ActiveX controls can't share the single `Click` event procedure you are going to create, so you need to ensure that you don't already have a control in the worksheet.

Use the name `Combo` for your ActiveX control. The quickest way to determine if there is already a control called `Combo` is to create an object variable referring to it. If this attempt fails, you know that the control does not exist. The error recovery code is used to ensure that the macro does not display an error message and stop running if the control does not exist. It would be friendlier to display an explanatory message before exiting the sub, but that is not the main point of this exercise. Setting `Cancel` to `True` suppresses the normal right-click menu from appearing.

If all is well, add a new combo box in the active cell. You need to know that an ActiveX object is not added directly onto a worksheet. It is contained in an `OLEObject` object, in the same way that a chart embedded in a worksheet is contained in a `ChartObject` object, as you saw in Chapter 8. The return value from the `Add` method of the `OLEObjects` collection is assigned to `ole` to make it easy to refer to the `OLEObject` object later. The `Name` property of the `OleObject` is changed to `Combo` to make it easy to identify later.

Next, create an object variable, `ctl`, referring to the `ComboBox` object contained in the `OleObject`, which is returned by the `Object` property of the `OLEObject`. The next line of code assigns the name `Combo` to the `ComboBox` object. This is not necessary in Excel 2007. When you assign a name to the `OLEObject`, it is also automatically assigned to the embedded object in these versions. This is not the case in Excel 97, where the name needs to be explicitly assigned.

Now you need to create the list of items to appear when the `ComboBox` is clicked. This can be done using the `AddItem` method, which needs to be executed for each item in the list.

Now create the `Click` event procedure code for the combo box. You can't create the event procedure in advance. It will cause compile errors if the ActiveX control it refers to does not exist. The methodology for creating event procedures programmatically is explained in detail in Chapter 26, so check that chapter for full details.

`oCodeModule` is assigned a reference to the class module behind the worksheet, and the `CreateEventProc` method of the code module is used to enter the first and last lines of the `Combo_Click` event procedure, with a blank line between. This method returns the line number of the first line of the procedure, which is assigned to `lLine`. The `ReplaceLine` method replaces the blank second line of the procedure with a call to a sub procedure named `ProcessComboClick`, which is listed in the next code snippet. The code for `ProcessComboClick` already exists in the worksheet's code module.

Set `Cancel` to `True` to ensure that the popup menu normally associated with a right-click in a cell does not appear.

Unfortunately, when you add code to a code module as done here, the code module is activated and the user could be left staring at a screen full of code. By setting the Excel application window's `Visible` property to `False` and then `True`, you ensure that the Excel window is active at the end

of the procedure. There will still be some screen flicker, even though screen updating was suppressed at the start of the macro. It is possible to suppress this flicker by calls to the Windows API (discussed in Chapter 27).

The `Click` event procedure code that is created by the preceding code looks like the following:

```
Private Sub Combo_Click()
    ProcessComboClick
End Sub
```

When the user selects a value in the combo box, the `Click` event procedure executes and, in turn, executes `ProcessComboClick`. The code for `ProcessComboClick`, which is a permanent procedure in the worksheet's code module, contains the following:

```
Private Sub ProcessComboClick()
    Dim lLine As Long
    Dim lCount As Long
    Dim oCodeModule As Object

    'Enter the chosen value
    With Me.OLEObjects("Combo")
        .TopLeftCell.Value = .Object.Value
        .Delete
    End With

    'Delete the combo box click event procedure
    Set oCodeModule = _
        ThisWorkbook.VBProject.VBComponents(Me.CodeName).CodeModule

    With oCodeModule
        lLine = .ProcStartLine("Combo_Click", 0)
        lCount = .ProcCountLines("Combo_Click", 0)
        .DeleteLines lLine, lCount
    End With

End Sub
```

The combo box is the object contained in the `OLEObject` named `Combo`. The preceding code enters the selected value from the combo box into the cell underneath the combo box, and then deletes the `OLEObject` and its contents.

The code then deletes the event procedure. `oCodeModule` is assigned a reference to the worksheet's code module. The `ProcStartLine` method returns the line number of the `Combo_Click` event procedure, including any blank lines or comment lines that precede it. The `ProcCountLines` method returns the number of lines in the procedure, including the lines that precede it. The `Delete` method removes all the lines in the procedure and the blank lines that precede it.

As you can see, dynamically coding an ActiveX control is quite complex. It is simpler to use a Forms toolbar control if you don't really need the extra power of an ActiveX control.

Controls on Charts

Figure 10-4 shows a chart that contains a button to remove or add the profit series from the chart, which is based on the Profit Planner figures of the Profit sheet. The control is a Forms toolbar Button object belonging to the Buttons collection (remember, ActiveX controls cannot be used in charts).

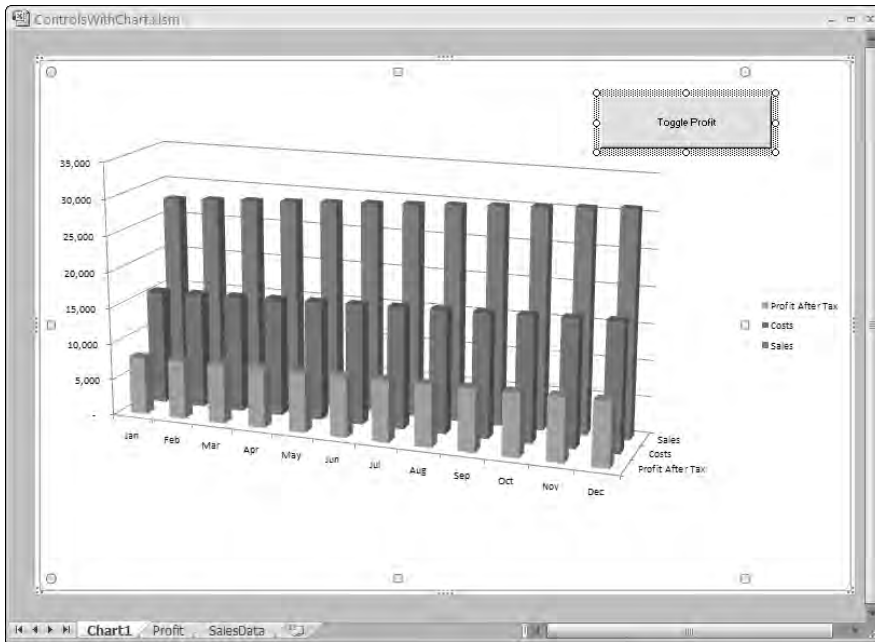


Figure 10-4

The code assigned to the OnAction property of the Button object is as follows:

```
Sub Button1_Click()  
  With ActiveChart  
    If .SeriesCollection.Count = 3 Then  
      .SeriesCollection(1).Delete  
    Else  
      With .SeriesCollection.NewSeries  
        .Name = Sheet1.Range("A13")  
        .Values = Sheet1.Range("B13:M13")  
        .XValues = Sheet1.Range("B12:M12")  
        .PlotOrder = 1  
      End With  
    End If  
  End With  
End Sub
```

If the `SeriesCollection.Count` property is 3, the first series is deleted. Otherwise, a new series is added and the appropriate ranges are assigned to it to show the profit after tax figures. The new series is added as the last series, which would plot behind the existing series, so the `PlotOrder` property of the new series is set to 1 to place it in front of the others.

Summary

This chapter explained some of the differences between ActiveX controls embedded in worksheets and Form controls embedded in worksheets, and you've also seen how to work with them. You have seen how scrollbars, spin buttons, checkboxes, and option buttons can be used to execute macros that can harness the full power of VBA and that do not need to depend on a link cell.

In addition, this chapter demonstrated how Form controls and ActiveX controls can be created and manipulated programmatically. ActiveX controls are more difficult to create programmatically, because they are contained within `OLEObjects` and you can't create their event procedures in advance.

Text Files and File Dialog

Text files provide one way to communicate information between different types of computers. There is no universal format for the binary files that usually provide the most efficient format for working within a particular computer system, so text files are often used for this. This chapter examines how to create text files and how to read them.

Increasingly, XML is becoming the standard way to exchange data across the Internet. XML files are text files, but in a highly organized format. They are covered in Chapter 12. However, there are many legacy systems, particularly where mainframe computers are concerned, where text files in a variety of formats are used.

Excel is capable of importing text files and can save data in .csv (comma separated variables) and .prn (print) files, as well as other formats. Often these features are not flexible enough to cater to specific needs. Using VBA, you can produce text files in whatever format you like and read text files in whatever format is provided.

This chapter also discusses the `FileDialog` object, which allows you to display the Office dialogs for opening and saving files and browsing folders.

Opening Text Files

Before you can read or write to a file, you need to open it using the `Open` statement. When opening a text file for sequential access, `Open` has the following options:

```
Open file [For mode] As [#]filename
```

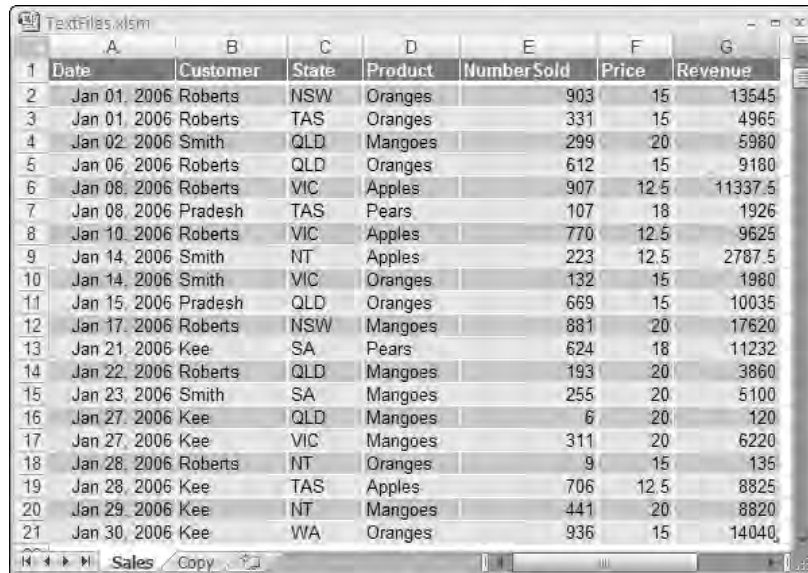
The mode can be `Input`, `Output`, or `Append`. If you specify `Output` and there is an existing file with the same name, it will be overwritten. You can use `Append` to write to the end of an existing file. The `filename` must be an integer between 1 and 511. If you are opening more than one file, you can ensure that each file has a unique file number by using the `FreeFile` function to return the next available file number.

Writing to Text Files

You can write to a text file with the `Write` statement or the `Print` statement. `Write` produces a line of values separated by commas, and puts hash marks (#) around dates and quotes (") around strings, as shown in the example that follows. `Print` produces a line that is suitable to be printed, with the data arranged in columns with spaces between. You will first see how `Write` performs and look at `Print` a little later.

Say you have a spreadsheet like the one shown in Figure 11-1. You want to create a file containing some of the data. You can use code like the following:

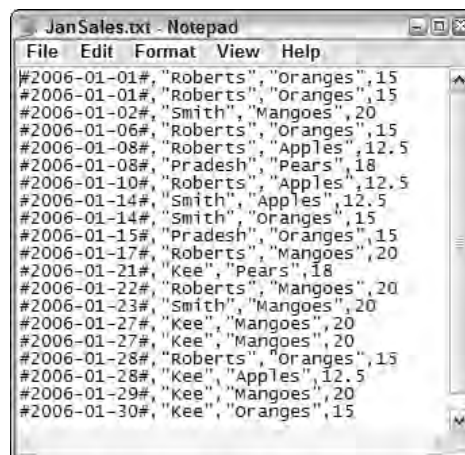
```
Sub WriteFile()  
    Dim dDate As Date  
    Dim sCustomer As String  
    Dim sProduct As String  
    Dim dPrice As Double  
    Dim sFName As String      'Path and name of text file  
    Dim iFNumber As Integer   'File number  
    Dim lRow As Long         'Row number in worksheet  
  
    sFName = "C:\VBA_Prog_Ref\Chapter12\JanSales.txt"  
  
    'Get an unused file number  
    iFNumber = FreeFile  
  
    'Create new file or overwrite existing file  
    Open sFName For Output As #iFNumber  
    lRow = 2  
  
    Do  
        'Read data from worksheet  
        With Sheet1  
            dDate = .Cells(lRow, 1)  
            sCustomer = .Cells(lRow, 2)  
            sProduct = .Cells(lRow, 4)  
            dPrice = .Cells(lRow, 6)  
        End With  
  
        'Write data to file  
        Write #iFNumber, dDate, sCustomer, sProduct, dPrice  
  
        'Address next row of worksheet  
        lRow = lRow + 1  
  
        'Loop until an empty cell is found  
        Loop Until IsEmpty(Sheet1.Cells(lRow, 1))  
  
        'Close the file  
        Close #iFNumber  
  
End Sub
```



	A	B	C	D	E	F	G
1	Date	Customer	State	Product	Number Sold	Price	Revenue
2	Jan 01, 2006	Roberts	NSW	Oranges	903	15	13545
3	Jan 01, 2006	Roberts	TAS	Oranges	331	15	4965
4	Jan 02, 2006	Smith	QLD	Mangoes	299	20	5980
5	Jan 06, 2006	Roberts	QLD	Oranges	612	15	9180
6	Jan 08, 2006	Roberts	VIC	Apples	907	12.5	11337.5
7	Jan 08, 2006	Pradesh	TAS	Pears	107	18	1926
8	Jan 10, 2006	Roberts	VIC	Apples	770	12.5	9625
9	Jan 14, 2006	Smith	NT	Apples	223	12.5	2787.5
10	Jan 14, 2006	Smith	VIC	Oranges	132	15	1980
11	Jan 15, 2006	Pradesh	QLD	Oranges	669	15	10035
12	Jan 17, 2006	Roberts	NSW	Mangoes	881	20	17620
13	Jan 21, 2006	Kee	SA	Pears	624	18	11232
14	Jan 22, 2006	Roberts	QLD	Mangoes	193	20	3860
15	Jan 23, 2006	Smith	SA	Mangoes	255	20	5100
16	Jan 27, 2006	Kee	QLD	Mangoes	6	20	120
17	Jan 27, 2006	Kee	VIC	Mangoes	311	20	6220
18	Jan 28, 2006	Roberts	NT	Oranges	9	15	135
19	Jan 28, 2006	Kee	TAS	Apples	706	12.5	8825
20	Jan 29, 2006	Kee	NT	Mangoes	441	20	8820
21	Jan 30, 2006	Kee	WA	Oranges	936	15	14040

Figure 11-1

FreeFile is used to get the next available file number, and the file is opened in Output mode so it can be written to. Use a Do...Loop to process each row of the worksheet until you find an empty cell in column A. The required data on each row is read into four variables, which are then written to the file. You must Close the file when you have finished using it, or it might not be completed properly and will be left open. You can use Notepad to view the file that's created, as shown in Figure 11-2.



```

JanSales.txt - Notepad
File Edit Format View Help
#2006-01-01#, "Roberts", "Oranges", 15
#2006-01-01#, "Roberts", "Oranges", 15
#2006-01-02#, "Smith", "Mangoes", 20
#2006-01-06#, "Roberts", "Oranges", 15
#2006-01-08#, "Roberts", "Apples", 12.5
#2006-01-08#, "Pradesh", "Pears", 18
#2006-01-10#, "Roberts", "Apples", 12.5
#2006-01-14#, "Smith", "Apples", 12.5
#2006-01-14#, "Smith", "Oranges", 15
#2006-01-15#, "Pradesh", "Oranges", 15
#2006-01-17#, "Roberts", "Mangoes", 20
#2006-01-21#, "Kee", "Pears", 18
#2006-01-22#, "Roberts", "Mangoes", 20
#2006-01-23#, "Smith", "Mangoes", 20
#2006-01-27#, "Kee", "Mangoes", 20
#2006-01-27#, "Kee", "Mangoes", 20
#2006-01-28#, "Roberts", "Oranges", 15
#2006-01-28#, "Kee", "Apples", 12.5
#2006-01-29#, "Kee", "Mangoes", 20
#2006-01-30#, "Kee", "Oranges", 15

```

Figure 11-2

Reading Text Files

You can read text files with the `Input` statement or the `Line Input` statement. `Input` expects data like that produced by `Write` and reads the data into a list of variables. `Line Input` reads the whole line of data as a single string into a single variable. The following code reads the `JanSales.txt` file and inserts the data into a worksheet:

```
Sub ReadFile()  
    Dim dDate As Date  
    Dim sCustomer As String  
    Dim sProduct As String  
    Dim dPrice As Double  
    Dim sFName As String 'Path and name of text file  
    Dim iFNumber As Integer 'File number  
    Dim lRow As Long 'Row number in worksheet  
  
    sFName = "C:\VBA_Prog_Ref\Chapter12\JanSales.txt"  
  
    'Get an unused file number  
    iFNumber = FreeFile  
  
    'Prepare file for reading  
    Open sFName For Input As #iFNumber  
  
    Sheet2.Cells.Clear  
    lRow = 2  
  
    Do  
        'Read data from file  
        Input #iFNumber, dDate, sCustomer, sProduct, dPrice  
  
        'Write data to worksheet  
        With Sheet2  
            .Cells(lRow, 1) = dDate  
            .Cells(lRow, 2) = sCustomer  
            .Cells(lRow, 3) = sProduct  
            .Cells(lRow, 4) = dPrice  
        End With  
  
        'Address next row of worksheet  
        lRow = lRow + 1  
  
        'Loop until end of file  
        Loop Until EOF(iFNumber)  
  
        'Close the file  
        Close #iFNumber  
  
End Sub
```

The `Do . . . Loop` processes each line in the file until the `EOF` function detects that the end of file has been reached. The `Input` statement reads each line of the file into four variables. It is important to have variables of the correct type to match the data, or unexpected results can occur.

If you just want to record data for your own purposes or exchange it with users of the same systems, `Write` and `Input` would probably be all you need. Unfortunately, other systems can use formats that are not quite the same. They can use different separator characters and different delimiter characters, or none. Some work in fixed-sized fields for different variables. You need to find more flexible ways to produce these files and read them.

Writing to Text Files Using Print

`Print` enables you to write text files in any format; you just have to do a bit more work. To see the effect of using `Print` instead of `Write`, change the `WriteFile` sub as follows:

```
Print #iFNumber, dDate, sCustomer, sProduct, dPrice
```

The output looks like that in Figure 11-3.

Date	Customer	Product	Price
1/01/2006	Roberts	Oranges	15
1/01/2006	Roberts	Oranges	15
2/01/2006	Smith	Mangoes	20
6/01/2006	Roberts	Oranges	15
8/01/2006	Roberts	Apples	12.5
8/01/2006	Pradesh	Pears	18
10/01/2006	Roberts	Apples	12.5
14/01/2006	Smith	Apples	12.5
14/01/2006	Smith	Oranges	15
15/01/2006	Pradesh	Oranges	15
17/01/2006	Roberts	Mangoes	20
21/01/2006	Kee	Pears	18
22/01/2006	Roberts	Mangoes	20
23/01/2006	Smith	Mangoes	20
27/01/2006	Kee	Mangoes	20
27/01/2006	Kee	Mangoes	20
28/01/2006	Roberts	Oranges	15
28/01/2006	Kee	Apples	12.5
29/01/2006	Kee	Mangoes	20
30/01/2006	Kee	Oranges	15

Figure 11-3

If you want to read data in this format, you can read each line of the file using the `Line Input` statement. You then need to have code to parse out the data. Taking a hint from `Write`, you might want to use a separator character, but you want to be able to use any character that does not appear in the data. You might also introduce some flexibility with the characters used to delimit items. The following code shows how you can assemble your own strings and write them to a file. Code that is specific to `WriteFile` is highlighted:

```
Sub WriteStrings()
    Dim sLine As String
    Dim sFName As String 'Path and name of text file
    Dim iFNumber As Integer 'File number
    Dim lRow As Long 'Row number in worksheet

    sFName = "C:\VBA_Prog_Ref\Chapter12\JanSalesStrings.txt"

    'Get an unused file number
```

Chapter 11: Text Files and File Dialog

```
ifNumber = FreeFile

'Create new file or overwrite existing file
Open sFName For Output As #ifNumber
lRow = 2

Do
    'Read data from worksheet
    With Sheet1
        sLine = Format(.Cells(lRow, 1), "yyyy-mm-dd") & ";"
        sLine = sLine & .Cells(lRow, 2) & ";"
        sLine = sLine & .Cells(lRow, 4) & ";"
        sLine = sLine & Format(.Cells(lRow, 6), "0.00")
    End With

    'Write data to file
    Print #ifNumber, sLine

    'Address next row of worksheet
    lRow = lRow + 1

    'Loop until an empty cell is found
    Loop Until IsEmpty(Sheet1.Cells(lRow, 1))

'Close the file
Close #ifNumber

End Sub
```

The code assembles each line of the file in a variable `sLine`. For data other than strings, it uses the `Format` function to convert the data to a string. A semicolon is used as a separator. The `Print` statement writes the string to the file. The result is shown in Figure 11-4.



Figure 11-4

Reading Data Strings

To read this data, you need to split the strings to locate the values as shown in the following code. Code that is specific to `ReadFile` is highlighted:

```

Sub ReadStrings()
    Dim sLine As String
    Dim sFName As String    'Path and name of text file
    Dim iFNumber As Integer 'File number
    Dim lRow As Long        'Row number in worksheet
    Dim lColumn As Long     'Column number in worksheet
    Dim vValues As Variant 'Hold split values
    Dim iCount As Integer   'Counter

    sFName = "C:\VBA_Prog_Ref\Chapter12\JanSalesStrings.txt"

    'Get an unused file number
    iFNumber = FreeFile

    'Prepare file for reading
    Open sFName For Input As #iFNumber

    Sheet2.Cells.Clear

    'First row for data
    lRow = 2

    Do
        'Read data from file
        Line Input #iFNumber, sLine

        'Split values apart into array
        vValues = Split(sLine, ";")

        With Sheet2

            'First column for data
            lColumn = 1

            'Process each value in array
            For iCount = LBound(vValues) To UBound(vValues)

                'Write value to worksheet
                .Cells(lRow, lColumn) = vValues(iCount)

                'Increase column count
                lColumn = lColumn + 1

            Next iCount

        End With

        'Address next row of worksheet

```

Chapter 11: Text Files and File Dialog

```
lRow = lRow + 1

'Loop until end of file
Loop Until EOF(iFNumber)

'Close the file
Close #iFNumber

End Sub
```

Line Input reads an entire line of the file into `sLine`. The `Split` function breaks the text apart using the delimiter character, creating an array of values in `vValues`. The `For...Next` loop processes each value in the array, writing the value into the worksheet.

Flexible Separators and Delimiters

Now that you have seen the basic techniques in action, you will set up some code that allows you to choose the separator and delimiter characters to suit the situation. The following code defines them using constants, which can be varied. The code inserts the delimiters and separators as required:

```
Sub WriteStringsWithDelimiters()
    Dim sLine As String
    Dim sFName As String    'Path and name of text file
    Dim iFNumber As Integer 'File number
    Dim lRow As Long       'Row number in worksheet

    Const sVS As String = ";" 'Variable separator character
    Const sTD As String = """" 'Text delimiter character
    Const sDD As String = "#" 'Date delimiter character

    sFName = "C:\VBA_Prog_Ref\Chapter12\JanSalesStringsDelimited.txt"

    'Get an unused file number
    iFNumber = FreeFile

    'Create new file or overwrite existing file
    Open sFName For Output As #iFNumber
    lRow = 2

    Do
        'Read data from worksheet
        With Sheet1
            sLine = sDD & Format(.Cells(lRow, 1), "yyyy-mm-dd") & sDD & sVS
            sLine = sLine & sTD & .Cells(lRow, 2) & sTD & sVS
            sLine = sLine & sTD & .Cells(lRow, 4) & sTD & sVS
            sLine = sLine & Format(.Cells(lRow, 6), "0.00")
        End With

        'Write data to file
        Print #iFNumber, sLine

        'Address next row of worksheet
```



```

lRow = lRow + 1

'Loop until an empty cell is found
Loop Until IsEmpty(Sheet1.Cells(lRow, 1))

'Close the file
Close #iFNumber

End Sub

```

The file produced by the code is shown in Figure 11-5.

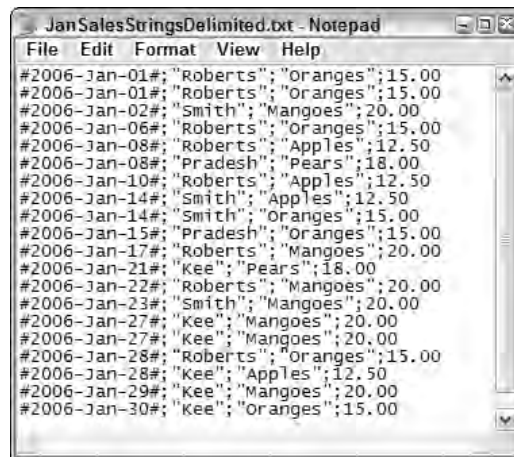


Figure 11-5

The following code is used to read the file. It uses the delimiter characters to decide the data type of each item and treat it appropriately:

```

Sub ReadStringsWithDelimiters()
    Dim sLine As String
    Dim sFName As String 'Path and name of text file
    Dim iFNumber As Integer 'File number
    Dim lRow As Long 'Row number in worksheet
    Dim lColumn As Long 'Column number in worksheet
    Dim vValues As Variant
    Dim vValue As Variant
    Dim iCount As Integer

    Const sVS As String = ";" 'Variable separator character
    Const sTD As String = "\"" 'Text delimiter character
    Const sDD As String = "#" 'Date delimiter character

    sFName = "C:\VBA_Prog_Ref\Chapter12\JanSalesStringsDelimited.txt"

    'Get an unused file number

```

Chapter 11: Text Files and File Dialog

```
iFNumber = FreeFile

'Prepare file for reading
Open sFName For Input As #iFNumber

Sheet2.Cells.Clear

'First row for data
lRow = 2

Do
    'Read data from file
    Line Input #iFNumber, sLine

    'Split values apart into array
    vValues = Split(sLine, sVS)

    'First column for data
    lColumn = 1

    'Process each value in array
    For Each vValue In vValues

        'Determine value type using first character
        Select Case Left(vValue, 1)

            'String
            Case sTD
                Sheet2.Cells(lRow, lColumn) = Mid(vValue, 2, Len(vValue) - 2)

            'Date
            Case sDD
                Sheet2.Cells(lRow, lColumn) = _
                    DateValue(Mid(vValue, 2, Len(vValue) - 2))

            'Other
            Case Else
                Sheet2.Cells(lRow, lColumn) = vValue

        End Select

        lColumn = lColumn + 1
    Next vValue

    'Address next row of worksheet
    lRow = lRow + 1

    'Loop until end of file
    Loop Until EOF(iFNumber)

    'Close the file
    Close #iFNumber

End Sub
```

The `For . . . Next` loop of `ReadStrings` has been replaced by a `For Each . . . Next` loop, to show a slightly different approach. The `Select Case` structure determines the data type of each item by examining the first character. For strings and dates, it strips off the delimiters. For dates, it converts the character string to a VBA date type.

FileDialog

The `FileDialog` object allows you to display the `File ↻ Open` and `File ↻ Save As` dialog boxes using VBA. Excel 2007 users can also use the `GetOpenFileName` and `GetSaveAsFileName` methods of the `Application` object to carry out similar tasks. One advantage of `FileDialog` is that it has one extra capability that allows you to display a list of directories, rather than files and directories. `FileDialog` also has the advantage of being available to all Office applications.

Set up a worksheet to display images and allow the user to choose them through the `File ↻ Open` dialog box. Figure 11-6 shows how the application looks.

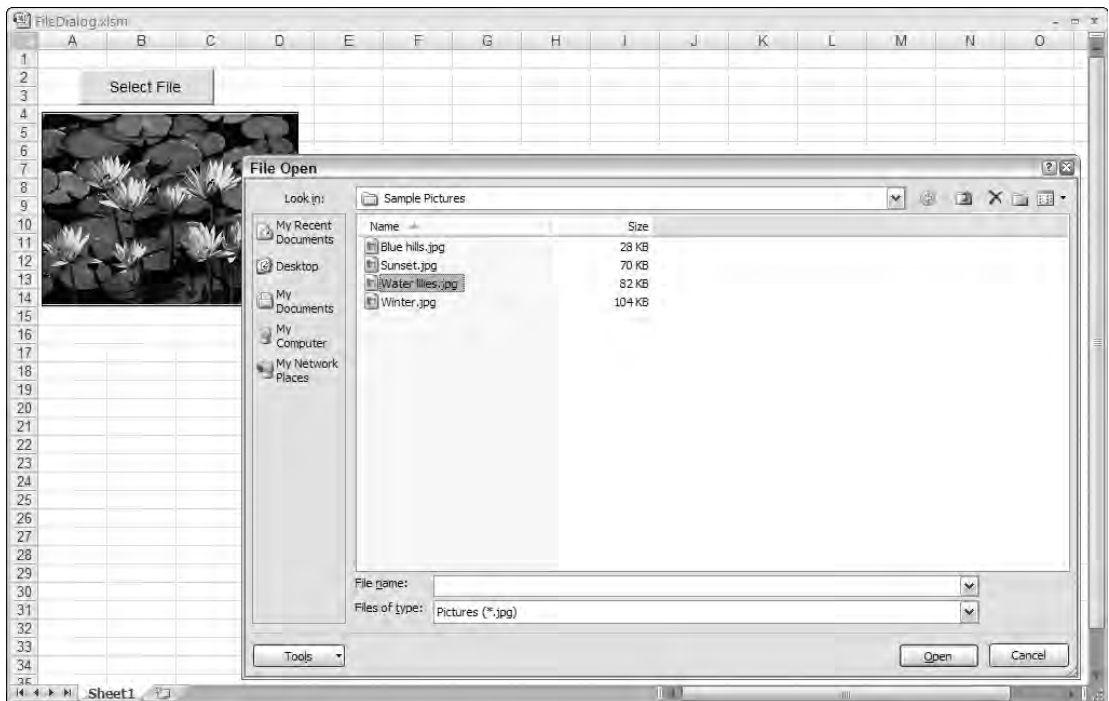


Figure 11-6

The worksheet contains an `Image` control created using the ActiveX controls from the `Developer` tab of the `Ribbon`, with the default name of "Image1". The `pictureSizeMode` property of the control is set to `zoom` so the picture is automatically fitted in the control. The command button above it has been named `cmdGetFile`.

Chapter 11: Text Files and File Dialog

The class module behind Sheet1 contains the following event procedure:

```
Private Sub cmdGetFile_Click()  
    Dim fd As FileDialog  
    Dim ffs As FileDialogFilters  
  
    On Error GoTo Problem  
  
    'Set up File | Open dialog  
    Set fd = Application.FileDialog(msoFileDialogOpen)  
  
    With fd  
        'Clear default filters and create picture filter  
        Set ffs = .Filters  
  
        With ffs  
            .Clear  
            .Add "Pictures", "*.jpg"  
        End With  
  
        'Allow only one file selection  
        .AllowMultiSelect = False  
  
        'Show the dialog. Exit if Cancel is pressed  
        If .Show = False Then Exit Sub  
  
        'Load selected file into Image  
        Image1.Picture = LoadPicture(.SelectedItems(1))  
    End With  
  
    Exit Sub  
  
Problem:  
    MsgBox "That was not a valid picture"  
  
End Sub
```

The `FileDialog` property of the `Application` object returns a reference to the Office `FileDialogs` object. Use the following `msoFileDialogType` constants to specify the type of dialog.

msoFileDialog Constants	Value
<code>msoFileDialogOpen</code>	1
<code>msoFileDialogSaveAs</code>	2
<code>msoFileDialogFilePicker</code>	3
<code>msoFileDialogFolderPicker</code>	4

FileDialogFilters

Use the `Filters` property of the `FileDialog` object to return a reference to the `FileDialogFilters` collection for the `FileDialog`. The filters control the types of files that are displayed. By default, there are 24 preset filters that the user can select from the drop-down menu at the bottom of the File ⇄ Open dialog box. The `Clear` method of the `FileDialogFilters` collection removes the preset filters, and you add your own filter that shows only `.jpg` files.

The `Show` method of the `FileDialog` object displays the dialog box. When the user clicks the Open button, the `Show` method returns a value of `True`. If the user clicks the Cancel button, the `Show` method returns `False` and you exit from the procedure.

FileDialogSelectedItem

The `Show` method does not actually open the selected file, but places the filename and path into a `FileDialogSelectedItem` collection. As you will see later, it is possible to allow multiple file selection. In the present example, the user can only select one file. The name of the file is returned from the first item in the `FileDialogSelectedItem` collection, which is referred to by the `SelectedItem` property of the `FileDialog` object.

Use the `LoadPicture` function to assign the file to the `Picture` property of the `Image` control.

Dialog Types

There is very little difference among the four possible dialog types, apart from the heading at the top of the dialog. The file picker and folder picker types show `Browse` in the title bar, and the others show `File ⇄ Open` and `File ⇄ Save As` dialogs, as appropriate. All the dialogs show directories and files except the folder picker dialog, which shows only directories.

Execute Method

As you have seen, the `Show` method displays the `FileDialog`, and the items chosen are placed in the `FileDialogSelectedItem` object without any attempt to open or save any files. You can use the `Execute` method with the `File ⇄ Open` and `File ⇄ Save As` dialogs to carry out the required Open or Save As operations after the user clicks the Open or Save button, as shown in the following code:

```
With Application.FileDialog(xlDialogOpen)
```

```
    If .Show Then .Execute
```

```
End With
```

MultiSelect

The application in Figure 11-7 has been modified to allow the user to select multiple filenames by holding down Shift or Ctrl while clicking filenames. The filenames are then loaded into the combo box, called `ComboBox1`, at the top of the screen, from which the files can be chosen for viewing.

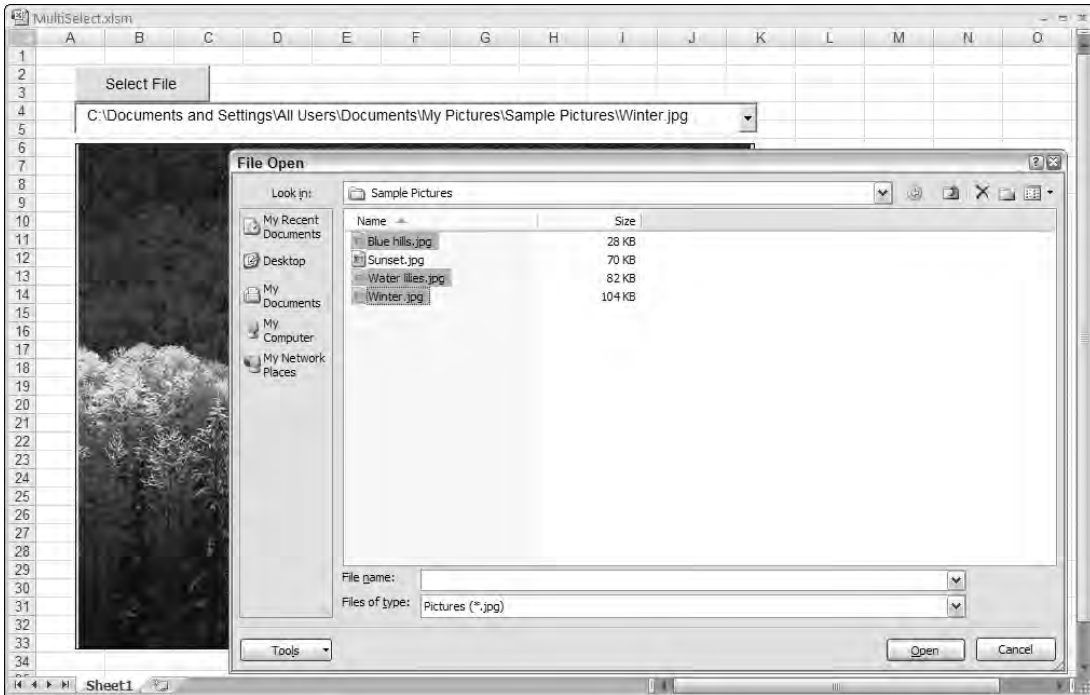


Figure 11-7

The code has been modified as follows:

```
Private Sub cmdGetFile_Click()  
    Dim fd As FileDialog  
    Dim ffs As FileDialogFilters  
  
    Dim vItem  
  
    On Error GoTo Problem  
  
    'Set up File | Open dialog  
    Set fd = Application.FileDialog(msoFileDialogOpen)  
  
    With fd
```

```
'Clear default filters and create picture filter
Set ffs = .Filters

With ffs
    .Clear
    .Add "Pictures", "*.jpg"
End With
```

```
'Allow multiple file selection
.AllowMultiSelect = True
```

```
'Show the dialog. Exit if Cancel is pressed
If .Show = False Then Exit Sub
```

```
'Load selected files into combo box
ComboBox1.Clear

For Each vItem In .SelectedItems
    ComboBox1.AddItem vItem
Next vItem

'Display first file
ComboBox1.ListIndex = 0
```

```
End With
```

```
Exit Sub
```

```
Problem:
MsgBox "That was not a valid picture"
```

```
End Sub
```

```
Private Sub ComboBox1_Change()
    Image1.Picture = LoadPicture(ComboBox1.Text)
End Sub
```

You set the `AllowMultiSelect` property to `True`. The combo box list is cleared of any previous items, and you use a `For Each...Next` loop to add the items in the `FileDialog.SelectedItems` collection to the combo box list. When you set the combo box `ListIndex` property to 0, it triggers the `Change` event and the event procedure loads the first picture into the image control.

Summary

Although Excel has many facilities for importing and exporting text files, it can't handle every possible format. Using VBA, you can import and export text with much greater flexibility.

`FileDialog` allows you to display the File ⇄ Open and File ⇄ Save As dialog boxes as well as a directory browser. It provides more powerful facilities than the `GetOpenFileName` and `GetSaveAsFileName` functions, which are only available in Excel.

`FileDialog` could be used in conjunction with the text file handling procedures that were presented to allow users to specify the location of input and output files.

12

Working with XML and the Open XML File Formats

XML (Extensible Markup Language) functionality has been available in various forms since Office 2000. It made its debut in the Office suite of applications in 1999 with relatively little fanfare, waiting there quietly until the release of Office 2003, where it was touted as one of the most significant improvements in Excel. Office 2003 came with many new XML capabilities and the promise of major changes in the way businesses would work with data. In addition to seamless exchange of data, XML promised easy analysis, dynamic reporting, and the ability to consume data from an untold number of external sources.

Unfortunately, XML has failed to find a place in the hearts of many Excel programmers. The problem is that many Excel programmers still look at XML as a solution to a problem that they haven't quite encountered yet. This is because much of the functionality offered by XML can be handled by existing technologies and processes that programmers are already comfortable with. In addition, most Excel developers don't live in environments where XML shines the brightest — environments where data is routinely exchanged between disparate platforms (such as the web). The reality is that most Excel programmers live in a world where the Office suite of applications and a few SQL Server databases are as diverse as it gets. The bottom line is that there has never been that one compelling reason to leave the comfort of existing technologies and processes to go to XML. That is, not until now.

Why has XML suddenly become so important? Two words: Open XML. With Office 2007, Microsoft gives XML a leading role by introducing the Open XML file format. These new file formats are XML-based, meaning that each Excel workbook you create in Office 2007 is essentially a group of XML documents. These XML documents are saved as a collection of parts, compressed into a Zip container, and given a file extension (for example, .xlsx, .xlsm, .xlam, and so on).

As illogical as Microsoft's move toward XML may seem, the decision to bet on XML is a fairly rational one. An XML-based Office will be able to move into an increasing number of environments as XML becomes a widely adopted standard. An XML-based Office can be integrated with much wider array of XML-capable software and web-based applications. An XML-based Office opens up new opportunities for programmers to develop applications that revolve around the Office suite.

Chapter 12: Working with XML and the Open XML File Formats

With this move toward XML, Microsoft takes a huge step toward making Excel spreadsheets universal widgets that can be integrated into almost any application or web-based solution. Within the next few years, XML-based solutions will start materializing everywhere until XML becomes a part of the Excel developer's everyday vernacular.

So as Microsoft pushes us all into a new realm of development, it's important to start to get a grasp of the XML technology. In this chapter, you will get a firm understanding of XML as it pertains to both Excel and the new Open XML file formats. That being said, it's important to note that the goal of this chapter is not to make you an expert XML developer. Indeed, the topic of XML is a robust one that cannot be fully covered in one chapter. The goal of this chapter is to give you a solid understanding of all the aspects of XML you will need to be familiar with when working with XML in Excel.

The Basics of Using XML Data in Excel

As intimidating as an XML document may seem, it's really nothing more than a text file that contains data wrapped in *markup* (tags that denote structure and meaning). These tags essentially make the text file machine-readable. The term *machine-readable* essentially means that any application or web-based solution designed to read XML files will be able to discern the structure and content of your file.

Because XML is text-based, it is not platform-dependent. That is to say, XML is not dependent on a specific application for construction, reading, or editing. This versatility promotes application interoperability, collaboration, and data sharing. In addition, because of their text-based nature, XML documents tend to compress at a higher compression rate than binary files, making them ideal for storing and archiving data.

Another benefit of XML documents is that they internally describe their own content and structure in parent/child hierarchies. This allows applications to search and extract data far more efficiently than standard text files. And because XML documents are intrinsically open, programmers don't have to spend valuable time developing processes to work around the ugly internal details of proprietary components.

This section gives you a solid understanding of the fundamentals of XML. You will also get some context for XML functionality in Excel by exploring some of the ways Excel allows you to work with XML data through the user interface.

XML Fundamentals

XML files are made up of several syntactic constructs. Take a moment to explore the fundamental components of a standard XML document.

The XML Declaration

The first line of an XML document is called the XML declaration. The following line of code shows an example of a typical XML declaration:

```
<?xml version="1.0"? encoding="UTF-8" standalone="Yes"?>
```

The XML declaration typically contains three parts: a `version` attribute, and optional `encoding` and `standalone` attributes.

The `version` attribute tells the processing application that this text file is an XML document. You will rarely see XML documents that go beyond version 1.0, primarily for two reasons: version 1.0 has been around since 1998, and changes to XML since version 1.0 have been relatively minor.

The `encoding` attribute is primarily used to work around character encoding issues. Because XML documents are inherently Unicode, the `encoding` attribute is optional if the character encoding used to create the document is UTF-8, UTF-16, or ASCII. Indeed, you will find that the character encoding is omitted from many of the XML documents you may encounter.

The `standalone` attribute tells the processing application whether the document references an external data source. If the document contains no reference to external data sources, it is deemed to be standalone; thus the "Yes" value. Because every XML document is inherently standalone, this attribute is optional for documents that do not reference an external source.

Processing Instructions

As their namesake implies, processing instructions provide explicit instructions to the processing application. These can be identified by distinctive tags comprised of left and right angle brackets coupled with question marks (`<?, ?>`). These instructions are typically found directly under the XML declaration and can provide any number of directives. For example, the following processing instruction would direct the use of Excel to open the given XML document:

```
<?mso-application progid="Excel.Sheet" ?>
```

Comments

Comments allow XML developers to enter plain-language explanations or remarks about the contents of the document. Just as in VBA, where the single quote signifies a comment, XML has its own syntax to denote a comment. Comments in XML begin with the `<!--` characters and end with the `-->` characters, as follows:

```
<!--Document created by Mike Alexander-->
```

Comments are also useful when you want to disable a particular construct or processing instruction in the XML document. For instance, imagine your XML document contained a processing instruction that forced the document to open with Excel. You may want to disable this processing instruction so the document opens in Internet Explorer by default. Instead of removing the processing instruction, you can simply comment it out by wrapping it in a comment as shown here:

```
<!-- <?mso-application progid="Excel.Sheet" ?> -->
```

Elements and the Root Element

An element is defined by a start tag (such as `<MyData>`) and an end tag (such as `</MyData>`). Any data you enter between the start and end tags makes up the contents of that element. As you can see in the following example, the document begins with `<MyTable>` and ends with `</MyTable>`; all of the syntax you see between these tags makes up the content of the `MyTable` element:

```
<?xml version="1.0"?>
<MyTable>
  <Customer>
    <Quarter>Q1</Quarter>
    <Region>North</Region>
    <Revenue>25000</Revenue>
  </Customer>
</MyTable>
```

The concept of tags will be a familiar one to those who have worked with HTML. However, unlike HTML, tags in XML are not predefined. That is to say, the text *MyTable* has no predefined utility or meaning. You can change that text to *Pork* and it would be all the same to the XML document. And herein, you stumble on the beauty of XML. XML allows you to create custom tags: tags to which you give definition and purpose. As long as you adhere to a few basic rules, you can create and describe any number of elements by creating your own custom tags. Here are the basic syntactic rules that must be followed when creating elements:

- ❑ Every element must have a start tag, represented by left and right angle brackets (<>), as well as a corresponding end tag represented by a left angle bracket, forward slash, and right angle bracket (</>).
- ❑ Names in XML are case sensitive, so the start and end tags of an element must match in case as well as in syntax. For example, an element defined by the tags <Data> </data> would cause a parsing error. XML would look for the end tag for <Data> as well as the start tag for </data>.
- ❑ You must begin all element names with a letter or an underscore; never a digit. In addition, names that begin with any permutation of *xml* are reserved and cannot be used.

The `MyTable` element is the *root element* for this particular XML document. The root element (which is always the topmost element in an XML document) serves as the container for all of the contents within the document. Every XML document must have one (and only one) root element.

Below the root element, you will see four elements, each one containing its own content. Elements can contain numbers, text, and even other elements.

Elements are normally framed in a parent/child hierarchy. For instance, in the previous example, the `Customer` element is a child of the `MyTable` root element. Likewise, the `MyTable` element is the parent of the `Customer` element. Following that logic, the `Quarter`, `Region`, and `Revenue` elements are the children of the `Customer` element. This parent/child hierarchy allows the XML document to describe the structure of the data as well as the content. Later in this chapter, you will discover how this parent/child hierarchy is leveraged to programmatically move around in XML documents.

Attributes

Attributes in XML documents come in two flavors: data attributes and metadata attributes. Data attributes are used to provide the actual data for an element. For example, the following attributes (name and age) provide the data for the `Pet` element:

```
<Pet name='Shnuckums' age="4">Dog</Pet>
```

Notice that the `age` attribute is wrapped in quotes even though the value itself is a number. This is because unlike elements, attributes are textual. This means that attributes must be wrapped in either single or double quotes.

You may also see attributes that exist in an empty element that contains no nested children. In these situations, you will see the attributes formatted as such:

```
<name='Doodoo' age="4" />
```

Metadata attributes provide descriptive information about the contents of elements. For instance, in the following example, the `Customer` element has an attribute called `id` that provides that `Customer` with a unique identifier:

```
<?xml version="1.0"?>
<MyTable>

  <Customer id="1"/>
  <Quarter>Q1</Quarter>
  <Region>North</Region>
  <Revenue>25000</Revenue>

</MyTable>
```

Many new users of XML find the concept of attributes versus elements a bit confusing. After all, most elements can be easily converted to attributes (or vice versa) and the XML document would parse just fine. For example, the `Customer id` attribute could just as easily be presented in an element as such: `<id>1</id>`. There are, however, general rules of thumb that most XML documents seem to adhere to when it comes to elements versus attributes:

- ❑ If the content is not an actual data item, but a descriptor of the data (record number, index number, unique identifier, and so forth), then an attribute is typically used.
- ❑ Elements are used for any content that consists of multiple values.
- ❑ If there is a chance that the content will expand in structure to include children, then elements are typically used.

Namespaces

The idea behind namespaces is simple. Because XML lets developers create and name their own elements and attributes, there is a possibility that a particular name could be used in different contexts. For instance, an XML document may use the name `ID` to describe both a customer ID and an invoice ID. Namespaces associate overlapping identifiers with Uniform Resource Identifiers (URI), allowing applications that process XML documents to make a distinction between similar names.

A URI is typically made up of a URL and a relative descriptor. For instance, the following line of code defines a namespace. As you can imagine, `xmlns` stands for XML namespace:

```
xmlns="http://www.datapigtechnologies.com/customers"
```

Chapter 12: Working with XML and the Open XML File Formats

The fact that URLs are used to define namespaces leads many to believe that namespaces point to some sort of online source. The fact is that URLs are only used to provide some semblance of ownership to anyone reading the XML file. The goal of a namespace is merely to create a unique string. So you could technically use something like `xmlns="arbitrary_namespace"`, although it wouldn't be very useful in identifying ownership or utility.

As you can imagine, using a URL can lead to some fairly long namespace strings. Most XML developers get around this problem by creating namespace prefixes. A prefix is nothing more than an alias for the namespace. For instance, the following namespace uses the prefix `dpc`. Then the `dpc` prefix is applied to an attribute:

```
xmlns:dpc="http://www.datapigtechnologies.com/customers"
<Invoice dpc:id="201">
```

In most XML documents, the first namespace is considered to be the default namespace for the document. That is to say, the names in the document that are not explicitly associated with other namespaces will be associated with this catch-all namespace.

You will notice that in the example illustrated in the following listing, the namespace is placed directly into the root element. Any namespace declared within an element is automatically applied to all child elements. You may be wondering why a namespace would be needed in a document where there are no duplicate names. Many XML documents contain a default namespace to avoid overlapping names with other XML documents that may be consumed in the same process or application:

```
<?xml version="1.0"?>
<Customer xmlns="http://www.datapigtechnologies.com">
  <Quarter>Q1</Quarter>
  <Region>North</Region>
  <Revenue>25000</Revenue>
</Customer>
```

Now take a look at an example of namespaces in action. In this XML document, you are using the name `id` to describe both the invoice ID and the customer ID. As you can see, you have applied three namespaces; one namespace is the default for the document, one is a prefixed namespace used for the orders ID, and one is used directly in the `id` element:

```
<?xml version="1.0" encoding="UTF-8"?>
<Order xmlns="http://www.datapigtechnologies.com"
xmlns:O="http://www.datapigtechnologies.com/orders">
  <Invoice O:id="1-A-234">
    <Customer>Zalex Corp</Customer>
    <id xmlns="http://www.datapigtechnologies.com/customers">21112</id>
    <Amount>1000</Amount>
  </Invoice>
</Order>
```

The first namespace is considered to be the default namespace for the document. The next namespace uses the letter `o` as a prefix. The invoice ID is then qualified with the association of the `o` prefix. Finally, the last namespace declaration can be found within the `i` element.

Viewing and Editing an XML Document

Double-clicking a standard XML document will typically open it in Internet Explorer. When XML documents are presented through a browser, they are read-only. To edit XML documents, you must open them with either a text editor such as Notepad or an XML editor.

Although you can use simple text editors, you will find that an XML editor structures your XML documents into reader-friendly trees that make finding and editing content much easier. You can find both free and commercial XML editors online. XML Marker and XML Cook Top are both excellent free editors.

It's generally a good idea to open any new XML documents you encounter with Internet Explorer so they can be checked with Internet Explorer's built-in XML parser. Internet Explorer 5 and above has a built-in DLL that can read XML documents and check for well-formedness. This is akin to the VBA compiling your code and checking for syntax errors. If the XML document you are trying to open is not well formed, the parser will throw an error message. The idea here is that if Internet Explorer cannot open your XML file, you will not be able to use it in Excel.

Every version of Internet Explorer from Internet Explorer 5 on has its version of the built-in MSXML.DDL parser. The MSXML.DLL comes with its own object model called the Document Object Model (DOM). Later, you will learn how to leverage this object model to read and edit data from XML files.

Just for the record, a well-formed document has the following characteristics:

- There is one root element encapsulating all content in the XML document.
- All element tags have a start tag and a matching end tag, as in `<element></element>`.
- None of the element tags have mismatched cases (such as `<Element></element>`).
- All child elements are closed before their parents. For instance, `<Parent><Child></Parent></Child>` would cause a parsing error because the child element is closed after the parent element has been closed. The correct sequence would be `<Parent><Child></Child></Parent>`.
- All attributes have one value wrapped in either single or double quotes.

Once your XML document has been validated by Internet Explorer, opening with no parsing errors, it is ready to be used.

Consuming XML Data Directly

Once you have a well-formed XML document, you can start using the data it contains. One of the simplest ways to use an XML document is to open it directly from Excel. To help demonstrate this, open `EmployeeSales.xml`, shown here. This XML document contains data revolving around the invoices filed by the employee in an organization:

```
<?xml version="1.0"?>
<EmployeeSales>
  <Employee>
    <Empid>2312</Empid>
    <FirstName>Mike</FirstName>
    <LastName>Alexander</LastName>
    <InvoiceNumber>100</InvoiceNumber>
    <InvoiceAmount>2300</InvoiceAmount>
  </Employee>

  <Employee>
    <Empid>24601</Empid>
    <FirstName>Stephen</FirstName>
    <LastName>Bullen</LastName>
    <InvoiceNumber>200</InvoiceNumber>
    <InvoiceAmount>3211</InvoiceAmount>
  </Employee>
</EmployeeSales>
```

You can find the sample files used in the various walkthroughs for this chapter at www.wrox.com, in this chapter's XMLSampleFiles folder.

Start Excel, select File → Open, and then open the XML document titled `EmployeeSales.xml`. You will immediately see the Open XML dialog box, shown here in Figure 12-1.

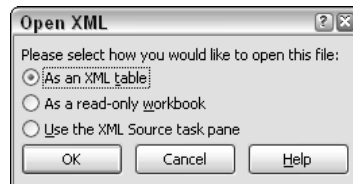


Figure 12-1

Select As an XML table and click the OK button. This will activate the dialog box, shown here in Figure 12-2, where Excel tells you that it could not find a schema for your XML document so it will create one for you.

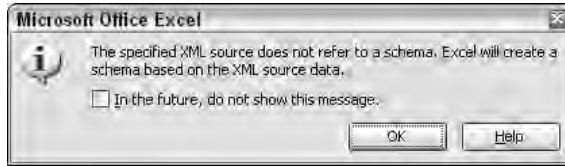


Figure 12-2

In XML terms, the word *schema* refers to an XML Schema Description (XSD). An XSD is a file typically associated with an XML file in order to provide rules for the document. XSD files dictate the layout and sequencing for the data in an XML document, as well as the data types and default values for each element and attribute.

The topic of XSD is a subject that is worthy of its own book. This chapter is, alas, focused on areas outside the scope of XSD, so XSD is not covered in detail here. Although there are plenty of books that cover this subject, a visit to www.w3shools.com/schema will get you off on the right foot. This site will give you a solid (and free) start on learning more about XSD.

Because the `EmployeeSales.xml` file does not have an associated schema file (XSD), Excel will infer a schema from your XML document. This means Excel essentially creates an internal schema that will dictate the rules for the document.

Once you click the OK button, Excel will create a new workbook and populate it with your XML data table. At this point, you should see the same table shown in Figure 12-3.

	A	B	C	D	E
1	Empid	FirstName	LastName	InvoiceNumber	InvoiceAmount
2	2312	Mike	Alexander	100	2300
3	24601	Stephen	Bullen	200	3211

Figure 12-3

Excel automatically creates an XML list, mapping a range of cells to the elements in the source XML document. These cells are linked back to the XML document and can be refreshed with the latest data by right-clicking inside the XML list and selecting XML ⇨ Refresh XML Data.

You can also refresh using the Refresh Data button, found in the XML Group under the Developer tab of the Ribbon.

Chapter 12: Working with XML and the Open XML File Formats

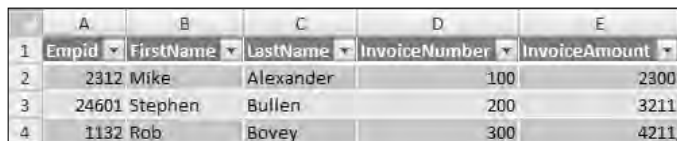
To test this out, save your Excel file and close it. Now edit the `EmployeeSales.xml` file to add a new employee record, as demonstrated here:

```
<?xml version="1.0"?>
<EmployeeSales>
  <Employee>
    <Empid>2312</Empid>
    <FirstName>Mike</FirstName>
    <LastName>Alexander</LastName>
    <InvoiceNumber>100</InvoiceNumber>
    <InvoiceAmount>2300</InvoiceAmount>
  </Employee>

  <Employee>
    <Empid>24601</Empid>
    <FirstName>Stephen</FirstName>
    <LastName>Bullen</LastName>
    <InvoiceNumber>200</InvoiceNumber>
    <InvoiceAmount>3211</InvoiceAmount>
  </Employee>

  <Employee>
    <Empid>1132</Empid>
    <FirstName>Rob</FirstName>
    <LastName>Bovey</LastName>
    <InvoiceNumber>300</InvoiceNumber>
    <InvoiceAmount>4211</InvoiceAmount>
  </Employee>
</EmployeeSales>
```

Once you save your edits in the XML document, return to the Excel file and refresh the XML map. As you can see in Figure 12-4, the newly added employee will be included in the mapped range.



	A	B	C	D	E
1	Empid	FirstName	LastName	InvoiceNumber	InvoiceAmount
2	2312	Mike	Alexander	100	2300
3	24601	Stephen	Bullen	200	3211
4	1132	Rob	Bovey	300	4211

Figure 12-4

Take a moment to think about how this functionality could be useful to you as an Excel programmer. Once your XML data is mapped to a range of cells, it can be used just as other data in Excel. For instance, you can use XML data as variables in formulas, as feeds for charts, and as the source data for pivot tables. Imagine building an Excel-based reporting system where all data that feeds your pivot tables and charts links back to XML files on a network server. You can imagine that those XML files could be updated on a nightly basis, while your client's workbooks could be designed to automatically refresh on open. Later in this chapter, you will discover how you can leverage VBA to create XML documents and automate many of the actions you have taken here thus far.

Keep in mind that when Excel infers a schema for you, the source XML document is automatically rendered read-only. In this context, the term read-only means that you cannot make changes to the source XML document via the XML map created in Excel. This prevents your users from editing or adding data in the source XML document.

Creating and Managing Your Own XML Maps

Although it *is* convenient to let Excel handle the mapping of XML data to your spreadsheet, creating your own XML maps gives you a bit more flexibility and control over how your data is used. In particular, creating your own XML maps allows you to write back to your XML document, selectively map elements individually, and integrate data from multiple XML documents. In this section, you walk through a scenario where creating a custom XML map will help you build a simple data entry template for employee invoices. As you go through this example, keep in mind that this is one of many possible scenarios that benefit from a custom XML mapping.

Creating Your Own XML Schema Description

One of the requirements for the data entry template is that it needs to be able to edit and add content in an XML document. To meet this requirement, you will have to provide Excel with an XML Schema Description (XSD). Although you could try to create your own XSD from scratch, there is enough complexity around developing XSD files to make that option a rather unappealing one. Instead, you can leverage Excel to create an XSD for you.

First, create a simple XML document using a simple text or XML editor. The goal here is to enter the elements that make up the data points you need to capture. In this example, you want to capture a set of five data points: employee ID, first name, last name, invoice number, and invoice amount. In that light, create the simple XML document shown here:

```
<?xml version="1.0"?>
<EmployeeSales>
  <Employee>
    <Empid>999</Empid>
    <FirstName>Text</FirstName>
    <LastName>Text</LastName>
    <InvoiceNumber>999</InvoiceNumber>
    <InvoiceAmount>999</InvoiceAmount>
  </Employee>

  <Employee></Employee>
</EmployeeSales>
```

Note that fake data is placed into the elements. This will allow Excel to identify the data type for each element when it infers a schema. You will also note the inclusion of an empty `Employee` element. This will ensure that the `Employee` element is tagged as a repeating element in Excel's schema. Why do you need this particular element to repeat? The data entry template will need to accept multiple entries at the employee level; a repeating employee element will satisfy this requirement. Be sure to save your file as an XML document.

XML Map Properties

It's important to note that your XML map has certain properties that can be adjusted to suit your needs. To access the properties dialog box, simply right-click anywhere inside the XML list and select XML ⇄ XML Map Properties. Again, later in this chapter, you will discover how you adjust these properties dynamically via VBA.

- Name:** The name property specifies the name of the XML map.
- Validate data against schema for import and export:** When selected, this property will direct Excel to validate any data entered into the XML map against its corresponding schema, ensuring that the data conforms to the rules specified in the schema before importing or exporting.
- Save data source definition in workbook:** When selected, this property ensures that your XML map is linked to the source XML document, thus allowing the document to be refreshed. Deselecting this property will make the XML map static.
- Adjust column width:** When selected, Excel will automatically adjust the width of columns to fit the mapped XML data.
- Preserve column filter:** When selected, Excel preserves the sorting and filtering applied to the XML list when refreshing the XML map.
- Preserve number formatting:** When selected, Excel preserves the formatting applied to numbers in the XML list when refreshing the XML map.
- Overwrite existing data with new data:** When selected, existing data in the XML list will be overwritten when the XML map is refreshed.
- Append new data to existing XML lists:** When selected, any new data from the source XML document will be added to the bottom of the existing data in the XML list when the XML map is refreshed.

Next, open the XML document from Excel, allowing Excel to infer its own schema. Once the XML document has been mapped, open the XML Source task pane to inspect the XML map. To do so, select the Developer tab in the Ribbon, and then select XML Source. This will activate the XML Source task pane, shown in Figure 12-5, where you will see the elements that make up your XML document.

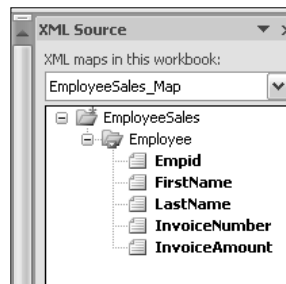


Figure 12-5

Notice that this particular XML map is named `EmployeeSales_Map`. Every XML map must have a name. When Excel infers a schema for you, it automatically assigns a name made up of the root element name and `_Map`. You can change this name by adjusting the `Name` property in the XML Map Properties dialog box. It is generally good practice to use the assigned name when referencing an XML map via VBA.

Excel has successfully created a schema for the data structure you specified via the XML document. Your goal now is to get that schema out of Excel and into an XSD file. Unfortunately, there is no way to extract the inferred schema via the user interface. You will have to use a bit of VBA to get to the schema. Start a new module and enter the following code:

```
Sub GetSchema()  
  
Dim MySchema As String  
  
'Get the schema  
MySchema = ActiveWorkbook.XmlMaps("EmployeeSales_Map").Schemas(1).XML  
  
'Create and fill an xsd file with your schema  
Open "C:\MySchema.xsd" For Output As #1  
Print #1, MySchema  
Close #1  
  
End Sub
```

In this code, you first use the `XmlMaps` collection to identify the XML map and schema you are targeting. As you can see, the target schema is the primary schema for the XML map named "EmployeeSales_Map" in the active workbook. Next, you create empty text file and save it as "C:\MySchema.xsd". Finally, you output your schema to the newly created XSD file. Your reward for creating and running this code is an XSD file that took very little effort to create.

Creating Your Own XML Map

With your newly created XSD in hand, you can create your XML-based data entry template. Start by opening a new Excel workbook and activating the XML Source task pane by selecting XML Source in the Developer tab. Next, click the XML Maps button to activate the XML Maps dialog box, shown here in Figure 12-6.



Figure 12-6

Chapter 12: Working with XML and the Open XML File Formats

Here, you will click the Add button to link to the `MySchema.xsd` you created in the `C:\` drive. Once your XSD is referenced, click OK to create the map. When your XML map is created, the XML Source will show you the elements that are available for use in your spreadsheet.

Keep in mind that you are not limited to one XML mapping. You can map multiple XML documents and schemas. Simply activate the XML Maps dialog box and add another link to an XML document or schema.

At this point, you can drag the `Employee` parent element to your spreadsheet to create a data entry table. From here, you can add a little formatting to achieve a particular look and feel. Now here comes the impressive bit. Enter some data into the template, as demonstrated here in Figure 12-7.

	A	B	C	D	E	F	G
		Enter Employee	Enter Employee	Enter Employee	Enter Invoice	Enter Invoice	
1		ID	First Name	Last Name	Number	Amount	
3		24601	Mike	Alexander	100	\$2,400	
4		211321	John	Green	200	\$5,400	
5		203400	Rob	Bovey	300	\$3,300	

Figure 12-7

Now you need to export the data entered into your template as an XML document by right-clicking anywhere inside the mapped range and selecting XML ⇄ Export. This will open a dialog box where you will be asked to specify a name and location for the XML export. Once the document is exported, you can open it to see that your data has indeed been output into the XML structure specified by your custom schema. Figure 12-8 demonstrates the output for the data entered in Figure 12-7.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
- <EmployeeSales xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
- <Employee>
  <Empid>24601</Empid>
  <FirstName>Mike</FirstName>
  <LastName>Alexander</LastName>
  <InvoiceNumber>100</InvoiceNumber>
  <InvoiceAmount>2400</InvoiceAmount>
</Employee>
- <Employee>
  <Empid>211321</Empid>
  <FirstName>John</FirstName>
  <LastName>Green</LastName>
  <InvoiceNumber>200</InvoiceNumber>
  <InvoiceAmount>5400</InvoiceAmount>
</Employee>
- <Employee>
  <Empid>203400</Empid>
  <FirstName>Rob</FirstName>
  <LastName>Bovey</LastName>
  <InvoiceNumber>300</InvoiceNumber>
  <InvoiceAmount>3300</InvoiceAmount>
</Employee>
</EmployeeSales>
```

Figure 12-8

You can imagine how automating the export functionality would allow a user to send entered data to a specified location with the click of a button. Once the data is in an XML document, any procedure or application that can process the XML document will be able to consume the data.

As mentioned at the beginning of this section, the scenario you just walked through is just one of countless ways you can choose to implement your own XML maps. You are encouraged to take some time to think about your data processes in terms of XML.

Using VBA to Program XML Processes

The functionality gained by using XML technologies has less to do with actual XML documents than the processes you build to leverage their benefits. Although many of the tasks involved in these XML-based processes can be done manually through Excel's UI, the very nature of XML functionality lends itself to automation. This section contains examples demonstrating how VBA can help you program many of the common tasks associated with an XML-based process.

Programming XML Maps

Fortunately, Excel exposes a robust set of objects, properties, and methods that make it possible for many of the tasks associated with mapping to XML data to be accomplished via VBA. Here, you will perform many of the tasks you performed in the last section programmatically.

Open an XML Document Directly into a List

Through the user interface, you can open an XML document directly from Excel, automatically creating an XML list that is mapped to the elements in the source XML document. This functionality can be replicated via code by using the `OpenXML` method of the `Workbooks` collection. This method returns a workbook object with the XML data mapped to your spreadsheet:

```
Sub ImportXMLtoList()  
Dim strTargetFile As String  
  
'Inhibit schema warning  
Application.DisplayAlerts = False  
  
'Select target XML document  
strTargetFile = ThisWorkbook.Path & "\EmployeeSales.xml"  
  
'Use the OpenXML method to open the target file  
Workbooks.OpenXML Filename:=strTargetFile, LoadOption:=xlXmlLoadImportToList  
  
'Turn alerts back on  
Application.DisplayAlerts = True  
  
End Sub
```

Chapter 12: Working with XML and the Open XML File Formats

First you inhibit the schema warning (where Excel tells you it will infer a schema for you) by setting the `DisplayAlerts` property to `False`. This prevents user confusion when custom-made schemas are not involved in your automated XML processes. You then assign the target XML document to a variable, allowing for some flexibility when incorporating this code into a larger process. Next, you pass the target filename to the `OpenXML` expression and include the `xlXmlLoadImportToList` variant, telling Excel to import the XML data directly into a list object. Finally, you set the `DisplayAlerts` property back to `True`.

Programmatically Changing XML Map Properties

Once an XML map exists, there may be cause to change a few of its properties programmatically. In the following code snippet, you pass the name of your map the `XMLMaps` collection, and then adjust each available property. These properties coincide with those discussed in the section called “Consuming XML Data Directly”:

```
Sub ChangeXmlMapProperties()  
    With ActiveWorkbook.XmlMaps("EmployeeSales_Map")  
  
        'Change map name  
        .Name = "New_Name"  
  
        'XML schema definition validation  
        .ShowImportExportValidationErrors = False  
  
        'Data source property  
        .SaveDataSourceDefinition = True  
  
        'Data formatting and layout properties  
        .AdjustColumnWidth = True  
        .PreserveColumnFilter = True  
        .PreserveNumberFormatting = True  
  
        'Overwrite or Append data  
        'False Overwrites while True Appends  
        .AppendOnImport = False  
  
    End With  
  
    'This example references the XML map by index number  
    ThisWorkbook.XmlMaps(1).Name = "EmployeeSales_Map"  
  
End Sub
```

Refresh Your XML Data

Once your XML map has been created, you can refresh it by using the following code. Here, you are using the `Refresh` method to update the data binding for a specified XML map:

```
Sub RefreshXML()  
    ThisWorkbook.XmlMaps("EmployeeSales_Map").DataBinding.Refresh  
End Sub
```


Turn Your XML Lists into Hard Data

In the context of XML lists, *hard data* means that the data residing in the list object is no longer mapped to the XML schema you are using. You may often find that once the needed XML data is in your spreadsheet, there is no need for the data to be refreshed. In these situations, you can simply delete the XML map to disconnect the cells in your spreadsheet from the schema. To do so, use the `Delete` method, as demonstrated here:

```
Sub RemoveMap()  
    ThisWorkbook.XmlMaps("EmployeeSales_Map").Delete  
End Sub
```

Creating Your Own XSD

For many Excel developers, creating a custom XML schema from scratch is simply out of the question. In the last section, you discovered that instead of developing an XML schema from scratch, you could leverage Excel to help you create and output a schema. The good news is that even this task can be automated, as demonstrated here:

```
Sub Create_XSD()  
    Dim StrMyXml As String, MyMap As XmlMap  
    Dim StrMySchema As String  
  
    'Fill a string with your template XML  
    StrMyXml = "<EmployeeSales>"  
    StrMyXml = StrMyXml & "<Employee>"  
    StrMyXml = StrMyXml & "<Empid>999</Empid>"  
    StrMyXml = StrMyXml & "<FirstName>Text</FirstName>"  
    StrMyXml = StrMyXml & "<LastName>Text</LastName>"  
    StrMyXml = StrMyXml & "<InvoiceNumber>999</InvoiceNumber>"  
    StrMyXml = StrMyXml & "<InvoiceAmount>999</InvoiceAmount>"  
    StrMyXml = StrMyXml & "</Employee>"  
    StrMyXml = StrMyXml & "<Employee></Employee>"  
    StrMyXml = StrMyXml & "</EmployeeSales>"  
  
    'Use your XML string to add an XML map  
    Application.DisplayAlerts = False  
    Set MyMap = ThisWorkbook.XmlMaps.Add(StrMyXml)  
    Application.DisplayAlerts = True  
  
    'Get the schema  
    StrMySchema = ThisWorkbook.XmlMaps(1).Schemas(1).XML  
  
    'Create and fill an xsd file with your schema  
    Open "C:\StrMySchema.xsd" For Output As #1  
    Print #1, StrMySchema  
    Close #1  
  
End Sub
```

Chapter 12: Working with XML and the Open XML File Formats

First, declare a string called `MyXML`, and then assign to it the elements that will make up the content and parent/child hierarchy of the XML document. What you're looking for here is structure: a format that makes the XML easy to read and manage within the VBE. The first line starts the string. Each subsequent line is concatenated to the previous line. By the last line, the `MyXML` variable contains the entire XML string.

Next, pass your newly created XML string to the `Add` method of the `XmlMaps` collection. Notice that again, the `DisplayAlerts` property is set to `False` to inhibit any schema warning messages. From here, you can use the newly created XML map's index number to identify and pass the inferred schema to a string variable. Finally, create an empty XSD and output your schema file.

Creating a Custom XML List

When you create your own XML lists, you gain the ability to control what information is included in your schema, which elements are mapped to the spreadsheet, and where the XML data is mapped. The following code demonstrates how to create a custom XML list, allowing you to manage each aspect of the mapping process; from the schema, you use to the location of the XML list:

```
Sub CreateXMLList()
    Dim oMyMap As XmlMap
    Dim strXPath As String
    Dim oMyList As ListObject
    Dim oMyNewColumn As ListColumn

    'Add a schema map
    ThisWorkbook.XmlMaps.Add (ThisWorkbook.Path & "\Myschema.xsd")

    ' Identify the target schema map.
    Set oMyMap = ThisWorkbook.XmlMaps("EmployeeSales_Map")

    ' Create a new list in A1.
    Range("A1").Select
    Set oMyList = ActiveSheet.ListObjects.Add

    'Find the first element to map.
    strXPath = "/EmployeeSales/Employee/Empid"
    ' Map the element.
    oMyList.ListColumns(1).XPath.SetValue oMyMap, strXPath

    ' Add a column to the list.
    Set oMyNewColumn = oMyList.ListColumns.Add
    ' find the next element to map.
    strXPath = "/EmployeeSales/Employee/InvoiceNumber"
    ' Map the element.
    oMyNewColumn.XPath.SetValue oMyMap, strXPath

    ' Add a column to the list.
    Set oMyNewColumn = oMyList.ListColumns.Add
    ' find the next element to map.
    strXPath = "/EmployeeSales/Employee/InvoiceAmount"
```

```
' Map the element.
oMyNewColumn.XPath.SetValue oMyMap, strXPath

'Give the columns logical names
oMyList.ListColumns(1).Name = "EmployeeId"
oMyList.ListColumns(2).Name = "Invoice Number"
oMyList.ListColumns(3).Name = "Invoice Amount"
End Sub
```

Take a moment to analyze what you are doing here.

You are using the `Add` method of the `XmlMaps` collection to map a custom-made schema. You then specify the XML map you are using. In this scenario, the map is the first map in the spreadsheet, thus the index (1) is used. Next, you create a new worksheet and then add a list object to cell B5. This list object will contain the range of cells that will be mapped back to your schema. At this point, the list object you added only has one column.

There is a lot going on here. The net result of this line of code is that the `EmpId` element is mapped to the first column in your XML list. This is accomplished by using the `SetValue` method of the `XPath` object. The `XPath` object is used to connect a range or list column to an XML schema that has been mapped to a workbook. The `SetValue` expression requires two variables: an XML map and an XPath string. The XML map (represented here by the `MyMap` object variable) identifies the source schema to connect to, and the XPath string tells Excel which elements in the source schema to employ.

Don't be too concerned if you do not yet understand the XPath expression you see in the previous code. You will take a more detailed look at XPath in the next section of this chapter.

Add subsequent columns to the list object, and then use `XPath.SetValue` to map the new columns to the `InvoiceNumber` and `InvoiceAmount`, respectively. Finally, you give each column in the XML list a logical name.

Importing Data into an Existing XML Map

As you may have guessed, the `Import` method imports data from an XML file into an XML list or cells that have been mapped to a particular XML map. This method is particularly useful when building automated XML-based reporting processes where data is programmatically imported from XML documents in shared locations. In the example shown here, the code imports data from the `EmployeeSales.xml` file to the specified XML map:

```
Sub ImportXmlFromFile()
ThisWorkbook.XmlMaps("EmployeeSales_Map").Import _
(ThisWorkbook.Path & "\EmployeeSales.xml")
End Sub
```

XML Events

The `Workbook` object provides a few XML events that allow you to define and manage what happens before and after import and export procedures:

- ❑ **BeforeXMLImport:** This event is typically used to trap XML data before it is imported, allowing for data and environment evaluations.
- ❑ **AfterXMLImport:** This event is typically used to perform some post-processing operations after XML data has been imported.
- ❑ **BeforeXMLExport:** This event is typically used to trap XML data before it is exported, allowing for data and environment evaluations.
- ❑ **AfterXMLExport:** This event is typically used to perform some post-processing operations after XML data has been exported.

These events are also exposed through the `Application` object in the following forms: `WorkbookBeforeXMLImport`, `WorkbookBeforeXMLExport`, `WorkbookAfterXMLImport`, and `WorkbookAfterXMLExport`.

Exporting to an XML File

Similar to the `Import` method, the `Export` method allows for the exporting of data from an XML list or range of cells into an XML document. As you can see in the following code, the data is exported to a URL of your choice. Although the URL shown here points to a local location, the export URL can be on a network drive or web server. This method comes in handy when integrating data inputs from various users into a standardized data-gathering exercise:

```
Sub ExportToXmlFile()  
    ActiveWorkbook.XmlMaps("EmployeeSales_Map").Export _  
        URL:=ThisWorkbook.Path & "\Exported.xml"  
End Sub
```

Leveraging DOM and XPath to Manipulate XML Files

The Document Object Model (DOM) is a programming interface that exposes the contents of an XML document, allowing you to access and manipulate the data within. DOM is based on the Microsoft XML Parser (MSXML) that comes with Internet Explorer. Every version of Internet Explorer from Internet Explorer 5 on has its version of the built-in MSXML.DDL. The concept behind DOM is reasonably simple. After MSXML parses an XML document, the contents of the XML document are placed in data containers that can be navigated and surveyed by other applications. DOM exposes its own set of properties, methods, and parameters that allow any application that supports DOM to program against these parsed XML documents.

This section explores both DOM and XPath; discovering these two interfaces allows you to explore, modify, and manipulate XML documents. As with all XML technologies, both DOM and XPath are rich in scope and complexity. In that light, you only touch the surface of these technologies. However, after this section, you will be armed with enough information to explore and code against the Open XML file formats.

Loading XML into a DOM Document

Before you can do anything with DOM, you will have to set a reference to the MSXML object library. To do so, open the Visual Basic Editor and select Tools ⇨ References. In the References dialog box, select the latest version of Microsoft XML, as demonstrated in Figure 12-9.

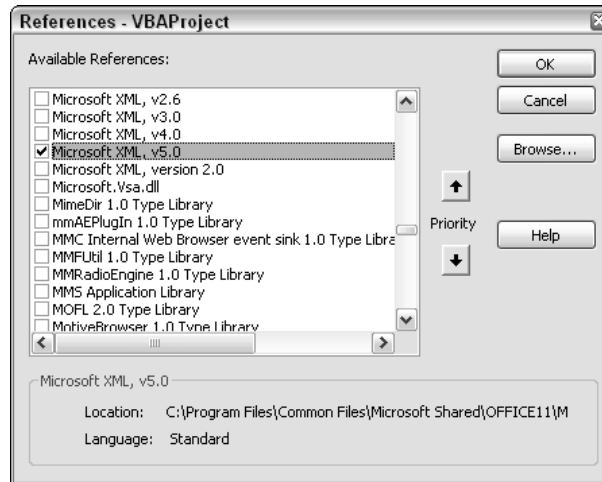


Figure 12-9

You will notice that there are four different versions of MSXML shown in Figure 12-9. A fifth version (Microsoft XML, v6.0) may be available to you if you have installed SQL Server 2005. You generally want to set your reference to the latest version available. However, if you are building XML-based solutions that will be distributed to other users, you will want to take into account the version of Internet Explorer your users have installed. If there is a good chance that some of your users are using Internet Explorer 5.0, you will want to set the reference to Microsoft XML, v3.0.

In any case, the differences between the various versions, for the most part, will not impact most of the tasks you as an Excel developer will need to accomplish. For instance, you can be confident that the rest of the procedures in this chapter would run fine with Microsoft XML, v3.0. If you have a burning need to find out what are the exact differences between the various versions of MSXML, you can find out here: <http://windowssdk.msdn.microsoft.com/en-us/library/ms753751.aspx>.

Once your reference is set, take a look at the code shown here. In this procedure, you are loading an XML file into a `DOMDocument` object. The `DOMDocument` object is the top-level container for the parsed XML file, serving as the parent for all nodes in the XML file's node structure. To programmatically explore and manipulate XML documents, they must first be exposed through the `DOMDocument` object.

```
Sub Load_ReadXMLDoc()  
Dim oMyDoc As DOMDocument  
  
'Create an instance of the DOMDocument  
Set oMyDoc = New DOMDocument  
  
'Disable asynchronous loading  
oMyDoc.async = False  
  
'Load XML information from a file  
oMyDoc.Load (ThisWorkbook.Path & "\SalesByRegion.xml")  
  
'Use the DOMDocument object's XML property to retrieve the raw data  
Debug.Print oMyDoc.XML  
  
'Cleanup  
Set oMyDoc = Nothing  
  
End Sub
```

First, instantiate a `DOMDocument` object, assigning it to the `MyDoc` variable. Next, disable asynchronous loading. Asynchronous loading is a process that MSXML uses by default to load documents in stages, allowing for cancellation and feedback during load. You generally will want to disable asynchronous loading to ensure that the document loads in its entirety before a result is returned.

Use the `Load` method to load the chosen XML document into the `DOMDocument` object. This essentially takes a snapshot of the XML document and loads an in-memory version of the XML document that you can explore and modify via VBA. In this example, you retrieve the raw XML data from the document and output it to the Immediate window. After you are done with any `DOMDocument`, you should always release it from memory.

Using DOM with ADO to Convert Excel Data to XML

A useful aspect of a `DOMDocument` object is that it can serve as the container for any hierarchical XML structure. This allows you to load any valid XML construct into a `DOMDocument` object. Coincidentally, ADO has an XML persistence constant that enables any recordset to persist in an XML stream.

ADO (ActiveX Data Objects) is a data access technology that is installed with Microsoft Data Access Components, and it's covered in detail in Chapter 20.

To use the procedure demonstrated here, you will need to first set a reference to Microsoft ActiveX Data Access 2.6 Library (any later version is also valid).

To help demonstrate this, open the `Programming XML.xlsm` file, found in the `XMLSampleFiles` folder in this chapter's download page at www.wrox.com. In this file, you will find the following procedure. This procedure loads an Excel range (in this case range A1:D43) into an ADO recordset, and then saves the recordset into the `DOMDocument` object, which is then output to an XML file:

```
Sub Convert_Excel_Data_to_XML()

Dim oMyconnection As Connection
Dim oMyrecordset As Recordset
Dim oMyXML As DOMDocument
Dim oMyWorkbook As String

Set oMyconnection = New Connection
Set oMyrecordset = New Recordset
Set oMyXML = New DOMDocument

'Identify the workbook you are referencing
oMyWorkbook = Application.ThisWorkbook.FullName

'Open connection to the workbook
oMyconnection.Open "Provider=Microsoft.Jet.OLEDB.4.0;" & _
    "Data Source=" & oMyWorkbook & ";" & _
    "Extended Properties=excel 8.0;" & _
    "Persist Security Info=False"

'Load the selected range into the recordset
oMyrecordset.Open "Select * from [Sheet1$A1:D43]", oMyconnection, adOpenStatic

'Load the recordset into the DOM Document
oMyrecordset.Save oMyXML, adPersistXML

'Save DOM Document to an xml file
oMyXML.Save (ThisWorkbook.Path & "\Output.xml")

'Clean up
oMyrecordset.Close
Set oMyconnection = Nothing
Set oMyrecordset = Nothing
Set oMyXML = Nothing

End Sub
```

Once the procedure is run, you will find the output XML file in the same directory as the `Programming XML.xlsm` file. When you open the output XML documents, you will notice that it does not look like the ones you have experienced here so far. This is because ADO produces attribute-based XML. An attribute-based XML document is almost exclusively made up of attribute nodes and is self-describing — they contain metadata that describes both the structure of the recordset and the data inside the recordset.

An XML file generated by ADO typically contains one root element and two child nodes: `Schema` and `Data`. The `Schema` node contains information about the recordset structure: field names, data type, field length, position, and so on. The `Data` node contains the actual data. Although attribute-based documents are difficult for humans to read, they are well formed and pose no problem for Excel.

When mapping an ADO-produced XML document, the XML Source pane will look similar to Figure 12-10. Remember to map the `Data` node to your spreadsheet.

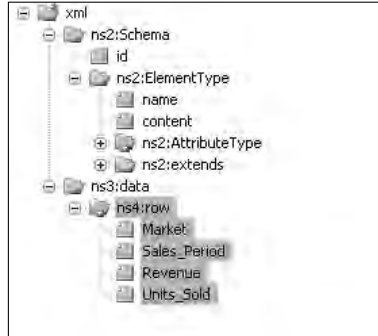


Figure 12-10

Traversing and Modifying XML Files with DOM and XPath

You'll remember from the start of this chapter that the construct of XML ensures that a parent/child hierarchy exists between all elements and attributes within an XML document. This gives XML documents an inherent logical structure that allows each construct in the document to be parsed into nodes.

XPath is a language that allows you to locate the component parts in an XML document by specifying a path to each node in the tree. With XPath, you can build an expression called a location path. A location path is similar to the file path you use to locate a file on your PC, starting at a given location, stepping through each directory in the file tree until you reach the file you want.

You compose your Xpath expression with location steps, evaluating the nodes at each step until all the location steps have been evaluated. The final step results in a node set that can be extracted, modified, deleted, and so on.

Take a moment to go through a few examples where combining DOM and XPath allows you to find, extract, and edit the data in a `DOMDocument`.

Keep in mind that the concepts presented here are key to your ability to program against the XML parts that make up your Excel file. Each one of the examples demonstrated here is employed later in this chapter when programming the Open XML file formats.

Return all Employee IDs

The first thing you will notice in the following example is that in addition to declaring a `DOMDocument` object, you are also declaring `IXMLDOMNode` and `IXMLDOMNodeList`. The `IXMLDOMNode` object is used to hold and pass a single node, whereas the `IXMLDOMNodeList` object is used to hold and pass a collection of nodes.

In this procedure, your goal is to return all Employee IDs contained in the chosen XML document. First, load the document in a `DOMDocument` object, as demonstrated in the previous section. Next, pass an XPath expression to the `SelectNodes` method of the `DOMDocument`:

```
"/EmployeeSales/Employee/Empid"
```


This expression steps through each element node in the document until it reaches the `Empid` element. As you can see, the XPath starts with the forward slash (`/`). A single forward slash establishes the root element of the document as the starting point for the location steps in the XPath expression. Each single forward slash from there steps through the hierarchy of elements until the one you need (the `Empid`) is reached. Once the destination path is reached, the `SelectNodes` method executes the query, trapping each element node in the `xmlnodes` object variable. It then iterates through `xmlnodes` to retrieve each single element node and output it:

```
Sub FindNode()  
Dim oXmlDoc As DOMDocument  
Dim oXmlNode As IXMLDOMNode  
Dim oXmlNodes As IXMLDOMNodeList  
  
'Load your XML Document into a DOM Document  
Set oXmlDoc = New DOMDocument  
oXmlDoc.async = False  
oXmlDoc.Load (ThisWorkbook.Path & "\EmployeeSales.xml")  
  
'Find and select the all EMPID nodes in the document  
Set oXmlNodes = oXmlDoc.SelectNodes("/EmployeeSales/Employee/Empid")  
  
'Iterate through the nodes and output each Empid  
For Each oXmlNode In oXmlNodes  
Debug.Print oXmlNode.Text  
Next  
End Sub
```

Return all Nodes for Any Employee with an Invoice Amount over \$3000

In this procedure, the goal is to return all the element nodes for any employees who have an invoice over \$3,000:

```
"//Employee[InvoiceAmount>3000]"
```

Notice in this expression the use of the double forward slashes (`//`). The double forward slash is an abbreviated syntax that can be used in any expression to jump directly to the desired node. This allows programmers to avoid the redundancy of declaring each step in the node hierarchy. In this example, you are selecting all `Employee` elements in the document that meet a specific criterion. You can pass a criterion through XPath by using a predicate placed in brackets (`[]`), as shown in previous procedure. Here, you are limiting the employees that are returned to only those whose `InvoiceAmount` element node has a value greater than 3000:

```
Sub FindNode()  
Dim oXmlDoc As DOMDocument  
Dim oXmlNode As IXMLDOMNode  
Dim oXmlNodes As IXMLDOMNodeList  
  
'Load your XML Document into a DOM Document  
Set oXmlDoc = New DOMDocument  
oXmlDoc.async = False  
oXmlDoc.Load (ThisWorkbook.Path & "\EmployeeSales.xml")  
  
'Find and select the all Employee that meet the criteria
```

```
Set oXmlNodes = oXmlDoc.SelectNodes("//Employee[InvoiceAmount>3000]")

'Iterate through the nodes and output each Employee
For Each oXmlNode In oXmlNodes
    Debug.Print oXmlNode.Text
Next
End Sub
```

Trap the Node that Contains the FirstName Mike

This procedure uses the same concepts to trap the node that contains the `FirstName` Mike. Note that you are using the `SelectSingleNode` method to trap and return one specific node. Other things to notice here are the use of the predicate `text()` followed by the criterion wrapped in single quotes, *not* double quotes. Because the `IXMLDOMNode` object does not hold a collection of nodes, there is no need to iterate through the nodes; you can simply output the node:

```
Sub FindNode()
Dim oXmlDoc As DOMDocument
Dim oXmlNode As IXMLDOMNode

    Set oXmlDoc = New DOMDocument
    oXmlDoc.async = False
    oXmlDoc.Load (ThisWorkbook.Path & "\EmployeeSales.xml")

    Set oXmlNode = oXmlDoc.SelectSingleNode("//FirstName[text()='Mike']")
    Debug.Print oXmlNode.XML
End Sub
```

Find and Edit the Node that Contains the FirstName Mike

In this procedure, you go a step further and actually change the text of the returned node. Because you are working with an in-memory version of the XML document, you must use the `Save` method of the `DOMDocument` to save changes back to the XML document:

```
Sub ChangeNode()
Dim oXmlDoc As DOMDocument
Dim oXmlNode As IXMLDOMNode

'Create a test copy of the EmployeeSales.xml file
FileCopy ThisWorkbook.Path & "\EmployeeSales.xml", _
    ThisWorkbook.Path & "\EmployeeSalesTest.xml"

'Load your XML Document into a DOM Document
Set oXmlDoc = New DOMDocument
oXmlDoc.async = False

    oXmlDoc.Load (ThisWorkbook.Path & "\EmployeeSalesTest.xml")

'Find and select the all Employee that meet the criteria
Set oXmlNode = oXmlDoc.SelectSingleNode("//FirstName[text()='Mike']")

'Edit the text and save back to the XML document
oXmlNode.Text = "Michael"
oXmlDoc.Save ThisWorkbook.Path & "\EmployeeSalesTest.xml"
End Sub
```

Find and Delete all Nodes for the Employee Bullen

In this example, first find the nodes where the employee's last name is Bullen. Once the correct set of nodes has been identified, iterate through the collection to remove each node. Here's how it works. Use each node as a starting point to find its parent by using the `parentnode` property. For instance, `xmlnode.parentnode` will return the parent of the `xmlnode` in focus. Once the parent is identified, use the `RemoveChild` method to remove the given `xmlnode`. After iterating through all the nodes in the collection, save the changes back to the XML document:

```
Sub DeleteNode()  
Dim oXmlDoc As DOMDocument  
Dim oXmlNode As IXMLDOMNode  
Dim oXmlNodes As IXMLDOMNodeList  
  
'Load your XML Document into a DOM Document  
Set oXmlDoc = New DOMDocument  
oXmlDoc.async = False  
oXmlDoc.Load (ThisWorkbook.Path & "\EmployeeSalesTest.xml")  
  
'Find and select the all Employee that meet the criteria  
Set oXmlNodes = oXmlDoc.SelectNodes("//Employee[LastName='Bullen']")  
  
'Find and select the all Employee that meet the criteria  
For Each oXmlNode In oXmlNodes  
oXmlNode.parentnode.RemoveChild oXmlNode  
Next  
oXmlDoc.Save ThisWorkbook.Path & "\EmployeeSalesTest.xml"  
End Sub
```

Using VBA to Program Open XML Files

You may be wondering why anyone would try to manipulate an Excel file by programming its XML parts. After all, doesn't Excel have a perfectly good object model? Well, there are a few benefits to be gained by programming against the Open XML files directly:

- ❑ **Encoding text files is always faster:** Remember that when you are programming against the Open XML files, you are essentially working with text files. Because there is very little overhead involved in programming XML files, your procedures will run more efficiently and far faster than they would using Excel's object model.
- ❑ **Find it, change it, get out:** Efficient relationship management, indexing, and shared files make it easy to find the exact component you need to manipulate. For instance, changing one string in the `Sharedstring.xml` file will apply that change to every instance of that string in your workbook. Changing a connection to external data is as easy as editing text within the `connections.xml`. Removing macros from a workbook is as easy as deleting the VBA project in the Excel container and updating the `.rels` XML file. With the Excel object model, you must negotiate over various objects, methods, and properties to make any change in the workbook.
- ❑ **Working with multiple workbooks:** When you need to make changes to dozens of files at one time, you typically apply some automation to instantiate a new instance of Excel to make the needed changes to that workbook. You then save that workbook and commence to open the

Chapter 12: Working with XML and the Open XML File Formats

next one to apply the same changes. With the Open XML files, there is no need to call Excel at all. You simply unpack the compressed container, edit the XML files, and repack. This not only simplifies the updating of multiple workbooks, but it speeds up the process considerably.

In this section, you get a glimpse of these benefits by walking through some examples of how you can use the techniques you have learned thus far to program Excel via its XML parts.

Programming Open XML Files with VBA

In this section, you will encounter three examples demonstrating how to automate the manipulation of the Open XML files. But first, let's take a look at how to unzip and zip your Excel containers via VBA.

Programmatically Unzipping an Excel Container

The unzip procedure is little more than a series of simple steps that duplicate the manual act of copying files out of an Excel container and saving them into a destination folder.

There are probably dozens of different methods and utilities that can be used to programmatically zip and unzip a compressed file. The procedures demonstrated here leverage the built-in file compression functionality within Windows XP. If you do not have Windows XP, you can use any one of dozens of compression software packages that provide command line and shell utilities for managing zip files. To find one, simply enter "Zip Command Line" in your favorite search engine.

First, load the name of your target Excel file to the `TargetFile` variable. The `NewFileName` is defined as the `TargetFile` string concatenated with the `.zip` extension. Next, use these variables in a `FileCopy` statement, essentially copying the target Excel file and saving it with a `.zip` file extension. This converts the target Excel file to a temporary `.zip` file while keeping the target file intact. Then create a destination folder and copy each of the XML parts located in the temporary `.zip` file into the destination folder. Once all XML parts have been copied, delete the temporary `.zip` file:

```
Sub UnzipPackage()  
  
Dim o As Object  
Dim TargetFile, NewFileName, DestinationFolder, ofile  
  
'Define the source file path and the path for the new file  
TargetFile = ThisWorkbook.Path & "\SalesByPeriod.xlsx"  
NewFileName = TargetFile & ".zip"  
  
'Create the temp zip File  
FileCopy TargetFile, NewFileName  
  
'Define a destination folder path and Make the destination folder  
DestinationFolder = "C:\MyUnzipped"  
On Error Resume Next  
MkDir (DestinationFolder)  
  
'Copy each file to the destination folder  
Set o = CreateObject("Shell.Application")  
For Each ofile In o.Namespace(NewFileName).items
```

```
        o.Namespace(DestinationFolder).CopyHere (ofile)
    Next ofile

'Clean up
    Kill NewFileName
    Set o = Nothing

End Sub
```

Programmatically Zipping an Excel Container

In the `ZipPackage` procedure demonstrated here, you are creating an empty `.zip` file and then filling it with the contents of a source directory. Notice that you are using the `Sleep` API function here. This lets you pause Excel for a specified number of milliseconds. Pausing Excel allows each file to be completely compressed and saved before moving on the next file. In this procedure, you are making Excel sleep for 500 milliseconds each time you copy a file to the `.zip` container:

```
Public Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long)

Sub ZipPackage()
    Dim ZipFile, TargetFolder, NewFileName, ofile
    Dim o As Object

    'Create Empty Zip Package
    ZipFile = "C:\UpdatedFile.zip"
    Open ZipFile For Output As #1
    Print #1, Chr$(80) & Chr$(75) & Chr$(5) & Chr$(6) & String(18, 0)
    Close #1

    'Identify Folder with Source Files
    TargetFolder = "C:\MyUnzipped"

    'Check for empty folder
    If Len(Dir$(TargetFolder & "\*.*)" ) < 1 Then
        MsgBox "There are no files in your target folder"
        Kill ZipFile
        Exit Sub
    End If

    'Copy each file to the zip file
    On Error Resume Next
    Set o = CreateObject("Shell.Application")
    For Each ofile In o.Namespace(TargetFolder).items
        o.Namespace(ZipFile).CopyHere (ofile)
        Sleep 500
    Next ofile

    'Rename the container to change the file extension to.xlsx
    Name ZipFile As Replace$(ZipFile, ".zip", ".xlsx")

    'Clean up
    Kill ZipFile
    Set o = Nothing

End Sub
```

Edit the sharedStrings XML File to Implement Mass Updates to Text

As mentioned before, the `sharedStringsXML` part holds all of the strings used in the Excel file. These strings are referenced via the shared index number by each sheet in the container to apply them to the correct cell. One nifty trick is to change a string in the `sharedStrings` file and watch that change take effect in your Excel file.

For example, in the `SalesByPeriod.xlsx` sample file, certain records are tagged with the market South America. Suppose you wanted to change all instances of South America to Latin America. All you would have to do is open the `sharedStrings` file, find South America, and change it to Latin America. After repackaging the Excel file, you will see that every instance of South America has been changed to Latin America. Although you could make this change manually, imagine doing this for dozens of files.

The good news is that you can automate this process using MSXML DOM and XPath. The procedure shown here demonstrates how:

```
Sub Change_SharedString_File()
    Dim oXmlDoc As DOMDocument
    Dim oXmlNode As IXMLDOMNode

    'Run the Unzip procedure
    Call UnzipPackage

    'Create an instance of the DOMDocument and load XML file
    Set oXmlDoc = New DOMDocument
    oXmlDoc.async = False
    oXmlDoc.Load ("C:\MyUnzipped\xl\sharedstrings.xml")

    'Pass XPath to find the text that needs to be changed
    Set oXmlNode = oXmlDoc.SelectSingleNode("//t[text()='South America']")

    'Make sure text exists
    If oXmlNode Is Nothing Then
        Exit Sub
    End If

    'Change the text and save your changes
    oXmlNode.Text = "Latin America"
    oXmlDoc.Save "C:\MyUnzipped\xl\sharedstrings.xml"

    'Run the Zip procedure
    Call ZipPackage

    'Ready message
    MsgBox "Find your updated file here:" & vbCrLf & "C:\UpdatedFile.xlsx"
    Set oXmlNode = Nothing
    Set oXmlDoc = Nothing

End Sub
```

To understand what is going on here, evaluate this code in steps:

1. Run the `UnzipPackage` procedure you created previously.
2. Create an instance of the `DOMDocument` object and load the `sharedstrings.xml` file.
3. Use an XPath expression to find the `t` node that contains the text 'South America'.
4. Check to make sure the text exists; if it doesn't, exit the procedure.
5. Edit the text to "Latin America" and save the change back to the `sharedstrings.xml` file.
6. Repackage the Excel file using the `ZipPackage` procedure created previously.
7. Clean up and output a message.

After running the procedure, open the `C:\UpdatedFile.xlsx` file to see that all of the records that were tagged as South America are now tagged Latin America.

Unprotect a Worksheet via Open XML Manipulation

It may seem rather amazing that all you have to do to unprotect a worksheet is delete the `sheetProtection` element in that sheet's XML part. That's right; simply removing the `sheetProtection` element from the XML part negates all protections placed on that sheet. The following procedure demonstrates how to unprotect a worksheet by manipulating its XML part; here you will unprotect `Sheet1` in the `SalesByPeriod.xlsx` sample file:

```
Sub RemovePasswordProtection()  
Dim oXmlDoc As DOMDocument  
Dim oXmlNode As IXMLDOMNode  
  
'Run the Unzip procedure  
Call UnzipPackage  
  
'Create an instance of the DOMDocument and load XML file  
Set oXmlDoc = New DOMDocument  
oXmlDoc.async = False  
oXmlDoc.Load ("C:\MyUnzipped\xl\worksheets\sheet1.xml")  
  
'Find and remove the sheetprotection element  
Set oXmlNode = oXmlDoc.SelectSingleNode("//sheetProtection")  
oXmlNode.parentNode.RemoveChild oXmlNode  
  
'Save Changes  
oXmlDoc.Save "C:\MyUnzipped\xl\worksheets\sheet1.xml"  
  
'Run the Zip procedure  
Call ZipPackage  
  
'Ready message and clean up  
MsgBox "Find your updated file here:" & vbCrLf & "C:\UpdatedFile.xlsx"  
Set oXmlNode = Nothing  
Set oXmlDoc = Nothing  
  
End Sub
```

Chapter 12: Working with XML and the Open XML File Formats

In this procedure, you:

1. Run the `UnzipPackage` procedure created previously.
2. Create an instance of the `DOMDocument` object and load the `sharedstrings.xml` file.
3. Use an XPath expression to find the `sheetProtection` node and then remove the node.
4. Save the change back to the sheet's XML file.
5. Repackage the Excel file using the `ZipPackage` procedure created previously.
6. Clean up and output a message.

Once your procedure has run its course, open the `C:\UpdatedFile.xlsx` file to see that `Sheet1` is now unprotected.

Updating Connection Strings

A particularly useful trick is to use the techniques in this chapter to change the connection strings to the external data sources in your files. For example, the sample file `SalesByPeriod.xlsx` contains an external data source that comes from the Facility Services Access database in the same directory. If you were to move the database, the connection to the external data source would be severed. Suppose you had dozens of files that were linked to a data source that moved or was renamed. You would have dozens of files that would have severed links that would need to be fixed.

In this walkthrough, you update the connection strings in an Excel file by coding against the workbook's `connections.xml` file. To prepare for this walkthrough, create a new directory on your `C:\` drive and call it `NewLocation`. Next, move the Facility Services Access database found in the sample folder for this chapter to your newly created directory.

Next, run the code you see here:

```
Sub Change_ConnectionString()  
    Dim oxmlDoc As DOMDocument  
    Dim oxmlNode As IXMLDOMNode  
    Dim StrOldLocation As String  
    Dim StrNewLocation As String  
  
    'Define the old and new location paths  
    StrOldLocation = "C:\VBA Reference\SampleFiles\Facility Services.accdb"  
    StrNewLocation = "C:\StrNewLocation\Facility Services.accdb"  
  
    'Run the Unzip procedure  
    Call UnzipPackage  
  
    'Create an instance of the DOMDocument and load XML file  
    Set oxmlDoc = New DOMDocument  
    oxmlDoc.async = False  
    oxmlDoc.Load ("C:\MyUnzipped\xl\connections.xml")  
  
    'Pass Xpath to find the SourceFile attribute
```



```
Set oxmlNode = oxmlDoc.SelectSingleNode("/connections/connection/@sourceFile")

'Replace the old string with new string and save changes
  oxmlNode.Text = Replace$(oxmlNode.Text, StrOldLocation, StrNewLocation)
  oxmlDoc.Save ("C:\MyUnzipped\xl\connections.xml")

'Pass Xpath to find the connection string attribute
Set oxmlNode =
oxmlDoc.SelectSingleNode("/connections/connection/dbPr/@connection")

'Replace the old string with new string and save changes
  oxmlNode.Text = Replace$(oxmlNode.Text, StrOldLocation, StrNewLocation)
  oxmlDoc.Save ("C:\MyUnzipped\xl\connections.xml")

'Run the Zip procedure
  Call ZipPackage

'Ready message
  MsgBox "Find your updated file here:" & vbCrLf & "C:\UpdatedFile.xlsx"
  Set oxmlNode = Nothing
  Set oxmlDoc = Nothing

End Sub
```

Here are the steps involved in this procedure:

1. Store the old and new location paths in strings for later use.
2. Run the `UnzipPackage` procedure.
3. Create an instance of the `DOMDocument` object and load the `connections.xml` file.
4. Use an XPath expression to locate and trap the `sourcefile` attribute for the `connection` element.

The at symbol (@) is the abbreviated syntax for *attribute*. Placing @ directly in front of a location step in an XPath expression identifies that step as an attribute node.

5. Use the `Replace` expression to find any part of the attribute's text containing the old location path and replace that path with the new location path. Then save the change to the `connections.xml` file.
6. Use an XPath expression to locate and trap the `connection` attribute for the `dbPR` element.
7. Use the `Replace` expression to find any part of the attribute's text containing the old location path and replace that path with the new location path. Then save the change to the `connections.xml` file.
8. Repackage the Excel file using the `ZipPackage` procedure created previously.
9. Clean up and output a message.

Chapter 12: Working with XML and the Open XML File Formats

After running the procedure, open the `C:\UpdatedFile.xlsx` file and refresh the employee table located on Sheet2.

Summary

This chapter brought you closer to understanding XML and seeing the potential impact that XML, if adopted, could have on your Excel processes and procedures. Can you continue to work with Excel without using XML? Sure you can. But XML is not the paperless office. XML will not go away anytime soon. It is fast becoming an industry standard that is used in an increasing number of environments. The new Open XML file formats promote interoperability, exposing new opportunities for Excel programmers to integrate with and expand to platforms that were previously out of scope. Congratulations, you have taken your first steps toward being able to develop the next generation of Excel solutions.

UserForms

UserForms are essentially user-defined dialog boxes. You can use them to display information and to allow the user to input new data or modify the displayed data. The `MsgBox` and `InputBox` functions provide simple tools to display messages and get input data, respectively, but UserForms take you to a new dimension. With these, you can implement nearly all the features that you are accustomed to seeing in normal Windows dialog boxes.

You create a UserForm in the VBE window using `Insert ⇨ UserForm`. You add controls from the Toolbox in the same way that you add controls to a worksheet. If the Toolbox is not visible, use `View ⇨ Toolbox`.

UserForms can contain Labels, TextBoxes, ListBoxes, ComboBoxes, CommandButtons, and many other ActiveX controls. You have complete control over the placement of controls and can use as many controls as you need. Naturally, each control can respond to a wide variety of events.

Displaying a UserForm

To load a UserForm called `UserForm1` into memory, without making it visible, you use the `Load` statement:

```
Load UserForm1
```

You can remove `UserForm1` from memory using the `Unload` statement:

```
Unload UserForm1
```

To make `UserForm1` visible, use the `Show` method of the `UserForm` object:

```
UserForm1.Show
```

Chapter 13: UserForms

If you show a UserForm that has not been loaded, it will be automatically loaded. You can use the `Hide` method to remove a UserForm from the screen without removing it from memory:

```
UserForm1.Hide
```

Figure 13-1 shows a simple UserForm in action that will be developed over the course of this chapter. It has been designed to allow you to see the current values in cells B2:B6 and to make changes to those values. It is linked directly to the cells in the worksheet, which makes it very easy to set up with a minimum of VBA code.

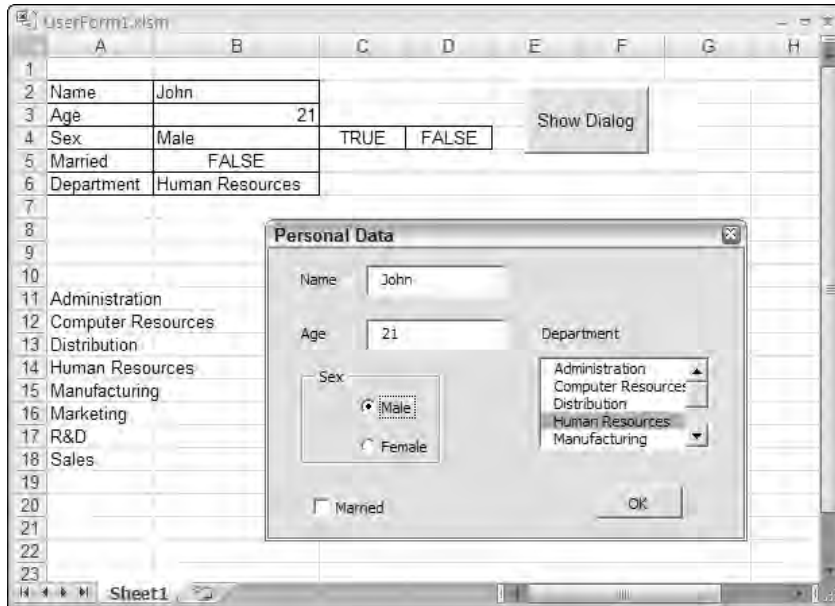


Figure 13-1

The ActiveX command button in the worksheet, with the caption `Show Dialog`, contains the following event procedure:

```
Private Sub cmdShowUserForm_Click()  
    frmPersonal.Show  
End Sub
```

The UserForm is modal by default. This means that the UserForm retains the focus until it is unloaded or hidden. The user cannot activate the worksheet or click Ribbon buttons until the UserForm is closed.

Modeless UserForms, which do allow the user to perform other tasks while they are visible, are discussed later in this chapter.

Creating a UserForm

Figure 13-2 shows the UserForm in the VBE window.

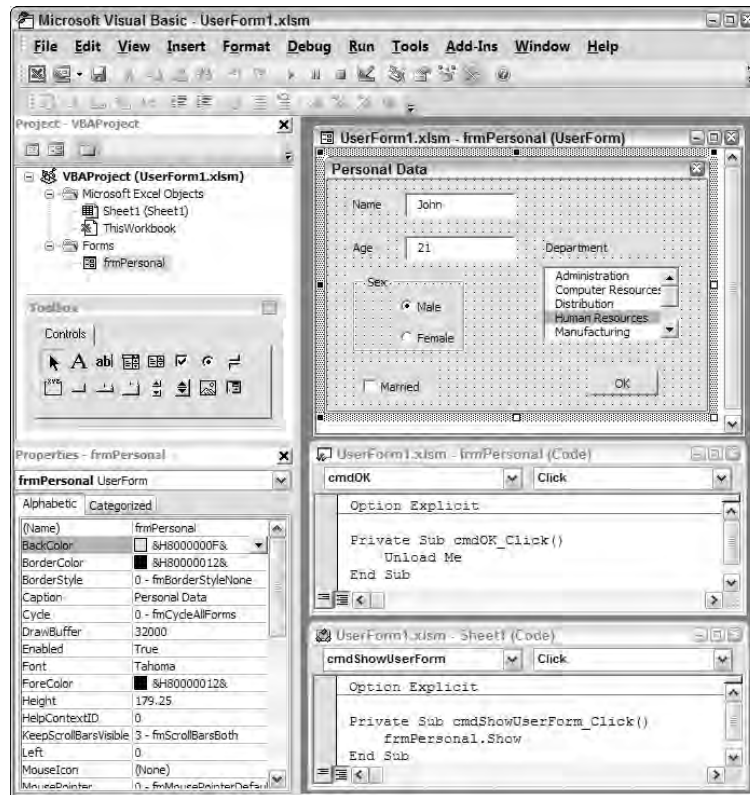


Figure 13-2

The name of the UserForm was changed from the default name `UserForm1` to `frmPersonal`. You do this in the first entry, (Name), in the Properties window. The `Caption` property is changed to `Personal Data`. The controls were added from the Toolbox.

There are two `TextBox` controls at the top of the form for name and age data. There are two option buttons (also known as radio buttons) for `Male` and `Female`, which are inside a frame control.

When you want to have a frame around other controls, you must insert the frame first, and then insert the controls into the frame.

There is also a `CheckBox` for `Married`, a `ListBox` for `Department`, and a `CommandButton` for `OK`.

It is a good idea to give your UserForms and controls descriptive names that identify what type of object they are and what their purpose is. The lowercase three-character prefix identifies the object type. For example, you use `frm` for a UserForm, `scb` for a scrollbar, and `txt` for a TextBox. The capitalized words that follow identify the control's purpose. This makes it much easier to write and maintain the VBA code that manipulates these objects.

The name of the first TextBox was changed to `txtName`, and the `ControlSource` property of `txtName` was entered as `Sheet1!B2`. The name of the second TextBox was changed to `txtAge`, and the `ControlSource` property of `txtAge` was entered as `Sheet1!B3`. Similar changes were made to the other main controls. The changes are summarized in the following table.

Control	Name	ControlSource
TextBox	<code>txtName</code>	<code>Sheet1!B2</code>
TextBox	<code>txtAge</code>	<code>Sheet1!B3</code>
OptionButton	<code>optMale</code>	<code>Sheet1!C4</code>
OptionButton	<code>optFemale</code>	<code>Sheet1!D4</code>
CheckBox	<code>chkMarried</code>	<code>Sheet1!B5</code>
ListBox	<code>lstDepartment</code>	<code>Sheet1!B6</code>
CommandButton	<code>cmdOK</code>	

When you assign a `ControlSource` property to a worksheet cell, the cell and the control are linked in both directions. Any change to the control affects the cell, and any change to the cell affects the control.

The descriptive titles on the form to the left of the TextBoxes and above the ListBox show that the departments are Label controls. The `Caption` properties of the Label controls were changed to `Name`, `Age`, and `Department`. The `Caption` property of the frame around the OptionButton controls was changed to `Sex`, and the `Caption` properties of the option buttons were changed to `Male` and `Female`. The `Caption` property of the CheckBox was changed to `Married`.

The Male and Female option buttons can't be linked to B4. It is not appropriate to display the values of these controls directly, so the following IF function in cell B4 converts the `True` or `False` value in cell C4 to the required Male or Female result:

```
=IF (C4=TRUE, "Male", "Female")
```

Although you only need to set cell C4 to get the required result, you need to link both option buttons to separate cells if you want the buttons to display properly when the UserForm is shown.

The `RowSource` property of `lstDepartment` was entered as `Sheet1!A11:A18`. It is good practice to create names for the linked cells and use those names in the `ControlSource`, rather than the cell references used here, but this extra step has been omitted to simplify this example.

The following `Click` event procedure was created for the button in the code module behind the UserForm:

```
Private Sub cmdOK_Click()
    Unload Me
End Sub
```

`Me` is a shortcut keyword that refers to the `UserForm` object containing the code. `Me` can be used in any class module to refer to the object the class module represents. If you want to access the control values later in your VBA code, you must use the `Hide` method, which leaves the UserForm in memory. Otherwise, the `Unload` statement removes the UserForm from memory and the control values are lost. You will see examples that use `Hide` shortly.

Clicking the `x` in the top-right corner of the UserForm will also dismiss the UserForm. This unloads the UserForm so that it is removed from memory. You will see how to prevent this later.

Directly Accessing Controls in UserForms

Linking UserForm controls to cells is not always the best way to work. You can gain more flexibility by directly accessing the data in the UserForm. Figure 13-3 shows a revised version of the previous example. You want to display essentially the same UserForm, but you want to store the resulting data as shown. `Sex` will be stored as a single-letter code, `M` or `F`. The `Department` name will be stored as a two-character code.

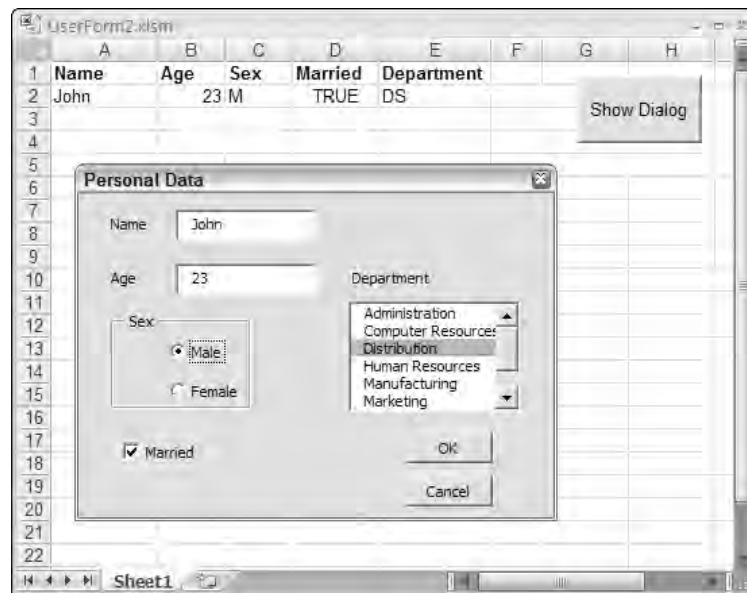


Figure 13-3

Chapter 13: UserForms

A Cancel button has been added to the UserForm so that any changes made to the controls while the UserForm is being shown can be discarded if the user wishes, rather than being automatically applied to the worksheet. The module behind `frmPersonal` now contains the following code:

```
Option Explicit
Public Cancelled As Boolean

Private Sub cmdCancel_Click()
    Cancelled = True
    Me.Hide
End Sub

Private Sub cmdOK_Click()
    Cancelled = False
    Me.Hide
End Sub
```

The Public variable `Cancelled` will provide a way to detect that the Cancel button has been clicked. If the OK button is clicked, `Cancelled` is assigned the value `False`. If the Cancel button is clicked, `Cancelled` is assigned a value of `True`. Both buttons hide `frmPersonal` so it remains in memory. The following event procedure has also been added to the module behind `frmPersonal`:

```
Private Sub UserForm_Initialize()
    Dim vDepartment As Variant
    Dim vDeptCode As Variant
    Dim saDeptList() As String
    Dim i As Integer

    'Department name array
    vDepartment = VBA.Array("Administration", _
        "Computer Resources", _
        "Distribution", _
        "Human Resources", _
        "Manufacturing", _
        "Marketing", _
        "R&D", _
        "Sales")

    'Department code array
    vDeptCode = VBA.Array("AD", _
        "CR", _
        "DS", _
        "HR", _
        "MF", _
        "MK", _
        "RD", _
        "SL")

    'Assign array values to string array
    ReDim saDeptList(0 To UBound(vDepartment), 0 To 1)

    For i = 0 To UBound(vDepartment)
        saDeptList(i, 0) = vDeptCode(i)
        saDeptList(i, 1) = vDepartment(i)
    End For
End Sub
```



```

Next i

'Assign string array to list box
lstDepartment.List = saDeptList
End Sub

```

The `UserForm_Initialize` event is triggered when the UserForm is loaded into memory. It does not occur when the form has been hidden and is shown again. It is used here to load `lstDepartment` with two columns of data. The first column contains the department codes, and the second contains the department names to be displayed.

`vDepartment` and `vDeptCode` are assigned arrays in the usual way using the `Array` function, except that `VBA.Array` has been used to ensure that the arrays are zero-based. `saDeptList` is a dynamic array, and `ReDim` is used to dimension it to the same number of rows as in `vDepartment` and two columns, once again zero-based.

The `For . . . Next` loop assigns the department codes and names to the two columns of `saDeptList`. `saDeptList` is then assigned directly to the `List` property of `lstDepartment`. If you prefer, you can maintain a table of departments and codes in a worksheet range and set the `ListBox` control's `RowSource` property equal to the range you saw in the first example in this chapter.

When you have a multi-column `ListBox`, you need to specify which column contains the data that will appear in a link cell and be returned in the control's `Value` property. This column is referred to as the bound column. The `BoundColumn` property of `lstDepartment` has been set to 1. This property is one-based, so the bound column is the department code. The `ColumnCount` property has been set to 2, because there are two columns of data in the list.

However, you only want to see the department names in the `ListBox`, so you want to hide the first column. You can do that by setting the column width of the first column to 0. To do this, you only need to enter a single 0 in the `ColumnWidths` property, rather than, for example, 0;40. Entering a single 0 sets the first column to a width of 0 and leaves the second column to fill the `ListBox` width.

The following code has been placed in the module behind `Sheet1`:

```

Private Sub cmdShowUserForm_Click()
    Dim rngData As Range
    Dim vData As Variant

    'First block of code

    'Get current data values
    Set rngData = Range("Database").Rows(2)
    vData = rngData.Value

    With frmPersonal
        'Load data into controls
        .txtName.Value = vData(1, 1)
        .txtAge.Value = vData(1, 2)
        Select Case vData(1, 3)
        Case "F"
            .optFemale.Value = True

```

```
Case "M"
    .optMale.Value = True
End Select
.chkMarried.Value = vData(1, 4)
.lstDepartment.Value = vData(1, 5)

'Second block of code

'Display UserForm
.Show

'Third block of code

'Continue if Cancel not clicked
If Not .Cancelled Then

    'Assemble data in vData
    vData(1, 1) = .txtName
    vData(1, 2) = .txtAge
    Select Case True
    Case .optFemale.Value
        vData(1, 3) = "F"
    Case .optMale.Value
        vData(1, 3) = "M"
    End Select
    vData(1, 4) = .chkMarried.Value
    vData(1, 5) = .lstDepartment.Value

    'Transfer data to worksheet
    rngData.Value = vData

End If

End With

Unload frmPersonal

End Sub
```

The code is in three blocks after the initial declaration statements. The first block loads the data from the worksheet into `frmPersonal`. The second block (only two lines) displays `frmPersonal`, then checks to see if the Cancel button is clicked. The third block copies the data in `frmPersonal` back to the worksheet.

At the start of the first block, `rngData` is assigned a reference to cells A2:E2. The range A1:E2 has been given the name `Database`, so `Range("Database").Rows(2)` refers to the required data range. The values in `rngData` are then assigned directly to the variant `vData`. This creates a two-dimensional, one-based array of values having one row and five columns. It is much more efficient to access the worksheet data in this way, rather than to access each cell individually.

Most of the remaining code is within the `With... End With` structure, which makes it possible to use shorter and more efficient references to the controls, properties, and methods associated with `frmPersonal`. The first reference to `frmPersonal` also causes `frmPersonal` to be loaded into memory, although it remains hidden at this point.

The `Value` properties of the controls on `frmPersonal` are then assigned the values in `vData`. The option buttons are an exception, because the `M` and `F` code values need to be translated to `True` values as appropriate. It is only necessary to set one of the option buttons to `True`, because the other will automatically be set to `False`. You can group option buttons by assigning them the same value in their `GroupName` property, or by placing them in the same frame. The option buttons here do not have a value in their `GroupName` property. They are considered to be in the same group because they are in the same frame.

The `Show` method displays `frmPersonal`. Control then passes to `frmPersonal` until it is hidden, which occurs when the user clicks `OK` or `Cancel`, or the `x` at the top of the `UserForm`. The user can also press `Esc` to activate the `Cancel` button because it has had its `Cancel` property set to `True`. The user can also press `Enter` to activate the `OK` button, as long as the `Cancel` button does not have the focus, because the `OK` button's `Default` property has been set to `True`.

When `frmPersonal` is hidden, the `cmdShowUserForm_Click` event procedure regains control, and checks to see if the `Cancel` button was clicked. It does this by examining the value of the `Public` variable `Cancelled` on `frmPersonal`.

The code modules behind UserForms (as well as those behind sheets and workbooks) are class modules. When you define a Public variable in a class module, the variable behaves as a property of the object associated with the class module. See Chapter 16 for more details.

If `Cancelled` is `False`, the procedure loads the values of the controls back into `vData`, translating the option button settings back into an `F` or `M` value, and the values in `vData` are directly assigned back to the worksheet. The final step is to unload `frmPersonal` from memory.

Stopping the Close Button

One problem with the previous code is that, if the user clicks the `x`, which is the `Close` button at the top of `frmPersonal`, the event procedure does not exit. Instead, it transfers any changes back to the worksheet. This is because the default value for `Cancelled` is `False`. Normally, clicking the `x` would also unload the form and the code would fail when it tries to access the controls on the form. However, in this case the `With...End With` structure keeps `frmPersonal` in scope, and `frmPersonal` is not unloaded until after the `End With` statement.

There are a number of simple ways in which the preceding problem could be corrected, but the following method gives you total control over that little `x`. You can use the `QueryClose` event of the `UserForm` object to discover what is closing the `UserForm` and cancel the event if necessary. Adding the following code to the `frmPersonal` module blocks the `Close` button exit:

```
Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)
    If CloseMode = vbFormControlMenu Then
        MsgBox "Please use only the OK or Cancel buttons", vbCritical
        Cancel = True
    End If
End Sub
```

Chapter 13: UserForms

The `QueryClose` event can be triggered in four ways. You can determine what caused the event by using the following intrinsic constants to test the `CloseMode` parameter.

Constant	Value	Reason for the Event
<code>vbFormControlMenu</code>	0	The user clicked the x in the Control menu on the UserForm.
<code>vbFormCode</code>	1	The <code>Unload</code> statement was used to remove the UserForm from memory.
<code>vbAppWindows</code>	2	Windows is shutting down.
<code>vbAppTaskManager</code>	3	The application is being closed by the Windows Task Manager.

Maintaining a Data List

The code you have developed can now be extended to maintain a data list without too much extra effort. However, the last example takes a different approach. This time you will build all the code into `frmPersonal`, apart from the code behind the command button in the worksheet that shows the UserForm. The code behind this button now becomes the following:

```
Private Sub cmdShowUserForm_Click()  
    frmPersonal.Show  
End Sub
```

It is really much easier to maintain a data list in a proper database application, such as Microsoft Access, but it can be done in Excel without too much trouble if your requirements are fairly simple.

If you are going to manage more than one row of data, you need to be able to add new rows, delete existing rows, and navigate through the rows. `frmPersonal` needs some extra controls, as shown in Figure 13-4.

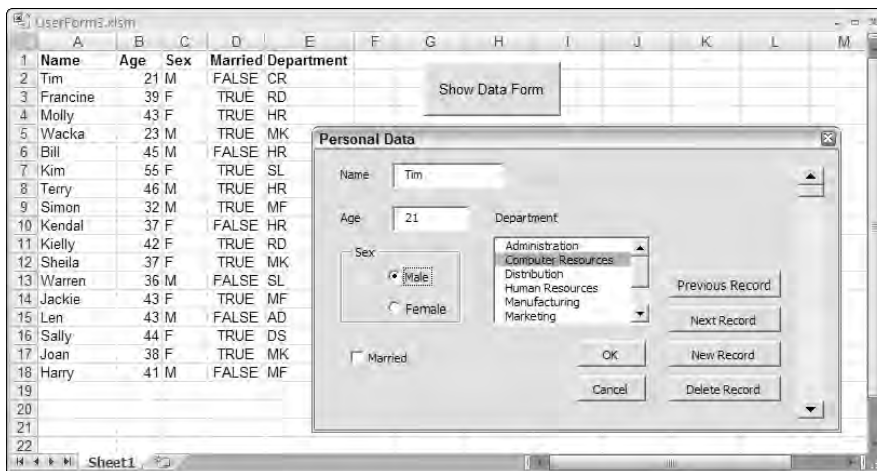


Figure 13-4

The scrollbar is a handy way to navigate through many records quickly. It also makes it easy to get to the last or first record. It can also be used to go to the next or previous record. For variety, buttons to go to the next and previous records are included as well. The New Record button adds a record to the end of the data list and initializes some of the values in the new record. The Delete button deletes the record that is currently showing in `frmPersonal`.

The code in `frmPersonal` is discussed next. It is important to note first that the following module-level variables have been declared in the (declarations) section at the top of the `frmPersonal` code module:

```
Dim mrngData As Range
Dim mvData As Variant
```

These variables are used in exactly the same way as they were used in the previous example, except that the row referred to can vary. The object variable `mrngData` is always set to the current row of data in the named range `Database`, which currently refers to `A1:E18` in the worksheet shown in Figure 13-4. `mvData` always holds the values from `mrngData` as a VBA array.

The code from the command button event procedure in the previous example has been converted to two utility procedures that reside in `frmPersonal`'s code module:

```
Private Sub LoadRecord()
    'Copy values in mrngData from worksheet to mvData array
    mvData = mrngData.Value

    'Assign array values to frmPersonal controls
    txtName.Value = mvData(1, 1)
    txtAge.Value = mvData(1, 2)

    Select Case mvData(1, 3)
        Case "F"
            optFemale.Value = True
        Case "M"
            optMale.Value = True
    End Select

    chkMarried.Value = mvData(1, 4)
    lstDepartment.Value = mvData(1, 5)

End Sub

Private Sub SaveRecord()
    'Copy values from frmPersonal controls to Data array
    mvData(1, 1) = txtName.Value
    mvData(1, 2) = txtAge.Value

    Select Case True
        Case optFemale.Value
            mvData(1, 3) = "F"
        Case optMale.Value
            mvData(1, 3) = "M"
```

Chapter 13: UserForms

```
End Select

mvData(1, 4) = chkMarried.Value
mvData(1, 5) = lstDepartment.Value

'Assign Data array values to current record in Database
mrngData.Value = mvData

End Sub
```

Because the code is in the `frmPersonal` module, there is no need to refer to `frmPersonal` when referring to a control, so all controls are directly addressed in the code.

`LoadRecord` and `SaveRecord` are the only procedures tailored to the data list structure and the controls. As long as the data list has the name `Database`, none of the other code in `frmPersonal` needs to change if you decide to add more fields to the data list or remove fields. It also means that you can readily apply the same code to a completely different data list. All you have to do is redesign the UserForm controls and update `LoadRecord` and `SaveRecord`.

The key navigation device in `frmPersonal` is the scrollbar, which has been named `scbNavigator`. It is used by the other buttons when a change of record is required, as well as being available to the user directly. The `Value` property of `scbNavigator` corresponds to the row number in the range named `Database`.

The minimum value of `scbNavigator` is fixed permanently at 2, because the first record is the second row in `Database`, so you need to set the `Min` property of the scrollbar in the Properties window. The maximum value is altered as needed by the other event procedures in `frmPersonal` so it always corresponds to the last row in `Database`:

```
Private Sub scbNavigator_Change()

    'When Scrollbar value changes, save current record and load
    'record number corresponding to scrollbar value
    Call SaveRecord

    Set mrngData = Range("Database").Rows(scbNavigator.Value)

    Call LoadRecord

End Sub
```

When the user changes the `scbNavigator.Value` property (or when it is changed by other event procedures), the `Change` event fires and saves the current record in `frmPersonal`, redefines `mrngData` to be the row in `Database` corresponding to the new value of `scbNavigator.Value`, and loads the data from that row into `frmPersonal`.

The `UserForm_Initialize` event procedure has been updated from the previous exercise to set the correct starting values for `scbNavigator`:

```

Private Sub UserForm_Initialize()
    Dim vDepartment As Variant
    Dim vDeptCode As Variant
    Dim saDeptList() As String
    Dim i As Integer

    vDepartment = VBA.Array("Administration", _
        "Computer Resources", _
        "Distribution", _
        "Human Resources", _
        "Manufacturing", _
        "Marketing", _
        "R&D", _
        "Sales", _
        "None")

    vDeptCode = VBA.Array("AD", _
        "CR", _
        "DS", _
        "HR", _
        "MF", _
        "MK", _
        "RD", _
        "SL", _
        "NA")

    ReDim saDeptList(0 To UBound(vDepartment), 0 To 1)

    For i = 0 To UBound(vDepartment)
        saDeptList(i, 0) = vDeptCode(i)
        saDeptList(i, 1) = vDepartment(i)
    Next i

    lstDepartment.List = saDeptList

    'Load 1st record in Database and initialize scrollbar
    With Range("Database")
        Set mrngData = .Rows(2)
        Call LoadRecord
        scbNavigator.Value = 2
        scbNavigator.Max = .Rows.Count
    End With

End Sub

```

After initializing the `lstDepartment.List` property, the code initializes `mrngData` to refer to the second row of `Database`, row two being the first row of data under the field names on row one, and loads the data from that row into `frmPersonal`. It then initializes the `Value` property of `scbNavigator` to 2 and sets the `Max` property of `scbNavigator` to the number of rows in `Database`. If the user changes the scrollbar, he can navigate to any row from row two through the last row in `Database`.

Chapter 13: UserForms

The buttons captioned Next Record and Previous Record have been named `cmdNext` and `cmdPrevious`. The `Click` event procedure for `cmdNext` is as follows:

```
Private Sub cmdNext_Click()

    With Range("Database")
        If mrngData.Row < .Rows(.Rows.Count).Row Then
            'Load next record only if not on last record
            scbNavigator.Value = scbNavigator.Value + 1
            'Note: Setting sbNavigator.Value runs its Change event procedure
        End If
    End With

End Sub
```

The `If` test checks that the current row number in `Database` is less than the last row number in `Database`, to ensure that you don't try to go beyond the data. If there is room to move, the value of `scbNavigator` is increased by one. This change triggers the `Change` event procedure for `scbNavigator`, which saves the current data, resets `mrngData`, and loads the next row's data.

The code for `cmdPrevious` is similar to `cmdNext`, except that there is no need for the `With...End With` because you don't need to keep repeating the reference to `Range("Database")`:

```
Private Sub cmdPrevious_Click()

    If mrngData.Row > Range("Database").Rows(2).Row Then
        'Load previous record if not on first record
        scbNavigator.Value = scbNavigator.Value - 1
        'Note: Setting scbNavigator.Value runs its Change event procedure
    End If

End Sub
```

The check ensures that you don't try to move to row numbers lower than the second row in `Database`. This, and the `cmdNext` check, could have also been carried out using the `Value`, `Max`, and `Min` properties of `scbNavigator`, but the method used in `cmdNext_Click` shows you how to determine the row number of the last row in a named range, which is a technique that it is very useful to know. It is important to carry out these checks, because trying to set the `scbNavigator.Value` property outside the `Min` to `Max` range causes a run-time error.

The code for `cmdDelete` is as follows:

```
Private Sub cmdDelete_Click()
    'Deletes current record in frmPersonal

    If Range("Database").Rows.Count = 2 Then
        'Don't delete if only one record left
        MsgBox "You cannot delete every record", vbCritical
        Exit Sub
    ElseIf mrngData.Row = Range("Database").Rows(2).Row Then
        'If on 1st record, move down one record and delete 1st record
        Set mrngData = mrngData.Offset(1)
        mrngData.Offset(-1).Delete shift:=xlUp
    End If
End Sub
```



```

        Call LoadRecord
    Else
        'If on other than 1st record, move to previous record before delete
        scbNavigator.Value = scbNavigator.Value - 1
        'Note: Setting scbNavigator.Value runs its Change event procedure
        mrngData.Offset(1).Delete shift:=xlUp
    End If

    scbNavigator.Max = scbNavigator.Max - 1

End Sub

```

This procedure carries out the following actions:

1. It aborts if you try to delete the last remaining record in Database.
2. If you delete the first record, `mrngData` is assigned a reference to the second record. `ScbNavigator.Value` is not reset, because row 2 becomes row 1, once the original row 1 is deleted. `LoadRecord` is called to load the data in `mrngData` into the UserForm.
3. If you delete a record that is not the first one, `scbNavigator.Value` is reduced by 1. This causes the previous record to be loaded into the UserForm.
4. At the end, the count of the number of rows in Database, held in `scbNavigator.Max`, is decreased by 1.

The code for `cmdNew` is as follows:

```

Private Sub cmdNew_Click()
    'Add new record at bottom of database
    Dim iRowCount As Integer

    With Range("Database")
        'Add extra row to name Database
        iRowCount = .Rows.Count + 1
        .Resize(iRowCount).Name = "Database"
        scbNavigator.Max = iRowCount
        scbNavigator.Value = iRowCount
        'Note: Setting scbNavigator.Value runs its Change event procedure
    End With

    'Set default values
    optMale.Value = True
    chkMarried = False
    lstDepartment.Value = "NA"

End Sub

```

This event procedure defines `iRowCount` to be one higher than the current number of rows in Database. It then generates a reference to a range with one more row than Database and redefines the name Database to refer to the larger range. It then assigns `iRowCount` to both the `Max` property of `scbNavigator` and the `Value` property of `scbNavigator`. Setting the `Value` property fires the `Change` event procedure for `scbNavigator`, which makes the new empty row the current row and loads the empty values into `frmPersonal`. Default values are then applied to some of the `frmPersonal` controls.

Chapter 13: UserForms

The only remaining code in `frmPersonal` is for the `cmdOK_Click` and `cmdCancel_Click` events, as follows:

```
Private Sub cmdOK_Click()  
    'Save Current Record and unload frmPersonal  
    Call SaveRecord  
    Unload Me  
End Sub  
  
Private Sub cmdCancel_Click()  
    'Unload frmPersonal without saving current record  
    Unload Me  
  
End Sub
```

Both buttons unload `frmPersonal`. Only the OK button saves any changes to the current record in the UserForm.

Modeless UserForms

The modal UserForms you have dealt with so far do not allow the user to change the focus away from the UserForm while it is being displayed. You cannot activate a worksheet, menu, or toolbar, for example, until the UserForm has been hidden or unloaded from memory. If you have a procedure that uses the `Show` method to display a modal UserForm, that procedure cannot execute the code that follows the `Show` method until the UserForm is hidden or unloaded.

A modeless UserForm does allow the user to activate worksheets, menus, and toolbars. It floats in the foreground until it is hidden or unloaded. The procedure that uses the `Show` method to display a modeless UserForm will immediately continue to execute the code that follows the `Show` method. `frmPersonal`, from the previous example that maintains a data list, can easily be displayed modeless. All you need to do is change the code that displays it, as follows:

```
Private Sub CommandButton1_Click()  
  
    frmPersonal.Show vbModeless  
  
End Sub
```

When the UserForm is modeless, you can carry on with other work while it is visible. You can even copy and paste data from TextBoxes on the UserForm to worksheet cells.

Progress Indicator

One feature that has been lacking in Excel is a good progress indicator that lets you show how much work has been done, and remains to be done, while a lengthy task is carried out in the background. You can display a message on the status bar using `Application.StatusBar`, as discussed in Chapter 2, but this message is not very obvious.

You can set up a good progress indicator very easily using a modeless UserForm. Figure 13-5 shows a simple progress bar indicator that moves from left to right to give a graphic indication of progress.

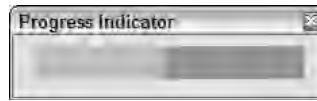


Figure 13-5

The progress indicator is a normal UserForm with two `Label` controls, one on top of the other, which have been given contrasting background colors. The `Caption` properties of both labels are blank.

This UserForm has been given the name `frmProgress`. The longer label is named `lblFixed`, because it extends over almost all the width of the UserForm and never changes. The shorter label, which is on top of the fixed label, is named `lblIndicate`. Initially, it is given a width of 0, and its width is gradually increased until it equals the width of the fixed label. The UserForm module contains the following procedure:

```
Public Sub Progress(dPerCent As Double)

    'Change width of indicator label
    lblIndicate.Width = dPerCent * lblFixed.Width

    'Allow Operating system to update screen
    DoEvents

End Sub
```

When you execute `Progress`, you pass a number between 0 and 1 as the input argument `dPerCent`. `Progress` sets the width of `lblIndicate` to `dPerCent` times the width of `lblFixed`. The `DoEvents` statement instructs the operating system to update the UserForm.

The operating system gives priority to the running macro and holds back on updating the modeless UserForm. `DoEvents` tells the operating system to stop the macro and complete any pending events. This technique often corrects problems with screen updating, or where background tasks need to be completed before a macro can continue processing. In this case you can alternatively use `Refresh`.

The progress indicator can be used with a procedure like the following, which counts how many cells contain errors within a range:

```
Sub TakesAWhile()
    Dim rng As Range
    Dim lErrorCount As Long

    For Each rng In Range(Cells(1, 1), Cells(4000, 250))
        If IsError(rng.Value) Then lErrorCount = lErrorCount + 1
    Next rng

    MsgBox "Error count = " & lErrorCount

End Sub
```

Chapter 13: UserForms

To incorporate the progress indicator, you can add the following code to the procedure:

```
Sub TakesAWhile()  
    'Routine to demonstrate progress indicator  
  
    Dim rng As Range  
    Dim lErrorCount As Long  
    Dim lCount As Long  
    Dim lRows As Long  
    Dim lColumns As Long  
    Dim lTotalIterations As Long  
    Dim lInterval As Long  
  
    'Adjust these numbers to adjust duration of demonstration  
    lRows = 4000  
    lColumns = 250  
  
    'Show the progress indicator UserForm  
    frmProgress.Show vbModeless  
  
    'Calculate interval needed for 100 progress updates  
    lTotalIterations = lRows * lColumns  
    lInterval = lTotalIterations / 100  
  
    For Each rng In Range(Cells(1, 1), Cells(lRows, lColumns))  
        If IsError(rng.Value) Then lErrorCount = lErrorCount + 1  
  
        'Each lInterval, update the indicator  
        If lCount Mod lInterval = 0 Then  
  
            frmProgress.Progress lCount / lTotalIterations  
  
        End If  
  
        lCount = lCount + 1  
  
    Next rng  
  
    Unload frmProgress  
    MsgBox "Error count = " & lErrorCount  
  
End Sub
```

The changes include showing `frmProgress` as a modeless UserForm at the start. `lTotalIterations` is the number of times the loop will be repeated. `lInterval` is the number of iterations between each update of the indicator. Here it is calculated so as to give 100 updates. Within the `For Each...Next` loop, the variable `lCount` is used to count the loops. `lCount Mod lInterval` has a value of 0 when `lCount` is 0 and every multiple of `lInterval` loops, so `frmProgress` is updated 100 times, with the `Progress` input parameter varying from 0 to .99 in steps of .01.

When a class module (such as the module behind a UserForm) contains a public procedure, you can execute the procedure as a method of the object represented by the class module.

The time taken by the macro will vary according to your processor. For demonstration purposes, you can alter the time taken by the procedure by changing the values of `lRows` and `lColumns`.

Variable UserForm Name

All the examples of UserForms have referred to the UserForm by its programmatic name (such as `frmProgress`). There can be situations where you need to run a number of different forms with the same code, or you don't know the programmatic name of the UserForm before the code is executed. In these cases, you need to be able to assign the UserForm name to a variable and use the variable as an argument. The following code allows you to do this:

```
FormName = "frmPersonal"  
VBA.UserForms.Add(FormName).Show
```

Summary

This chapter introduced the topic of UserForms. You have seen how to:

- ❑ Directly link controls on a form to a worksheet
- ❑ Use VBA code to access UserForm controls and copy data between the form and a worksheet
- ❑ Prevent closure of a UserForm by modifying the code executed when the x button is clicked
- ❑ Set up a form to maintain a data list and the difference between modal and modeless UserForms
- ❑ Construct a progress indicator using a modeless UserForm

14

RibbonX

One of the biggest changes in Office 2007 is, of course, the Ribbon. Early in the design of the Ribbon, Microsoft realized that there had to be a way for it to be customized by developers and (to a certain extent) end users. That realization led to *RibbonX*, the Ribbon's programmability mechanism. This chapter provides an introduction to RibbonX and explains how you can customize the Ribbon, both for yourself and within your applications.

Overview

In previous versions of Office, you created menus and toolbars by using VBA to manipulate the objects that make up the `CommandBars` object model (see Chapter 15). The code to do that for a non-trivial application often extended to hundreds and sometimes thousands of lines of VBA that proved hard to maintain when menus were added, removed, or rearranged. For some time, best practice has been to use a table-driven approach to building menus, in which the menus and toolbars were defined by filling in a table on a worksheet and a dedicated (and reusable) VBA procedure interpreted the table to create the menus and toolbars. Even when using a table-driven approach, you still needed quite a bit of custom VBA to ensure that specific menus were visible only when their workbook was active, and had to be extremely careful about removing customizations when the workbook was closed.

When designing the programmability model for the Ribbon, Microsoft started with the current best practices, identified the remaining pain points, and removed them. It took the resultant architecture on a world tour of key clients and other interested parties, listened to all their issues, and modified the RibbonX design to resolve most of the issues that were encountered. The result is an entirely new paradigm for the Excel developer, in which:

- ❑ The customizations are defined at design-time, rather than coded individually. Instead of using a table in a worksheet (which would only be available in Excel), they're defined using XML and stored as a custom part in the XML file formats (for all the Office applications that have the Ribbon).

- ❑ When the workbook is opened, Excel automatically reads the XML part and applies the customizations to the Ribbon.
- ❑ If a standard workbook is used, its Ribbon customizations are only applied and visible when that workbook is active.
- ❑ If an add-in workbook is used, its Ribbon customizations are always applied and available.
- ❑ Whenever a workbook is closed, its Ribbon customizations are automatically removed.
- ❑ Even though the customizations are defined at design-time, most of the controls' attributes can be modified at run time, using VBA (such as enabled, visible, label, and so on).
- ❑ A few of the controls can be totally dynamic—so their structure as well as their attributes can be defined at run time, using VBA.
- ❑ All the built-in controls are available for our use and can be overridden, executed, and queried for their images, caption, and so forth.

Prerequisites

If you intend to spend more than ten minutes investigating RibbonX, there are a few key downloads you'll need and web sites you'll need to know:

- ❑ The official RibbonX site is at <http://msdn.microsoft.com/office/tool/ribbon>.
- ❑ The Office 2007 Custom UI Editor is available from <http://openxmldeveloper.org/articles/CustomUIeditor.aspx>.
- ❑ There are two invaluable files available on the MSDN web site. The first contains a list of the names of all Excel's built-in tabs, groups, and controls; the second is an Excel add-in that adds a gallery of all the available built-in images that can be used for your custom controls.
- ❑ If you want to get into the guts of RibbonX, the `customui.xsd` schema is also available on MSDN. This is the official schema used to validate your customizations, and it details exactly which controls have which attributes and contents.
- ❑ To be informed of any errors in your RibbonX XML, check the box at Office Menu ⇨ Excel Options ⇨ Advanced ⇨ Show add-in user interface errors.

Adding the Customizations

Adding RibbonX customizations to a workbook requires just two steps:

1. Create the XML to define the required customization.
2. Insert the XML into the workbook's file (which must be using one of the XML file formats).

The first step is the subject of the remainder of this chapter. You can add the XML part to the workbook by hand or programmatically, using the techniques shown in Chapter 12. The following changes need to be made:

- ❑ Add the XML file to the workbook's zipped structure. By convention, it has the name `/customUI/customUI.xml`, though any other name can be used. It's a good idea to put the XML part into its own folder, because you may need to store button images in there as well.
- ❑ Edit the root `rels` file to include a reference to the new XML part, such as:
 - ❑ `<Relationship Type="http://schemas.microsoft.com/office/2006/relationships/ui/extensibility" Target="/customUI/customUI.xml" Id="rID5" />`
 - ❑ The important thing to get right is the `relationship Type` attribute, because that is what Excel looks for to see if the relationship is for a RibbonX customization. Note that because this is XML, it is case-sensitive, so it's critical to get the correct capitalization.

These changes are easily made by hand using the Office 2007 Custom UI Editor utility, available to download from <http://openxmldeveloper.org/articles/CustomUIEditor.aspx>.

XML Structure

One of the main criticisms of the Ribbon is that all the controls are grouped according to related functionality, so the text formatting controls are in the Font group on the Home tab, and the formula auditing tools are all in the Formula Auditing group on the Formulas tab. Yes, it's logical when viewed from a functional perspective, but totally illogical if you view the Ribbon from a process perspective. For example, it's quite common for an Excel user to be asked to look at a workbook created by someone else. When faced with such a challenge, most people work their way through the file, tracing formula precedents and dependents, checking defined names, applying different formatting to the cells they identify, adding comments, and regularly switching between the original and a working copy of the file. Unfortunately, every one of those actions is found on a different tab of the Ribbon.

In previous versions of Excel, you could bring all these actions together by creating a custom toolbar and adding the required buttons to it; when you finished auditing the workbook, you'd close the toolbar and forget about it until you were next asked to look at someone else's file.

In Office 2007, the concept of custom toolbars has been dropped. Instead, you can define some RibbonX XML to create a custom tab containing all the built-in groups and/or controls you need for your auditing, then add that XML to an otherwise empty add-in workbook. Now when you're asked to audit a workbook, you can load the Auditing add-in and have all the controls you need conveniently located on one tab; when you're finished, you can unload the add-in to remove the custom tab. No VBA required.

Start by creating a new workbook. Click Office Menu ⇨ Prepare ⇨ Properties to give it a Title and Comment (shown in the Add-ins dialog) and save it as `Auditing.xlam` (using the Excel Add-in .xlam file type in the Save As dialog).

Chapter 14: RibbonX

Now start the Office 2007 Custom UI Editor, use it to open the `Auditing.xml` file, enter the following XML, and click Save to add the XML to the file. Note that everything in the XML file is case-sensitive, so be careful to get the capitalization correct:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
  <ribbon>
    <tabs>
      <tab id="rxAuditing" label="Auditing" >
        <group id="rxAuditMisc" label="Miscellaneous" >
          <control idMso="Copy" />
          <control idMso="PasteMenu" />
          <separator id="rxAuditMiscSeparator1"/>
          <control idMso="NameManager" />
          <control idMso="ViewFreezePanelsGallery" />
          <control idMso="WindowSwitchWindowsMenuExcel" />
        </group>
        <group idMso="GroupFormulaAuditing" />
        <group idMso="GroupFont" />
        <group idMso="GroupNumber" />
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

Back in Excel, navigate to the Add-ins dialog (Office Menu ⇨ Excel Options ⇨ Add-Ins ⇨ Manage: Excel Add-Ins ⇨ Go) and load the `Auditing.xml` add-in. When you OK out of the dialog, you should see an extra Auditing tab on the Ribbon containing the groups and controls defined in the XML, as depicted in Figure 14-1.

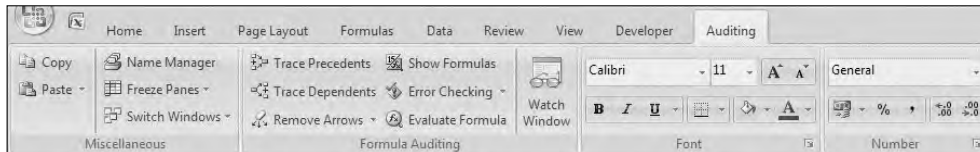


Figure 14-1

The following paragraphs work through each line of the XML definition and relate it to the resultant changes to the Ribbon.

The `<customUI . . . >` element is the root container for the XML, and the namespace identifies it as a RibbonX document.

The `<ribbon>` element is a container for all changes related to the visible Ribbon. The `<customUI>` element could also contain a `<commands>` element that you could use to repurpose built-in controls (see later in this chapter).

The `<tabs>` element is a container for all changes related to existing or new tabs on the Ribbon. The `<ribbon>` element could also contain `<officeMenu>`, `<qat>`, and/or `<contextualTabs>` elements to control the corresponding parts of the Ribbon. Note that you don't have `<miniToolBar>` or `<statusBar>` elements, and indeed, those are off limits to RibbonX.

The `<tab id="rxAuditing" label="Auditing">` element is where customizations really begin, by creating the custom tab. Every item you include in your customizations has to have at least an ID. There are three types of ID attributes: `id`, `idMso`, and `idQ`, specifying a custom item, a built-in item, or an item shared across multiple files, respectively. In this case, you're creating a custom tab, so you use the `id` attribute and give it a unique name. It's a good idea to use a standard prefix for all your custom items, to easily distinguish between them and the built-in names. I tend to use `rx` to indicate it's a RibbonX item, which also helps to further distinguish it from other types of controls when you refer to it in VBA.

The `<group id="rxAuditMisc" label="Miscellaneous">` element creates the first of the groups and opens the definition of its contents. Groups are displayed on the tab in the same order they're defined in the RibbonX file, and they display controls below each other for three rows, then across — again, in the order they're defined.

The `<control idMso="Copy"/>` element adds the built-in Copy button to the custom group. The generic `control` element type can be used for all built-in controls regardless of their actual type, and the `idMso` ID type provides the actual control name. As mentioned earlier, the names of all Excel's controls are listed in the `ExcelRibbonControls.xls` file available from MSDN.

The `<control idMso="PasteMenu"/>` element adds the standard Paste split button/drop-down.

The `<separator id="rxAuditMiscSeparator1"/>` element adds a vertical separator to the group, and starts a second column of controls. Note that even though it's a "do nothing" visual element, it still has to have a unique custom ID.

The next three elements add the built-in Name Manager, Freeze Panes, and Switch Windows controls to the custom group.

The `</group>` line completes the definition of the first group.

The `<group idMso="GroupFormulaAuditing"/>` element adds the entire built-in Formula Auditing group to the custom tab, again using the `idMso` and the correct name to identify it as built-in.

The `<group idMso="GroupFont"/>` and `<group idMso="GroupNumber"/>` elements add the built-in Font and Number Format groups.

The `</tab>` line completes the definition of your custom tab, and the remaining lines close out the containers for the `tabs`, `ribbon`, and `customUI` elements.

Using this technique, you can create multiple add-ins that only contain RibbonX definitions, each one creating custom tabs that bring together different sets of built-in controls, appropriate for different high-level tasks. If you do so, you'll probably want to add the Add-ins dialog to your QAT — it's listed as *Add-Ins* in the *All commands* section of the QAT Customization dialog.

Chapter 14: RibbonX

Although it's easy to add entire built-in groups to your tabs, you'll usually get better results by creating custom groups and adding specific controls to them. There are a number of different container controls you can include, to provide layout control or to create custom drop-down menus of built-in controls, and a number of display attributes you can set to control their appearance. For example, you could include a custom `<box>` element and use the `showLabel` attribute to show the standard Sort buttons as a horizontal set of icons above the Copy menu you have already:

```
<group id="rxAuditMisc" label="Miscellaneous" >
  <box id="rxSortBox">
    <control idMso="SortAscendingExcel" showLabel="false"/>
    <control idMso="SortDescendingExcel" showLabel="false"/>
    <control idMso="SortDialog" showLabel="false"/>
  </box>
  <control idMso="Copy" />
</group>
```

A full list of the available control types and their attributes is included later in the chapter.

Including the `<box>` element results in the group shown in Figure 14-2.

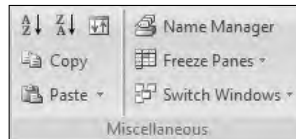


Figure 14-2

RibbonX and VBA

As well as having the ability to create custom tabs and groups containing built-in controls, Microsoft has provided the ability to add many types of custom controls to the Ribbon, and to hook their actions and most of their attributes to VBA procedures and functions. This is done using a mechanism known as a *callback*. A callback means simply that, as part of a RibbonX definition, you provide the name of a procedure to run when the control is clicked, changed, and so on. It's exactly the same as you've been doing for years with `Application.OnKey`, `CommandBarButton.OnAction`, and so forth. The main difference in RibbonX is that it passes a number of parameters to the function being called, so the function signature must be declared correctly for the call to work. Again, that is no different than how you've been coding UserForm control event procedures (other than the function signature usually being written for you).

Callbacks are also used when you need to be able to change a control's attributes at run time. Rather than defining a specific value for the attribute in the XML, provide the name of a procedure that Excel should call whenever it needs to know the attribute's value; it's up to that procedure to work out (and remember) what the value should be and return it to Excel. For example, rather than including the `label="Miscellaneous"` attribute in your earlier custom group, you could have specified a `getLabel` callback:

```
<group id="rxAuditMisc" getLabel="rxAuditMisc_getLabel" >
```

When the tab is first displayed, Excel will call the `rxAuditMisc_getLabel` VBA procedure (in a standard module), which should provide the text for Excel to use. If you want to change that text sometime later, you can't just update an object's property; RibbonX does not have an object model. Instead, there's an interface to tell Excel that the information provided in an earlier callback is no longer valid; when Excel next needs to display the group or control, it will call the procedure again to get the new value. Most attributes of most controls can be set dynamically in this way. All the available callbacks and their function signatures are detailed later in the chapter, but first, have a look at all the available types of custom controls you can add to the Ribbon and all the attributes you can set to modify their appearance and behavior.

Control Types

Previous versions of Office had a relatively limited set of control types that could be added to the menus or toolbars. Office 2007 allows you to add much more than a simple button or popup menu, and includes control types for almost every display mechanism that can be found in Excel's built-in Ribbons. The table on the following pages lists all the available control types, with a brief description and example picture of a built-in control of each type.

Basic Controls

The following table lists all the basic controls that you can add to custom groups or that can be contained in other control types.

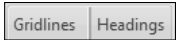
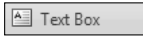


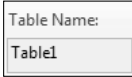

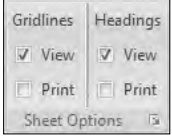
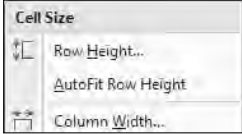

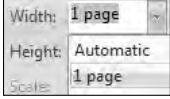

Control Type	Description	Example from Excel's Ribbon
<code><control .../></code>	The generic control type is used whenever you want to add a built-in control to a custom group.	
<code><labelControl .../></code>	A <code>labelControl</code> is a textual element and has no actions. It's typically used to provide headers to columns of related buttons.	
<code><button .../></code>	The most common control, a button is a single clickable item, having an image and/or caption.	
<code><toggleButton .../></code>	A toggle-button is a clickable item that toggles between pressed and not pressed with each click. It is most often used in groups to switch an attribute between multiple possible states, where only one of the group can be "down" at any time.	
<code><checkbox .../></code>	A clickable control that toggles between on and off, often used to control whether or not a UI element is visible.	

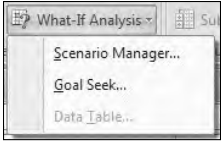
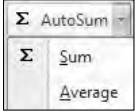
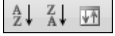
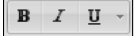
Table continued on following page

Control Type	Description	Example from Excel's Ribbon
<code><editBox .../></code>	A control that can be typed into.	
<code><gallery .../></code>	A drop-down control that drops to show a grid of other controls. The gallery can contain many different types of controls within the grid, and is one of the most flexible RibbonX controls.	
<code><dynamicMenu .../></code>	A popup menu, whose content is provided at run time, using a callback.	
<code><separator .../></code>	A vertical bar used to provide visual separation of controls in a group	
<code><menuSeparator .../></code>	A horizontal bar within a popup menu, providing a title for a group of related menu items. If no title is provided, displays as a thin horizontal line.	
<code><item .../></code>	An item in the drop-down list of a <code>comboBox</code> or <code>dropDown</code> .	
<code><dialogBoxLauncher .../></code>	Adds a Launcher button to the bottom-right corner of a group.	

Container Controls

The following table lists all the container controls you can add to custom groups. By nesting container controls within other containers, you can create hierarchical structures.

<code><dropDown ...></code> contents <code></dropDown></code>	Can contain <code><item></code> or <code><button></code> controls.	A control that provides a drop-down list of items to select between, such as the Width <code>dropDown</code> .	
<code><comboBox ...></code> contents <code></comboBox></code>	Can contain only <code><item></code> controls.	A control that can be typed into, but also provides a drop-down list to pick from. The contents of the drop-down are provided as a set of item elements.	

<pre><menu ...> contents </menu></pre>	<p>Can contain <code><control></code>, <code><button></code>, <code><toggleButton></code>, <code><checkbox></code>, <code><gallery></code>, <code><dynamicMenu></code>, <code><menuSeparator></code>, <code><menu></code> and/or <code><splitButton></code> controls</p>	<p>A popup menu, whose constituent items are defined in the RibbonX file. Menus may contain buttons or other menus, allowing you to create hierarchical menu structures.</p>	
<pre><splitButton ...> <button .../> <menu ...> menu contents </menu> </splitButton></pre>	<p>Must have the structure show here, though could contain either a <code><button></code> or a <code><toggleButton></code> control.</p>	<p>A combined button or toggle-button and menu. Clicking the button part usually performs a default action, and clicking the drop-down arrow shows a list of related alternatives.</p>	
<pre><box ...> contents </box></pre>	<p>Can contain any other control type.</p>	<p>The box control has no visual display, but is used to control the layout of other buttons, such as in your Auditing group.</p>	
<pre><buttonGroup ...> contents </buttonGroup></pre>	<p>Can contain <code><control></code>, <code><button></code>, <code><toggleButton></code>, <code><gallery></code>, <code><menu></code>, <code><dynamicMenu></code> and/or <code><splitButton></code> controls.</p>	<p>A container control that displays its content controls as a related group, with a border and breaks between the controls.</p>	

Control Attributes

All of the control types have numerous attributes you can use to modify their appearance. All the available attributes are listed in the following table in alphabetical order, with their allowed values and the controls that they can be used with.

Attribute	Description	Allowed Values	Applies To
boxStyle	Whether a box control arranges icons horizontally (the default) or vertically	horizontal, vertical	box
columns	The number of columns in a gallery	1 to 1024 columns	gallery

Table continued on following page

Attribute	Description	Allowed Values	Applies To
description	A long description of a control, shown in menus when the menu's <code>itemSize</code> is set to large	1 to 4096 characters	button, toggleButton, splitButton, checkBox, menu, dynamicMenu, gallery
Enabled	Whether a control is enabled	true, false	All controls
Id	A custom control ID	1 to 1024 characters	All controls
idMso	A built-in control ID	1 to 1024 characters	All controls
idQ	A qualified control (see later)	1 to 1024 characters	tab, group or menu
Image	The name of a custom image within the workbook file	1 to 1024 characters	All controls that have an image
imageMso	The name of a built-in control, whose image should be used	1 to 1024 characters	All controls that have an image
invalidateContentOnDrop	Whether to fire the content-related callbacks whenever a control is dropped	true, false	comboBox, gallery, dynamicMenu
itemHeight	The height of a gallery item, in pixels	1 to 4096	gallery
itemSize	The size of items in a menu; large items show their descriptions as well as their labels	normal, large	menu
itemWidth	The width of a gallery item, in pixels	1 to 4096	gallery
keytip	The shortcut key combination used to access the control	1 to 3 characters	All controls, tab and group
label group	The control's caption	1 to 1024 characters	All controls, tab and
maxLength	The maximum length of textual input	1 to 1024	editBox, comboBox
rows	The number of rows in a gallery	1 to 1024 rows	gallery
screentip	The small tip that shows when the mouse hovers over a control	1 to 1024 characters	All controls
showImage	Whether a control's image is displayed	true, false	All controls that have an image
showItemImage	Whether images are displayed for drop-down items	true, false	comboBox, dropDown, gallery
showItemLabel	Whether labels are displayed for drop-down items	true, false	comboBox, dropDown, gallery

Attribute	Description	Allowed Values	Applies To
showLabel	Whether a control's label is displayed	true, false	All controls
size	The size of a control: normal-size takes up one row; large-size takes up three rows	normal, large	All controls
sizeString	A representative string used to set the width of a control	1 to 1024 characters	editBox, comboBox, dropDown
supertip	The large tip that shows when the mouse hovers over a control	1 to 1024 characters	All controls
tag	Arbitrary text	1 to 1024 characters	All controls
title	The text for a menu's title	1 to 1024 characters	menu, menuSeparator
visible	Whether a control is visible	true, false	All controls, tab and group

Control Callbacks

Most RibbonX customization can be defined at design-time and can thereby be included directly within the XML file. If, however, there are some attributes that need to be set at startup or can change at run time, you can use the equivalent `get` attribute to provide the name of a callback function. When Excel starts, it calls the function and the function provides the value of the attribute. Unlike the `OnAction` calls you're used to, Excel does *not* automatically scope the callback to the workbook containing the RibbonX definition; if multiple workbooks contain a procedure with the same callback name, there is no guarantee which one will be called!

For example, the following line was originally included in the XML to hard code the group label:

```
<group id="rxAuditMisc" label="Miscellaneous">
```

To use a callback instead, add the following VBA procedure to the `Auditing.xlam` workbook to provide the label at run time. The VBA procedure's parameters must match those that RibbonX expects to provide. Most of them pass in a reference to the RibbonX control and a `ByRef` parameter for the return value:

```
Sub rxAuditMisc_getLabel(ByRef Control As IRibbonControl, _
    ByRef ReturnValue As Variant)

    ReturnValue = "Miscellaneous - " & Format(Date, "dddd")
End Sub
```

Save the add-in, unload it, and use the Custom UI Editor to change the RibbonX XML to use the `getLabel` attribute, calling the procedure you just added:

```
<group id="rxAuditMisc" getLabel="rxAuditMisc_getLabel">
```

Chapter 14: RibbonX

If you reload the add-in and click the Auditing tab, the first group should now include the day of the week in its name.

The Control reference passed into the callback is an extremely simple object, having only three read-only properties and no methods:

- ❑ `id`—The control's `id` attribute
- ❑ `tag`—The control's `tag` attribute, if defined in the XML
- ❑ `context`—Not used in Excel

The `id` property can be used to distinguish between controls if you specify a common callback name for multiple controls. For example, in a multilingual application, you could include `getLabel="rxGetLabel"` in the definition for all controls and read the appropriate text from a language lookup table, matching on the control ID:

```
Sub rxGetLabel(ByRef Control As IRibbonControl, _
              ByRef ReturnValue As Variant)

    ReturnValue = Application.WorksheetFunction.VLookup(Control.ID, _
              shtLanguages.Range("rngLabels"), glLanguageID, False)

End Sub
```

In addition to the `get` equivalents of all the design-time attributes, the following control callbacks are available only at run time.

Callback	Used By	Description
<code>getContent</code>	dynamicMenu	Provides the XML for the menu's content.
<code>getPressed</code>	toggleButton, checkBox	Specifies whether or not the control is pressed/ticked.
<code>getItemCount</code>	comboBox, dropdown, gallery	Specifies how many items there are in a list populated at run time.
<code>getItemID</code> , <code>getItemLabel</code> , <code>getItemImage</code> , <code>getItemScreentip</code> , <code>getItemSupertip</code>	comboBox, dropdown, gallery	Called once for each item, to provide the attributes for that item.
<code>getSelectedItemID</code>	dropdown, gallery	Specifies which is the selected item, by providing the item's ID.
<code>getSelectedItemIndex</code>	dropdown, gallery	Specifies which is the selected item, by providing the item's index in the list.
<code>getText</code>	comboBox, editBox	Provides the text shown in the control
<code>onAction</code>	button, toggleButton, checkBox, dropdown, gallery	Called when the control is clicked. Note that the signatures are different depending on the control (see the following table).
<code>onChange</code>	editBox, comboBox	Called when the text of the control has changed.

The following tables show the function signatures for all the available control callbacks. If you use the Office 2007 Custom UI Editor, it can generate the correct callback signatures for any callbacks included in the XML, ready for you to paste into your VBA project.

Callback	getContent, getDescription, getEnabled, getImage, getItemCount, getItemHeight, getItemWidth, getKeytip, getLabel, getPressed, getSize, getScreentip, getSelectedItemID, getSelectedItemIndex, getShowImage, getShowLabel, getSupertip, getText, getTitle, getVisible
Signature	Sub ProcName (ByRef Control As IRibbonControl, _ ByRef ReturnValue As Variant)
Callback	getItemID, getItemImage, getItemLabel, getItemScreentip, getItemSupertip
Signature	Sub ProcName (ByRef Control As IRibbonControl, ByRef Index As Integer, __ ByRef ReturnValue As Variant)
Callback	onAction for a button control
Signature	Sub ProcName (ByRef Control As IRibbonControl)
Callback	onAction for a checkBox and toggleButton control
Signature	Sub ProcName (ByRef Control As IRibbonControl, ByRef Pressed As Boolean)
Callback	onAction for a dropDown and gallery control
Signature	Sub ProcName (ByRef Control As IRibbonControl, ByRef SelectedID As String, _ ByRef SelectedIndex As Integer)
Callback	onChange for an editBox or comboBox
Signature	Sub ProcName (ByRef Control As IRibbonControl, _ ByRef Text As String)

Managing Control Images

Most of the control types can have an associated image. The choice of image and display style is controlled by the `imageMso`, `image`, `getImage`, `showImage`, `getShowImage`, `showItemImage`, `getShowItemImage`, `size`, and `getSize` attributes.

The `imageMso` attribute is used when you want to use one of the built-in icons for a custom control. The value of the attribute must be the name of the built-in control, which can be found by downloading the `Office2007IconsGallery.xlsm` file from the MSDN web site. That file adds a set of galleries to Excel's Developer tab that together show all of the 2,586 available images. For example, you can show a button with a smiley face using the XML:

```
<button id="rxButtonSmiley" imageMso="HappyFace" />
```

Chapter 14: RibbonX

The `image` attribute is used when you want to provide custom images for your controls. Those images are usually contained inside the workbook file. The Ribbon drawing engine is designed to work best with full-color (24-bit) images that also have an alpha channel to control each pixel's transparency. The best graphics format to use for custom images is therefore the Portable Network Graphics (.png) format, as that supports an alpha channel and results in relatively small file sizes.

The Office 2007 Custom UI Editor can be used to add custom images to workbook files—just click the Insert Icons button and select the file to add. The Editor will show the icon in a pane on the right-hand side and give it a default ID. The ID can be changed by right-clicking the icon and is then used in the `image` attribute for the control:

```
<button id="rxButtonCustom" image="MyPNG" />
```

Within the file, the editor stores all the images in a separate `customUI\images` folder and creates a `customUI.xml.rels` file in a `customUI_rels` folder to relate the IDs used in the XML to the image files:

```
<Relationships
xmlns="http://schemas.openxmlformats.org/package/2006/relationships">
  <Relationship
Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/image"
Target="images/myCustomImage.png" Id="MyPNG" />
</Relationships>
```

If there are likely to be a large number of workbooks all using the same custom images, it may be more efficient to store them all on a network drive, rather than copy them within each workbook file, and load them at run time. You can do this with the `loadImage` callback of the `customUI` element:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
loadImage="rxcustomUI_loadImage">
...
  <button id="rxButtonCustom" image="Custom1.bmp">
...
</customUI>
```

And some VBA code to load the images and provide them as an `IPicture` object:

```
Sub rxcustomUI_loadImage(imageID as String, ByRef returnedVal)
  Set returnedVal = LoadPicture("X:\Images\" & imageID)
End Sub
```

When Excel loads the workbook, it calls the `loadImage` callback for every `image` attribute it finds that doesn't point to an image contained within the file, passing the value of the `image` attribute as the `imageID`. The `returnedVal` parameter must be set to either a standard `IPicture` image object or the name of a built-in control. The code uses the standard `LoadPicture` function to load the image from the network share as the correct object type. Unfortunately, `LoadPicture` does not handle the .png file format, so if you want to load .png files using this technique, you have to use Windows API calls into the GDI+ libraries. That is beyond the scope of this chapter, but a drop-in module is available for download from www.wrox.com, providing a `LoadPictureGDI` function that handles .png files.

The `getImage` callback is used if you want to change a custom control's image while your application is running. An example is the `splitButton` control used to apply cell borders; when an item is selected from the menu part of the button (for example, double-bottom border), the button image (and behavior) is updated to match the selected item. This is achieved by assigning an `onAction` callback to all the menu items, which records the appropriate image in a module-level variable and marks the button as needing to be refreshed (see later for details). When Excel next needs to display the button, it calls the `getImage` callback again, which returns the new image from the module-level variable.

The `showImage` attribute and `getShowImage` callback can be `true` or `false`, and they control whether or not a control's image is displayed.

The `showItemImage` attribute and `getShowItemImage` callback can be `true` or `false` and control whether or not an image is displayed for a `dropDown`, `comboBox`, or `gallery` item.

The `size` attribute and `getSize` callback can be `normal` or `large`. The normal size takes up one row of the Ribbon (with the caption beside the icon), and the large size takes up all three rows (with the caption below the icon).

Other RibbonX Elements, Attributes, and Callbacks

Though most of the elements, attributes, and callbacks in RibbonX are used to set properties of controls, there are a few more you need to understand to fully use the Ribbon.

If you omit the XML elements that define the controls, the full RibbonX structure is shown as follows, where the ellipses (...) indicate one or more optional attributes. These elements and their attributes are described in the sections that follow:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
  ... >
  <commands>
    <command ... />
  </commands>
  <ribbon ... >
    <officeMenu>
      Any control type that can be in a menu
    </officeMenu>
    <qat>
      <sharedControls>
        <control>, <button> or <separator> control types
      </sharedControls>
      <documentControls>
        <control>, <button> or <separator> control types
      </documentControls>
    </qat>
  </ribbon>
  <tabs>
```

```
<tab ... >
  <group ... >
    All control types
  </group>
</tab>
</tabs>
<contextualTabs>
  <tabSet idMso="TabSetChartTools">
    <tab ... >
      <group ... >
        All control types
      </group>
    </tab>
  </tabSet>
</contextualTabs>
</ribbon>
</customUI>
```

Sharing Controls among Multiple Workbooks

When you create custom tabs, groups, and controls using the `id` attribute, you always get a brand new tab, group, or control created for you—even if one already exists with the same ID or label. This is usually beneficial, because it prevents multiple add-ins accidentally changing each other's items. Occasionally, though, you may want to share those items among many add-ins or workbooks. For example, you might have an add-in that creates a basic tab, group, and menu structure, and many individual workbooks that should add items to that structure. This is achieved by using the `idQ` attribute to provide *qualified* IDs—IDs associated with a specific namespace. The namespace is provided within the `customUI` element and just needs to be a unique string. It is typically given an alias (Q in this case) to make the XML easier to read:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
  xmlns:Q="Excel 2007 VBA Prog Ref">
```

Any elements you want to share between workbooks are then defined using the `idQ` attribute and including the namespace alias within the ID:

```
<ribbon>
  <tabs>
    <tab idQ="Q:rxShared" label="Shared Tab">
      <group id="Group1" label="Group Not Shared">
```

Any workbooks using the same namespace string to qualify controls with the same ID will share those controls rather than getting their own copies. In this example, any workbooks that included a namespace of `xmlns:Q="Excel 2007 VBA Prog Ref"` and a qualified tab ID of `idQ="Q:rxShared"` would all use the same tab for their controls. However, because you've used an unqualified ID for the group, that would not be shared between the workbooks. The ability to use qualified IDs in this way applies to all the container-type controls, so an add-in could create a complex menu structure using qualified IDs for all the tabs, groups, and menus, and individual workbooks could add their customizations to the shared menus.

Updating Controls at Run Time

Callbacks aren't just used to get controls' attributes when the workbook is loaded; you can also use them to change the attribute values at any time. This is done by using a special interface (`IRibbonUI`) to mark a control as invalid. The next time Excel needs to display the control, all its callbacks will be called again to get the latest values.

You tell Excel to give you an `IRibbon` interface by adding the `onLoad` callback to the `customUI` element:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
  xmlns:Q="Excel 2007 VBA Prog Ref"
  onLoad="rxcustomUI_onLoad">
```

The signature for the `onLoad` callback includes an `IRibbon` parameter, which is stored in a module-level variable for later use:

```
Dim moRibbon As IRibbonUI

Sub rxcustomUI_onLoad(ribbon as IRibbonUI)
  Set moRibbon = ribbon
End Sub
```

The `IRibbonUI` interface has only two methods:

- ❑ The `InvalidateControl("ControlID")` method marks an individual control as invalid, and hence should be refreshed when it's next displayed. In general, it is best to only invalidate those controls that are necessary.
- ❑ The `Invalidate` method marks your entire Ribbon customization as invalid, so every callback for every control is called again. This might be used if you provide a choice of UI language and use `getLabel` for every control; when the user changes their language choice, every control can be invalidated, and thereby updated to show the new text.

As an example, the following XML creates a `splitButton` with `Up`, `Goto`, and `Down` menus, using built-in images, with callbacks defined such that you can change the button's image and action to repeat the last selected menu (just like the `Border splitButton`). Create a new workbook, save it as an add-in, and use the Custom UI Editor to add the XML:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
  onLoad="rxcustomUI_onLoad">
  <ribbon>
    <tabs>
      <tab id="rxExcelVBA" label="VBA Prog Ref">
        <group id="rxDemo" label="Demo">
          <splitButton id="rxSplit" size="large">
            <button id="rxButton" getImage="rxButton_getImage"
              onAction="rxButton_onAction"/>
            <menu id="rxMenu">
              <button id="rxMenuOutlineMoveUp" label="Up"
                imageMso="OutlineMoveUp"
                onAction="rxMenu_onAction"/>
              <button id="rxMenuGoTo" label="Goto"
                imageMso="GoTo"
                onAction="rxMenu_onAction"/>
            </menu>
          </splitButton>
        </group>
      </tab>
    </tabs>
  </ribbon>
</customUI>
```

```
        <button id="rxMenuOutlineMoveDown" label="Down"
              imageMso="OutlineMoveDown"
              onAction="rxMenu_onAction" />
    </menu>
  </splitButton>
</group>
</tab>
</tabs>
</ribbon>
</customUI>
```

Now open the add-in (ignoring any errors) and add the following VBA code to a standard module, to implement all the callbacks defined in the XML:

```
'Variable to invalidate controls
Dim moRibbon As IRibbonUI

'Variable to store current splitButton style,
'which is the same as the image name
Dim msSplitStyle As String

'Callback for customUI.onLoad
Sub rxcustomUI_onLoad(ribbon As IRibbonUI)
    Set moRibbon = ribbon
End Sub

'Callback for rxButton getImage
Sub rxButton_getImage(control As IRibbonControl, ByRef returnedVal)

    'Default the style to GoTo
    If msSplitStyle = "" Then msSplitStyle = "GoTo"

    'Return the built-in image name
    returnedVal = msSplitStyle
End Sub

'Callback for rxButton onAction
Sub rxButton_onAction(control As IRibbonControl)
    DoSplitAction msSplitStyle
End Sub

'Callback for all rxMenu onActions
Sub rxMenu_onAction(control As IRibbonControl)

    'Get the style from the control ID
    msSplitStyle = Mid$(control.ID, 7)

    'Tell the ribbon that the button needs to be refreshed
    moRibbon.InvalidateControl "rxButton"

    'Do the appropriate action
    DoSplitAction msSplitStyle
End Sub

'Do the action
```



```
Private Sub DoSplitAction(ByVal sStyle As String)

    Select Case sStyle
    Case "OutlineMoveUp": MsgBox "Up"
    Case "GoTo": MsgBox "Goto"
    Case "OutlineMoveDown": MsgBox "Down"
    End Select

End Sub
```

Save, close, and reopen the file. You should now have a tab called VBA Prog Ref with a single large split button, whose image and action changes to match the menu selected (see Figure 14-3).



Figure 14-3

Hooking Built-In Controls

Whereas most of RibbonX concentrates on customizing the visual appearance of the Ribbon, the `commands` and `command` elements provide a mechanism for overriding any built-in menu, to modify its enabled state or to intercept any button clicks. To achieve that, you can include the following XML to, say, override the Print button so you can set up some formatting:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
  <commands>
    <command idMso="FilePrint" onAction="rxPrint_onAction" />
  </commands>
</customUI>
```

The `command` element can only have an `enabled` attribute or `getEnabled` and/or `onAction` callbacks. You could, for example, disable a control by including `enabled="false"`, or use a `getEnabled` callback to determine whether or not to enable the control depending on the state of the application. The `onAction` callback is used to intercept the default behavior, allowing you to do some preprocessing and optionally cancel the action:

```
'Callback for Print onAction, with ability to cancel
Sub rxPrint_onAction(control as IRibbonControl, ByRef cancelDefault)
    If Cdbl(Time) > 0.5 Then
        MsgBox "Printing can only be done in the morning!"
        cancelDefault = True
    End If
End Sub
```

Chapter 14: RibbonX

Unfortunately, overriding Ribbon controls in this way only affects the user actually clicking the control or using the Alt+Key shortcuts to trigger the control; it does not intercept the many operations that can be achieved using the function keys and Ctrl+Key combinations (such as Ctrl+P in this case).

If two add-ins attempt to override the same control, the last-loaded add-in wins. For these two reasons, the ability to override a built-in control should be used with extreme caution.

RibbonX in Dictator Applications

Most dictator-style Excel applications typically start by removing all Excel's menus and as many other UI elements as possible, so as to present a locked-down interface to the user. To do this for the Ribbon, use the `startFromScratch` attribute of the `ribbon` element to give a minimal Office Menu with New, Open, Save As, Recent Files, and Excel Options:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
  <ribbon startFromScratch="true">
  </ribbon>
</customUI>
```

Obviously, you'd then include lots more XML to build custom tabs, groups, and so on!

Customizing the Office Menu

The Office Menu is treated as a special case within the RibbonX definitions, in that it's an element in its own right (rather than, say, being a special tab). You can add controls to it through the `officeMenu` element:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
  <ribbon>
    <officeMenu>
      <button id="rxOMBtn" label="Office Menu" onAction="rxOMBtn_onAction" />
    </officeMenu>
  </ribbon>
</customUI>
```

You can also add buttons to the built-in items on the Office Menus, such as the Send menu, by nesting the appropriate control IDs. Note that to do this, you have to use the correct control type, rather than the generic control element:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
  <ribbon>
    <officeMenu>
      <menu idMso="FileSendMenu">
        <button id="rxSend" label="Send Menu" onAction="rxSend_onAction" />
      </menu>
    </officeMenu>
  </ribbon>
</customUI>
```

Customizing the QAT

The Office 2007 design philosophy is that the QAT belongs to the user and applications should never add their controls directly to it. If the users consider your feature to be useful, they'll put it on the QAT and Excel will automatically handle the interaction.

In certain scenarios it would be highly beneficial to be able to add controls to the QAT — such as a user creating add-ins similar to the Auditing add-in presented at the start of this chapter, but by moving groups of controls on and off the QAT rather than adding custom tabs. Unfortunately, Microsoft chose to disable this capability, and restricted QAT customization to Dictator Applications that also set `startFromScratch="true"`.

The QAT has two sections — one for controls shared across all open documents, and one for controls that should only appear on the QAT when this document has the focus. You can add built-in controls using the generic `<control>` type, `<button>` controls, and separators to either of the two areas:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
  <ribbon startFromScratch="true">
    <qat>
      <sharedControls>
        <control>,<button> or <separator> types
      </sharedControls>
      <documentControls>
        <control>,<button> or <separator> types
      </documentControls>
    </qat>
  </ribbon>
</customUI>
```

Controlling Tabs, Tab Sets, and Groups

The `<tabs>`, `<tab>`, `<contextualTabs>`, `<tabSet>`, and `<group>` elements are all used to provide the structure to the Ribbon. You've been using tabs and groups all the way through this chapter; the `tabSet` element provides access to the built-in sets of contextual tabs. For example, when a chart is selected, Excel displays a Chart Tools set of three contextual tabs. You could add a custom group to the Chart Tools Design tab with the following XML:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui">
  <ribbon>
    <contextualTabs>
      <tabSet idMso="TabSetChartTools">
        <tab idMso="TabChartToolsDesign">
          <group id="MyChartGroup">
            <!-- All control types -->
          </group>
        </tab>
      </tabSet>
    </contextualTabs>
  </ribbon>
</customUI>
```

Chapter 14: RibbonX

Microsoft has placed a few restrictions on what you can do with tabs, contextual tabs, and groups, including:

- You can't create your own contextual `tabSet`, but can only refer to built-in ones
- You can add custom tabs to a tab set, and custom groups to a built-in tab
- You can't add controls to built-in groups
- You can add built-in groups to different tabs (either built-in or custom)
- You can make a built-in tab or group invisible
- You can move a tab or group to be before or after another one

These are all achieved by setting appropriate attributes on the `<tab>` and `<group>` elements, such as:

- `visible="false"` to hide the tab or group
- `insertAfterMso="TabData"` to move or add the tab or group after a built-in tab or group
- `insertBeforeMso="GroupZoom"` to move or add the tab or group before a built-in tab or group
- `insertAfterQ="Q:SharedTab"` to move or add the tab or group after a qualified custom tab or group
- `insertBeforeQ="Q:SharedGroup"` to move or add the tab or group in front of a qualified custom tab or group

Dynamic Controls

The Ribbon is designed to be a static structure of tabs, groups, and controls; while you're working with Excel, all the controls in the Ribbon stay in the same place and have the same structure, size, and actions. The only parts of the Ribbon that could be considered dynamic are the contextual tabs, the Window list, and the File MRU — and even then, it's only the visibility of the contextual tabs that changes; their structure remains constant.

This philosophy is reflected in the design of RibbonX, with the restriction that your XML customization has to be defined up front and hard coded into your document files; you do not have a mechanism in VBA for providing the XML at run time. Indeed, if you agree with the philosophy, you wouldn't need to provide the XML at run time, because the `getLabel`, `getImage`, and other callbacks would be sufficient for any localization requirements.

There are, however, times when the philosophy breaks down — the Window menu is a built-in example. There might also be problems when migrating `CommandBar` code to use the Ribbon, if that code changed the command bar structures as workbooks were opened or closed, or worksheets were added and removed.

Fortunately, Microsoft recognized the requirement for a certain amount of dynamism in our UIs and provided four controls whose contents can be provided at run time. They are the `comboBox`, `dropDown`, `gallery`, and `dynamicMenu` controls. The content is provided using a set of callbacks, called when the controls are first displayed and when they're explicitly marked as invalid. They also have the extremely

useful `invalidateContentOnDrop` attribute, which if true ensures the callbacks get called every time, just before the control is dropped down.

dropDown, comboBox, and gallery

These controls are essentially three styles of drop-down controls, where the `gallery` drops to show a 2D grid of images and/or labels. The `dropDown` and `comboBox` controls can be static, by including their content as `<item>` controls in the XML. You can populate all three controls at run time by using their `getItemCount`, `getItemID`, `getItemLabel`, `getItemImage`, `getItemScreenTip`, or `getItemSupertip` callbacks. At a minimum, you have to use `getItemCount` to provide the number of items in the list. The remaining `getItem` callbacks will be called once for each item, with the item index passed in. You are required to provide an ID for each item with `getItemID`, and will typically provide an image or a label with `getItemImage` or `getItemLabel`, respectively. The `dropDown` and `comboBox` controls usually look best with images and labels, while the `gallery` is designed to look best as a grid of images. In all cases, the `onAction` callback of the `dropDown` or `gallery` provides both the selected item's ID and its index as parameters (but not the `comboBox`, which only has an `onChange` callback to provide the text).

dynamicMenu

The `dynamicMenu` is a unique control in RibbonX, because it is the only one whose content's *structure* can be changed at run time. The `dropDown`, `comboBox`, and `gallery` are essentially flat lists; the `dynamicMenu` can contain a full control hierarchy of both custom and built-in controls—including other `dynamicMenus`. This control was created to satisfy the specific requirement for dynamic content that changes radically as workbooks are opened, closed, and changed.

Imagine a workbook containing multiple sheets of different types (for example, raw data, intermediate calculations, summaries, and reports), and each sheet requiring a different set of menus. You might define it statically using the following menu definition (where extra attributes and callbacks such as `Image` and `onAction` have been omitted for clarity):

```
<menu id="rxSheetOperations" label="Sheet Operations">
  <menu id="rxData1Menu" label="Data Sheet 1">
    <button id="rxData1Refresh" label="Refresh Sheet 1 data"/>
  </menu>
  <menu id="rxData2Menu" label="Data Sheet 2">
    <button id="rxData2Refresh" label="Refresh Sheet 2 data"/>
  </menu>
  <menu id="rxCalcMenu" label="Calculation Sheet">
    <button id="rxCalcRecalc" label="Recalculate"/>
  </menu>
  <menu id="rxReport1Menu" label="Report Sheet 1">
    <button id="rxReport1Show" label="Show Report Sheet"/>
    <button id="rxReport1Print" label="Print Report Sheet"/>
  </menu>
  <menu id="rxReport2Menu" label="Report Sheet 2">
    <button id="rxReport1Show" label="Show Report Sheet"/>
    <button id="rxReport1Print" label="Print Report Sheet"/>
  </menu>
</menu>
```

Chapter 14: RibbonX

The problem is that you don't know at design-time how many sheets of each type there might be in the final workbook, or in what order they'll be shown. The `dynamicMenu`'s `getContent` callback provides the mechanism through which you can use VBA to create the preceding XML at run time (by examining the workbook directly). Within the XML definition, you include a single entry for the `dynamicMenu`:

```
<dynamicMenu id="rxSheetOperations" label="Sheet Operations"
  getContent="rxSheetOperations_getContent">
```

The `rxSheetOperations_getContent` procedure builds the XML definition for the contained controls and returns it, wrapped in a containing `<menu>` element:

```
Sub rxSheetOperations_getContent(ByRef control As IRibbonControl, _
    ByRef returnedVal)

    Dim sXML As String
    Dim wks As Worksheet

    'Start with a container <menu> element
    sXML = "<menu xmlns=""http://schemas.microsoft.com/office/2006/01/customui"">"

    For Each wks In ThisWorkbook.Worksheets
        'TODO: Add the control definitions for each sheet to the sXML string
    Next

    'End by closing the container <menu> element
    sXML = sXML & "</menu>"

    'Return the full XML to the dynamicMenu
    returnedVal = sXML
End Sub
```

CommandBar Extensions for the Ribbon

With the design of RibbonX based around XML and callbacks instead of an object model, it is the responsibility of the customizing application to maintain the state information for all its controls. That raises the question of how to query that state information — without an object, you have nothing to read the properties from! Fortunately, the `CommandBars` object has been extended with `GetEnabledMso`, `GetImageMso`, `GetLabelMso`, `GetPressedMso`, `GetScreenTipMso`, `GetSuperTipMso`, and `GetVisibleMso` properties to expose all the state information for the built-in controls. In each case, you pass in the name of the built-in control. These functions are likely to be most useful to add-in writers who want to include control images on their UserForms or ensure that they're using the correct control labels when directing users to click a Ribbon control. Note that you don't have a `GetTextMso` to return the text of a `dropDown`, `comboBox`, or `editBox`, nor can you call these functions with the ID of a custom control to query other add-ins' customizations.

As well as being able to retrieve the state information, you can also click any built-in control using the new `ComandBars.ExecuteMso` method. This is extremely useful for triggering those actions that don't have an object model equivalent — such as putting the user in drawing mode to draw a text box on a sheet:

```
Application.CommandBars.ExecuteMso "TextBoxInsertExcel"
```

RibbonX Limitations

The first version of any new technology can never hope to be perfect for everyone. To their great credit, the RibbonX team at Microsoft implemented a huge number of the suggestions that were submitted during the Office 2007 Beta testing (within the limits imposed on them by the overall Ribbon design) and have done an amazing job of making RibbonX as feature-rich, well-rounded, and stable as it is. The remaining major limitations are, in our view and in no particular order, as follows:

- ❑ There is no way for VBA to provide the full XML at run time. It would have been nice to see a standard `GetCustomUI` event added to the `Workbook` object, or a `CreateCustomUI` method added to the `CommandBars` object.
- ❑ The only way to remove a workbook's customizations is to close the workbook. A better alternative might have been a `RemoveCustomUI` method added to the `CommandBars` object.
- ❑ The `getImage` callback can be fed either an `idMso` name or an `IPicture` object to provide a built-in or custom image. You should also be able to provide an internal relationship ID to use a custom image contained in the workbook file.
- ❑ There are a number of controls on Excel's Ribbon that can't be created using RibbonX—such as the Page Layout Scale spinner, the galleries that display directly in the Ribbon area, and the list boxes from the Filter popup.
- ❑ You cannot create custom contextual tab sets. It would be beneficial to be able to define your own contexts (for example, inside a P&L report) and display appropriate context-sensitive tabs.
- ❑ The only way to activate a tab is to use `SendKeys`; there should be an `ActivateMso` method to select rather than execute a control—or perhaps using `ExecuteMso` on a tab name should activate it.
- ❑ You can't modify the Mini Toolbar or the status bar.
- ❑ You cannot add custom ActiveX controls to the Ribbon (though you can now create Task Panes with them).
- ❑ You cannot modify the built-in groups—though that's as much a Ribbon design issue as a RibbonX one.
- ❑ You can't read the text of built-in `dropDown`, `comboBox`, or `editBox` controls, identify the currently active tab, group, or control, or identify the selected item from a gallery; indeed, missing are all the `getItem` equivalents to those properties that were added to the `CommandBars` object.
- ❑ There is no way to execute an item from a `comboBox`, `dropdown`, or `gallery`. `ExecuteMso` should take an optional `ItemID` or `ItemIndex` parameter.
- ❑ Every RibbonX container item, from the root `customUI` element to a deeply nested menu, should support having its contents set dynamically, using a `getContent` callback.
- ❑ Both the RibbonX XML and any images in the file should be exposed through the `Workbook` object's `CustomXMLParts` collection, so they can be easily read or updated with VBA.
- ❑ There is no way for a custom group to fit in with the `resize/collapse` behavior of the built-in groups. You should be able to determine how large a custom group is, and how much spare space there is on a tab—thereby allowing you to change what you display to make the best use of the available space.

Summary

In RibbonX, Microsoft has done an extremely good job of exposing the capabilities of the Office 2007 Ribbon for developers to use.

You can easily create custom tabs, groups, menus, buttons, toggle buttons, drop-downs, galleries, checkboxes, and dynamic menus to add your application's features to the Ribbon.

You have a comprehensive and cohesive set of attributes that can be applied to all control types, and (nearly) every attribute has an equivalent `getAttribute` callback.

Using those controls and attributes, you can create an entirely new style of interface for your applications—significantly better than previous versions' CommandBars.

Though very good, RibbonX is still a v1.0 technology, and you might encounter a number of limitations when trying to push its limits.

Command Bars

The `CommandBars` collection is an object contained in the Office Object Model, documented in Appendix C. It contains all the menus, toolbars, and shortcut popup menus that are already built into Excel and the other Office applications, as well as any of those objects that you create yourself. You access command bars through the `CommandBars` property of the `Application` object.

Command bars were first introduced into Office in Office 97. Excel 5 and 95 supported menu bars and toolbars as separate object types. Shortcut menus, or popups, such as those that appear when you right-click a worksheet cell, were a special type of menu bar. In Excel 97 and later versions, *command bar* is a generic term that includes menu bars, toolbars, and shortcut menus.

Excel 2007 has made a giant leap forward and replaced menus and toolbars with the Ribbon to provide access to commands. Only the popup type command bars are included in the standard Excel 2007 user interface. However, Excel 2007 still maintains all the old command bars internally and still allows you to create your own command bars, although the way menus and toolbars are exposed is different compared with previous versions. User-defined menus and toolbars now appear in the Add-Ins tab of the Ribbon.

The Visual Basic Editor window in Excel 2007 still has the menu and toolbar interface it has always supported. You can manipulate the VBE command bars using the same techniques that are outlined in this chapter. See Chapter 26 for more information.

Command bars contain items that are called *controls*. When clicked, some controls execute operations, such as copy. Until this chapter gets down to the nuts and bolts, these types of controls will be referred to as *commands*. There are other controls, such as File, that produce an additional list of controls when clicked. These controls are referred to as *menus*. You can create new controls, or you can use the built-in controls that come with Excel.

This chapter shows you how to create and manipulate these useful tools. It takes a look at how command bars were used in Excel 2003 and then shows how to use them in Excel 2007.

Toolbars, Menu Bars, and Popups

Figure 15-1 shows the standard Worksheet menu bar at the top of the Excel window in Excel 2003.

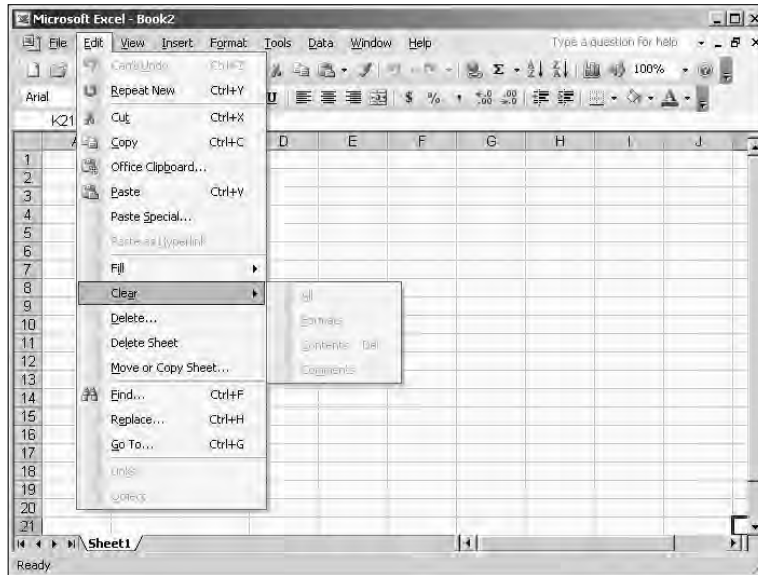


Figure 15-1

The Worksheet menu bar contains menus, such as File and Edit. When you click a menu, you see another list containing commands and menus:

- Cut and Copy are examples of commands in the Edit menu.
- Clear is an example of a menu contained within the Edit menu.

Figure 15-2 shows the Standard toolbar in Excel 2003.



Figure 15-2

Toolbars contain controls that can be clicked to execute Excel commands. For example, the button with the scissors icon carries out a cut. Toolbars can also contain other types of controls, such as the zoom combo box (two from the end of the Standard toolbar in Figure 15-2) that allows you to select or type in a zoom factor, displayed as a percentage. Some toolbars contain buttons that display menus.

Figure 15-3 shows the shortcut menu that appears when you right-click a worksheet cell in Excel 2003.

This shortcut menu contains commands, such as Paste, and menus, such as Delete, for those operations appropriate to the selected context, in this case a worksheet cell.

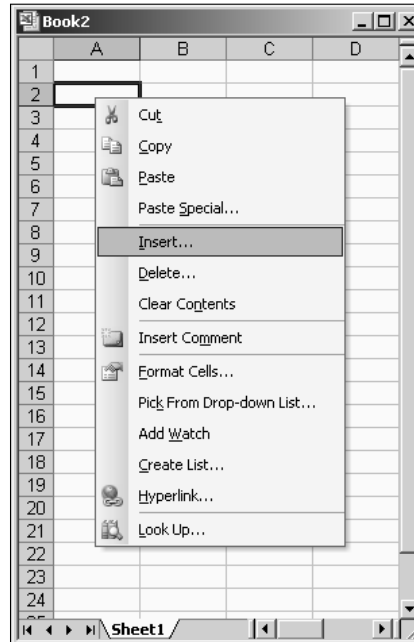


Figure 15-3

To summarize, a command bar can be any one of three types. It can be a menu, toolbar, or shortcut popup menu. When you create a command bar using VBA, you specify which of the three types it will be, using the appropriate parameters of the `Add` method of the `CommandBars` collection. You will see examples of this later in the chapter. You can find out what type an existing command bar is by testing its `Type` property, which will return a numeric value equal to the value of one of the following intrinsic constants.

Constant	CommandBar Type
<code>msoBarTypeNormal</code>	Toolbar
<code>msoBarTypeMenuBar</code>	Menu bar
<code>msoBarTypePopup</code>	Shortcut menu

Controls on command bars also have a `Type` property similar to the `msoXXX` constants shown in the preceding table. The control that is used most frequently has a `Type` property of `msoControlButton`, which represents a command such as the Copy command on the Edit menu of the Worksheet menu bar, or a command button on a toolbar, such as the Cut button on the Standard toolbar. This type of control runs a macro or a built-in Excel action when it is clicked.

The second most common control has a `Type` property of `msoControlPopup`. This represents a menu on a menu bar, such as the Edit menu on the Worksheet menu bar, or a menu contained in another a menu, such as the Clear submenu on the Edit menu on the Worksheet menu bar. This type of control contains its own `Controls` collection, to which you can add further controls.

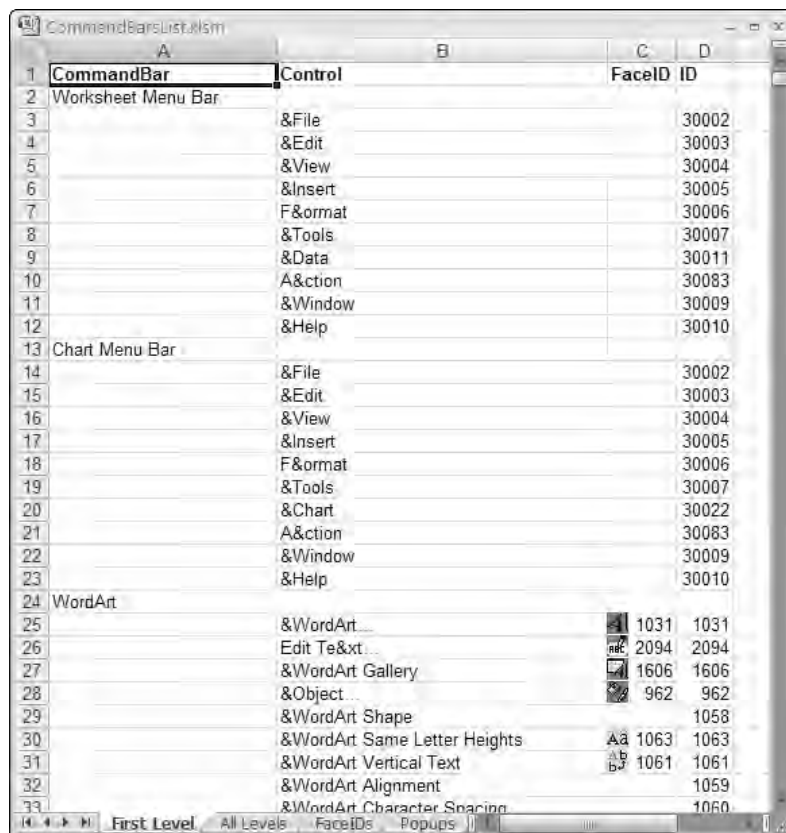
Chapter 15: Command Bars

Controls have an `Id` property. For built-in controls, the `Id` property determines the internal action carried out by the control. When you set up a custom control, you assign the name of a macro to its `OnAction` property to make it execute that macro when it is clicked. Custom controls have an `Id` property of 1.

Many built-in menu items and most built-in toolbar controls have a graphic image associated with them. The image is defined by the `FaceId` property. The `Id` and `FaceId` properties of built-in commands normally have the same numeric value. You can assign the built-in `FaceId` values to your own controls, if you know what numeric value to use. You can determine these values using VBA, as you will see in the next example.

Excel's Built-in Command Bars

Before launching into creating your own command bars, it will help to understand how the built-in command bars are structured. You can use the following code to list the existing command bars and any that you have added yourself. It lists the name of each command bar in column A and the names of the controls in the command bars `Controls` collection in column B, as shown in Figure 15-4. The code does not attempt to display lower-level controls that belong to controls such as the File menu on the Worksheet menu bar, so the procedure has been named `ListFirstLevelControls`.









	A	B	C	D
1	CommandBar	Control	FaceID	ID
2	Worksheet Menu Bar			
3		&File		30002
4		&Edit		30003
5		&View		30004
6		&Insert		30005
7		F&ormat		30006
8		&Tools		30007
9		&Data		30011
10		A&ction		30083
11		&Window		30009
12		&Help		30010
13	Chart Menu Bar			
14		&File		30002
15		&Edit		30003
16		&View		30004
17		&Insert		30005
18		F&ormat		30006
19		&Tools		30007
20		&Chart		30022
21		A&ction		30083
22		&Window		30009
23		&Help		30010
24	WordArt			
25		&WordArt...	 1031	1031
26		Edit Te&xt...	 2094	2094
27		&WordArt Gallery	 1606	1606
28		&Object...	 962	962
29		&WordArt Shape		1058
30		&WordArt Same Letter Heights	 1063	1063
31		&WordArt Vertical Text	 1061	1061
32		&WordArt Alignment		1059
33		&WordArt Character Spacing		1060

Figure 15-4

The macro also shows the control's `Id` property value, in all cases, and its image and its `FaceId` property value when such an image exists. Although you can no longer display the built-in command bars, you can use the controls and images they contain in your own command bars.

Make sure you are in an empty worksheet when you run this macro and the following two examples. They contain tests to make sure they will not overwrite any data in the active sheet.

If you are testing this code, it should be placed in a standard code module, not in a class module. Don't put the code in the `ThisWorkbook` module or a class module behind a worksheet. You should also include the `IsEmptyWorksheet` function listed further down.

Here is the code to list the first-level controls:

```
Sub ListFirstLevelControls()
    Dim ctl As CommandBarControl
    Dim cbr As CommandBar
    Dim iRow As Integer

    If Not IsEmptyWorksheet(ActiveSheet) Then Exit Sub

    'Ignore errors and freeze screen
    On Error Resume Next
    Application.ScreenUpdating = False

    'Enter headings
    Cells(1, 1).Value = "CommandBar"
    Cells(1, 2).Value = "Control"
    Cells(1, 3).Value = "FaceID"
    Cells(1, 4).Value = "ID"
    Cells(1, 1).Resize(1, 4).Font.Bold = True

    'Start at row 2
    iRow = 2

    'Loop through all commandbars
    For Each cbr In CommandBars
        Application.StatusBar = "Processing Bar " & cbr.Name
        Cells(iRow, 1).Value = cbr.Name
        iRow = iRow + 1

        'Loop through controls on commandbar
        For Each ctl In cbr.Controls
            Cells(iRow, 2).Value = ctl.Caption

            'Try to get image
            ctl.CopyFace
            If Err.Number = 0 Then
                ActiveSheet.Paste Cells(iRow, 3)
                Cells(iRow, 3).Value = ctl.FaceId
            End If
        Next
    Next
End Sub
```

```
End If

Cells(iRow, 4).Value = ctl.ID
Err.Clear

iRow = iRow + 1

Next ctl

Next cbr

Range("A:D").EntireColumn.AutoFit

Application.StatusBar = False

End Sub
```

This example, and the two following, can take a long time to complete. You can watch the progress of the code on the status bar. If you only want to see part of the output, press Ctrl+Break after a minute or so to interrupt the macro, click Debug, and then choose Run ↻ Reset.

`ListFirstLevelControls` first checks that the active sheet is an empty worksheet, using the `IsEmptyWorksheet` function that is shown in the following code. It then uses `OnError Resume Next` to avoid run-time errors when it tries to access control images that do not exist. In the outer `ForEach...Next` loop, it assigns a reference to each command bar to `cbr`, shows the `Name` property of the command bar on the status bar so you can track what it is doing, and places the `Name` in the A column of the current row, defined by `iRow`.

The inner `ForEach...Next` loop processes all the controls on `cbr`, placing the `Caption` property of each control in column B. It then attempts to use the `CopyFace` method of the control to copy the control's image to the clipboard. If this does not create an error, it pastes the image to column C and places the value of the `FaceId` property in the same cell. It places the `ID` property of the control in column D. It clears any errors, increments `iRow` by one, and processes the next control.

The `IsEmptyWorksheet` function, shown next, checks that the input parameter object `sht` is a worksheet. If so, it checks that the count of entries in the used range is 0. If both checks succeed, it returns `True`. Otherwise, it issues a warning message and the default return value, which is `False`, is returned:

```
Function IsEmptyWorksheet(sht As Object) As Boolean

    'If sht is a worksheet, count the non empty cells
    If TypeName(sht) = "Worksheet" Then

        If WorksheetFunction.CountA(sht.UsedRange) = 0 Then

            IsEmptyWorksheet = True
```

```

Exit Function

End If

End If

MsgBox "Please make sure that an empty worksheet is active"

End Function

```

Controls at All Levels

Figure 15-5 and the following code take the previous procedure to greater levels of detail. All controls are examined to see what controls are contained within them. Where possible, the contained controls are listed. Some controls, such as those containing graphics, can't be listed in greater detail. The information on sub-controls is indented across the worksheet. The code is capable of reporting to as many levels as there are, but Excel 2007 does not have controls beyond the fourth level.

Row	Control Name	Level	Sub-Control	Count	Icon	Value
1	Worksheet Menu Bar	&File	10 &New...	1	[icon]	18
2		&Open...		1	[icon]	23
3		&Close...		1	[icon]	106
4		&Save...		1	[icon]	3
5		Save &As...		1	[icon]	748
6		Save as Web Pa&ge...		1	[icon]	3823
7		Save &Workspace...		1	[icon]	846
8		File Searc&h...		1	[icon]	5905
9		Per&mission...		1	[icon]	7994
10		Per&mission	10 &Unrestricted Access	1	[icon]	7990
11			&Restricted Access	1	[icon]	7991
12			&Manage Credentials	1	[icon]	10014
13		Ch&eck Out		1	[icon]	6127
14		Ch&eck In...		1	[icon]	6128
15		Ve&rsion History...		1	[icon]	7799
16		We&b Page Preview		1	[icon]	3655
17		Page Set&up...		1	[icon]	247
18		Prin&t Area	10 &Set Print Area	1	[icon]	364
19			&Clear Print Area	1	[icon]	1584
20		Print Pre&view		1	[icon]	109
21		&Print		1	[icon]	4
22		Sen&d To	10 &Mail Recipient	1	[icon]	3738
23			Original &Sender...	1	[icon]	6139
24			Mail Re&ipient (for Review)...	1	[icon]	5958
25			M&ail Recipient (as Attachment)...	1	[icon]	2188
26			&Exchange Folder	1	[icon]	938
27			&Online Meeting Participant	1	[icon]	3728
28			Recipient using Internet Fa&x Service...	1	[icon]	7392
29		Propert&ies		1	[icon]	750

Figure 15-5

Here is the code to list controls at all levels:

```

Sub ListAllControls()
    Dim cbr As CommandBar
    Dim rng As Range

```

Chapter 15: Command Bars

```
Dim ctl As CommandBarControl

'Test for empty worksheet and freeze screen
If Not IsEmptyWorksheet(ActiveSheet) Then Exit Sub
Application.ScreenUpdating = False

'Start in A1 cell
Set rng = Range("A1")

'Loop through all commandbars
For Each cbr In Application.CommandBars
    Application.StatusBar = "Processing Bar " & cbr.Name

    'List name of bar
    rng.Value = cbr.Name

    'Loop through controls on bar
    For Each ctl In cbr.Controls

        'Call ListControls function
        Set rng = rng.Offset(ListControls(ctl, rng))

    Next ctl

Next cbr

'Fit columns to data
Range("A:J").EntireColumn.AutoFit

Application.StatusBar = False

End Sub
```

`ListAllControls` loops through the `CommandBars` collection, using `rng` to keep track of the current A column cell of the worksheet it is writing to. It posts the name of the current command bar in a message on the status bar, so you can tell where it is up to, and also enters the name of the command bar at the current `rng` location in the worksheet. It then loops through all the controls on the current command bar, executing the `ListControls` function, which is shown in the next code snippet.

In Chapter 26, you will need to get listings of the VBE command bars. You can easily accomplish this by changing the following line:

```
For Each cBar in Application.CommandBars
```

To:

```
For Each cBar in Application.VBE.CommandBars
```

`ListControls` is responsible for listing the details of each control it is passed and the details of any controls under that control, starting at the current `rng` location in the worksheet. When it has performed its tasks, `ListControls` returns a value equal to the number of lines that it has used for its list. `Offset` is used to compute the new `rng` cell location for the start of the next command bar's listing:


```

Function ListControls(ctl As CommandBarControl, rng As Range) As Long
    Dim lOffset As Long    'Tracks current row relative to rng
    Dim ctlSub As CommandBarControl    'Control contained in ctl

    'Ignore Errors
    On Error Resume Next

    'Start in rng cell
    lOffset = 0

    'List control name and type
    rng.Offset(lOffset, 1).Value = ctl.Caption
    rng.Offset(lOffset, 2).Value = ctl.Type

    'Attempt to copy control face. If error, don't paste
    ctl.CopyFace
    If Err.Number = 0 Then
        ActiveSheet.Paste rng.Offset(lOffset, 3)
        rng.Offset(lOffset, 3).Value = ctl.FaceId
    End If
    Err.Clear

    'Check Control Type
    Select Case ctl.Type

    Case 1, 2, 4, 6, 7, 13, 18
        'Do nothing for these control types

    Case Else
        'Call function recursively if current control contains other controls
        For Each ctlSub In ctl.Controls
            lOffset = lOffset + _
                ListControls(ctlSub, rng.Offset(lOffset, 2))
        Next ctlSub

        lOffset = lOffset - 1

    End Select

    ListControls = lOffset + 1

End Function

```

ListControls is a recursive function, and it runs itself to process as many levels of controls as it finds. It uses lOffset to keep track of the rows it writes to, relative to the starting cell rng. It uses very similar code to ListFirstLevelControls, but records the control type as well as the caption, icon, and face ID. Most of the control types are:

- 1—msoControlButton
- 10—msoControlPopup

Chapter 15: Command Bars

However, you will see other types in the list as well:

- 2—msoControlEdit
- 4—msoControlComboBox
- 6—msoControlSplitDropdown
- 7—msoControlOCXDropdown
- 13—msoControlSplitButtonPopup
- 18—msoControlGrid

The `Select Case` construct is used to avoid trying to list the sub-controls where this is not possible.

When `ListControls` finds a control with sub-controls it can list, it calls itself with a `rng` starting point that is offset from its current `rng` by `10Offset` lines down and two columns across. `ListControls` keeps calling itself as often as necessary to climb down into every level of sub-control, and then it climbs back to continue with the higher levels. Each time it is called, it returns the number of lines it has written to, relative to `rng`.

Facelids

The following code gives you a table of the built-in button faces, as shown in Figure 15-6. There are more than 15,000 faces in Office 2007. Note that many `FaceId` values represent blank images, and that the same images appear repeatedly as the numbers increase.

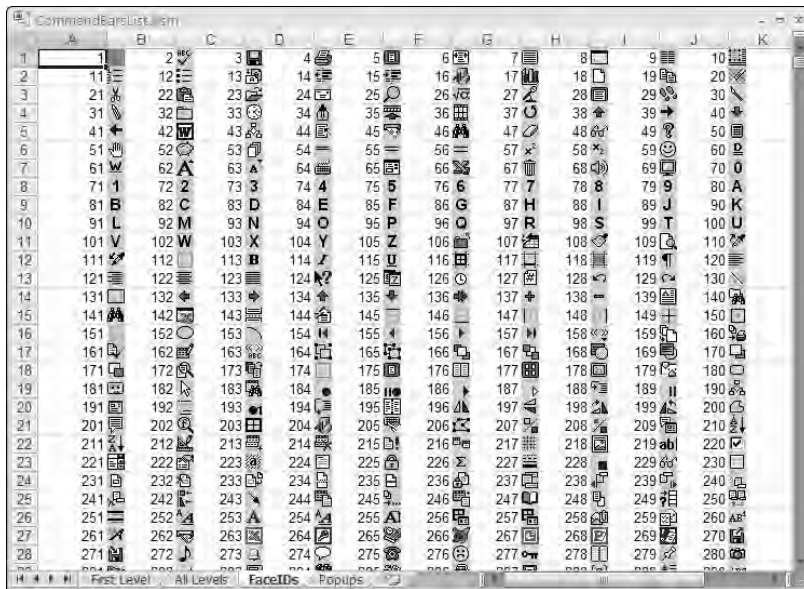


Figure 15-6

Here is the code to list all the FaceIds:

```

Sub ListAllFaces()
    Dim iFaceId As Integer    'Tracks current FaceId
    Dim iColumn As Integer    'Tracks current column in worksheet
    Dim iRow As Integer      'Tracks current row in worksheet
    Dim ctl As CommandBarControl
    Dim cbr As CommandBar

    If Not IsEmptyWorksheet(ActiveSheet) Then Exit Sub

    'Ignore errors and freeze screen
    On Error GoTo Recover
    Application.ScreenUpdating = False

    'Create temporary command bar with single control button
    'to hold control button face to be copied to worksheet
    Set cbr = CommandBars.Add(Position:=msoBarFloating, _
                               MenuBar:=False, _
                               temporary:=True)
    Set ctl = cbr.Controls.Add(Type:=msoControlButton, _
                               temporary:=True)

    iRow = 1

    Do

        For iColumn = 1 To 10
            iFaceId = iFaceId + 1
            Application.StatusBar = "FaceID = " & iFaceId
            'Set control button to current FaceId
            ctl.FaceId = iFaceId
            'Attempt to copy Face image to worksheet
            ctl.CopyFace
            ActiveSheet.Paste Cells(iRow, iColumn + 1)
            Cells(iRow, iColumn).Value = iFaceId
        Next iColumn
        iRow = iRow + 1

    Loop

Recover:
    If Err.Number = 1004 Then Resume Next

    Application.StatusBar = False
    cbr.Delete

End Sub

```

Note that when you run this macro, your computer may well freeze for a while. The CPU has to do a lot of hard work! You can follow the macro's progress by watching the status bar.

Chapter 15: Command Bars

ListAllFaces creates a temporary toolbar, *cbr*, using the Add method of the CommandBars collection. The toolbar is declared:

- Temporary, which means that it will be deleted when you exit Excel, if it has not already been deleted
- Floating, rather than docked at an edge of the screen or a popup
- Not to be a menu bar, which means that *cbBar* will be a toolbar

A temporary control is added to *cbr*, using the Add method of the Controls collection for the command bar, and assigned to *ctl*.

The Do...Loop continues looping until there are no more valid FaceId values. The Do...Loop increments *iRow*, which represents the row numbers in the worksheet. On every row, *iColumn* is incremented from 1 to 10. *j* represents the columns of the worksheet. The value of *iRow* is increased by 1 for every iteration of the code in the For...Next loop. *iFaceId* represents the FaceId. The FaceId property of *ctl* is assigned the value of *iFaceId*, and the resulting image is copied to the worksheet.

Some button images are blank and some are missing. The blank images are copied without error, but the missing images cause an error number 1004. When an error occurs, the code branches to the error trap at *Recover*:. If the error number is 1004, the code resumes executing at the statement after the one that caused the error, leaving an empty slot for the button image. Eventually the code gets to the last FaceId in Office. This causes error number -2147467259. At this point the code clears the status bar, removes the temporary command bar, and exits.

The information you have gathered with the last three exercises is not documented in any easily obtainable form by Microsoft. It is a valuable guide to the built-in button faces at your disposal. There is an add-in application, *CBList.xla*, available with the code that accompanies this book that makes it easy to generate these lists.

Creating New Menus

In versions of Office prior to Office 2007, you can add a new menu to the built-in command bars. The code is still valid in Office 2007, but produces a different result. The new menu is displayed in the Add-Ins tab of the Ribbon, as shown in Figure 15-7.

The code to create this menu is as follows:

```
Public Sub AddCustomMenu()  
    Dim cbr As CommandBar  
    Dim ctlMenu As CommandBarControl  
  
    'Add new menu control  
    Set cbr = Application.CommandBars("Worksheet Menu Bar")  
    Set ctlMenu = cbr.Controls.Add(Type:=msoControlPopup)  
  
    'Add controls to new menu control
```

```

With ctlMenu
    .Caption = "Custom"

    With .Controls.Add(Type:=msoControlButton)
        .Caption = "Show Data Form"
        .OnAction = "ShowDataForm"
    End With

    With .Controls.Add(Type:=msoControlButton)
        .Caption = "Print Data List"
        .OnAction = "PrintDataList"
    End With

    With .Controls.Add(Type:=msoControlButton)
        .Caption = "Sort Names Ascending"
        .BeginGroup = True
        .OnAction = "SortList"
        .Parameter = "Asc"
    End With

    With .Controls.Add(Type:=msoControlButton)
        .Caption = "Sort Names Descending"
        .OnAction = "SortList"
        .Parameter = "Dsc"
    End With

    With .Controls.Add(Type:=msoControlButton)
        .Caption = "Show Products"
        .OnAction = "'ShowProduct "' & "Apple", 3, 4'"
    End With

End With

End Sub

```

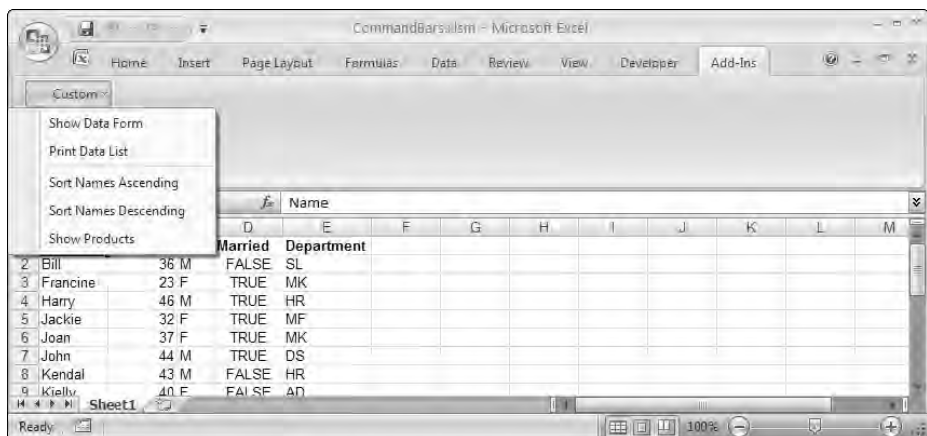


Figure 15-7

Chapter 15: Command Bars

`AddCustomMenu` creates an object variable `cbr` referencing the Worksheet menu bar, and uses the `Add` method of the menu bar's `Controls` collection to add the new menu. The `Type` property is declared `msoControlPopup` so that other controls can be attached to the menu. The `Caption` property of the new menu is assigned `Custom`.

The `Add` method of the new menu's `Controls` collection is then used to add four commands to the menu. They are all of type `msoControlButton` so they can each run a macro. Each is given an appropriate `Caption` property. The `OnAction` property of each command is assigned the name of the macro it is to run. The first of the sort menu items has its `BeginGroup` property set to `True`. This places the dividing line above it to mark it as the beginning of a different group. Both sort commands are assigned the same `OnAction` macro, but also have their `Parameter` properties assigned text strings that distinguish them.

The `Parameter` property is a holder for a character string. You can use it for any purpose. Here it is used to hold the strings `"Asc"`, for ascending, and `"Dsc"`, for descending. As you will see in the next section, the `SortList` procedure will access the strings to determine the sort order required.

The OnAction Macros

The macro assigned to the `OnAction` property of the Show Data Form menu item is as follows:

```
Private Sub ShowDataForm()  
    frmPersonal.Show  
End Sub
```

It displays exactly the same data form as in Chapter 13. The macro assigned to the Print Data List menu item is as follows:

```
Private Sub PrintDataList()  
    Range("Database").PrintPreview  
End Sub
```

`PrintDataList` shows a print preview of the list, from which the user can elect to print the list.

The macro assigned to the Sort menu items is as follows:

```
Private Sub SortList()  
    Dim lAscDsc As Long  
  
    Select Case CommandBars.ActionControl.Parameter  
  
        Case "Asc"  
            lAscDsc = xlAscending  
  
        Case "Dsc"
```

```

        lAscDsc = xlDescending

    End Select

    Range("Database").Sort Key1:=Range("A2"), Order1:=lAscDsc, Header:=xlYes

End Sub

```

`SortList` uses the `ActionControl` property of the `CommandBars` collection to get a reference to the command bar control that caused `SortList` to execute. This is similar to `Application.Caller`, used in user-defined functions to determine the `Range` object that executed the function.

Knowing the control object that called it, `SortList` can examine the control's `Parameter` property to get further information. If the `Parameter` value is "Asc", `SortList` assigns an ascending sort. If the `Parameter` value is "Dsc", it assigns a descending sort. Controls also have a `Tag` property that can be used, in exactly the same way as the `Parameter` property, to hold another character string. You can use the `Tag` property as an alternative to the `Parameter` property, or you can use it to hold supplementary data.

Passing Parameter Values

The previous example used the `Parameter` property of the control on the menu to store information to be passed to the `OnAction` macro, and pointed out that you can also use the `Tag` property. If you have more than two items of information to pass, it is more convenient to use a macro procedure that has input parameters.

Say you wanted to pass three items of data, such as a product name and its cost and selling price. The macro might look like the following:

```

Sub ShowProduct(sName As String, dCost As Double, dPrice As Double)

    MsgBox "Product: " & sName & vbCr & _
        "Cost: " & Format(dCost, "$0.00") & vbCr & _
        "Price: " & Format(dPrice, "$0.00")

End Sub

```

To execute this macro from a command bar control, you need to assign something like the following code to the `OnAction` property of the control:

```
'ShowProduct "Apple", 3, 4'
```

The entire expression is enclosed in single quotes. Any string parameter values within the expression are enclosed in double quotes. To define this as the `OnAction` property of a control referred to by an object variable, `ctl`, for example, you need to use the following code:

```
ctl.OnAction = "'ShowProduct ""Apple"", 3, 4''
```

Chapter 15: Command Bars

The mix of single and double quotes is tricky to get right. The entire string is enclosed in double quotes, and any internal double quotes need to be shown twice.

Deleting a Menu

Built-in and custom controls can be deleted using the control's `Delete` method. The following macro deletes the Custom menu:

```
Public Sub RemoveCustomMenu()  
    Dim cbr As CommandBar  
  
    On Error Resume Next  
  
    Set cbr = CommandBars("Worksheet Menu Bar")  
    cbr.Controls("Custom").Delete  
  
End Sub
```

`On Error` is used in case the menu has already been deleted.

You can use a built-in command bar's `Reset` method to make the entire command bar revert to its default layout and commands. This is not a good idea if users have other workbooks or add-ins that alter the setup, because all their work will be lost.

The following event procedures should be added to the `ThisWorkbook` module to add the Custom menu when the workbook is opened, and delete it when the workbook is closed:

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)  
  
    Call RemoveCustomMenu  
  
End Sub  
  
Private Sub Workbook_Open()  
  
    Call AddCustomMenu  
  
End Sub
```

It is important to recognize that command bar changes are permanent. If you do not remove the Custom menu in this example, it will stay in the Excel Worksheet menu bar during the current session and future sessions. Trying to use this menu with another workbook active could cause unexpected results.

Creating a Toolbar

In previous versions of Office, you could manually create a simple toolbar with buttons and drop-downs. Now you can only do this using VBA code. The more complex controls, such as those of type `msoControlEdit`, `msoControlDropdown`, and `msoControlComboBox`, have always required VBA code. As with the new menu created earlier, Excel 2007 displays the new toolbar in the Add-Ins tab of the Ribbon. The toolbar in Figure 15-8 contains three controls.

The first is of type `msoControlButton` and displays the user form for the data list.

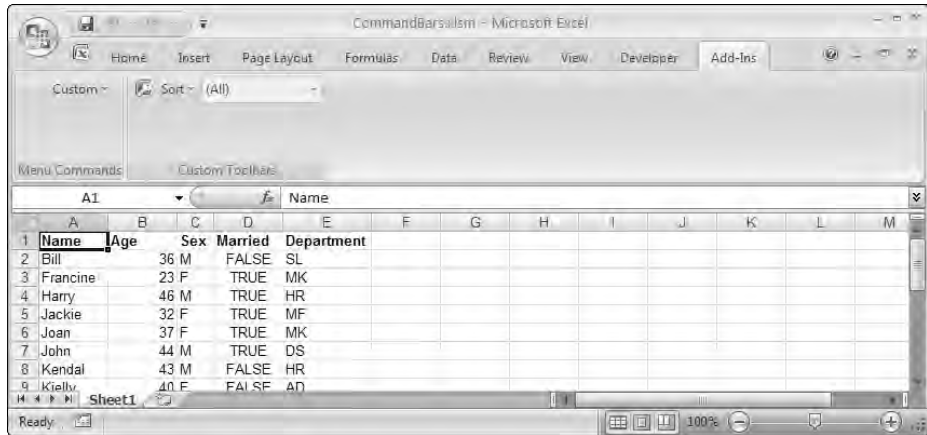


Figure 15-8

The second control is of type `msoControlPopup` and displays two controls of type `msoControlButton`, as shown in Figure 15-9.

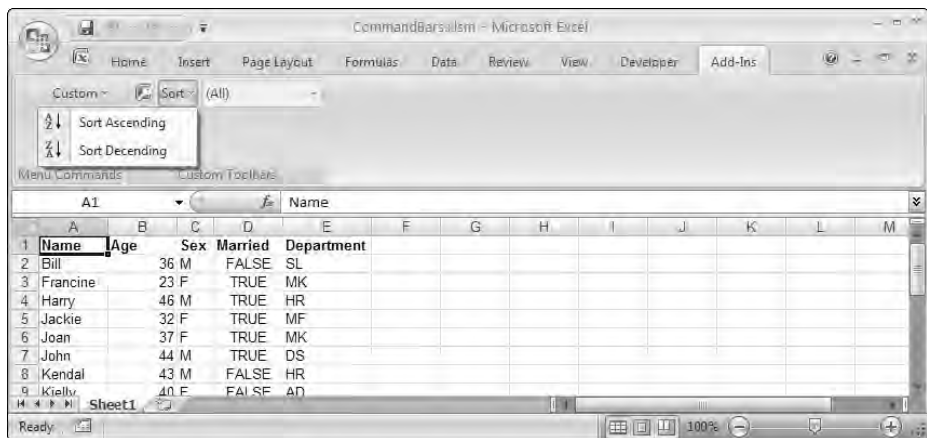


Figure 15-9

Chapter 15: Command Bars

The third control is of type `msoControlDropdown` and applies an `AutoFilter` on `Department`, as shown in Figure 15-10.

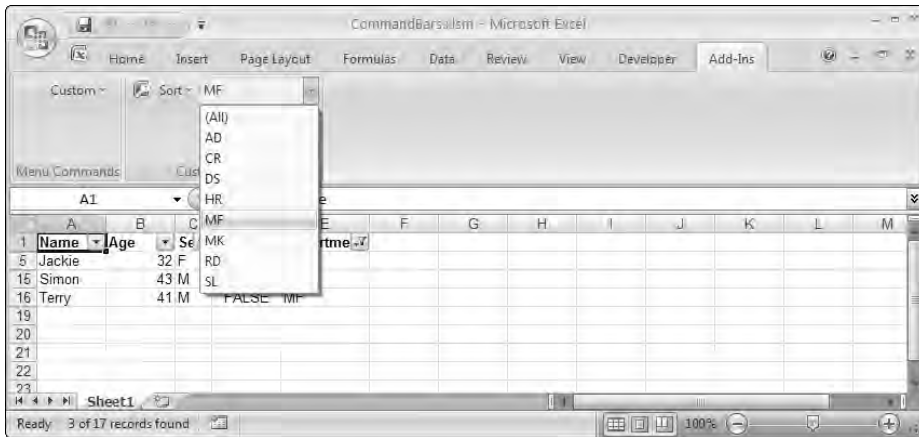


Figure 15-10

The following code creates the toolbar:

```
Public Sub CreateToolbar()  
'Get rid of any existing toolbar called Manage Data  
On Error Resume Next  
CommandBars("Manage Data").Delete  
On Error GoTo 0  
  
'Create new toolbar  
With CommandBars.Add(Name:="Manage Data")  
  
    With .Controls.Add(Type:=msoControlButton)  
        .OnAction = "ShowDataForm"  
        .FaceId = 264  
        .TooltipText = "Show Data Form"  
    End With  
  
    With .Controls.Add(Type:=msoControlPopup)  
        .Caption = "Sort"  
        .TooltipText = "Sort Ascending or Descending"  
  
        With .Controls.Add(Type:=msoControlButton)  
            .Caption = "Sort Ascending"  
            .FaceId = 210  
            .OnAction = "SortList"  
            .Parameter = "Asc"  
        End With  
  
        With .Controls.Add(Type:=msoControlButton)  
            .Caption = "Sort Descending"  
            .FaceId = 211  
            .OnAction = "SortList"  
            .Parameter = "Dsc"  
        End With  
    End With  
End With
```

```

        End With

    End With

    With .Controls.Add(Type:=msoControlDropdown)
        .AddItem "(All)"
        .AddItem "AD"
        .AddItem "CR"
        .AddItem "DS"
        .AddItem "HR"
        .AddItem "MF"
        .AddItem "MK"
        .AddItem "RD"
        .AddItem "SL"
        .OnAction = "FilterDepartment"
        .TooltipText = "Select Department"
    End With

    .Visible = True

End With

End Sub

```

The toolbar itself is very simple to create. `CreateToolbar` uses the `Add` method of the `CommandBars` collection and accepts all the default parameter values, apart from the `Name` property. The first control button is created in much the same way as a menu item, using the `Add` method of the `Controls` collection. It is assigned an `OnAction` macro, a `FaceId`, and a `ToolTip`.

The second control is created as type `msoControlPopup`. It is given the `Caption` of `Sort` and a `ToolTip`. It is then assigned two controls of its own, of type `msoControlButton`. They are assigned the `SortList` macro and `Parameter` values, as well as `FaceIds` and captions.

Finally, the control of type `msoControlDropdown` is added. Its drop-down list is populated with department codes and its `OnAction` macro is `FilterDepartment`. It is also given a `ToolTip`. The last action is to set the toolbar's `Visible` property to `True` to display it.

The `FilterDepartment` macro follows:

```

Sub FilterDepartment()
    Dim sDept As String

    With CommandBars.ActionControl
        sDept = .List(.ListIndex)
    End With

    If sDept = "(All)" Then
        Range("Database").Parent.AutoFilterMode = False
    Else
        Range("Database").AutoFilter Field:=5, Criteria1:=sDept
    End If

End Sub

```

Chapter 15: Command Bars

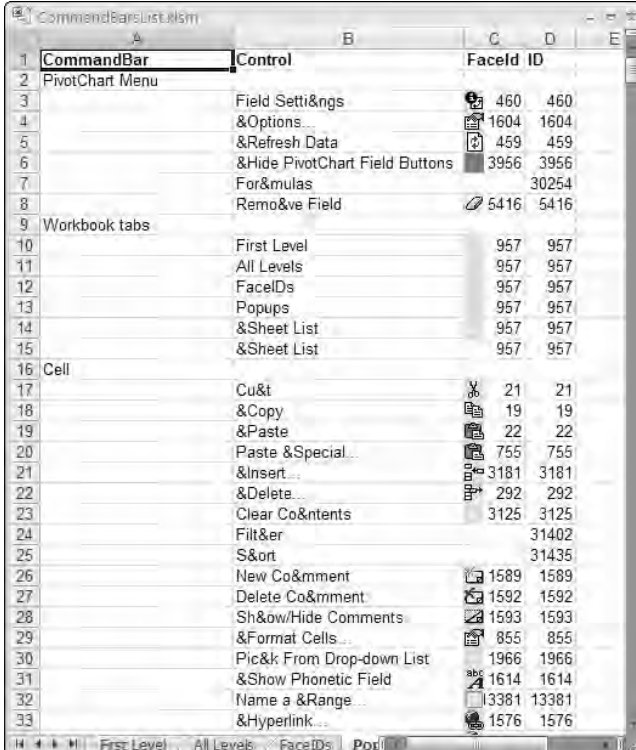
A drop-down control has a `List` property that is an array of its list values and a `ListIndex` property that is the index number of the current list value. The `ActionControl` property of the `CommandBar` object, which refers to the currently active control, is a quick way to reference the control and access the `List` and `ListIndex` properties to get the department code required. The code is then used to perform the appropriate `AutoFilter` operation. If the (All) option is chosen, the `AutoFilterMode` property of the worksheet that is the parent of the `Database Range` object is set to `False`, removing the `AutoFilter` drop-downs and showing any hidden rows.

It is a good idea to run `CreateToolbar` from the `Workbook_Open` event procedure, and to delete the toolbar in the `Workbook_BeforeClose` event procedure. The toolbar will remain permanently in Excel if it is not deleted, and will give unexpected results if its buttons are pressed when other workbooks are active. If you do refer to command bars directly in workbook event procedures, you need to qualify the reference with `Application`:

```
Application.CommandBars("Manage Data").Delete
```

Popup Menus

Excel's built-in shortcut menus are included in the command bar listing created by the macro `ListFirstLevelControls`, which you saw earlier in this chapter. The following modified version of this macro shows only the command bars of type `msoBarTypePopup`, as shown in Figure 15-11.



The screenshot shows an Excel spreadsheet titled "CommandBarsList1511sm". The spreadsheet contains a list of command bars and their properties. The columns are labeled "CommandBar", "Control", and "FaceId ID". The data is as follows:

CommandBar	Control	FaceId	ID
PivotChart Menu			
	Field Setti&ngs	460	460
	&Options...	1604	1604
	&Refresh Data	459	459
	&Hide PivotChart Field Buttons	3956	3956
	For&mulas		30254
	Remo&ve Field	5416	5416
Workbook tabs			
	First Level	957	957
	All Levels	957	957
	FaceIDs	957	957
	Popups	957	957
	&Sheet List	957	957
	&Sheet List	957	957
Cell			
	Cu&t	21	21
	&Copy	19	19
	&Paste	22	22
	Paste &Special...	755	755
	&Insert...	3181	3181
	&Delete...	292	292
	Clear Co&ntents	3125	3125
	Filt&r		31402
	S&ort		31435
	New Co&mment	1589	1589
	Delete Co&rment	1592	1592
	Sh&ow/Hide Comments	1593	1593
	&Format Cells...	855	855
	Pic&k From Drop-down List	1966	1966
	&Show Phonetic Field	1614	1614
	Name a &Range...	13381	13381
	&Hyperlink...	1576	1576

Figure 15-11

The code to display the popups is shown here:

```

Sub ListPopups ()
    Dim ctl As CommandBarControl
    Dim cbr As CommandBar
    Dim iRow As Integer

    If Not IsEmptyWorksheet(ActiveSheet) Then Exit Sub

    'Ignore errors and freeze screen
    On Error Resume Next
    Application.ScreenUpdating = False

    'Enter headings
    Cells(1, 1).Value = "CommandBar"
    Cells(1, 2).Value = "Control"
    Cells(1, 3).Value = "FaceId"
    Cells(1, 4).Value = "ID"
    Cells(1, 1).Resize(1, 4).Font.Bold = True

    'Set row to 2
    iRow = 2

    'Loop through all commandbars
    For Each cbr In CommandBars
        Application.StatusBar = "Processing Bar " & cbr.Name

        'Only list popups
        If cbr.Type = msoBarTypePopup Then

            Cells(iRow, 1).Value = cbr.Name
            iRow = iRow + 1

            'Loop through controls on popup commandbar
            For Each ctl In cbr.Controls

                Cells(iRow, 2).Value = ctl.Caption
                ctl.CopyFace
                If Err.Number = 0 Then
                    ActiveSheet.Paste Cells(iRow, 3)
                    Cells(iRow, 3).Value = ctl.FaceId
                End If

                Cells(iRow, 4).Value = ctl.ID

                Err.Clear

                iRow = iRow + 1

            Next ctl

        End If
    
```

Chapter 15: Command Bars

```
Next cbr

Range("A:B").EntireColumn.AutoFit

Application.StatusBar = False

End Sub
```

The listing is identical to `ListFirstLevelControls`, apart from the introduction of a block `If` structure that processes only command bars of type `msoBarTypePopup`. If you look at the listing produced by `ListPopups`, you will find you can identify the common shortcut menus. For example, there are command bars named `Cell`, `Row`, and `Column` that correspond to the shortcut menus that pop up when you right-click a worksheet cell, row number, or column letter.

You might be confused about the fact that the `Cell`, `Row`, and `Column` command bars are listed twice. The first set is for a worksheet in Normal view. The second set is for a worksheet in Page Break Preview.

Another tricky one is the Workbook tabs command bar. This is not the shortcut that you get when you click an individual worksheet tab. It is the shortcut for the workbook navigation buttons to the left of the worksheet tabs. The shortcut for the tabs is the Ply command bar.

Having identified the shortcut menus, you can tailor them to your own needs using VBA code. For example, Figure 15-12 shows a modified `Cell` command bar that includes an option to `Clear All`.

The `Clear All` control was added using the following code:

```
Public Sub AddShortCut()
    Dim cbr As CommandBar
    Dim ctl As CommandBarButton
    Dim lIndex As Long

    Set cbr = CommandBars("Cell")

    lIndex = cbr.Controls("Clear Contents").Index
    Set ctl = cbr.Controls.Add(Type:=msoControlButton, _
                              ID:=1964, Before:=lIndex)
    ctl.Caption = "Clear &All"

End Sub
```

`AddShortCut` starts by assigning a reference to the `Cell` command bar to `cbr`.

If you want to refer to the `Cell` command bar that is shown in Page Break view in Excel 2007, you can use its `Index` property:

```
Set cbBar = CommandBars(39)
```

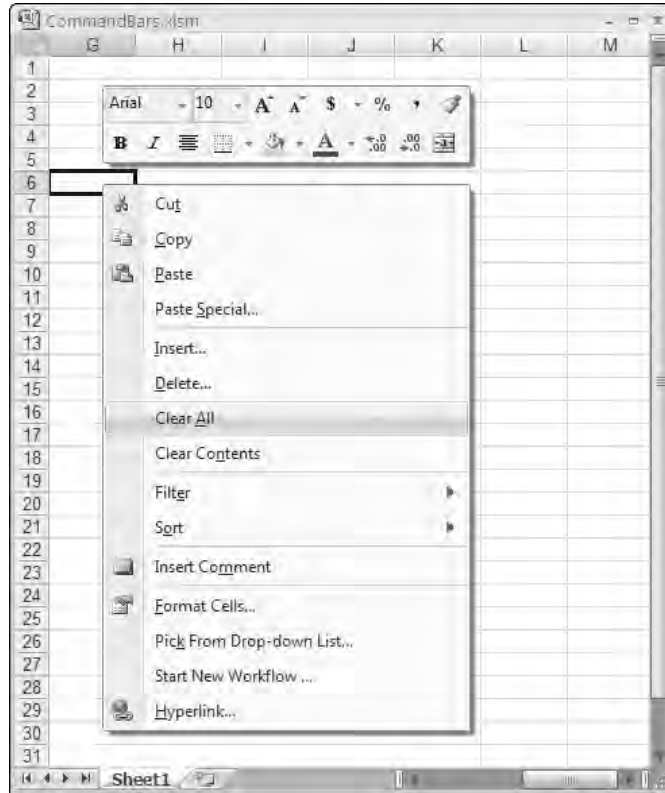


Figure 15-12

You need to take care here, if you want code compatible with other versions of Office. In Excel 2003, the `Index` property of the `Cell` command bar in Page Break view is 32, in Excel 2000 it is 26, and in Excel 97 it is 24.

`AddShortcut` records the `Index` property of the `Clear Contents` control in `lIndex`, so that it can add the new control before the `Clear Contents` control. `AddShortcut` uses the `Add` method of the `Controls` collection to add the new control to `cbBar`, specifying the `Id` property of the built-in `Edit ⇄ Clear ⇄ All` menu item on the Worksheet menu bar.

The `Add` method of the `Controls` collection allows you to specify the `Id` property of a built-in command. The listing from `ListAllControls` allows you to determine that the `Id` property, which is the same as the `FaceId` property, of the `Edit ⇄ Clear ⇄ All` menu item is 1964.

The built-in `Caption` property for the newly added control is `All`, so `AddShortcut` changes the `Caption` to be more descriptive.

You can safely leave the modified `Cell` command bar in your `CommandBars` collection. It is not tied to any workbook and does not depend on having access to macros in a specific workbook.

Showing Popup Command Bars

If you want to display a shortcut menu without having to right-click a cell, or chart, you can create code to display the shortcut in a number of ways. For example, you might like to display the shortcut Cell command bar from the keyboard, using Ctrl+Shift+C. You can do this using the following code:

```
Sub SetShortcut()  
    Application.OnKey "^+c", "ShowCellShortcut"  
End Sub  
  
Private Sub ShowCellShortcut()  
    CommandBars("Cell").ShowPopup x:=0, y:=0  
End Sub
```

ShowCellShortcut uses the ShowPopup method to display the Cell shortcut menu at the top-left corner of the screen. The parameters are the x and y screen coordinates for the top-left corner of the menu.

You can also create a popup menu from scratch. The popup in Figure 15-13 appears when you right-click inside the range named Database. Outside the range, the normal Cell popup menu appears.

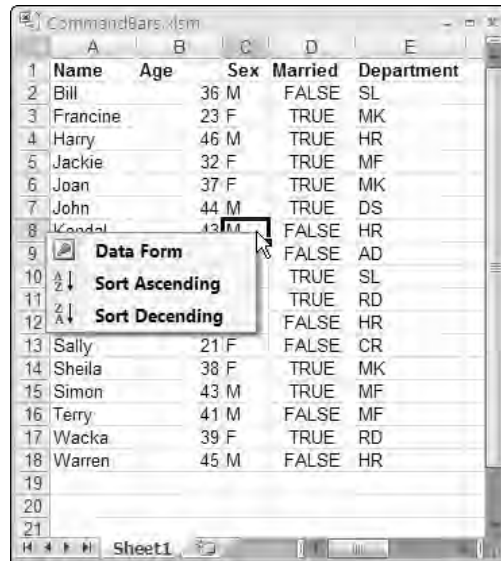


Figure 15-13

The following code created the popup menu:


```

Sub MakePopup()
'Get rid of any existing toolbar called Data Popup
  On Error Resume Next
  CommandBars("Data Popup").Delete
  On Error GoTo 0

  'Add new popup commandbar
  With CommandBars.Add(Name:="Data Popup", Position:=msoBarPopup)

    'Add controls
    With .Controls.Add(Type:=msoControlButton)
      .OnAction = "ShowDataForm"
      .FaceId = 264
      .Caption = "Data Form"
      .TooltipText = "Show Data Form"
    End With

    With .Controls.Add(Type:=msoControlButton)
      .Caption = "Sort Ascending"
      .FaceId = 210
      .OnAction = "SortList"
      .Parameter = "Asc"
    End With

    With .Controls.Add(Type:=msoControlButton)
      .Caption = "Sort Descending"
      .FaceId = 211
      .OnAction = "SortList"
      .Parameter = "Dsc"
    End With

  End With
End Sub

```

The code is similar to the code that created the custom menu and toolbar in previous examples. The difference is that, when the popup is created by the `Add` method of the `CommandBars` collection, the `Position` parameter is set to `msoBarPopup`. The `Name` property here is set to `Data Popup`.

You can display the popup with the following `BeforeRightClick` event procedure in the code module behind the worksheet that displays the Database range:

```

Private Sub Worksheet_BeforeRightClick(ByVal Target As Range, _
                                       Cancel As Boolean)
  If Not Intersect(Range("Database"), Target) Is Nothing Then
    CommandBars("Data Popup").ShowPopup
    Cancel = True
  End If
End Sub

```

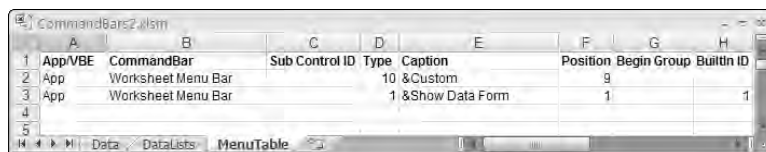
Chapter 15: Command Bars

When you right-click the worksheet, the event procedure checks to see if `Target` is within `Database`. If so, it displays `Data Popup` and cancels the right-click event. Otherwise the normal `Cell` shortcut menu appears.

Table-Driven Command Bar Creation

Very few professional Excel developers write code to add their menu items and toolbars one-by-one. Most use a table-driven approach, whereby they fill out a table with information about the items they want to add, then have a routine that generates all the items based on this table. This makes it much easier to define and modify the design of your command bars.

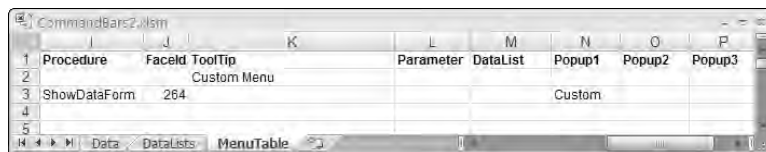
Say you want to create the `Custom` menu, which you set up earlier in this chapter, using this new method. The first thing that is needed is a table for the menu information. Insert a new worksheet, change its name to `MenuTable`, and fill out the sheet as shown in Figure 15-14 and Figure 15-15. The worksheet named `Data` contains the employee database, and `DataLists` will be used later to define a list of departments.



The screenshot shows a table in the `CommandBars2.vsm` project. The table has the following data:

	A	B	C	D	E	F	G	H
1	App/VBE	CommandBar	Sub Control ID	Type	Caption	Position	Begin Group	Builtin ID
2	App	Worksheet Menu Bar		10	&Custom	9		
3	App	Worksheet Menu Bar		1	&Show Data Form	1		1
4								
5								

Figure 15-14



The screenshot shows a table in the `MenuTable` worksheet. The table has the following data:

		K	L	M	N	O	P	
1	Procedure	Faceld	ToolTip	Parameter	DataList	Popup1	Popup2	Popup3
2		Custom Menu						
3	ShowDataForm	264			Custom			
4								
5								

Figure 15-15

The columns of the `MenuTable` are:

Column	Title	Description
A	App / VBE	Either App to add items to Excel's menus or VBE to add them to the VBE. The code to handle VBE entries is provided in Chapter 26.
B	CommandBar	The name of the top-level command bar to add the menu to. Get these names from the listings generated earlier in this chapter.
C	Sub Control ID	The ID number of a built-in popup bar to add menu to. For example, 30002 is the ID of the File popup menu.

Column	Title	Description
D	Type	The type of control to add: 1 for a normal button, 10 for a popup, and so on. These correspond to the <code>msoControl...</code> types listed in the Object Browser.
E	Caption	The text to use for the menu item.
F	Position	The position in the command bar to add the menu item. Leave this blank to add the menu to the end of the bar.
G	Begin Group	<code>True</code> or <code>False</code> to specify whether to place a separator line before the item.
H	BuiltIn ID	If you're adding a built-in menu item, this is the ID of that menu. Use 1 for all custom menu items.
I	Procedure	The name of the procedure to run when a custom menu item is clicked.
J	FaceId	The ID number of the built-in tool face to use for the menu. This can also be the name of a picture in the worksheet to use for the button face. 18 is the number for the standard New icon.
K	ToolTip	The text of the popup ToolTip to show for the button.
L	Parameter	The string to be assigned to the <code>Parameter</code> property of the button.
M	DataList	Only used with controls that have drop-down lists, such as type <code>msoControlDropDown</code> (type 3). It contains the name of a range of cells in the worksheet called DataLists that contains items to be added to the drop-down list.
N+	Popup1-n	If you add your own popup menus, this is the caption of the custom popup to add further menu items to. You can include as many levels of popup as you like by simply adding more columns—the code will detect the extra columns.

Because the `MenuTable` sheet will be referred to a number of times in code, it is a good idea to give it a meaningful “code name,” such as `wksMenuTable`. To do this, locate and select the sheet in the Project Explorer in the VBE, and change its name in the Properties window. It should now be shown as `wksMenuTable (MenuTable)` in the Project Explorer. Using the code name allows you to refer directly to that sheet as an object, so the following two lines are equivalent:

```
Debug.Print ThisWorkbook.Worksheets("MenuTable").Name
Debug.Print wksMenuTable.Name
```

The `DataLists` sheet needs to be renamed as `wksDataLists` in the same way.

Chapter 15: Command Bars

The code to create the menu from this table is shown next. The code should be copied into a new module called `modSetupBars`.

At the top of the module, a number of constants are declared, which correspond to each column of the menu table, and you will use these throughout your code. If the menu table structure changes, all you need to do is renumber these constants — you don't need to search through the code:

```
'Constants for the columns in the commandbar creation table
Const miTABLE_APP_VBE           As Integer = 1
Const miTABLE_COMMANDBAR_NAME  As Integer = 2
Const miTABLE_CONTROL_ID       As Integer = 3
Const miTABLE_CONTROL_TYPE     As Integer = 4
Const miTABLE_CONTROL_CAPTION  As Integer = 5
Const miTABLE_CONTROL_POSITION As Integer = 6
Const miTABLE_CONTROL_GROUP    As Integer = 7
Const miTABLE_CONTROL_BUILTIN  As Integer = 8
Const miTABLE_CONTROL_PROC     As Integer = 9
Const miTABLE_CONTROL_FACEID   As Integer = 10
Const miTABLE_CONTROL_TOOLTIP  As Integer = 11
Const miTABLE_CONTROL_PARAMETER As Integer = 12
Const miTABLE_CONTROL_DATA_LIST As Integer = 13
Const miTABLE_POPUP_START      As Integer = 14

'Constant to determine whether commandbars are temporary or permanent
'If you set this to False, users will not loose any additional controls
'that they add to your custom commandbars
Const mbTEMPORARY              As Boolean = False

'The following Application ID is used to identify our menus, making it easy to
'remove them
Const psAppID As String = "TableDrivenCommandBars"
```

The `mbTEMPORARY` constant allows you to make the menu changes temporary or permanent. `psAppID` provides an identifying string that will be assigned to the `Tag` property of your added controls, which makes it easy to find and remove them.

The routine to actually set up the menus — called from the workbook's `Auto_Open` procedure or `Workbook_Open` event procedure:

```
' Subroutine: SetUpMenus
'
' Purpose:   Adds the commandbars defined in the wksMenuTable worksheet'

Sub SetUpMenus()
    Dim rngRow As Range
    Dim cbrAllBars As CommandBars
    Dim cbrBar As CommandBar
    Dim ctlButton As CommandBarControl
    Dim iBuiltInID As Integer, iPopupCol As Integer, vData As Variant
    Dim rng As Range

    On Error Resume Next 'Just ignore errors in the table definition

    'Remove all of our menus before adding them.
```

```
'This ensures we don't get any duplicated menus
RemoveMenus

'Loop through each row of our menu generation table
For Each rngRow In wksMenuTable.Cells(1).CurrentRegion.Rows
    'Ignore the header row
    If rngRow.Row > 1 Then
        'Read the row into an array of the cells' values
        vData = rngRow.Value

        Set cbrBar = Nothing
```

A single routine can be used to add menu items to both the Excel and VBE menus. The only difference is the `CommandBars` collection that is used — Excel's or the VBE's. This code does not contain all the elements necessary to add VBE menus. The additional requirements are discussed in Chapter 26:

```
'Get the collection of all command bars, either in the VBE or Excel
If vData(1, miTABLE_APP_VBE) = "VBE" Then
    Set cbrAllBars = Application.VBE.CommandBars
Else
    Set cbrAllBars = Application.CommandBars
End If

'Try to find the commandbar we want
Set cbrBar = cbrAllBars.Item(vData(1, miTABLE_COMMANDBAR_NAME))

'Did we find it - if not, we must be adding one!
If cbrBar Is Nothing Then
    Set cbrBar = cbrAllBars.Add( _
        Name:=vData(1, miTABLE_COMMANDBAR_NAME), temporary:=mbTEMPORARY)
End If
```

If you want to look for a built-in popup menu to add your control to, you can recursively search for it in the `CommandBars` collection. For example, if you want to add a menu item to the Cell shortcut under the Filter menu, you can enter the ID of the Filter menu (31402) in the Sub Control ID column of the table. Alternatively, you can enter one or more control name entries in the PopUp columns of the table. Entering Filter under PopUp1 accomplishes the same result as placing 31402 under Sub Control ID. The first method is convenient when adding controls to the built-in menus. The alternative method is necessary to add items to the menus you create yourself:

```
'If set, locate the built-in popup menu bar (by ID) to add our control to.
'e.g. Worksheet Menu Bar > Edit
If Not IsEmpty(vData(1, miTABLE_CONTROL_ID)) Then
    Set cbrBar = cbrBar.FindControl(ID:=vData(1, miTABLE_CONTROL_ID), _
        Recursive:=True).CommandBar
End If

'Loop through the PopUp name columns to navigate down the menu structure
For iPopUpCol = miTABLE_POPUP_START To UBound(vData, 2)
    'If set, navigate down the menu structure to the next popup menu
    If Not IsEmpty(vData(1, iPopUpCol)) Then
        Set cbrBar = cbrBar.Controls(vData(1, iPopUpCol)).CommandBar
    End If
Next
```

Chapter 15: Command Bars

If you are adding an existing Excel control, you can specify its `Id` property value in the `BuiltIn ID` column. If you want the control to run your own procedure, you specify the name of the procedure in the `Procedure` column:

```
'Get the ID number if we're adding a built-in control
iBuiltInID = vData(1, miTABLE_CONTROL_BUILTIN)

'If it's empty, set it to 1, indicating a custom control
If iBuiltInID = 0 Then iBuiltInID = 1

'Now add our control to the command bar
If IsEmpty(vData(1, miTABLE_CONTROL_POSITION)) Or _
    vData(1, miTABLE_CONTROL_POSITION) > cbrBar.Controls.Count Then
    Set ctlButton = cbrBar.Controls.Add(Type:=vData(1,
miTABLE_CONTROL_TYPE), _
                                        ID:=iBuiltInID,
temporary:=mbTEMPORARY)
    Else
        Set ctlButton = cbrBar.Controls.Add(Type:=vData(1,
miTABLE_CONTROL_TYPE), _
                                        ID:=iBuiltInID,
temporary:=mbTEMPORARY, _
                                        Before:=vData(1,
miTABLE_CONTROL_POSITION))
    End If

'Set the rest of button's properties
With ctlButton
    .Caption = vData(1, miTABLE_CONTROL_CAPTION)
    .BeginGroup = vData(1, miTABLE_CONTROL_GROUP)
    .TooltipText = vData(1, miTABLE_CONTROL_TOOLTIP)
```

You can either use one of the standard Office tool faces, by supplying the numeric `FaceId`, or provide your own picture to use. To use your own picture, just give the name of the `Picture` object in the `FaceId` column of the menu table:

```
'The FaceID can be empty for a blank button, the number of a standard
'button face, or the name of a picture object on the sheet, which
'contains the picture to use.
If Not IsEmpty(vData(1, miTABLE_CONTROL_FACEID)) Then
    If IsNumeric(vData(1, miTABLE_CONTROL_FACEID)) Then
        'A numeric face ID, so use it
        .FaceId = vData(1, miTABLE_CONTROL_FACEID)
    Else
        'A textual face ID, so copy the picture to the button
        wksMenuTable.Shapes(vData(1, miTABLE_CONTROL_FACEID)).CopyPicture
        .PasteFace
    End If
End If
```

It is a good idea to set a property for all your menu items that identifies them as yours. If you use the `Tag` property to do this, you can use the `FindControl` method of the `CommandBars` object to locate all of your menu items, without having to remember exactly where you added them. This is done in the `RemoveMenus` procedure later in the module:

```

'Set the button's tag to identify it as one we created.
'This way, we can still find it if the user moves or renames it
.Tag = psAppID

'Set the control's OnAction property.
'Surround the workbook name with quote marks,
'in case the name includes spaces
If Not IsEmpty(vData(1, miTABLE_CONTROL_PROC)) Then
    .OnAction = "" & ThisWorkbook.Name & "!" & vData(1, _
        miTABLE_CONTROL_PROC)
End If

```

If your procedure expects to find information in the control's `Parameter` property, you enter that information under the `Parameter` column of the table:

```

'Assign Parameter property value, if specified
If Not IsEmpty(vData(1, miTABLE_CONTROL_PARAMETER)) Then
    .Parameter = vData(1, miTABLE_CONTROL_PARAMETER)
End If

```

For a drop-down control or combo box, you enter a list of values in the `DataLists` worksheet and assign a name to the list. You enter the name in the `DataList` column of the table:

```

'Assign data list to ComboBox
If Not IsEmpty(vData(1, miTABLE_CONTROL_DATA_LIST)) Then
    For Each rng In wksDataLists.Range(vData(1, _
        miTABLE_CONTROL_DATA_LIST))
        .AddItem rng.Value
    Next rng
End If
End With
End If
Next rngRow
End Sub

```

When the application workbook is closed, you need to run some code to remove your menus. Some developers just use `CommandBars.Reset`, but this removes all other customizations from the command bars as well as their own. It is much better to locate all the menu items and command bars that were created for your application and delete them. This takes two routines. The first removes all the menus from a specific `CommandBars` collection, by searching by its `Tag` value:

```

Private Sub RemoveMenusFromBars(cbrBars As CommandBars)
    Dim ctl As CommandBarControl

    'Ignore errors while deleting our menu items
    On Error Resume Next

    'Using the application or VBE CommandBars ...
    With cbrBars
        'Find a CommandBarControl with our tag
        Set ctl = .FindControl(Tag:=psAppID)

        'Loop until we didn't find one
    End With
End Sub

```

Chapter 15: Command Bars

```
    Do Until ctl Is Nothing
        'Delete the one we found
        ctl.Delete

        'Find the next one
        Set ctl = .FindControl(Tag:=psAppID)
    Loop
End With
End Sub
```

The second removal routine calls the first to remove the menu items from the Excel command bars and the VBE command bars, and removes any custom bars that might have been created, as long as the user has not added his or her own controls to them:

```
Sub RemoveMenus()
    Dim cbrBar As CommandBar, rngRow As Range, sBarName As String

    'Ignore errors while deleting our menu items and commandbars
    On Error Resume Next

    'Delete our menu items from the Excel and VBE commandbars
    RemoveMenusFromBars Application.CommandBars
    RemoveMenusFromBars Application.VBE.CommandBars

    'Loop through each row of our menu generation table
    For Each rngRow In wksMenuTable.Cells(1).CurrentRegion.Rows
        'Ignore the header row
        If rngRow.Row > 1 Then
            sBarName = rngRow.Cells(1, miTABLE_COMMANDBAR_NAME)

            Set cbrBar = Nothing
            'Find the command bar, either in the VBE or Excel
            If rngRow.Cells(1, miTABLE_APP_VBE) = "VBE" Then
                Set cbrBar = Application.VBE.CommandBars(sBarName)
            Else
                Set cbrBar = Application.CommandBars(sBarName)
            End If
            'If we found it, delete it if it is not a built-in bar
            If Not cbrBar Is Nothing Then
                If Not cbrBar.BuiltIn Then
                    'Only delete blank command bars - in case user
                    'or other applications added menu items to the
                    'same custom bar
                    If cbrBar.Controls.Count = 0 Then cbrBar.Delete
                End If
            End If
        End If
    Next
End Sub
```


You should run the `SetUpMenus` procedure from the `Auto_Open` procedure or the `Workbook_Open` event procedure, and run the `RemoveMenus` procedure from the `Auto_Close` procedure or the `Workbook_BeforeClose` event procedure.

You now have a complete template, which can be used as the basis for any Excel application (or just in a normal workbook where you want to modify the menu structure).

See Chapter 26 for the extra code needed to create a template that can be used with Excel and the VBE.

The first table entry shown in Figure 15-14 and Figure 15-15 adds a new popup menu to the Worksheet menu bar called Custom. The second entry adds a menu item called Show Data Form to the Custom menu, as shown in Figure 15-16.

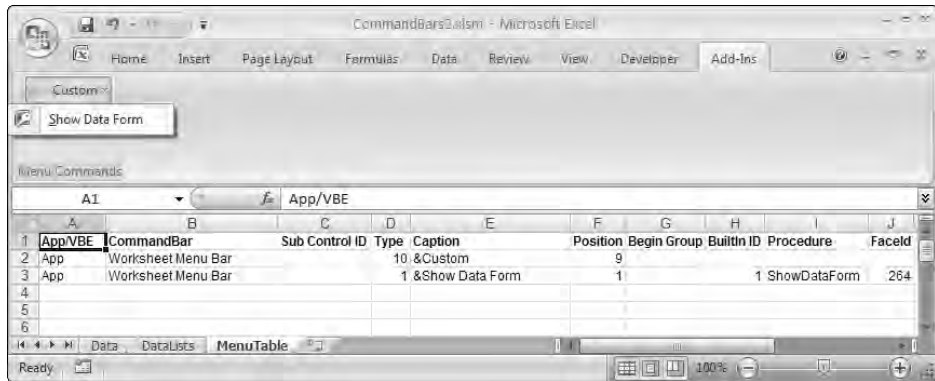


Figure 15-16

You can expand the table to add more items to the custom menu and create new command bars and controls, as shown in Figure 15-17 and Figure 15-18.

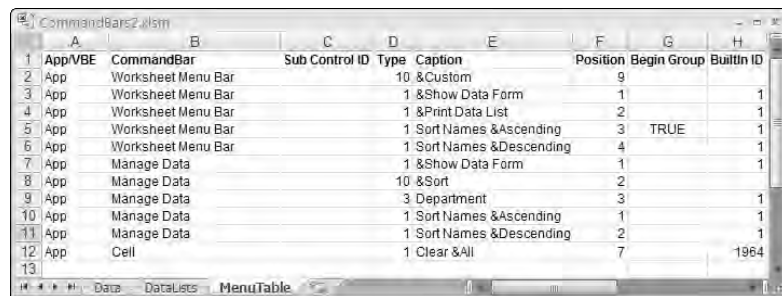


Figure 15-17

Procedure	FacelId	ToolTip	Parameter	DataList	Popup1	Popup2	Popup3
		Custom Menu					
ShowDataForm	264				Custom		
PrintDataList	4				Custom		
SortList	210		Asc		Custom		
SortList	211		Dsc		Custom		
ShowDataForm	264	Show Data Form					
		Sort Data					
FilterDepartment		Select Department		Departments			
SortList	210	Sort Names in Ascending Order	Asc		Sort		
SortList	211	Sort Names in Descending Order	Dsc		Sort		
		Clear Data and Formatting					

Figure 15-18

The following procedures will automate the running of the code as the workbook is opened and closed:

```
' Subroutine: Auto_Open
' Purpose:  Adds our menus and menuitems to the application
Sub Auto_Open()
    SetUpMenus
    CommandBars("Manage Data").Visible = True
End Sub

' Subroutine: Auto_Close
' Purpose:  Removes our menus and menu items from the application
Sub Auto_Close()
    RemoveMenus
End Sub
```

The data in rows 2 through 6 of the MenuTable table create the Custom menu shown in Figure 15-19, which is identical to the Custom menu created earlier in this chapter, apart from some added icons.

Rows 7 through 11 create a Manage Data toolbar identical to the one created earlier. The data required for the drop-down list of departments is in the DataLists worksheet, as shown here. The highlighted range has been given the name Departments, as shown in Figure 15-20.

Row 12 of the table creates a Clear All entry in the popup menu that appears when you right-click a worksheet cell.

Although the code presented here allows you to add items to existing shortcut menus, it is not capable of creating a new popup shortcut menu. However, you could easily add an extra column that allows you to specify this, as long as you adapt the code accordingly. The technique is flexible enough to accommodate whatever options you need.

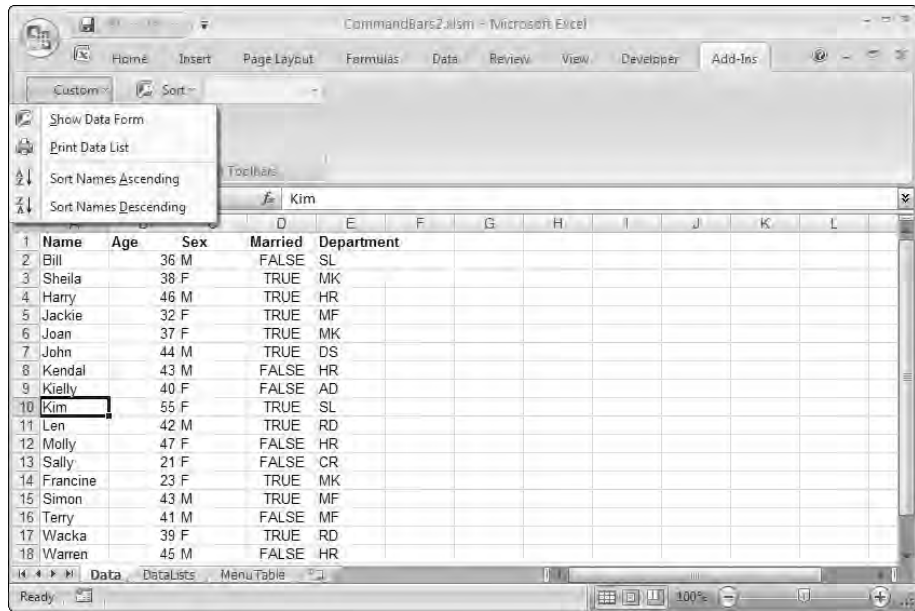


Figure 15-19

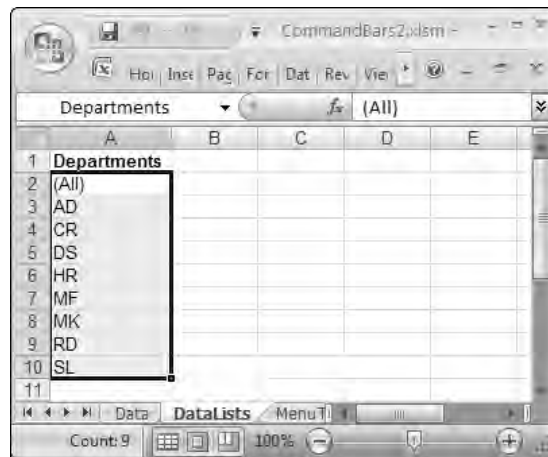


Figure 15-20

Summary

In this chapter, you have seen how the Excel command bars are structured and learned how to create:

- ❑ Lists of the built-in control images with their `Id` and `FaceId` properties
- ❑ An entire list of the `FaceIds` available
- ❑ A complete list of popup menu items

You have also seen how to create your own command bars and how to add controls to your command bars. The differences between the three types of command bars — that is, toolbars, menu bars, and popup menus — have been described, and methods of creating them programmatically have been presented.

Finally, you have seen how you can create a table to define the changes you want to make to a command bar structure while your application is open. This approach simplifies the task of customizing menus and makes it very easy to make changes.

Class Modules

Class modules are used in VBA to create your own customized objects. Most VBA users will never have to create their own objects because Excel already provides all of the objects they need. However, there are occasions when class modules can be very useful. You can use them to:

- ❑ Respond to application events; you can write code that is executed whenever any open workbook is saved or printed, for example
- ❑ Respond to embedded chart events
- ❑ Set up a single event procedure that can be used by a number of ActiveX controls, such as text boxes in a UserForm
- ❑ Encapsulate Windows API code so it is easy to use
- ❑ Encapsulate standard VBA procedures in a form that is easy to transport into other workbooks

In this chapter, you create some simple (if not terribly useful) objects, to get the idea of how class modules work. Then you apply the principles to some more useful examples. You are already familiar with Excel's built-in objects, such as the `Worksheet` object, and you know that objects often belong to collections such as the `Worksheets` collection. You also know that objects have properties and methods, such as the `Name` property and the `Copy` method of the `Worksheet` object.

Using a class module, you can create your own "blueprint" for a new object, such as an `Employee` object. You can define properties and methods for the object, such as a `Rate` property that records the employee's current rate of pay, and a `Training` method that consumes resources and increases the employee's skills. You can also create a new collection for the object, such as the `Employees` collection. The class module is a plan for the objects you want to create. From it you can create instances of your object. For example, Mary, Jack, and Anne could be instances of an `Employee` object, all belonging to the `Employees` collection.

You might not be aware of it, but you have been using some class modules already. The modules behind worksheets, charts, workbooks, and UserForms are class modules. However, they are special types of class modules that behave a little differently from those you create yourself. They are designed specifically to support the object with which they are associated, they give you access to the event procedures for that object, and they cannot be deleted without deleting the associated object.

Creating Your Own Objects

Proceed with creating the `Employee` object just discussed. You want to store the employee's name, hours worked per week, and rate of pay. From this information, you want to calculate the employee's weekly pay. You can create an `Employee` object with three properties to hold the required data and a method that calculates the weekly pay.

To do this, you create a class module named `CEmployee`, as shown in the top right of Figure 16-1.

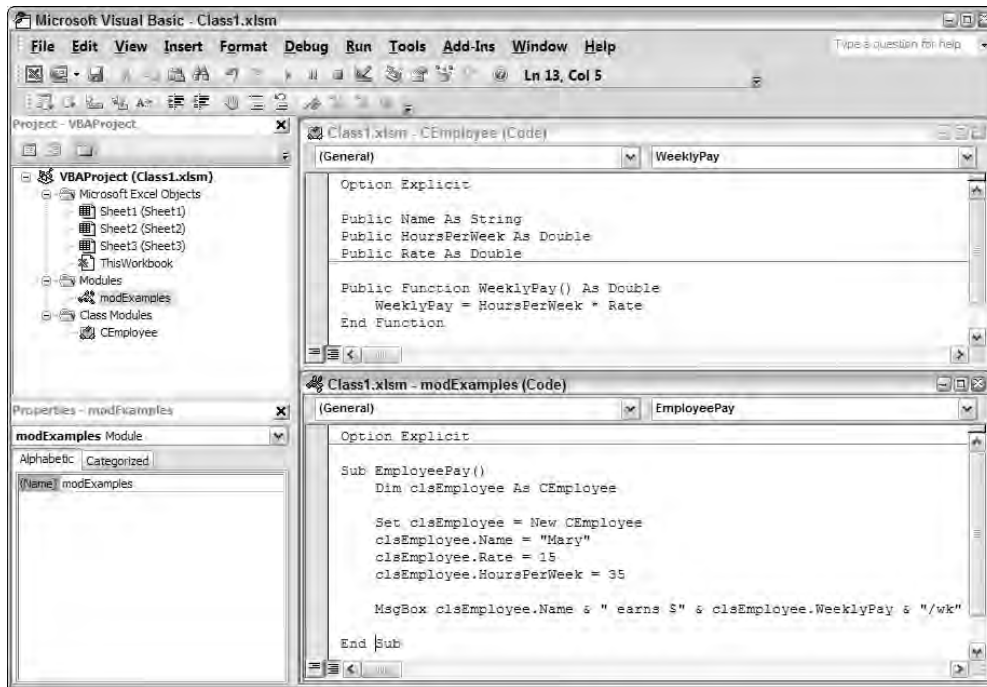


Figure 16-1

The class module declares three public variables — `Name`, `HoursPerWeek`, and `Rate` — which are the properties of the `Employee` object. There is also one public function, `WeeklyPay`. Recall that any public function or sub procedure in the class module behaves as a method of the object. A function is a method that can generate a return value. A sub is a method that does not return a value.

The code in the standard module `modExamples` (at the bottom right of Figure 16-1) generates an `Employee` object from the `CEmployee` blueprint. The module declares `clsEmployee` as a `CEmployee` type. The `EmployeePay` sub procedure uses the `Set` statement to assign a new instance of `CEmployee` to `clsEmployee`; that is, `Set` creates the new object.

The sub then assigns values to the three properties of the object, before generating the message that appears in the message box shown in Figure 16-2. To form the message, it accesses the `Name` property of the `Employee` object and executes the `WeeklyPay` method of the `Employee` object.



Figure 16-2

An alternative way of setting up the standard code module, when you only need to create a single instance of the object variable, is as follows:

```
Dim Employee As New CEmployee

Sub EmployeePay()
    Employee.Name = "Mary"
    Employee.Rate = 15
    Employee.HoursPerWeek = 35
    MsgBox Employee.Name & " earns $" & Employee.WeeklyPay & "/wk"
End Sub
```

Here, the keyword `New` is used on the declaration line. In this case, the `Employee` object is automatically created when it is first referenced in the code.

Property Procedures

If your properties are defined by public variables, they are read/write properties. They can be directly accessed and can be directly assigned new values, as you have seen in the previous section. If you want to perform checks or calculations on properties, you use the `Property Let` and `Property Get` procedures to define the properties in your class module, instead of using public variables.

`Property Get` procedures allow the class module to control the way in which properties are accessed. `Property Let` procedures allow the class module to control the way in which properties can be assigned values. You can also use `Property Set` procedures. They are similar to `Property Let` procedures, but they process objects instead of values.

For example, say you want to break up the employee hours into normal time and overtime, where the value of overtime is anything over 35 hours. You want to have an `HoursPerWeek` property, which includes both normal and overtime hours that can be read and can be assigned new values. You want the class module to split the hours into normal and overtime, and set up two properties, `NormalHours` and `OverTimeHours`, that can be read but cannot be directly assigned new values. You can set up the following code in the `CEmployee` class module:

```
Public Name As String
Public Rate As Double

Private dNormalHrs As Double
Private dOverTimeHrs As Double

'Return weekly pay
```

```
Public Function WeeklyPay() As Double
    WeeklyPay = dNormalHrs * Rate + dOverTimeHrs * Rate * 1.5
End Function

'Convert input hours to normal and overtime
Property Let HoursPerWeek(dHours As Double)
    dNormalHrs = WorksheetFunction.Min(35, dHours)
    dOverTimeHrs = WorksheetFunction.Max(0, dHours - 35)
End Property

'Return total hours per week
Property Get HoursPerWeek() As Double
    HoursPerWeek = dNormalHrs + dOverTimeHrs
End Property

'Return normal hours
Property Get NormalHours() As Double
    NormalHours = dNormalHrs
End Property

'Return overtime hours
Property Get OverTimeHours() As Double
    OverTimeHours = dOverTimeHrs
End Property
```

HoursPerWeek is no longer declared as a variable in the declarations section. Instead, two new private variables have been added — dNormalHrs and dOverTimeHrs. HoursPerWeek is now defined by a Property Let procedure, which processes the input when you assign a value to the HoursPerWeek property. It breaks the hours into normal time and overtime. The Property Get procedure for HoursPerWeek returns the sum of normal and overtime hours when you access the property value.

NormalHours and OverTimeHours are defined only by Property Get procedures that return the values in the Private variables, dNormalHrs and dOverTimeHrs, respectively. This makes the properties NormalHours and OverTimeHours read-only. There is no way they can be assigned values except through the HoursPerWeek property.

The WeeklyPay function has been updated to calculate pay as normal hours at the standard rate and overtime hours at 1.5 times the standard rate. You can change the standard module code as follows to generate the message shown in Figure 16-3:

```
Sub EmployeePay()
    Dim clsEmployee As CEmployee

    'Create instance of CEmployee
    Set clsEmployee = New CEmployee

    'Define properties
    clsEmployee.Name = "Mary"
    clsEmployee.Rate = 15
    clsEmployee.HoursPerWeek = 45

    'Display properties
```



```

MsgBox clsEmployee.Name & " earns $" _
      & clsEmployee.WeeklyPay & "/wk" _
      & " including " & clsEmployee.OvertimeHours _
      & " hrs overtime"

```

```
End Sub
```

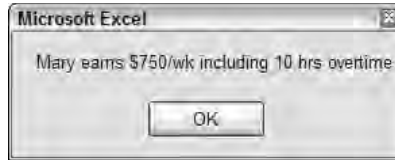


Figure 16-3

Creating Collections

Now that you have an `Employee` object, you will want to have many `Employee` objects, and what better way is there to organize them but in a collection? VBA has a `Collection` object that you can use as follows, in a standard module:

```

'Collection to hold Employee objects
Dim mcolEmployees As New Collection

Sub AddEmployees()
    Dim clsEmployee As CEmployee
    Dim lCount As Long

    'Ensure collection is empty
    For lCount = 1 To mcolEmployees.Count
        mcolEmployees.Remove 1
    Next lCount

    'Define Employee
    Set clsEmployee = New CEmployee
    clsEmployee.Name = "Mary"
    clsEmployee.Rate = 15
    clsEmployee.HoursPerWeek = 45

    'Add Employee to collection
    mcolEmployees.Add clsEmployee, clsEmployee.Name

    'Define Employee
    Set clsEmployee = New CEmployee
    clsEmployee.Name = "Jack"
    clsEmployee.Rate = 14
    clsEmployee.HoursPerWeek = 35

    'Add Employee to collection

```

```
mcolEmployees.Add clsEmployee, clsEmployee.Name

'Display data from collection
MsgBox "Number of Employees = " & mcolEmployees.Count
MsgBox "mcolEmployees(2).Name = " & mcolEmployees(2).Name
MsgBox "mcolEmployees("Jack").Rate = " & mcolEmployees("Jack").Rate

'Process all Employees
For Each clsEmployee In mcolEmployees
    MsgBox clsEmployee.Name & " earns $" & clsEmployee.WeeklyPay
Next clsEmployee

End Sub
```

At the top of the standard module, you declare `mcolEmployees` to be a new collection. The `AddEmployees` procedure uses the `Remove` method of the collection in the `For . . . Next` loop to remove any existing objects. It keeps removing the first object in the collection, because as soon as you remove it, the second object automatically becomes the first object, and so on—hence the `.Remove 1` statement. This step is normally not necessary, because the collection is initialized empty. It is only here to demonstrate the `Remove` method, and also to allow you to run the procedure more than once without doubling up the items in the collection.

`AddEmployees` creates the first employee, `Mary`, and uses the `Add` method of the collection to place the `Mary` object in the collection. The first parameter of the `Add` method is a reference to the object itself. The second parameter, which is optional, is an identifying key that can be used to reference the object later. In this case you have used the `Employee` object's `Name` property as the key. The same procedure is used with `Jack`.

If you supply a key value for each member of the collection, the keys must be unique. You will get a run-time error when you attempt to add a new member to the collection with a key value that is already in use. Using a person's name as the key is not recommended, because different people can have the same name. Use a unique identifier, such as a Social Security number.

The `MsgBox` statements illustrate that you can reference the collection in the same ways as you can reference Excel's built-in collections. For instance, the `Employees` collection has a `Count` property. You can reference a member of the collection by position or by key, if you have entered a key value.

Class Module Collection

You can also set up your collection in a class module. There are advantages and disadvantages to doing this. The advantages are that you get much more control over interaction with the collection, you can prevent direct access to the collection, and the code is encapsulated into a single module that makes it more transportable and easier to maintain. The disadvantages are that it takes more work to set up the collection, and you lose some of the shortcut ways to reference members of the collection and the collection itself.

The following shows the contents of a class module CEmployees:

```
'Collection to hold Employee instances
Private mcolEmployees As New Collection

'Method to add employees to collection
Public Function Add(clsEmployee As CEmployee)

    mcolEmployees.Add clsEmployee, clsEmployee.Name

End Function

'Return Count property
Public Property Get Count() As Long

    Count = mcolEmployees.Count

End Property

'Return collection
Public Property Get Items() As Collection

    Set Items = mcolEmployees

End Property

'Return a member of the collection
Public Property Get Item(vItem As Variant) As CEmployee

    Set Item = mcolEmployees(vItem)

End Property

'Remove a member of the collection
Public Sub Remove(vItem As Variant)

    mcolEmployees.Remove vItem

End Sub
```

When the collection is in its own class module, you can no longer directly use the collection's four methods (Add, Count, Item, and Remove) in your standard module. You need to set up your own methods and properties in the class module, even if you have no intention of modifying the collection's methods. On the other hand, you have control over what you choose to implement and what you choose to modify, as well as what you present as a method and what you present as a property.

In CEmployees, Function Add, Sub Remove, Property Get Item, and Property Get Count pass on most of the functionality of the collection's methods. There is one new feature in the Property Get Items procedure. Whereas Property Get Item passes back a reference to a single member of the collection, Property Get Items passes back a reference to the entire collection. This is to provide the capability to use the collection in a For Each...Next loop.

Chapter 16: Class Modules

The standard module code is now as follows:

```
Option Explicit
Sub AddEmployees()
    Dim clsEmployees As CEmployees
    Dim clsEmployee As CEmployee
    Dim lCount As Long
    Dim vNames As Variant
    Dim vRates As Variant
    Dim vHours As Variant
    Dim sText As String

    'Input data
    vNames = Array("Mary", "Jack", "Anne", "Harry")
    vRates = Array(15, 14, 20, 17)
    vHours = Array(45, 35, 40, 40)

    'Initialize collection
    Set clsEmployees = New CEmployees

    'Define and add employees to collection
    For lCount = LBound(vNames) To UBound(vNames)
        Set clsEmployee = New CEmployee
        clsEmployee.Name = vNames(lCount)
        clsEmployee.Rate = vRates(lCount)
        clsEmployee.HoursPerWeek = vHours(lCount)
        clsEmployees.Add clsEmployee
        Set clsEmployee = Nothing
    Next lCount

    'Display data from collection
    MsgBox "Number of Employees = " & clsEmployees.Count
    MsgBox "Employees.Item(2).Name = " & clsEmployees.Item(2).Name
    MsgBox "Employees.Item(""Jack").Rate = " & clsEmployees.Item("Jack").Rate

    For Each clsEmployee In clsEmployees.Items
        sText = sText & clsEmployee.Name & " earns $" & _
            clsEmployee.WeeklyPay & vbCrLf
    Next clsEmployee

    MsgBox sText

End Sub
```

`clsEmployees` is declared to be of type `CEmployees`. The code that follows defines three arrays as a convenient way to make it clear which data is being used. After initializing the `Employees` collection, you create the `Employee` instances and add them to the collection. As one small convenience, you no longer need to specify the key value when using the `Add` method of the `Employees` collection. The `Add` method code in `clsEmployees` does this for you.

The second, third, and fourth `MsgBox` statements show the new properties needed to reference the collection and its members. You need to use the `Item` property to reference a member and the `Items` property to reference the whole collection.

Encapsulation

Class modules allow you to encapsulate code and data in such a way that it becomes very easy to use, very easy to share, and much easier to maintain.

You hide the code that does the work from the user, who only needs to know what sort of object the class module represents, and what properties and methods are associated with the object. This is particularly useful when it is necessary to make calls to the Windows API (application programming interface) to perform tasks that are not possible in normal VBA. This topic is presented in Chapter 27, where you can see examples that encapsulate very complex code and create very usable objects.

Class modules provide a mechanism for encapsulating code that you can use in other workbooks or share with other programmers to reduce development time. You can easily copy a class module to another workbook. In the Project Explorer window, it is as straightforward as dragging the class module between the projects.

You can also export the code in the class module to a file by right-clicking the module in the Project Explorer and choosing `Export File` to create a text file that can be copied to another PC. The file can then be imported into another workbook by right-clicking its project in the Project Explorer and choosing `Import File`.

So far, this chapter has examined class modules from a general programming perspective. You will now see how to use class modules to gain more control over Excel.

Trapping Application Events

You can use a class module to trap application events. Most of these events are the same as the workbook events, but they apply to all open workbooks, not just the particular workbook that contains the event procedures. For example, in a workbook there is a `BeforePrint` event that is triggered when you start to print anything in that workbook. At the application level, there is a `WorkbookBeforePrint` event that is triggered when any open workbook starts to print.

To see what application events are available, you first insert a class module into your project. The class module can have any valid module name. The one shown in Figure 16-4 has been named `CAppEvents`. You then type in the following variable declaration at the top of the module:

```
Public WithEvents xlApp As Application
```

The object variable name, `xlApp`, can be any valid variable name, as long as you use it consistently in code that refers to the class module, as a property of the class. The `WithEvents` keyword causes the events associated with the application object to be exposed. You can now choose `xlApp` from the drop-down at the top left of the module and then use the drop-down at the top right to see the event list, as shown in Figure 16-4.

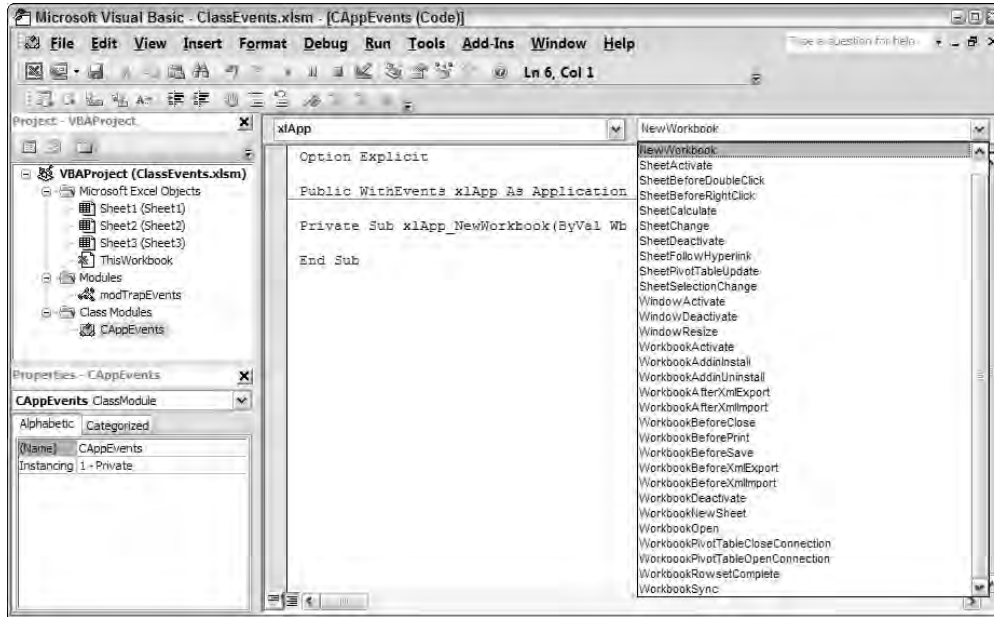


Figure 16-4

Choose the `WorkbookBeforePrint` event and extend the event procedure presented in Chapter 9, using the following code in `CAppEvents`:

```
Private Sub xlApp_WorkbookBeforePrint(ByVal Wbk As Workbook, _
                                     Cancel As Boolean)

    Dim wks As Worksheet
    Dim sFullFileName As String
    Dim sCompanyName As String

    With Wbk

        'Define footer data
        sCompanyName = "Execuplan Consulting"
        sFullFileName = .FullName

        'Process each worksheet
        For Each wks In .Worksheets
            With wks.PageSetup
                .LeftFooter = sCompanyName
                .CenterFooter = ""
                .RightFooter = sFullFileName
            End With
        Next wks

    End With

End Sub
```

Unlike sheet and workbook class modules, the event procedures you place in your own class modules do not automatically function. You need to create an instance of your class module and assign the `Application` object to the `xlApp` property of the new object. The following code must be set up in a standard module:

```
Public xlApplication As CAppEvents

Sub TrapApplicationEvents()
    'Create instance of class module
    Set xlApplication = New CAppEvents

    'Assign the Excel Application object to the xlApp property
    Set xlApplication.xlApp = Application
End Sub
```

All you need to do now is execute the `TrapApplicationEvents` procedure. The `WorkbookBeforePrint` event procedure will then run when you use any `Print` or `Preview` commands, until you close the workbook containing the event procedure.

It is possible to terminate application event trapping during the current session. Any action that resets module-level variables and public variables will terminate application event processing, because the class module instance will be destroyed. Actions that can cause this include editing code in the VBE and executing the `End` statement in VBA code.

If you want to enable application event processing for all Excel sessions, you can place your class module and standard module code in `Personal.xlsb` and execute `TrapApplicationEvents` in the `Workbook_Open` event procedure. You can even transfer the code in `TrapApplicationEvents` to the `Workbook_Open` event procedure. However, you must keep the `Public` declaration of `xlApplication` in a standard module.

To illustrate, you can place the following code in the declarations section of a standard module:

```
Public xlApplication As CAppEvents
```

You can place the following event procedure in the `ThisWorkbook` module:

```
Private Sub Workbook_Open()
    Set xlApplication = New CAppEvents
    Set xlApplication.xlApp = Application
End Sub
```

Embedded Chart Events

If you want to trap events for a chart embedded in a worksheet, you use a process similar to the process for trapping application events. First, insert a new class module into your project, or you could use the same class module that you used for the application events. You place the following declaration at the top of the class module:

```
Public WithEvents cht As Chart
```

Chapter 16: Class Modules

Set up the same `BeforeDoubleClick` event procedure used in Chapter 10. The class module should be as follows:

```
Public WithEvents cht As Chart

Private Sub cht_BeforeDoubleClick(ByVal ElementID As Long, _
    ByVal Arg1 As Long, ByVal Arg2 As Long, Cancel As Boolean)
    Dim se As Series

    'Determine what part of chart was double clicked
    Select Case ElementID

        'If legend, remove it
        Case xlLegend
            ActiveChart.HasLegend = False
            Cancel = True

        'If chart area, display legend
        Case xlChartArea
            ActiveChart.HasLegend = True
            Cancel = True

        'If series, determine which series
        Case xlSeries
            'Arg1 is the Series index
            'Arg2 is the point index (-1 if the entire series is selected)
            Set se = ActiveChart.SeriesCollection(Arg1)
            If Arg2 = -1 Then

                'Whole series selected
                With se.Border
                    If .ColorIndex = xlColorIndexAutomatic Then
                        .ColorIndex = 1
                    Else
                        .ColorIndex = (.ColorIndex Mod 56) + 1
                    End If
                End With

            Else

                'Data point selected
                With se.Points(Arg2)
                    .HasDataLabel = Not .HasDataLabel
                End With

            End If

            'Cancel double click
            Cancel = True

        End Select

    End Sub
```


This code allows you to double-click the chart legend to make it disappear, or double-click in the chart area to make it reappear. If you double-click a series line, it changes color. If you select a point in a series by clicking it, and then double-click it, it will toggle the data label on and off for that point.

Say your chart is contained in a `ChartObject` that is the only `ChartObject` in a worksheet called `Mangoes`, as shown in Figure 16-5, and you have named your class module `CChartEvents`. In your standard module, you enter the following:

```
Public chtMangoes As CChartEvents

Sub InitializeChartEvents()

    'Create instance of CChartEvents
    Set chtMangoes = New CChartEvents

    'Assign reference to chart to cht property
    Set chtMangoes.cht = ThisWorkbook.Worksheets("Mangoes").ChartObjects(1).Chart

End Sub
```

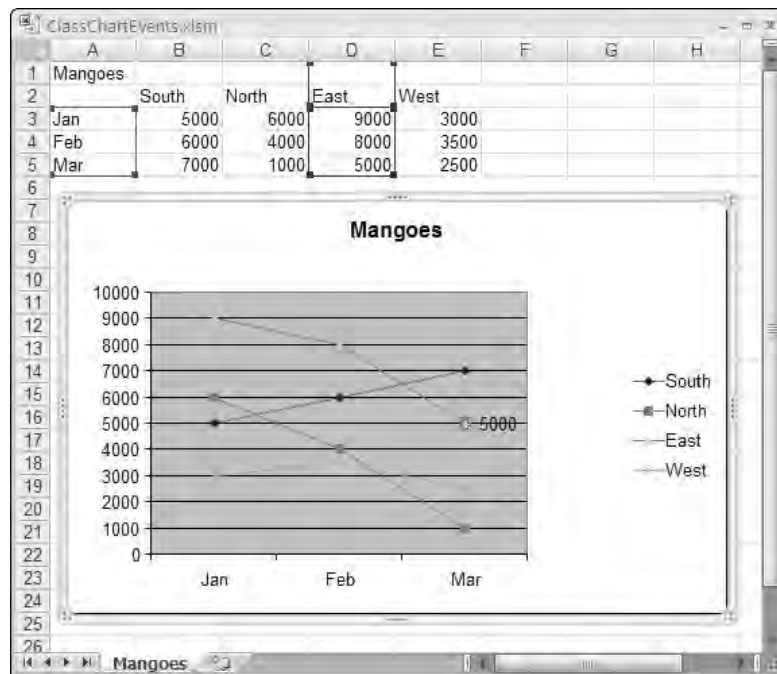


Figure 16-5

After executing `InitializeChartEvents`, you can double-click the series, points, and legend to run the `BeforeDoubleClick` event procedure.

A Collection of UserForm Controls

When you have a number of the same type of control on a form, you often write almost identical event procedures for each one. For example, say you want to be able to double-click the label to the left of each of the TextBox in the UserForm in Figure 16-6 to clear the TextBox and set the focus to the TextBox. You would normally write four almost identical event procedures, one for each label control.

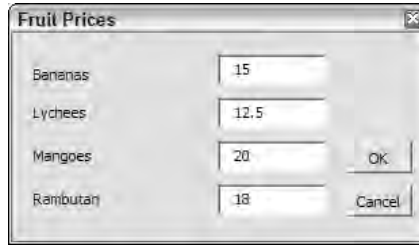


Figure 16-6

Using a class module, you can write a single generic event procedure to apply to all the label controls, or just those that need the procedure. The label controls and TextBox in the UserForm have been given corresponding names, as follows:

Label	TextBox
lblBananas	txtBananas
lblLychees	txtLychees
lblMangoes	txtMangoes
lblRambutan	txtRambutan

The following code is entered in a class module CControlEvents:

```
Public WithEvents lbl As MSForms.Label
Public frm As UserForm

Private Sub lbl_DblClick(ByVal Cancel As MSForms.ReturnBoolean)
    Dim sProduct As String
    Dim sTextBoxName As String

    'Get product name from label caption
    sProduct = lbl.Caption

    'Construct name of associated text box
    sTextBoxName = "txt" & sProduct

    'Assign zero length string and set focus
    With frm.Controls(sTextBoxName)
        .Text = ""
    End With
End Sub
```

```

        .SetFocus
    End With

End Sub

```

`lbl` is declared with events as a `UserForm` label. `frm` is declared to be a `UserForm`. The generic `Db1Click` event procedure for `lbl` gets the product name from the `Caption` property of the label, and converts this to the `TextBox` name by appending "txt" in front of the product name.

The `With...End With` structure identifies the `TextBox` object by using the `TextBox` name as an index into the `Controls` collection of the `UserForm`. It sets the `Text` property of the `TextBox` to a zero-length string and uses the `SetFocus` method to place the cursor in the `TextBox`.

The following code is entered into the class module behind the `UserForm`:

```

Dim mcolLabels As New Collection

Private Sub UserForm_Initialize()
    Dim ctl As MSForms.Control
    Dim clsEvents As CControlEvents

    'Loop through all controls on userform
    For Each ctl In Me.Controls

        'Only process labels
        If TypeOf ctl Is MSForms.Label Then
            'Instantiate class module and assign properties
            Set clsEvents = New CControlEvents
            Set clsEvents.lbl = ctl
            Set clsEvents.frm = Me

            'Add instance to collection
            mcolLabels.Add clsEvents

        End If

    Next ctl

End Sub

```

`mcolLabels` is declared as a new collection to hold the objects that will be created from the `CControlEvents` class module. In the `UserForm Initialize` event procedure, the label controls are associated with instances of `CControlEvents`.

The `For Each...Next` loop processes all the controls on the form. When it identifies a control that is a label, using the `TypeOf` keyword to identify the control type, it creates a new instance of `CControlEvents` and assigns it to `clsEvents`. The `lbl` property of the new object is assigned a reference to the control, and the `frm` property is assigned a reference to the `UserForm`. The new object is then added to the `mcolLabels` collection.

When the `UserForm` is loaded into memory, the `Initialize` event runs and connects the label controls to instances of the class module event procedure. Double-clicking any label clears the `TextBox` to the right and sets the focus to that `TextBox`, ready for new data to be typed in.

You need to be aware that some events associated with some controls are not made available in a class module using `With Events`. For example, the most useful events exposed by UserForm text boxes are `BeforeUpdate`, `AfterUpdate`, `Enter`, and `Exit`. None of these is available in a class module. You can only handle these events in the class module associated with the UserForm.

Referencing Classes Across Projects

When you want to run macros in another workbook, you can use `Tools` ⇔ `References` in the VBE window to create a reference to the other workbook's VBA project. The reference appears as a special entry in the Project Explorer, as shown in Figure 16-7.

`ClassReferences.xlsm` has a reference to `ClassUserFormControls.xlsm`, which contains the UserForm from the previous example. The reference allows you to run procedures in standard modules in `ClassUserFormControls.xlsm` from standard modules in `ClassReferences.xlsm`. However, the reference does not allow you to create instances of class modules or UserForms in the referenced workbook.

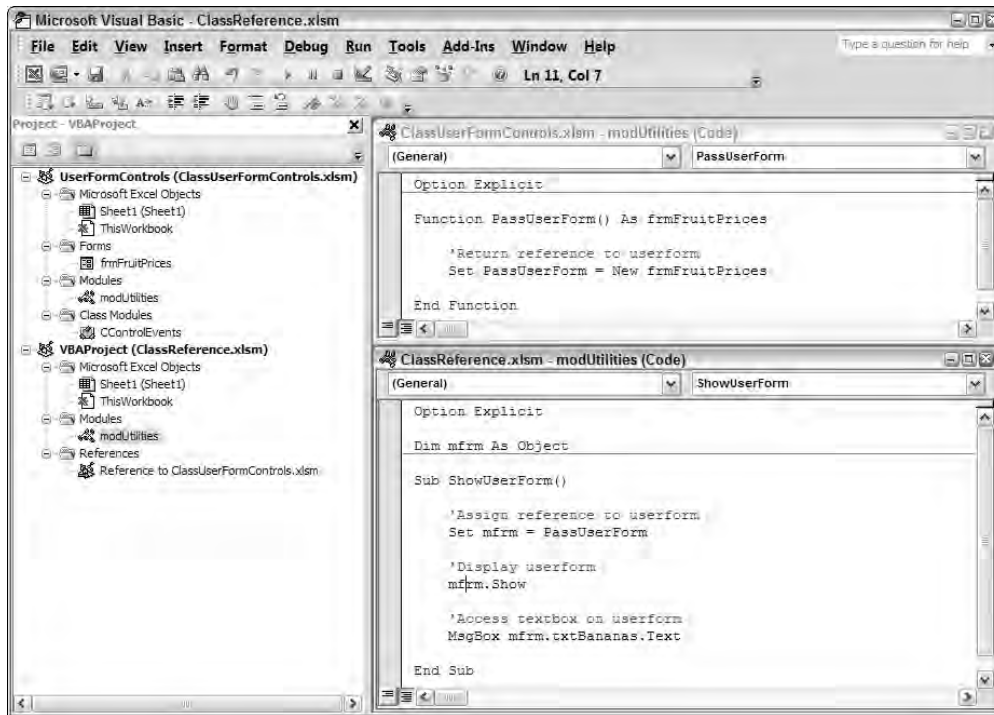


Figure 16-7

When you create a reference to another workbook, you should make sure that the VBA Project in the referenced workbook has a unique name. By default, it will be named VBA Project. Click Tools ⇨ VBA Project Properties and enter a new project name.

There is a way to get around this. You can indirectly access a UserForm in the referenced workbook if that workbook has a function that returns a reference to the UserForm. There is an example of this type of function in the top-right corner of Figure 16-7. `PassUserForm`, in `ClassUserFormControls.xlsm`, is a function that assigns a new instance of `frmFruitPrices` to its return value. In `ClassReferences.xlsm`, `mfrm` is declared as a generic `Object` type. `ShowUserform` assigns the return value of `PassUserForm` to `mfrm`. `mfrm` can then be used to show the UserForm and access its control values, as long as the UserForm is hidden, not unloaded.

Summary

Class modules are used to create blueprints for new objects, such as the `Employee` object presented in this chapter:

- ❑ Function and Sub procedures are used in the class module to create methods for the object.
- ❑ Public variables declare the properties for the object.
- ❑ However, if you need to take programmatic control when a property is assigned a value, you can define the property using a `Property Let` procedure.
- ❑ In addition, `Property Get` procedures allow you to control access to property values.

To use the code in your class module, you create one or more instances of your object. For example, you can create `Mary` and `Jack` as instances of an `Employee` object. You can further customize your objects by creating your own collection, where you add all the instances of your object.

Class modules are not used to create objects to the same extent in Excel VBA as they are used in a standalone programming language such as Visual Basic. This is because Excel already contains the objects that most Excel programmers want to use. However, Excel programmers can use class modules to:

- ❑ Trap application-level events, such as the `WorkbookBeforePrint` event that allows you to control the printing of all open workbooks
- ❑ Trap events in embedded charts
- ❑ Write a single event procedure that can be used by many instances of a particular object, such as a `TextBox` control on a UserForm
- ❑ Encapsulate difficult code and make it easier to use
- ❑ Encapsulate code so you can share the code among different projects and users

See Chapter 27 for examples of encapsulation of API code.

Add-ins

If you want to make your workbook invisible to the user in the Excel window, you can turn it into an Add-in file. An Add-in can be loaded into memory using Open under the Microsoft Office button, but it generally makes more sense to access it via the Add-Ins dialog box, which is covered later in this chapter. Either way, the file does not appear in the Excel Application window, but the macros it contains can be executed from the user interface. Any user-defined functions it contains can be used in worksheet calculations. The Add-ins macros can be attached to menu commands and toolbar buttons, and the Add-in can communicate with the user through UserForms and VBA functions such as `InputBox` and `MsgBox`.

It is widely believed that an Add-in is a compiled version of a workbook. In programming, compilation involves translating the human-readable programming code into machine language. This is *not* the case with an Excel Add-in. In fact, all that happens is that the workbook is hidden from the user interface. The Add-in's worksheets and charts can no longer be seen by anyone. Its code modules can still be viewed, as normal, in the VBE window and remain complete with comments as well as code.

However, it is possible to create a compiled version of an Add-in. This is referred to as a COM (Component Object Model) Add-in. COM Add-ins are discussed separately in Chapter 18.

This chapter has taken the `CommandBars2.xlsm` file used in Chapter 15, saved it as `AddIn.xlsm` prior to converting it to `AddIn.xlam`, and adapted the code to make it suitable for an Add-in. The code on popup menus has been removed because it is not relevant.

In previous versions of Excel, it is not necessary to give an Add-in filename any special extension. In Excel 2007, however, it is necessary to give an Add-in filename an `.xlam` extension. It is a good idea to do so, in any case, because it identifies the file as an Add-in and ensures that the Add-in icon appears against the file in the Windows File Manager. The conversion of a workbook file to an Add-in file is covered later in this chapter.

Hiding the Code

You cannot stop users from seeing a standard workbook's name, or an Add-in's name, in the Project Explorer window. However, you can stop users from expanding the workbook's name, or Add-in's name, to view the component modules and user forms and the code they contain.

You prevent access to your code by putting a password on the VBA project. Select the project and use Tools ⇨ <ProjectName> Properties (where <ProjectName> is the name of your particular project), or right-click the project in the Project Explorer window and click <ProjectName> Properties, to see the screen shown in Figure 17-1.

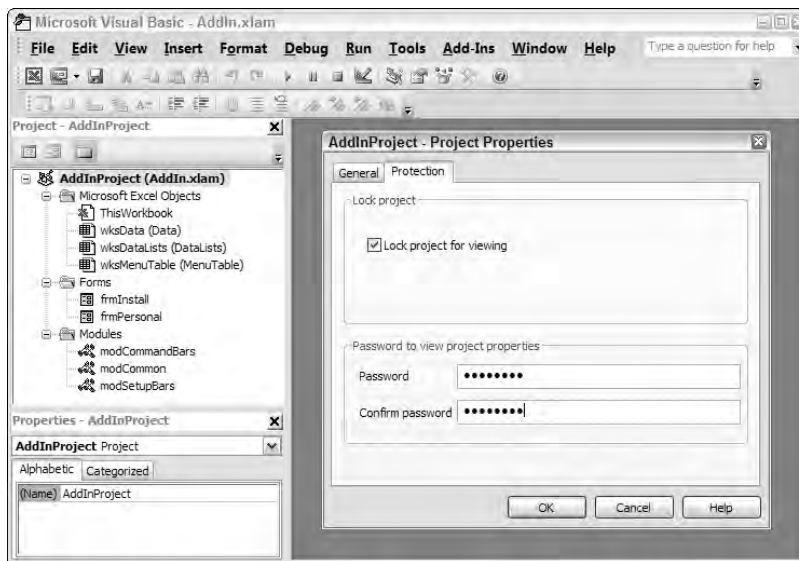


Figure 17-1

After you have entered the password and confirmed it, click OK and save the file. To see the effect, close and reopen the file. The file has been password-protected and cannot be expanded unless you supply the password. You are prompted for the password when you try to expand the project.

It is a common misconception that Excel's passwords cannot be broken. There are programs available that can decipher file, workbook, and worksheet passwords, as well as the VBA project passwords for all versions of Excel. Since the introduction of Excel 97, the workbook file password has proven a difficult nut to crack if it contains more than just a few characters. Unfortunately, this password is useless to developers who want users to be able to open their files and actually use them.

Creating an Add-in

Converting a workbook to an Add-in is a trivial exercise, on the face of it. Make sure that a worksheet is active in your workbook, click the Microsoft Office button, select Save As, scroll down the drop-down labeled Save as type:, and choose Excel Add-In (*.xlam). Excel will automatically position you

in a special Add-ins folder, although there is no requirement that you use it. The advantage of this method is that you do not overwrite the original `.x1sm` file, and you create a file with the `.x1am` extension that distinguishes it as an Add-in to the operating system.

Note that you must have a worksheet active when using Save As to create an Add-in. Otherwise, you will not find the `.x1am` type offered as an option.

An easier way to create an Add-in is to change the `IsAddin` property of `ThisWorkbook` to `True` in the Properties window, as shown in Figure 17-2.

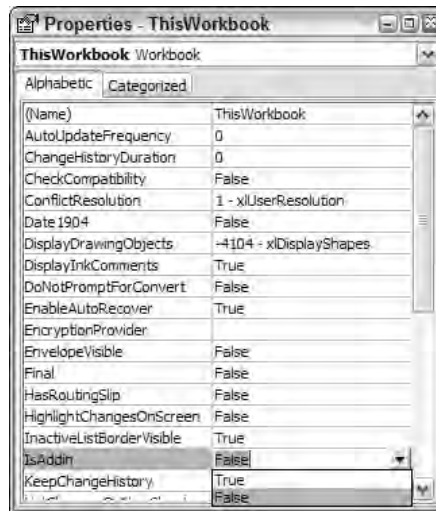


Figure 17-2

The disadvantage of this method is that you change the original `.x1sm` file to an Add-in, but its `.x1sm` extension remains unchanged and, when you open the file later, Excel 2007 objects to the mismatch between the file type and the extension. You need to change the file extension to `.x1am` manually, using the Windows Explorer.

Closing Add-ins

If you have just converted a workbook to an Add-in by changing its `IsAddin` property and saving it, or you have loaded the Add-in using `Open`, there is no obvious way to close the file from the menus without closing Excel. One way to close the Add-in is to go to the Immediate window and type in code that uses the `Close` method, treating the Add-in as a member of the `Workbooks` collection:

```
Workbooks("AddIn.x1sm").Close
```

Add-ins do not have an `Index` property value in the `Workbooks` collection and are not included in the `Count` property of the `Workbooks` collection, but they can be addressed by name as members of the `Workbooks` collection.

Chapter 17: Add-ins

Another method you can use to close an Add-in is to click the filename in the Recent Documents list while holding down Shift. You may get a message about overwriting the copy in memory (depending on whether or not it has changed), and then you will get a message about not being able to open an Add-in for editing (a hangover from previous versions). Click OK and the Add-in will be removed from memory.

Code Changes

In most cases, you need to make some changes to the VBA code that was written for a standard workbook to make it suitable for an Add-in. This is particularly true if you reference data within your Add-in workbook. Most Excel programmers write code that assumes that the workbook is the active workbook and that the worksheet is the active sheet. Nothing is active in an Add-in, so your code must explicitly reference the Add-in workbook and worksheet. For example, in Chapters 13 and 15, the code assumed that it was dealing with the active workbook, using statements like the following:

```
With Range("Database")
    Set rngData = .Rows(2)
    Call LoadRecord
    scbNavigator.Value = 2
    scbNavigator.Max = .Rows.Count
End With
```

This code only works if the worksheet containing the name `Database` is active. In your Add-in code, you need to include a reference to the workbook and worksheet. You could say:

```
With Workbooks("Addins.xlsm").Sheets("Data").Range("Database")
```

A more useful way to refer to the workbook containing the code is to use the `ThisWorkbook` property of the `Application` object that refers to the workbook containing the code. This makes the code much more flexible; you can save the workbook under any filename and the code still works:

```
With ThisWorkbook.Sheets("Data").Range("Database")
```

You can also use the object name for the sheet that you see in the Project Explorer:

```
With Sheet1.Range("Database")
```

You can edit both the workbook's programmatic name and the sheet's programmatic name in the Properties window. If you change the sheet's programmatic name, you must also change your code. If you change the workbook's programmatic name, you can use the new name if you wish, but `ThisWorkbook` remains a valid reference, because it is a property of the `Application` object and a member of `<globals>`.

If you want to be able to ignore the sheet name to allow the name `Database` to exist on any sheet, you can use the following construction:

```
With ThisWorkbook.Names("Database").RefersToRange
```

Saving Changes

Another potential problem with an Add-in that contains data is that changes to the data will not be saved automatically at the end of an Excel session. For example, `AddIn.xlam` allows users to edit the data in the range `Database`, so it is essential to save those changes before the Add-in is closed. One of the nice things about Add-ins is that users are never bothered with prompts about saving changes. Therefore, you need to ensure that data is saved by setting up a procedure in your VBA code. One way to do this is to add the following code to the `Workbook_BeforeClose` event procedure, or the `Auto_Close` procedure:

```
If Not ThisWorkbook.Saved Then ThisWorkbook.Save
```

Interface Changes

You need to bear in mind that the Add-in's sheets will not be visible and you will not see the names of the Add-in's macros in the Macro dialog box. You need to create menus, toolbars, or command buttons in other workbooks to execute your macros. Luckily, you have already built a menu and toolbar interface in your `CommandBars2.xlsm` application, which has been converted to the `AddIn.xlam` application.

It is also a good idea to make all your code as robust as possible. You should allow for abnormal events such as system crashes that could affect your code. The `CommandBars2.xlsm` application adds a menu and a toolbar to Excel when it is opened. Before doing this, `CommandBars2.xlsm` deletes any previously created versions of these command bars, which is a very good practice. It is possible that previous versions could exist following a system crash, for example.

Finally, `CommandBars2.xlsm` made the data list visible on the screen, so the results of an AutoFilter were obvious. The data list is not visible when it is in an Add-in, so you need another way to show the results of a filter. The Next and Previous buttons on the user form `frmPersonal` have been adapted to show only the filtered data. The Add-in user interface appears as shown in Figure 17-3.



Figure 17-3

Chapter 17: Add-ins

The captions on the Next and Previous buttons have been changed to Next in Dept and Previous in Dept. The code in the Click event procedure of the two buttons has been rewritten (from that in Chapter 15) to show only the rows in the data that are not hidden by the AutoFilter:

```
Private Sub cmdNext_Click()  
    Dim lBottomRow As Long  
    Dim lCheckRow As Long  
  
    With ThisWorkbook.Names("Database").RefersToRange  
        'Determine last row to check  
        lBottomRow = .Rows(.Rows.Count).Row + 1  
        'Start looking at row below current row  
        lCheckRow = rngData.Row + 1  
        'Look for first row down that is not hidden  
        Do Until lCheckRow = lBottomRow  
            If .Parent.Rows(lCheckRow).Hidden = False Then  
                Exit Do  
            Else  
                lCheckRow = lCheckRow + 1  
            End If  
        Loop  
        'If we found a visible row within the data, display it  
        If lCheckRow <> lBottomRow Then  
            scbNavigator.Value = lCheckRow + .Row - 1  
        End If  
    End With  
End Sub
```

When you search down through the data to find a row that is not hidden, you need to take into account that there might not be one. The first visible row will then be the row after the last row of data, so you need to search that far. The first row to check is the row after the current row, which is the one you are viewing. The code in the Do...Loop increments the check row until you either find a visible row or reach the bottom row. If you have found the bottom row, do nothing. Otherwise, change the scrollbar value to the found rows location in the database, which runs the scrollbar event procedure to show the data that was found:

```
Private Sub cmdPrevious_Click()  
    Dim lTopRow As Long  
    Dim lCheckRow As Long  
  
    With ThisWorkbook.Names("Database").RefersToRange  
        'The top row to check is the database header row  
        lTopRow = .Row  
        'Start looking at row above current row  
        lCheckRow = rngData.Row - 1  
        'Look for first row up that is not hidden  
        Do Until lCheckRow = lTopRow  
            If .Parent.Rows(lCheckRow).Hidden = False Then  
                Exit Do  
            Else  
                lCheckRow = lCheckRow - 1  
            End If  
        Loop  
    End With  
End Sub
```

```

'If we found a visible row within the data, display it
If lCheckRow <> lTopRow Then
    scbNavigator.Value = lCheckRow + .Row - 1
End If
End With
End Sub

```

Searching up through the data to find a row that is not hidden is a similar task to searching down. Set the top row to be checked to the header record in the database, because it might be the only unhidden row above the current row.

The operating procedure for these buttons may not be clear initially to users, so an explanatory screen will be included that appears when the Add-in is installed, as you will see later.

Installing an Add-in

An Add-in can be opened from the Office Open menu, as has been mentioned. However, you get better control over an Add-in if you install it using the Add-Ins dialog box. Click the Microsoft Office button and click Excel Options at the bottom of the dialog box. Click Add-Ins, which displays the dialog box depicted in Figure 17-4.

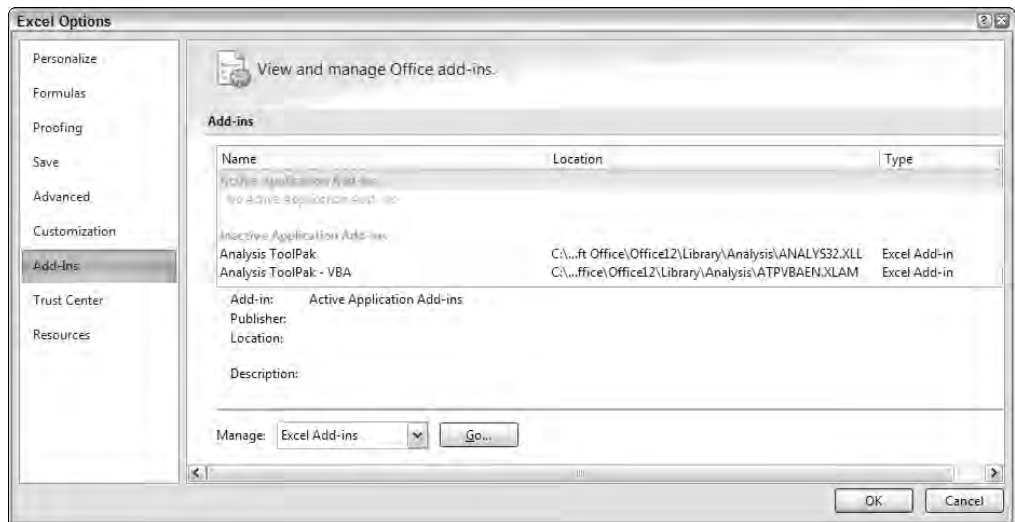


Figure 17-4

Select Excel Add-ins in the Manage drop-down and click Go to display the dialog box illustrated in Figure 17-5.

The Company Data List Add-in is the `Addin.xlam` file. If it does not already appear in the list, you can click the Browse button to locate it.

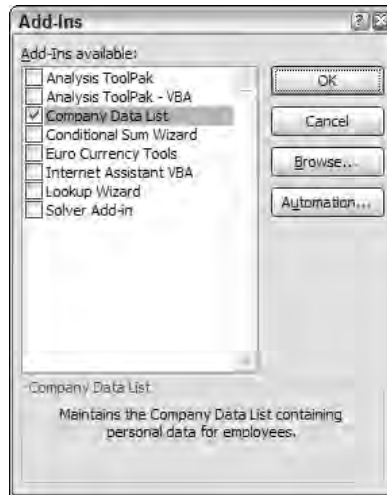


Figure 17-5

The friendly title and description are provided by filling in the workbook's Properties. If you have already converted the workbook to an Add-in, you can set its `IsAddin` property to `False` to make the workbook visible in the Excel window, click the Microsoft Office button, and then choose Prepare ⇨ Properties to display the dialog box shown in Figure 17-6.



Figure 17-6

The Title and Comments boxes supply the information for the Add-Ins dialog box. When you have added the required information, you can set the `IsAddin` property back to `True` and save the file.

If you change the Add-in workbook properties after adding it to the Add-Ins dialog box, the friendly text will not appear. You need to remove the Add-in from the list and add it back again. The removal process is covered later.

Once the Add-in is visible in the Add-Ins dialog box, you can install and uninstall the Add-in by checking and unchecking the checkbox beside the Add-in's description. When it is installed, it is loaded into memory and becomes visible in the VBE window, and will be automatically loaded in future Excel sessions. When it is uninstalled, it is removed from memory and is no longer visible in the VBE window, and will no longer be loaded in future Excel sessions.

AddinInstall Event

Two special events are triggered when you install and uninstall an Add-in. The following code, in the `ThisWorkbook` module, shows how to display a user form when the Add-in is installed:

```
Private Sub Workbook_AddinInstall()
    frmInstall.Show
End Sub
```

The user form displays the information shown in Figure 17-7 for the user.

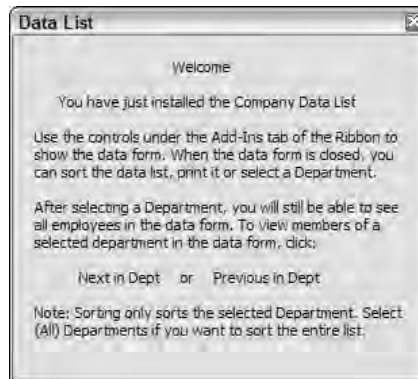


Figure 17-7

The other event is the `AddinUninstall` event.

Removing an Add-in from the Add-ins List

There is no easy way to remove an Add-in from the Add-Ins dialog box. One way you can do this is to move the Add-in file from its current folder using the Windows Explorer, before opening Excel. An

Chapter 17: Add-ins

alternative is to change the Add-in's filename before opening Excel. The message shown in Figure 17-8 will appear when you open Excel.

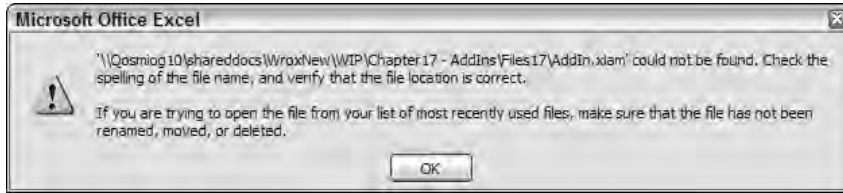


Figure 17-8

Open the Add-Ins dialog box and click the checkbox against the Add-in's entry. You will get the message depicted in Figure 17-9.

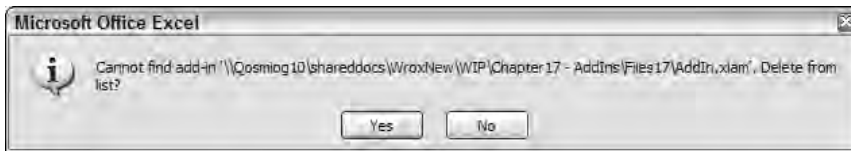


Figure 17-9

Click Yes and the Add-in will be deleted from the list.

Summary

You can provide users with all the power of VBA customization, without cluttering the Excel screen with a workbook, by creating an Add-in. Workbook files can be easily converted to Add-in files using Save As, or by changing the workbook's `IsAddIn` property to `True`. It is usually necessary to make some changes to the code when converting a workbook to an Add-in, to be able to refer to its hidden objects.

Once a file is an Add-in, it is no longer visible in the Excel window — its sheets still exist and can be used by the Add-in, but they are not displayed. You can still see the Add-in file in the Project Explorer in the VBE window. However, a password can be applied to lock the VBA project and prevent users from viewing or editing the project's modules and UserForms, just as you can lock the VBA project of a normal workbook.

An Add-in application can be accessed by users through menu commands, toolbar controls, or controls embedded in workbooks, though you cannot use popup menus. It can obtain and display information through functions such as `MsgBox` and `InputBox` and through UserForms. A workbook-based application usually needs some redesign in this area before it can be converted to an Add-in application.

Although Add-in files can be opened in the same way as workbooks, they work best when added to the Add-ins listed in the Add-Ins dialog box. Once added, they can be installed and uninstalled using the same dialog box. If they are installed, they will open automatically in every Excel session.

Automation Add-Ins and COM Add-Ins

With the release of Office 2000, Microsoft introduced a new concept for creating custom Add-Ins for all the Office applications. Instead of creating application-specific Add-Ins (*xlam* in Excel, *dotm* in Word, and so on), you can create DLLs using Visual Basic, C++, or .NET that all the Office applications can use. Because these DLLs conform to Microsoft's Component Object Model, they are known as COM Add-Ins. The second half of this chapter explains how to create and implement your own COM Add-Ins.

In Excel 2002, Microsoft extended the concept and simplified the implementation of the COM Add-In mechanism, so their functions could be used in the same way as worksheet functions and VBA user-defined functions. These Add-Ins are known as *Automation Add-Ins*.

In Excel 2007, Microsoft has further extended COM Add-Ins to support application-level customization of the Ribbon and the creation of custom task panes.

Automation Add-Ins

Automation Add-Ins are COM DLLs (ActiveX DLLs) that have a creatable class and a public function in the creatable class. For example, when Excel can do the following with your class, you can then call `FunctionName` from the worksheet:

```
Dim oAutoAddin As Object

Set oAutoAddin = CreateObject("TheProgID")
TheResult = CallByName(oAutoAddin, "FunctionName", _
    VbMethod, param1, param2, ...)
```

In other words, your function must satisfy the following conditions:

- ❑ The class must be publicly creatable (that is, have an `Instancing` property of `Multi-Use` or `Global-Multi-Use`).
- ❑ The procedure must be a `Function` (as opposed to a `Sub` or `Property` procedure).

A Simple Add-In — Sequence

For the Excel VBA developer, the easiest way to create Automation Add-Ins is still to use Visual Basic 6. This is because it uses the same syntax as VBA, and most Excel user-defined functions can be converted to Automation Add-Ins by simply copying the code to a VB6 class. Although it is possible to create Automation Add-Ins using .NET, there are a number of hoops to jump through, and creating Add-Ins with .NET is outside the scope of this book.

In this example, you create a simple Automation Add-In using VB6. It will contain a single function to provide a sequence of numbers as an array (which is often used in array formulas).

Start VB6 and create a new ActiveX DLL project. Rename the project `Excel2007ProgRef` and rename the class `Simple` in the Properties window. Set the class's `Instancing` property to `5-MultiUse` (this should be the default setting). By setting this property, you're making the class publicly creatable (that is, Excel can create instances of this class when or if you tell it to).

Type the following code into the `Simple` class, to calculate and return a sequence of numbers:

```
*****
' *
' * FUNCTION NAME: Sequence
' *
' * DESCRIPTION: Returns a sequence of numbers, often used in array formulas.
' *
' * PARAMETERS:  Items      The number of elements in the sequence
' *              Start      The starting value for the sequence, default = 1
' *              Step       The step value in the sequence, default = 1
' *
*****
Public Function Sequence(Items As Long, Optional Start As Double = 1, _
                        Optional Step As Double = 1) As Variant

    Dim vaResult As Variant
    Dim i As Long
    Dim dValue As Double

    ' Validate entries
    If Items < 1 Then
        Sequence = CVErr(2015)      '#Value
        Exit Function
    End If

    ' Create an array for the series
    ReDim vaResult(1 To Items)

    ' Get the initial value
    dValue = Start

    ' Calculate all the values, populating the array
    For i = 1 To Items
        vaResult(i) = dValue
```

```
dValue = dValue + Step
Next

' Return the array
Sequence = vaResult

End Function
```

By defining the function to be `Public`, Excel will be able to see it and you'll be able to call it from the worksheet. Save the project, then use `File` → `Make Excel2007ProgRef.dll` to create the DLL — you've just created an Automation Add-In.

Registering Automation Add-Ins with Excel

Before you can use the `Sequence` function in a worksheet, you need to tell Excel about the DLL. Microsoft has extended the Add-Ins paradigm to include Automation Add-Ins, making their usage extremely similar to normal `xla` or `xlam` Add-Ins. The main difference is that instead of a filename, Automation Add-Ins use the class's ProgID, which is the Visual Basic Project name, a period, then the class name. In this example, the ProgID of the `Simple` class is `Excel2007ProgRef.Simple`.

Through the Excel User Interface

To load an Automation Add-In through Excel's dialog, do the following:

1. Click `Office Menu` → `Excel Options` → `Add-Ins` to show the Add-Ins options.
2. Select `Excel Add-Ins` in the `Manage` drop-down and click the `Go` button to show the Add-Ins dialog.
3. Click the `Automation` button to show the Automation Add-Ins dialog.
4. Select the entry for `Excel2007ProgRef.Simple` in the list, and click `OK` to return to the Add-Ins dialog.

You should see that the Automation Add-In is now included in the list of known Add-Ins and you can load or unload it by checking or unchecking the checkbox — just like any other Add-In.

Using VBA

Automation Add-Ins are loaded in the same way as normal `xla` Add-Ins, but using the ProgID instead of the filename, as in the following code:

```
Sub InstallAutomationAddIn()
    AddIns.Add Filename:="Excel2007ProgRef.Simple"
    AddIns("Excel2007ProgRef.Simple").Installed = True
End Sub
```

In the Registry

If you are creating an installation routine for your Add-In, you may want to write directly to the registry in order to set the Automation Add-In as installed. To do so, you need to create the following registry entry (which will already exist if you've used the previous technique).

Chapter 18: Automation Add-Ins and COM Add-Ins

In the registry key:

```
HKEY_CURRENT_USER\Software\Microsoft\Office\12.0\Excel\Options
```

Create the string value:

```
Name = the first unused item in the series: Open, Open1, Open2, Open3, Open4, Open5  
etc.  
Value = /A "Excel2007ProgRef.Simple"
```

If you want to add the Automation Add-In to the list shown in the Add-Ins dialog, but not have it installed, create the following registry key instead:

```
HKEY_CURRENT_USER\Software\Microsoft\Office\12.0\Excel\Add-In Manager
```

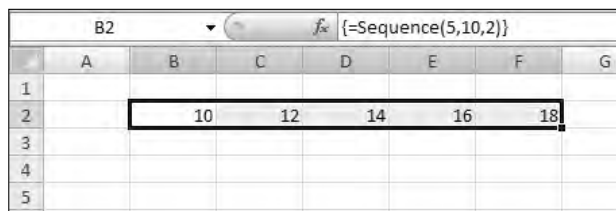
In this registry key, create an empty string value with the Name = Excel2007ProgRef.Simple.

Using Automation Add-Ins

Like normal `xlam` Add-Ins, the functions contained in Automation Add-Ins can be used both in the worksheet and within VBA routines.

In the Worksheet

Once installed, you can simply type the name of the function directly into the worksheet. To test it, start Excel 2007, install the Add-In as shown in the previous section, select a horizontal range of 5 cells, type the function `=Sequence(5,10,2)`, and enter the function as an array formula by pressing `Shift+Ctrl+Enter`. You should see a sequence of numbers, as shown in Figure 18-1.



	A	B	C	D	E	F	G
1							
2		10	12	14	16	18	
3							
4							
5							

Figure 18-1

If the function name in the Automation Add-In conflicts with a built-in Excel function or a function defined in a normal Excel Add-In, Excel will use the first one that it finds in the order of preference:

1. Built-in function
2. Function in `xla` or `xlam` Add-In
3. Function in Automation Add-In

To force Excel to use the function from the Automation Add-In, prefix the function name by the Add-In's ProgID when typing it in, as in:

```
=Excel2007ProgRef.Simple.Sequence(5,10,2)
```

As soon as you enter the function, Excel will remove the ProgID, but will still use it to reference the function correctly. This will probably cause some confusion if you subsequently edit the function, because you must remember to re-enter the ProgID each time (or Excel will think it is the built-in function).

In VBA

There are a number of alternatives for using the function in VBA. Because the DLL is a simple ActiveX DLL, you can create your own instance of it and use the function directly. This will work regardless of whether it is installed as an Add-In, as long as you have a reference to the Excel2007ProgRef library:

```
Private Sub CommandButton1_Click()

    ' Assumes a reference has been created to the Excel2007ProgRef library,
    ' using Tools | References
    Dim oSimple As Excel2007ProgRef.Simple
    Dim vaSequence As Variant

    ' Create our own instance of the class
    Set oSimple = New Excel2007ProgRef.Simple

    ' Get the sequence
    vaSequence = oSimple.Sequence(5, 10, 2)

    ' Write the sequence to the sheet
    ActiveCell.Resize(1, 5) = vaSequence
End Sub
```

If you know that the Add-In is installed, you can use the instance that Excel has created, by using `Application.Evaluate`:

```
Private Sub CommandButton1_Click()

    Dim vaSequence As Variant

    'Use Application.Evaluate - which doesn't require a reference to the DLL
    vaSequence = _
        Application.Evaluate("Excel2007ProgRef.Simple.Sequence(5,10,2)")

    'Or the shorthand:
    'vaSequence = [Excel2007ProgRef.Simple.Sequence(5,10,2)]

    'Write the sequence to the sheet
    ActiveCell.Resize(1, 5) = vaSequence
End Sub
```

When using `Application.Evaluate`, the full ProgID is only needed if there is a risk that the function name conflicts with a built-in function, or one in a loaded `xla` or `xlam` Add-In (or workbook), so the following works equally well:

```
vaSequence = Application.Evaluate("Sequence(5,10,2)")
```

It is generally safer and more robust to use the first method — creating and using your own instance of the class.

Introducing the IDTExtensibility2 Interface

The simple Add-In shown in the previous section is simple for one reason—it's self-contained and doesn't need to use Excel at all. In most real-world examples, you will want to use the Excel `Application` object in a number of ways:

- ❑ Use `Application.Caller` to identify the range that the function was called from
- ❑ Use `Application.Volatile` to mark an Automation Add-In function as volatile, and hence that Excel should call the function every time it recalculates the worksheet
- ❑ Use Excel's built-in functions within your Add-In

To use the Excel `Application` object within your Automation Add-In, you need to get (or be given) a reference to it, which you can store in a private variable within your Add-In class. This is achieved by implementing a specific interface within your Add-In class.

An *interface* is simply a predefined and fixed set of sub procedures, functions, and properties. *Implementing an interface* means that you are including all those predefined sub procedures, functions, and properties within your class. By doing this you are providing fixed, known, and predictable entry points through which Excel can call into your class.

When Excel loads an Automation Add-In, it checks to see if the Add-In has implemented an interface called `IDTExtensibility2`. If that interface has been implemented, Excel calls the `OnConnection` method defined in the interface, passing a reference to itself (that is, to the Excel `Application` object). In VBA terms, Excel is doing something like the following (don't type this in):

```
Dim oIDT2 As IDTExtensibility2
Dim oAutoAddIn As Object

' Create an instance of the Automation Add-In
Set oAutoAddIn = CreateObject("Excel2007ProgRef.Simple")

' Does it implement the special interface?
If TypeOf oAutoAddIn Is IDTExtensibility2 Then

    ' Yes it does, so get a reference to that
    ' interface within the Add-In class
    Set oIDT2 = oAutoAddIn

    ' And call the interface's OnConnection method,
    ' passing the Application
    oIDT2.OnConnection Me
End If
```

Within the Add-In's class, you can respond to the call to `OnConnection` by storing the reference to the `Application` object in a class-level variable, and using it in the Add-In's functions.

The `IDTExtensibility2` interface has five methods, each called at specific points in Excel's lifetime, though only two are used by Automation Add-Ins (the others are used by COM Add-Ins and are discussed later in the chapter). `OnConnection` has already been mentioned; the other method used here is `OnDisconnection`. Even though they are not used, you have to include code for every routine defined in the interface, as shown a bit later.

The first task in implementing an interface is to create a reference to the library in which the interface is defined. In this case, the `IDTExtensibility2` interface is defined in the Microsoft Add-In Designer library.

Open the `Excel2007ProgRef` project in Visual Basic, select `Project ⇄ References`, and put a check mark next to the Microsoft Add-In Designer item. Because you're interacting with Excel, you also need a reference to the Excel object library, so find the entry for Microsoft Excel 12.0 Object Library and check that one too. Now select `Project ⇄ Excel2007ProgRef Properties` and select the `Component` tab. Choose `Binary Compatibility` and select the file `Excel2007ProgRef.dll` that you created in the previous section (which might already be selected). This ensures that VB updates the current DLL registry entries rather than creating new ones each time you recompile.

Then add a new class module to the project, call it `Complex`, set its `Instancing` property to `5-MultiUse`, and copy in the following code to implement the `IDTExtensibility2` interface and respond to Excel calling its entry points:

```
' Implement the IDTExtensibility2 interface, so Excel can call into the class
Implements IDTExtensibility2

' Declare a private reference to the Excel application
Private moXL As Excel.Application

' Called by Excel when the class is loaded, passing a reference to the
' Excel object
Private Sub IDTExtensibility2_OnConnection(ByVal Application As Object, _
    ByVal ConnectMode As AddInDesignerObjects.ext_ConnectMode, _
    ByVal AddInInst As Object, custom() As Variant)

    ' Set a reference to the Excel application, for use in the functions
    Set moXL = Application

End Sub

' Called by Excel when the class is unloaded, so destroy the reference
' to Excel
Private Sub IDTExtensibility2_OnDisconnection( _
    ByVal RemoveMode As AddInDesignerObjects.ext_DisconnectMode, _
    custom() As Variant)

    Set moXL = Nothing
End Sub

' Not used by Automation Add-Ins, but have to be included in the class to
' implement the interface
Private Sub IDTExtensibility2_OnAddInsUpdate(custom() As Variant)
    ' Have a comment to stop VB removing the routine when doing its tidy-up
End Sub

' Not used by Automation Add-Ins, but have to be included in the class to
' implement the interface
Private Sub IDTExtensibility2_OnBeginShutdown(custom() As Variant)
    ' Have a comment to stop VB removing the routine when doing its tidy-up
```

```
End Sub

' Not used by Automation Add-Ins, but have to be included in the class to
' implement the interface
Private Sub IDTExtensibility2_OnStartupComplete(custom() As Variant)
    ' Have a comment to stop VB removing the routine when doing its tidy-up
End Sub
```

A Complex Add-In — RandUnique

Now that you have a reference to the Excel `Application` object, you can use it in a more complex function. The `RandUnique` function shown next returns a random set of integers between two limits, without any duplicates in the set. It uses the Excel `Application` object in two ways:

- ❑ It uses `Application.Caller` to identify the range containing the function, and hence the size and shape of the array to create and return.
- ❑ It uses `Application.Volatile` to ensure the function is recalculated each time Excel calculates the sheet.

The routine works by doing the following:

- ❑ It creates an array of all the integers between the given limits, with a random number associated with each item.
- ❑ It sorts the array by the random number, effectively putting the array into a random order.
- ❑ It reads the first n items from the jumbled-up array to fill the required range.

The function has also been written to take an optional `Items` parameter, enabling it to be called from VBA as well as from the worksheet. If the `Items` parameter is provided, the function returns a 2D array (1, n) of unique integers. If the `Items` parameter is not provided, the function uses `Application.Caller` to identify the size of array to create. Because you're using VB's random number generator, it's a good idea to add a `Randomize` statement in the initial `OnConnection` call to ensure that you get different numbers each time (type in the shaded rows):

```
' Called by Excel when the class is loaded, passing a reference to the
' Excel object
Private Sub IDTExtensibility2_OnConnection(ByVal Application As Object, _
    ByVal ConnectMode As AddInDesignerObjects.ext_ConnectMode, _
    ByVal AddInInst As Object, custom() As Variant)

    ' Set a reference to the Excel application, for use in the functions
    Set moXL = Application

    ' Initialize the VB random number generator
    Randomize
End Sub
```

```
' *****
' *
' * FUNCTION NAME:  RandUnique
' *
```



```

'* DESCRIPTION:      Returns an array of random integers between two limits,
'*                  without duplication
'*
'* PARAMETERS:      Min           The lower limit for the random numbers
'*                  Max           The upper limit for the random numbers
'*
'*****
Public Function RandUnique(Min As Long, Max As Long, _
                          Optional Items As Long) As Variant

    Dim oRng As Range
    Dim vaValues() As Double, vaResult() As Double
    Dim iItems As Long, i As Long, iValue As Long
    Dim iRows As Long, iCols As Long, iRow As Long, iCol As Long

    ' Tell Excel that this function is volatile, and should be called
    ' every time the sheet is recalculated
    moXL.Volatile

    ' If we've been given the number of items required, use it...
    If Items > 0 Then
        iRows = 1
        iCols = Items
    Else
        '... Otherwise get the range of cells that this function is in
        ' (as an array formula)
        Set oRng = moXL Caller

        iRows = oRng.Rows.Count
        iCols = oRng.Columns.Count
    End If

    ' How many cells in the range
    iItems = iRows * iCols

    ' We can't generate a unique set of numbers if there are more
    ' cells to fill than there are numbers to choose from,
    ' so return an error value in that case
    If iItems > (Max - Min + 1) Then
        RandUnique = CVErr(xlErrValue)
        Exit Function
    End If

    ' Fill an array with all the possible numbers to choose from,
    ' and a column of random numbers to sort on
    ReDim vaValues(Min To Max, 1 To 2)
    For i = Min To Max
        vaValues(i, 1) = i
        vaValues(i, 2) = Rnd()
    Next

    ' Sort by the array by the column of random numbers,
    ' jumbling up the array

```

```
Sort2DVert vaValues, 2, "A"

' Dimension an array to be the same size as the range we're called from
ReDim vaResult(1 To iRows, 1 To iCols)

' Start the counter at the beginning of the jumbled array
iValue = Min

' Fill the result array from the jumbled array of all values
For iRow = 1 To iRows
    For iCol = 1 To iCols
        vaResult(iRow, iCol) = vaValues(iValue, 1)
        iValue = iValue + 1
    Next
Next

' Return the result
RandUnique = vaResult

End Function
```

A QuickSort Routine

The `RandUnique` function uses a standard `QuickSort` algorithm to sort the array, reproduced next. This is one of the fastest sorting algorithms and uses a recursive divide-and-conquer approach to sorting:

- ❑ Choose one of the numbers in the array (usually the middle one).
- ❑ Group all the numbers less than it at the top of the array, and all the numbers greater than it at the bottom.
- ❑ Repeat for the top half of the array, then for the bottom half.

```
*****
'*
'* FUNCTION NAME:   SORT ARRAY - 2D Vertically
'*
'* DESCRIPTION:    Sorts the passed array into required order, using the
'*                  given key. The array must be a 2D array of any size.
'*
'* PARAMETERS:     avArray   The 2D array of values to sort
'*                  iKey     The column to sort by
'*                  sOrder   A-Ascending, D-Descending
'*                  iLow1    The first item to sort between
'*                  iHigh1   The last item to sort between
'*
*****
Private Sub Sort2DVert(avArray As Variant, iKey As Integer, _
                    sOrder As String, Optional iLow1, Optional iHigh1)

    Dim iLow2 As Integer, iHigh2 As Integer, i As Integer
    Dim vItem1, vItem2 As Variant

    On Error GoTo PtrExit

    If IsMissing(iLow1) Then iLow1 = LBound(avArray)
```

```

If IsMissing(iHigh1) Then iHigh1 = UBound(avArray)

' Set new extremes to old extremes
iLow2 = iLow1
iHigh2 = iHigh1

' Get value of array item in middle of new extremes
vItem1 = avArray((iLow1 + iHigh1) \ 2, iKey)

' Loop for all the items in the array between the extremes
Do While iLow2 < iHigh2

    If sOrder = "A" Then
        ' Find the first item that is greater than the mid-point item
        Do While avArray(iLow2, iKey) < vItem1 And iLow2 < iHigh1
            iLow2 = iLow2 + 1
        Loop

        ' Find the last item that is less than the mid-point item
        Do While avArray(iHigh2, iKey) > vItem1 And iHigh2 > iLow1
            iHigh2 = iHigh2 - 1
        Loop
    Else
        ' Find the first item that is less than the mid-point item
        Do While avArray(iLow2, iKey) > vItem1 And iLow2 < iHigh1
            iLow2 = iLow2 + 1
        Loop

        ' Find the last item that is greater than the mid-point item
        Do While avArray(iHigh2, iKey) < vItem1 And iHigh2 > iLow1
            iHigh2 = iHigh2 - 1
        Loop
    End If

    ' If the two items are in the wrong order, swap the rows
    If iLow2 < iHigh2 Then
        For i = LBound(avArray, 2) To UBound(avArray, 2)
            vItem2 = avArray(iLow2, i)
            avArray(iLow2, i) = avArray(iHigh2, i)
            avArray(iHigh2, i) = vItem2
        Next
    End If

    ' If the pointers are not together, advance to the next item
    If iLow2 <= iHigh2 Then
        iLow2 = iLow2 + 1
        iHigh2 = iHigh2 - 1
    End If
Loop

' Recurse to sort the lower half of the extremes
If iHigh2 > iLow1 Then Sort2DVert avArray, iKey, sOrder, iLow1, iHigh2

' Recurse to sort the upper half of the extremes

```

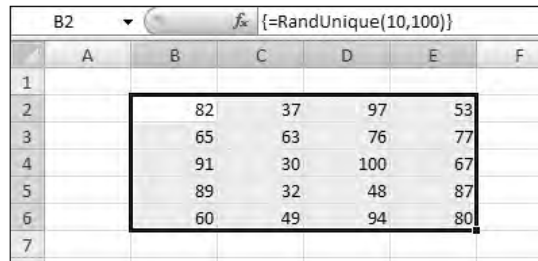
Chapter 18: Automation Add-Ins and COM Add-Ins

```
If iLow2 < iHigh1 Then Sort2DVert avArray, iKey, sOrder, iLow2, iHigh1  
  
PtrExit:  
  
End Sub
```

You must now save and recompile your project by using File ⇨ Make Excel2007ProgRef.dll to create the updated DLL.

Due to a bug in some versions of VB, you may be presented with an error message stating that there is a sharing violation with the file you are trying to replace. If this occurs, try closing VB (making sure you have saved your project) and then reopening it. You will also get an error if the Add-In is currently loaded in an Excel session, so you'll need to close your Excel session(s) before rebuilding the Add-In.

The complex Add-In is used in the same way as the simple `Sequence` function shown previously. The only difference is that you have to tell Excel to load the `Excel2007ProgRef.Complex` Add-In, by clicking Office Menu ⇨ Excel Options ⇨ Add-Ins ⇨ Manage: Excel Add-Ins ⇨ Automation Add-Ins and selecting it from the list. When entered as an array formula, the `RandUnique` function looks something like Figure 18-2.



	A	B	C	D	E	F
1						
2		82	37	97	53	
3		65	63	76	77	
4		91	30	100	67	
5		89	32	48	87	
6		60	49	94	80	
7						

Figure 18-2

COM Add-Ins

Whereas Automation Add-Ins enable you to create your own worksheet functions, COM Add-Ins provide a way to extend the user interface of Excel and all the other Office applications. They have a number of advantages over normal `xla` or `xlam` Add-Ins, including:

- ❑ They're much faster to open.
- ❑ They're less obtrusive (not showing up in the VBE Project Explorer).
- ❑ They're more secure (being compiled DLLs).
- ❑ They're not specific to a single application — the same mechanism works with all the Office applications and the VBE itself (and any other application that uses VBA 6), allowing you to create a single Add-In that can extend all the Office applications.

The IDTExtensibility2 Interface (Continued)

The previous section introduced the `IDTExtensibility2` interface, where you used the `OnConnection` and `OnDisconnection` methods to obtain a reference to the Excel Application. The remaining methods defined in the interface can be used by COM Add-Ins to respond to specific events in Excel's lifetime. The methods are outlined in the following table.

Method	Occurs	Typical Usage
<code>OnConnection</code>	When the COM Add-In is loaded by Excel.	Store a reference to the Excel application, add menu items to Excel's <code>CommandBars</code> , and set up event hooks.
<code>OnStartupComplete</code>	After Excel has finished loading all Add-Ins and initial files.	Show a startup dialog (such as those in Access and PowerPoint) or change behavior depending on whether other Add-Ins are loaded.
<code>OnAddInsUpdate</code>	Whenever any other COM Add-Ins are loaded or unloaded.	If the COM Add-In depends on another Add-In being loaded, this Add-In can unload itself.
<code>OnBeginShutdown</code>	When Excel starts its shutdown process.	Stop the shutdown in certain circumstances or perform any pre-shutdown tidy-up routines.
<code>OnDisconnection</code>	When the COM Add-In is unloaded, either by the user or by Excel shutting down.	Save settings. If unloaded by the user, delete any <code>CommandBar</code> items that were created at connection.

Most COM Add-Ins use only the `OnConnection` method (to add their menu items) and `OnDisconnection` method (to remove them), though code has to exist in the class module for all five methods to correctly implement the interface.

Registering a COM Add-In with Excel

For Automation Add-Ins, you told Excel that the Add-In exists by selecting it in the Automation Add-Ins dialog (resulting in some entries being written to the registry). You tell Excel that a COM Add-In exists by writing specific keys and values to specific places in the registry. When Excel starts, it looks in those keys to see which COM Add-Ins exist, then checks the values in those keys to see how to display them in the COM Add-Ins list, whether or not to load them, and so on. The keys for COM Add-Ins targeted to Excel are:

- Registered for the current user:

```
HKEY_CURRENT_USER\Software\Microsoft\Office\Excel\Addins\AddInProgID
```

- Registered for all users:

```
HKEY_USERS\.\DEFAULT\Software\Microsoft\Office\Excel\Addins\AddInProgID
```

Chapter 18: Automation Add-Ins and COM Add-Ins

- ❑ Registered for the machine:

HKEY_LOCAL_MACHINE\Software\Microsoft\Office\Excel\Addins\AddInProgID

The values are as follows.

Name	Type	Use
FriendlyName	String	The name shown in the COM Add-Ins list.
Description	String	The description shown in the COM Add-Ins dialog.
LoadBehavior	Number	Whether it is unloaded, loaded at startup, or demand-loaded.
SatelliteDllName	Number	The name of a resource DLL that contains localized names and descriptions. If used, the name and description will be #Num, where Num is the numeric resource ID in the Satellite DLL. Most of the standard Office Add-Ins use this technique for their localization.
CommandLineSafe	String	Whether the DLL could be called from the command line (not applicable to Office COM Add-Ins).

Once registered correctly, the COM Add-In will show up in Excel's COM Add-Ins dialog, where it can be loaded and unloaded like any other Add-In. You can find the COM Add-Ins dialog by clicking Office Menu ⇨ Excel Options ⇨ Add-Ins ⇨ Manage: COM Add-Ins ⇨ Go.

The COM Add-In Designer

Microsoft has provided a COM Add-In Designer class to assist in the creation and registration of COM Add-Ins. It provides the following benefits:

- ❑ Implements the `IDTExtensibility2` interface, exposing the methods as events that you can either hook or ignore. You don't, therefore, have to include code for unused interface methods in your class module.
- ❑ Provides a form to fill in to provide the values for the registry entries used to register the COM Add-In, and to select which application to target.
- ❑ When compiled, it adds code to the standard `DllRegisterServer` entry point in the DLL that writes all the registry entries for you when the DLL is registered on the system (though only for the Current User key). This greatly simplifies installation, because you can install the Add-In by running the following command: `RegSvr32 %c:\MyPath\MyComAddIn.DLL`.

By way of an example, you'll create a COM Add-In that provides a Wizard for entering the `RandUnique` Automation Add-In function created in the previous section. You will continue to use Visual Basic, building on the `Excel2007ProgRef` DLL from the previous section.

Open the `Excel2007ProgRef` project in Visual Basic. Add a new Add-In class to the project by clicking Project ⇨ Add Addin Class (if that menu item doesn't exist, click Project ⇨ Components ⇨ Designers and

check the Addin Class entry). This adds a new Designer class and gives it the name AddInDesigner1. Using the Properties window, change the name to COMAddIn and set the Public property to True (ignoring any warnings). Fill in the Designer form as follows:

Add-In Display Name	Excel 2007 Prog Ref Wizards
Addin Description	Displays a Wizard dialog for entering the Sequence and RandUnique Automation Addin functions, documented in the Excel 2007 VBA Programmers Reference
Application	Microsoft Excel
Application Version	Microsoft Excel 12.0
Initial Load Behavior	Startup

The Designer only creates registry entries for the current user. If you wish to install the Add-In for all users on the machine, you will need to add your own registry entries in the Advanced tab of the Designer form, as documented in Microsoft KnowledgeBase article Q290868 at <http://support.microsoft.com/KB/q290868/>.

Linking to Excel

Click View ⇄ Code to get to the Designer's code module, and copy in the following code to hook into the IDTExtensibility2 interface and link the COM Add-In to Excel by storing a reference to the Excel Application object, passed to the Add-In in the OnConnection method:

```
Dim WithEvents moXL As Excel.Application

' The IDTExtensibility2_OnConnection method is handled by the Designer,
' and exposed to us through the AddInInstance_OnConnection method
Private Sub AddInInstance_OnConnection( _
    ByVal Application As Object, _
    ByVal ConnectMode As AddInDesignerObjects.ext_ConnectMode, _
    ByVal AddInInst As Object, custom() As Variant)

    Set moXL = Application
    MsgBox "Connected"

End Sub

' The IDTExtensibility2_OnDisconnection method is handled by the Designer,
' and exposed to us through the AddInInstance_OnDisconnection method
Private Sub AddInInstance_OnDisconnection( _
    ByVal RemoveMode As AddInDesignerObjects.ext_DisconnectMode, _
    custom() As Variant)

    Set moXL = Nothing
    MsgBox "Disconnected"

End Sub
```

Chapter 18: Automation Add-Ins and COM Add-Ins

Save the project and make the Add-In DLL by clicking File ⇨ Make Excel2007ProgRef.dll, and then open Excel 2007 (note that you will not be able to subsequently rebuild the DLL if it is being accessed by Excel at the time). As Excel opens, you'll see a Connected message pop up as the Add-In is connected, and a Disconnected message when Excel is closed. You will also get these messages if you load or unload the Add-In using the COM Add-Ins dialog.

Responding to Excel's Events

The `Designer` code module is a type of class module that allows you to declare a variable `WithEvents`, to hook into their events. In the previous code, you hooked into the Excel Application events, enabling the COM Add-In to respond to the users opening or closing workbooks, changing data in cells, and so on, in the same way you can in a normal Excel Add-In. See Chapters 8 and 16 for more information about these events.

Adding CommandBar Controls

Prior to Excel 2007, you used the `CommandBars` objects to create all menus and toolbars. In Excel 2007, the Ribbon replaced the top-level menus and toolbars, but you still use the `CommandBars` objects for the popup menus. This example COM Add-In adds two menu items to the cell's right-click popup menu to show Wizard forms to assist in the entry of Automation Add-In formulas. Using the Ribbon with COM Add-Ins is explained later in the chapter.

Once you have a reference to the `Excel Application` object, you can add buttons to command bars in the same way as described in Chapter 15. The only difference is how the code responds to a button being clicked.

When adding a `CommandBarButton` from within Excel, set its `OnAction` property to be the name of the VBA procedure to run when the button is clicked.

When adding a `CommandBarButton` from outside Excel (from within a COM Add-In), hook the button's `Click` event using a variable declared `WithEvents` inside the Add-In.

To use `CommandBarButtons`, you need a reference to the Office object library, so click Project ⇨ References and check the Microsoft Office 12.0 Object Library.

Delete any code that may already exist in the `Designer`'s code module (such as the example code added in the previous section), and replace it with the following. This defines the class-level variables you'll be using to store the reference to the `Excel Application` object, and to hook the `CommandBarButton`'s events:

```
Dim WithEvents moXL As Excel.Application
Dim WithEvents moBtn As Office.CommandBarButton

Const msAddInTag As String = "Excel2007ProgRefTag"
```

When you hook a command bar button's events using the `WithEvents` keyword, the variable (`moBtn`) is associated with the `Tag` property of the button it's set to reference. All buttons that share the same `Tag` will cause the `Click` event to fire. In this way, you can handle the `click` events for all your buttons

using a single `WithEvents` variable, by ensuring they all have the same `Tag`. You can distinguish between buttons by giving them each a unique `Parameter` property as you create them in the `OnConnection` method, which should be copied into the Designer's code module:

```
' The IDTExtensibility2_OnConnection method is handled by the Designer,
' and exposed to us through the Add-InInstance_OnConnection method
Private Sub AddInInstance_OnConnection( _
    ByVal Application As Object, _
    ByVal ConnectMode As AddInDesignerObjects.ext_ConnectMode, _
    ByVal AddInInst As Object, custom() As Variant)

    Dim oToolBar As CommandBar, oBtn As CommandBarButton

    Set moXL = Application

    ' Get a reference to the cell right-click menus
    Set oToolBar = moXL.CommandBars("Cell")

    ' If our controls don't exist on the menu bar, add them

    ' Handle errors in-line (such as the button not existing)
    On Error Resume Next

    ' Check for, and add, the 'Sequence Wizard' button
    Set oBtn = oToolBar.Controls("Sequence Wizard")
    If oBtn Is Nothing Then
        Set oBtn = oToolBar.Controls.Add( _
            msoControlButton, , "SequenceWiz", , True)

        With oBtn
            .Caption = "Sequence Wizard"
            .Style = msoButtonCaption
            .Tag = msAddInTag
        End With
    End If

    ' Check for, and add, the 'RandUnique Wizard' button
    Set oBtn = Nothing
    Set oBtn = oToolBar.Controls("RandUnique Wizard")
    If oBtn Is Nothing Then
        Set oBtn = oToolBar.Controls.Add( _
            msoControlButton, , "RandUniqueWiz", , True)

        With oBtn
            .Caption = "RandUnique Wizard"
            .Style = msoButtonCaption
            .Tag = msAddInTag
        End With
    End If

    ' Set the WithEvents object to hook these buttons. All buttons
    ' that share the same Tag property will fire the moBtn_Click event
    Set moBtn = oBtn

End Sub
```

Chapter 18: Automation Add-Ins and COM Add-Ins

Although you set the `Temporary` parameter to `True` when adding the controls, it is good practice to delete them when the Add-In is unloaded, using the `OnDisconnection` event:

```
' The IDTExtensibility2_OnDisconnection method is handled by the Designer,
' and exposed to us through the AddInInstance_OnDisconnection method
Private Sub AddInInstance_OnDisconnection( _
    ByVal RemoveMode As AddInDesignerObjects.ext_DisconnectMode, _
    custom() As Variant)
    Dim oCtl As CommandBarControl

    ' Find and remove the buttons
    For Each oCtl In moXL.CommandBars.FindControls(Tag:=msAddInTag)
        oCtl.Delete
    Next

    Set moBtn = Nothing
    Set moXL = Nothing

End Sub
```

In the `Click` event, you check the `Parameter` property of the button that was clicked and show the appropriate form. For this example, just add two blank forms to the project, giving them the names `frmSequenceWiz` and `frmRandUniqueWiz`:

```
' The moBtn_Click event is fired when any of our commandbar buttons are
' clicked. This is because the event handler is associated with the Tag
' property of the button, not the button itself. Hence, all buttons that
' have the same Tag will fire this event.
Private Sub moBtn_Click(ByVal Ctrl As Office.CommandBarButton, _
    CancelDefault As Boolean)

    ' Check that a cell range is selected
    If TypeOf moXL.Selection Is Range Then

        ' Run the appropriate form, depending on the control's Parameter
        Select Case Ctrl.Parameter
            Case "SequenceWiz"
                frmSequenceWiz.Show vbModal

            Case "RandUniqueWiz"
                frmRandUniqueWiz.Show vbModal
        End Select
    Else
        ' Display an error message if a range is not selected
        MsgBox "A range must be selected to run the Wizard.", vbOKOnly, _
            "Excel 2007 Prog Ref Wizards"
    End If

End Sub
```

Save the project and use File ⇨ Make Excel2007ProgRef.dll to create the DLL, which also adds the registry entries for Excel to see it. Start Excel 2007, right-click a cell, and click the Sequence Wizard menu to show the Wizard form.

Using a COM Add-In from VBA

It is possible (though unfortunately quite rare) for the creator of a COM Add-In to provide programmatic access to the Add-In from VBA. This would be done either to:

- ❑ Expose the Add-In's functionality for use through code
- ❑ Provide a mechanism for controlling or customizing the Add-In

It is achieved by setting the Add-In instance's `Object` property to reference the COM Add-In class (or a separate class within the Add-In), and then exposing the required functionality using `Public` properties and methods, just like any other class. This example provides yet another way of getting to the `Sequence` and `RandUnique` functions.

Add the following lines to the bottom of the `AddInInstance_OnConnection` routine, to provide a reference to the Add-In class using the Add-In's `Object` property:

```
' Set the Add-In instance's Object property to be this class, providing
' access to the Com Add-In's object model from within VBA. Note that we
' don't use Set here!
AddInInst.Object = Me
```

And add the following code to the bottom of the Designer's class module, to create and return new instances of our `Simple` and `Complex` classes:

```
' Property to return a reference to our Simple class, providing access
' from VBA:
'vaSeq = Application.ComAddIns("Excel2007ProgRef.ComAddIn").Object _
' .SimpleFuncs.Sequence(...)
Public Property Get SimpleFuncs() As Simple
    Set SimpleFuncs = New Simple
End Property

' Property to return a reference to our Complex class, providing access
' from VBA:
'vaRU = Application.ComAddIns("Excel2007ProgRef.ComAddIn").Object _
' .ComplexFuncs.RandUnique(...)
Public Property Get ComplexFuncs() As Complex
    Set ComplexFuncs = New Complex
End Property
```

From within Excel, you can then use the following code to access the `Sequence` function, going through the COM Add-In and its `Object` property:

```
Private Sub CommandButton1_Click()

    Dim vaSequence As Variant

    ' Get the sequence using the COM Add-In
```

```
vaSequence = Application.ComAddIns("Excel2007ProgRef.ComAddIn") _  
    .Object.SimpleFuncs.Sequence(5, 10, 2)  
  
    ' Write the sequence to the sheet  
    ActiveCell.Resize(1, 5) = vaSequence  
End Sub
```

The key point about using this method is that you are accessing the same instance of the class that Excel is using for the Add-In, allowing you to manipulate, query, or control that Add-In from VBA. For more complex COM Add-Ins, the same method can be used to provide access to a full object model for controlling the Add-In.

Adding Ribbon Controls

Chapter 14 explained how VBA applications can modify the Ribbon by creating custom tabs, groups, or controls. This was done by creating a text file containing the XML for the custom UI definition and adding it to the XML workbook file. When Excel loads the workbook, it sees the custom part and processes it, creating custom controls. As designed, this allows you to create document-level RibbonX customizations, but that chapter demonstrated how to achieve application-level customizations by simply using a standard Excel XML Add-In (.xlam). The “official” approach to application-level UI customizations is to use a COM Add-In.

Obviously, COM Add-Ins don’t have an XML workbook that Excel can check for any custom UI XML. Instead, each time Excel 2007 loads a COM Add-In, it checks to see if the Add-In implements another specific interface, `IRibbonExtensibility`. If that interface is found, Excel calls its `GetCustomUI` function and the Add-In returns the custom UI XML as text. After that point, the behavior of a COM Add-In is exactly the same as the VBA code shown in Chapter 14, except that all the callbacks must exist in the same class that implements the `IRibbonExtensibility` interface. To demonstrate this concept, this section creates a simple COM Add-In that uses RibbonX to add a menu to Excel’s View tab. Start by creating a new VB6 Add-In project and performing the following steps to configure it correctly:

1. Remove the default form.
2. Delete all the code from the Designer Connect class.
3. Edit the Designer to give it a meaningful name and description, targeting Excel 12.0 and loading at startup.
4. Change the project name from MyAddIn to XLVBARibbonX.
5. Click Project ⇄ References and uncheck the reference to the Visual Basic 6.0 Extensibility library.
6. Check that the project references the Microsoft Excel 12.0 Object Library and the Microsoft Office 12.0 Object Library (which contains the definition of the `IRibbonExtensibility` interface), adding them if they’re missing.
7. Copy the following code into the Connect class (which can also be found in the XLVBARibbonX folder in the code download for this chapter):

```
'Implement an interface to tell Excel we're doing things with RibbonX  
Implements IRibbonExtensibility  
  
'Store a reference to the ribbon, so we can invalidate controls when needed
```

```

Dim moRibbon As IRibbonUI

'Called by Excel at startup. Provide the custom UI.
Private Function IRibbonExtensibility_GetCustomUI(ByVal RibbonID As String) _
    As String
    Dim sXML As String

    'Build the XML for the custom UI
    'Here, we're Adding a simple button to the middle of Excel's View tab
    'Typically, this would be read from a resource file
    sXML = ""
    sXML = sXML & "<customUI " & _
        "xmlns="&"http://schemas.microsoft.com/office/2006/01/customui" " & _
        "onLoad="&"CustomUI_OnLoad" ">"

    sXML = sXML & " <ribbon>"
    sXML = sXML & " <tabs>"
    sXML = sXML & " <tab idMso="&"TabView" ">"
    sXML = sXML & " <group id="&"XLVBABView" "
    sXML = sXML & " insertAfterMso="&"GroupViewShowHide" "
    sXML = sXML & " label="&"VBA Prog Ref" ">"
    sXML = sXML & " <button id="&"CTPTest" label="&"A Test" "
    sXML = sXML & " imageMso="&"DateAndTimeInsert" size="&"large" "
    sXML = sXML & " onAction="&"CTPTest_Click" />"
    sXML = sXML & " </group>"
    sXML = sXML & " </tab>"
    sXML = sXML & " </tabs>"
    sXML = sXML & " </ribbon>"
    sXML = sXML & "</customUI>"

    IRibbonExtensibility_GetCustomUI = sXML
End Function

'Called by Excel to provide the Ribbon object,
'which is used to invalidate controls, forcing a refresh.
Public Sub CustomUI_OnLoad(ribbon As IRibbonUI)
    Set moRibbon = ribbon
End Sub

'Show a message when the button is clicked
Public Sub CTPTest_Click(control As IRibbonControl)
    MsgBox "Clicked me!"
End Sub

```

Compile the DLL, start Excel 2007, click the View tab, and click the Date Picker button in the middle of the tab. You should get the Clicked me! message box.

Note that the `GetCustomUI` function is only called once in the life of the COM Add-In, at startup. That means you have only the one opportunity to provide your custom UI XML. Although you can change the visibility of controls, in practice that means you are extremely limited in the degree to which you can change your UI in response to changes within Excel (such as opening or closing workbooks). This is a serious deficiency of the COM Add-In RibbonX extensibility model. Instead of being asked to provide CustomUI XML at startup, Excel should provide a reference to a class factory the Add-In could use to create or modify UI customizations at any time during its life. As you'll see later in this chapter, that design is used for custom task panes and it works extremely well.

Creating Custom Task Panes

When task panes were introduced in Office XP, developers were soon eager to use them for their own content. In Office 2007, Microsoft has answered that request by adding the ability for COM Add-Ins to create custom task panes (CTPs), using custom ActiveX controls to define their content. Just like RibbonX, the COM Add-In tells Excel that it contains a custom task pane by implementing another interface, `ICustomTaskPaneConsumer`. A COM Add-In, therefore, needs to do the following to create a CTP:

- ❑ Implement the `ICustomTaskPaneConsumer` interface, which contains a single method, `CTPFactoryAvailable`.
- ❑ The `CTPFactoryAvailable` procedure is passed a reference to Excel's CTP factory class, which the Add-In stores in a module-level variable.
- ❑ Design a custom ActiveX control to provide the content of the CTP.
- ❑ Create, show, and hide the CTP in response to a user trigger.

From a VBA developer's perspective, the restriction that you can only use ActiveX controls to provide the CTP content is a rather nasty one, because you can't create ActiveX controls using VBA. There are, however, two ways to get around that. The first is to realize that there are a number of readily available ActiveX controls that you can drop into a CTP, such as the Web Browser control. All you need is a simple COM Add-In that exposes the capability to create new CTPs to VBA, then use VBA code to create a CTP containing a Web Browser control and automate the Web Browser control to show an HTML page. To create the COM Add-In, follow the steps listed in the "Adding Ribbon Controls" section, give the project the name `OACTPVBA`, and copy in the following code to implement the `ICustomTaskPaneConsumer` interface and expose CTP creation to VBA:

```
'Tell Excel that we're working with custom task panes
Implements ICustomTaskPaneConsumer

'Store a reference to Excel's CTP factory class
Dim moCTPFactory As ICTPFactory

'Expose the functions in this class to VBA
Private Sub AddInInstance_OnConnection(ByVal Application As Object, _
    ByVal ConnectMode As AddInDesignerObjects.ext_ConnectMode, _
    ByVal AddInInst As Object, custom() As Variant)
    AddInInst.object = Me
End Sub

'Called by Excel when the Add-In is loaded,
'providing a factory object that we use to
'create custom task panes.
Private Sub ICustomTaskPaneConsumer_CTPFactoryAvailable( _
    ByVal CTPFactoryInst As Office.ICTPFactory)
    'Store a reference to the factory object, for use when required
    Set moCTPFactory = CTPFactoryInst
End Sub

'Expose CTP-creation to VBA, e.g. to create a web browser task pane:
'Dim moCTP As CustomTaskPane
'Set moCTP = Application.ComAddIns("OACTPVBA.Connect").Object
```

```
' .CreateTaskPane("Internet Explorer","Shell.Explorer.2")
Public Function CreateTaskPane(ByVal sTitle As String, _
    ByVal sProgID As String) As Office.CustomTaskPane

    On Error Resume Next
    Set CreateTaskPane = moCTPFactory.CreateCTP(sProgID, sTitle)
End Function
```

Compile the DLL, start Excel, and use the following VBA code in a standard module to show Google in a custom task pane:

```
'Keep a reference to the CTP, so we can show/hide it
Dim moCTP As CustomTaskPane

Sub ShowGoogleCTP()

    'Create the CTP containing a Web Browser control
    Set moCTP = Application.COMAddIns("OACTPVBA.Connect").Object _
        .CreateTaskPane("Internet Explorer", "Shell.Explorer.2")

    'Show the CTP
    moCTP.Visible = True

    'Navigate to Google
    moCTP.ContentControl.navigate "http://www.google.com"
End Sub
```

It's enlightening to compare the way in which CTPs have been implemented to the RibbonX mechanism. For CTPs, when the COM Add-In starts up, Excel passes a factory class into the `CTPFactoryAvailable` method, which is stored in a module-level variable. You can then use the factory class to create new instances of custom task panes at any time you like. In turn, that allows you to very easily expose the new CTP functionality to VBA. In contrast, the RibbonX design requires you to specify all customization as soon as the COM Add-In starts, so you can only create one customization snippet and can neither expose the RibbonX features to VBA nor create dynamic interfaces that respond to the changing Excel environment.

Showing VBA UserForms as Task Panes

While CTPs are a very useful addition to Excel VBA, you're either limited by the set of generally available ActiveX controls or you need to learn how to create your own ActiveX controls. Although this is beyond the scope of this book, it is actually quite easy to do, using either VB6 or .NET.

There is, however, a third possibility, which is for someone to create a custom ActiveX control that can in turn host a normal VBA UserForm — and we've done exactly that! The `OACTPUserformHost` is a single ocx file containing both a COM Add-In and an ActiveX control, and it's available as one of the download files for this chapter from www.wrox.com.

Load the COM Add-In by using the Add... button on the COM Add-Ins dialog (Office Menu ⇨ Excel Options ⇨ Add-Ins ⇨ Manage: COM Add-Ins ⇨ Go), choosing Files of Type: All Files, and navigating to the `OACTPUserformHost.ocx` file.

Chapter 18: Automation Add-Ins and COM Add-Ins

You can show a standard VBA UserForm as a task pane by including the following code within the UserForm's code module. You'll first need to add a Project reference to the `OACTPUserformHost` library, listed as Custom Task Pane Userform Host in the Project References dialog. If not in the list, it can be added by clicking the Browse button, choosing Files of Type: ActiveX Controls, and navigating to the `OACTPUserformHost.ocx` file.

```
'A WithEvents object variable to refer to the Custom Task Pane Userform Host
'object. Requires a reference to OACTPUserformHost.OCX, listed as
'"Custom Task Pane Userform Host" in the Project References dialog
Dim WithEvents moCTP As CTPUserformHost

' Public method to show a VBA userform in a custom
' task pane. Typical usage is:
'
' UserformName.ShowAsTaskPane
'
Public Sub ShowAsTaskPane()

    If moCTP Is Nothing Then
        'Create a task pane with the required title
        Set moCTP = Application.COMAddIns("OACTPUserformHost.Connect").Object _
            .CreateUserformTaskPaneHost(Me, "Hello World")

        With moCTP
            'Set the task pane's properties

            '(default = msoCTPDockPositionRestrictNone)
            .DockPositionRestrict = msoCTPDockPositionRestrictNone

            '(default = msoCTPDockPositionRight)
            .DockPosition = msoCTPDockPositionRight

            'Tell the task pane whether to handle the userform's resizing
            '(default = False)
            .HandleResizing = True
        End With
    End If

    'Make the task pane visible
    moCTP.Visible = True

End Sub

'Close the CTP when the form is unloaded
Private Sub UserForm_Terminate()
    On Error Resume Next
    moCTP.Visible = False
    Set moCTP = Nothing
End Sub
```

A call to `Userform.ShowAsTaskPane` uses the `OACTPUserformHost` COM Add-In to create a custom task pane hosting the UserForm, which is then initialized and displayed. As well as acting as a host for the UserForm, the COM Add-In also handles the form's resizing, using the technique for resizable forms shown in Chapter 27.

Putting it all Together

By adding extra interfaces that a COM Add-In can choose to implement, Microsoft has created an easily extensible architecture that works across all the Office applications and across multiple development platforms — commonly VB6 and .NET. A feature-rich COM Add-In can implement all three interfaces (*IDTExtensibility2*, *IRibbonExtensibility*, and *ICustomTaskPaneConsumer*) to deliver its functionality. The sample *XLVBADatePicker* COM Add-In does this to create a custom task pane that contains the standard Month View control, with a RibbonX toggle button to control its visibility. To make it all work seamlessly, it handles the following events:

- When the toggle button is clicked, the task pane is shown or hidden.
- When the user closes the task pane, the toggle button is restored.
- When a date is selected in the task pane, the value is put into the active cell.
- When a cell containing a date is selected, the task pane shows that date in the calendar.

You can find the complete code for the COM Add-In at www.wrox.com, in the *XLVBADatePicker* folder in the sample downloads for this chapter.

Linking to Multiple Office Applications

The start of this chapter mentioned that one of the fundamental advantages of COM Add-Ins over *xla* or *xlam* Add-Ins is that the same DLL can target multiple Office applications. All you need to do to achieve this is to add a new *Add-In Designer* class for each application you want to target, in exactly the same way you added the *Designer* to target Excel previously in the chapter. Of course, you still have to handle the idiosyncrasies of each application separately.

In the following simple example, you make the *Sequence* function available through the COM Add-Ins collection in Access and use it to populate a list box on a form.

Start by opening the *Excel2007ProgRef* project and adding a new Add-In class to the project. In the Properties window, change its name to *AccessAddIn*, set its *Public* property to *True* (ignoring any warnings), and complete the *Designer's* form as follows:

Add-In Display Name	Excel 2007 Prog Ref Sequence
Addin Description	Example to expose the Sequence function through Access' COM Addins.
Application	Microsoft Access
Application Version	Microsoft Access 12.0
Initial Load Behavior	Startup

Click View ⇨ Code and copy the following into the *Designer's* code module:

```
' Simple COM Add-In to provide the Sequence function to MS Access,
' through Access' COMAdd-Ins collection

Private Sub Add-InInstance_OnConnection(ByVal Application As Object, _
```

Chapter 18: Automation Add-Ins and COM Add-Ins

```
        ByVal ConnectMode As Add-InDesignerObjects.ext_ConnectMode, _
        ByVal Add-InInst As Object, custom() As Variant)

    ' Set the Add-In instance's Object property to be this class,
    ' providing access to the Com Add-In's object model from within VBA.
    ' Note that we don't use Set here!
    Add-InInst.Object = Me

End Sub

' Property to return a reference to our Simple class, providing access
' from VBA:
'vaSeq = Application.ComAdd-Ins("Excel2007ProgRef.ComAdd-In").Object _
'    .SimpleFuncs.Sequence(...)
Public Property Get SimpleFuncs() As Simple
    Set SimpleFuncs = New Simple
End Property
```

Save the project and use File ⇨ Make Excel2007ProgRef.dll to build the DLL. Start Access 2007 with a blank database, create a new form, add a list box, and copy the following code into the form's code module:

```
Private Sub Form_Load()

    Dim vaSequence As Variant
    Dim i As Integer

    ' Use the COMAdd-In to get the sequence
    vaSequence = Application.COMAddIns("Excel2007ProgRef.AccessAddIn") _
        .Object.SimpleFuncs.Sequence(5, 10, 2)

    ' Add the sequence to the list box
    List0.RowSourceType = "Value list"
    For i = LBound(vaSequence) To UBound(vaSequence)
        List0.AddItem vaSequence(i)
    Next

End Sub
```

Save the form and run it to show the COM Add-In at work (see Figure 18-3).

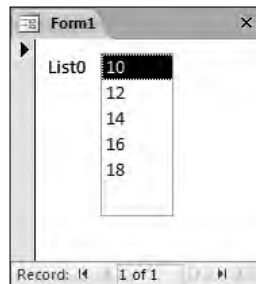


Figure 18-3

Summary

In Excel 2007, Microsoft has provided a number of ways to extend Excel using Add-Ins written in Visual Basic, or any other language that can produce Component Object Model (COM) DLLs, including .NET:

- ❑ With Automation Add-Ins, you can add new functions for use in Excel worksheets and VBA routines.
- ❑ The performance of Automation and COM Add-Ins is typically much faster than their VBA equivalents.
- ❑ With COM Add-Ins, you can add new menu items and respond to Excel's events. You can also use these to create Add-Ins that work across multiple Office applications and the VBE.
- ❑ The COM Add-In can provide programmatic access to the behavior of the Add-In, such as enabling or disabling its actions, or using its functions.
- ❑ In Excel 2007, COM Add-Ins can be used for application-level customization of the Ribbon or to create custom task panes.
- ❑ The Custom Task Pane extensibility model can be used to create multiple CTPs within a single COM Add-In and expose the CTP-creation features to VBA.
- ❑ The RibbonX extensibility model forces you to declare your UI customizations as soon as the COM Add-In is loaded, preventing you from exposing RibbonX to VBA or creating highly dynamic interfaces.

Interacting with Other Office Applications

The Office application programs Excel, Word, PowerPoint, Outlook, and Access all use the same VBA language. Once you understand VBA syntax in Excel, you know how to use VBA in all the other applications. Where these applications differ is in their object models.

One of the really nice things about the common VBA language is that all the Office applications are able to expose their objects to each other, and you can program interaction between all of the applications from any one of them. To work with Word objects from Excel, for example, you only need to establish a link to Word, and then you have access to its objects as if you were programming with VBA in Word itself.

This chapter explains how to create the link in a number of different ways, and presents some simple examples of programming the other application. In all cases, the code is written in Excel VBA, but it could easily be modified for any other Office application. The code is equally applicable to products outside Office that support the VBA language. These include other Microsoft products such as Visual Basic and SQL Server. There is also a growing list of non-Microsoft products that can be programmed in the same way.

Establishing the Connection

Once you have made a connection with an Office application, its objects are exposed for automation through a type library. There are two ways to establish such a connection: *late binding* and *early binding*. In either case, you establish the connection by creating an object variable that refers to the target application, or a specific object in the target application. You can then proceed to use the properties and methods of the object referred to by the object variable.

In late binding, you create an object that refers to the Office application before you make a link to the Office application's type library. In earlier versions of the Office applications, it was necessary to use late binding, and you will still see it used because it has some advantages over early

Chapter 19: Interacting with Other Office Applications

binding. One advantage is that you can write code that can detect the presence or absence of the required type library on the PC running your code, and link to different versions of applications based on decisions made as the code executes.

Late binding is very useful when you have to run an application under different versions of Office. Office happily upgrades an application's *early bound* links when those links are to an earlier version. Unfortunately, the opposite is not true. An application linked to a later version of Office than the version it is running under gives errors. The problem is compounded if users can save the changes to the links in applications on a network. Because late binding does not create the links until execution time, the links are always to the version it is running under.

The disadvantage of late binding is that the type library for the target application is not accessed when you are writing your code. Therefore, you get no help information regarding the application, you cannot reference the intrinsic constants in the application, and when the code is compiled, the references to the target application may not be correct, because they cannot be checked. The links are only fully resolved when you try to execute the code, and this takes time. It is also possible that coding errors may be detected at this point that cause your program to fail.

Early binding is supported by all the Office applications, from Office 97 onward. Code that uses early binding executes faster than code using late binding, because the target application's type library is present when you write your code. Therefore, more syntax and type checking can be performed, and more linkage details can be established, before the code executes.

It is also easier to write code for early binding because you can see the objects, methods, and properties of the target application in the Object Browser, and as you write your code, you will see automatic tips appear, such as a list of related properties and methods after you type an object reference. You can also use the intrinsic constants defined in the target application.

In the following examples that involve Outlook, you could be interrupted by your virus scanning software when accessing Outlook, because the scanner could identify your actions as possible virus activity.

Late Binding

The following code creates an entry in the Outlook calendar. The code uses the late binding technique:

```
Sub MakeOutlookAppointment()  
    'Example of Outlook automation using late binding  
    'Creates an appointment in Outlook  
  
    Dim olApp As Object 'Reference to Outlook  
    Dim olAppointment As Object 'Reference to Outlook Appointment  
    Dim olNameSpace As Object 'Reference to Outlook NameSpace  
    Dim olFolder As Object 'Dummy reference to initialize Outlook  
    Const olAppointmentItem = 1 'Outlook intrinsic constants not available  
    Const olFolderInbox = 6 'Outlook intrinsic constants not available  
  
    'Create link to Outlook
```

```
Set olApp = CreateObject("Outlook.Application")
Set olNameSpace = olApp.GetNamespace("MAPI")
Set olFolder = olNameSpace.GetDefaultFolder(olFolderInbox)
Set olAppointment = olApp.CreateItem(olAppointmentItem)

'Set details of appointment
With olAppointment
    .Subject = "Discuss Whitefield Contract"
    .Start = DateSerial(2007, 2, 26) + TimeSerial(9, 30, 0)
    .End = DateSerial(2007, 2, 26) + TimeSerial(11, 30, 0)
    .ReminderPlaySound = True
    .Save
End With

'Release object variable
Set olApp = Nothing

End Sub
```

The basic technique in programming another application is to create an object variable referring to that application. The object variable in this case is `olApp`. You then use `olApp` (as you would use the `Application` object in Excel) to refer to objects in the external application's object model. In this case, the `CreateItem` method of Outlook's `Application` object is used to create a reference to a new `AppointmentItem` object.

You have also created references to the `NameSpace` object and the `Inbox` folder. This is not because you want to use these objects. It is a way of initializing Outlook that has been found to be effective. If you don't do this, errors can occur.

Because Outlook's intrinsic constants are not available in late binding, you need to define your own constants, such as `olAppointmentItem` here, or substitute the value of the constant as the parameter value. You go on to use the properties and methods of the `Appointment` object in the `With...End With` structure. Note the times have been defined using the `DateSerial` and `TimeSerial` functions to avoid ambiguity or problems in an international context. See Chapter 25 for more details on international issues.

By declaring `olApp` and `olAppointment` as the generic `Object` type, you force VBA to use late binding. VBA cannot resolve all the links to Outlook until it executes the `CreateObject` function.

The `CreateObject` input argument defines the application name and class of object to be created. Outlook is the name of the application and `Application` is the class. Many applications allow you to create objects at different levels in the object model. For example, Excel allows you to create `Worksheet` or `Chart` objects from other applications, using `Excel.Worksheet` or `Excel.Chart` as the input parameter of the `CreateObject` function.

It is good programming practice to close the external application when you are finished with it and set the object variable to `Nothing`. This releases the memory used by the link and the application.

If you run this macro, nothing will happen in Excel at all. However, open up Outlook, and in the Calendar you will find that the appointment has been added for the morning of February 26, as shown in Figure 19-1.

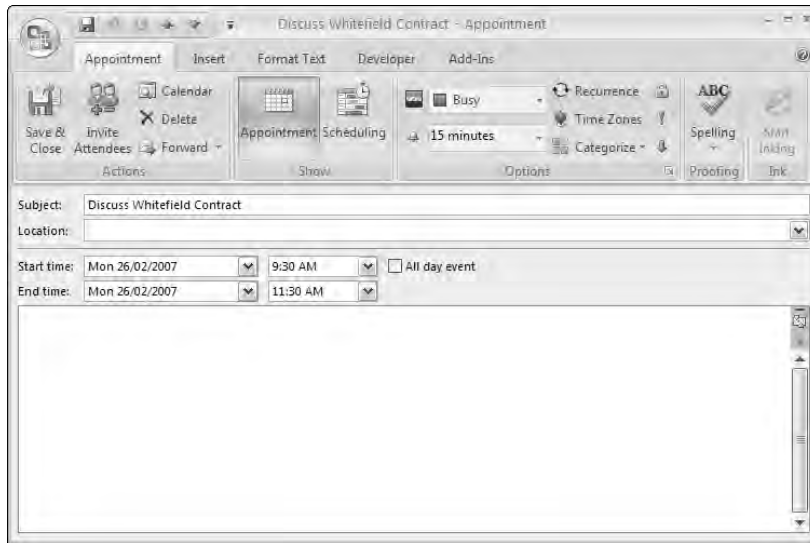


Figure 19-1

Early Binding

If you want to use early binding, you need to establish a reference to the type library of the external application in your VBA project. You do this from the VBE by clicking **Tools** → **References**, which displays the dialog box shown in Figure 19-2.

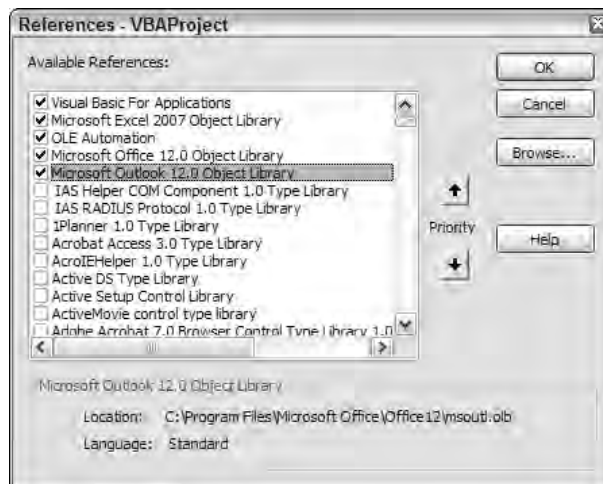


Figure 19-2

You create a reference by checking the box next to the object library. Once you have a reference to an application, you can declare your object variables as the correct type. For example, you could declare `olEntry` as an `AddressEntry` type, as follows:

```
Dim olEntry As AddressEntry
```

VBA will search through the type libraries, in the order shown from the top down, to find references to object types. If the same object type is present in more than one library, it will use the first one found. You can select a library and click the Priority buttons to move it up or down the list to change the order in which libraries are searched. There is no need to depend on priority, however. You can always qualify an object by preceding it with the name of the main object in the library. For example, instead of using `AddressEntry`, use `Outlook.AddressEntry`.

The following example uses early binding. It lists all the names of the entries in the Outlook Contacts folder, placing them in column A of the active worksheet. Make sure that you have created a reference to the Outlook object library before you try to execute it, or you will get the error “User-defined type not defined”:

```
Sub DisplayOutlookContactNames()  
    'Example of Outlook automation using early binding  
    'Lists all the Contact names from Outlook in the A column  
    'of the active sheet  
    Dim olApp As Outlook.Application  
    Dim olNameSpace As Outlook.NameSpace  
    Dim olFolder As Outlook.Folder  
    Dim olAddresslist As AddressList  
    Dim olEntry As AddressEntry  
    Dim i As Long  
  
    'Create link to Outlook Contacts folder  
    Set olApp = New Outlook.Application  
    Set olNameSpace = olApp.GetNamespace("MAPI")  
    Set olFolder = olNameSpace.GetDefaultFolder(olFolderInbox)  
    Set olAddresslist = olNameSpace.AddressLists("Contacts")  
    For Each olEntry In olAddresslist.AddressEntries  
        i = i + 1  
        'Enter contacts in A column of active sheet  
        Cells(i, 1).Value = olEntry.Name  
    Next  
  
    'Release object variable  
    Set olApp = Nothing  
End Sub
```

Here, you directly declare `olApp` to be an `Outlook.Application` type. The other `Dim` statements also declare object variables of the type you need. If the same object name is used in more than one object library, you can precede the object name by the name of the application, rather than depend on the priority of the type libraries. You did this with `Outlook.NameSpace` to illustrate the point. The `New` keyword is used when assigning a reference to `Outlook.Application` to `olApp` to create a new instance of Outlook.

Chapter 19: Interacting with Other Office Applications

The fact that you declare the variable types correctly makes VBA use early binding. You could use the `CreateObject` function to create the `o1App` object variable, instead of the `New` keyword, without affecting the early binding. However, it is more efficient to use `New`.

Opening a Document in Word

If you want to open a file created in another Office application, you can use the `GetObject` function to directly open the file. However, it is just as easy to open an instance of the application and open the file from the application. Another use of `GetObject` is examined shortly.

If you are not familiar with the Word object model, you can use the Word macro recorder to discover which objects, properties, and methods you need to use to perform a Word task that you can do manually.

The following code copies a range in Excel to the clipboard. It then starts a new instance of Word, opens an existing Word document, and pastes the range to the end of the document. Because the code uses early binding, make sure you establish a reference to the Word object library:

```
Sub CopyTableToWordDocument()  
    'Example of Word automation using early binding  
    'Copies range from workbook and appends it to existing Word document  
    Dim wdApp As Word.Application  
  
    'Copy A1:B6 in Table sheet  
    ThisWorkbook.Sheets("Table").Range("A1:B6").Copy  
  
    'Establish link to Word  
    Set wdApp = New Word.Application  
    With wdApp  
        'Open Word document  
        .Documents.Open Filename:="C:\VBA_Prog_Ref\Chapter19\Table.docx"  
        With .Selection  
            'Go to end of document and insert paragraph  
            .EndKey Unit:=wdStory  
            .TypeParagraph  
            'Paste table  
            .Paste  
  
        End With  
        .ActiveDocument.Save  
        'Exit Word  
        .Quit  
    End With  
    'Release object variable  
    Set wdApp = Nothing  
End Sub
```

The `New` keyword creates a new instance of Word, even if Word is already open. The `Open` method of the `Documents` collection is used to open the existing file. The code then selects the end of the document, enters a new empty paragraph, and pastes the range. The document is then saved and the new instance of Word is closed.

Accessing an Active Word Document

Say you are working in Excel, creating a table. You also have Word open with a document active, into which you want to paste the table you are creating. You can copy the table from Excel to the document using the following code. There is no need to establish a reference to Word if you declare `wdApp` as an `Object` type, because VBA will use late binding. On the other hand, you can establish a reference to Word, declare `wdApp` as a `Word.Application` type, and VBA will use early binding. In this example, you are using early binding:

```
Sub CopyTableToOpenWordDocument()  
    'Example of Word automation using late binding  
    'Copies range from workbook and appends it to  
    ' a currently open Word document  
  
    Dim wdApp As Word.Application  
  
    'Copy Range A1:B6 on sheet named Table  
    ThisWorkbook.Sheets("Table").Range("A1:B6").Copy  
  
    'Establish link to open instance of Word  
    Set wdApp = GetObject(, "Word.Application")  
    With wdApp.Selection  
        'Go to end of document and insert paragraph  
        .EndKey Unit:=wdStory  
        .TypeParagraph  
        'Paste table  
        .Paste  
  
    End With  
    'Release object variable  
    Set wdApp = Nothing  
End Sub
```

The `GetObject` function has two input parameters, both of which are optional. The first parameter can be used to specify a file to be opened. The second can be used to specify the application program to open. If you do not specify the first parameter, `GetObject` assumes you want to access a currently open instance of Word. If you specify a zero-length string as the first parameter, `GetObject` assumes you want to open a new instance of Word.

You can use `GetObject` with no first parameter, as in the previous code, to access a current instance of Word that is in memory. However, if there is no current instance of Word running, `GetObject` with no first parameter causes a run-time error.

Creating a New Word Document

Say you want to use a current instance of Word if one exists, or if there is no current instance, you want to create one. In either case, you want to open a new document and paste the table into it. The following code shows how to do this. Again, you are using early binding:

```
Sub CopyTableToAnyWordDocument()  
    'Example of Word automation using early binding  
    'Copies range from workbook and pastes it in  
    'a new Word document, in a active instance of  
    'Word, if there is one.  
    'If not, opens new instance of Word  
  
    Dim wdApp As Word.Application  
  
    'Copy Range A1:B6 on sheet named Table  
    ThisWorkbook.Sheets("Table").Range("A1:B6").Copy  
  
    On Error Resume Next  
    'Try to establish link to open instance of Word  
    Set wdApp = GetObject(, "Word.Application")  
  
    'If this fails, open Word  
    If wdApp Is Nothing Then  
        Set wdApp = GetObject("", "Word.Application")  
    End If  
    On Error GoTo 0  
  
    With wdApp  
        'Add new document  
        .Documents.Add  
        'Make Word visible  
        .Visible = True  
    End With  
  
    With wdApp.Selection  
        'Go to end of document and insert paragraph  
        .EndKey Unit:=wdStory  
        .TypeParagraph  
        'Paste table  
        .Paste  
  
    End With  
    'Release object variable  
    Set wdApp = Nothing  
End Sub
```

If there is no current instance of Word, using `GetObject` with no first argument causes a run-time error, and the code then uses `GetObject` with a zero-length string as the first argument, which opens a new instance of Word, and then creates a new document. The code also makes the new instance of Word visible, unlike the previous examples, where the work was done behind the scenes without showing the Word window. The table is then pasted at the end of the Word document. At the end of the procedure, the object variable `wdApp` is released, but the Word window is accessible on the screen so that you can view the result.

Access and ADO

If you want to copy data from Access to Excel, you can establish a reference to the Access object library and use the Access object model. However, this is overkill because you don't really need most of the functionality in Access. You can also use ADO (ActiveX Data Objects), which is Microsoft's technology for programmatic access to relational databases, and many other forms of data storage. For a comprehensive treatment of ADO, see Chapter 20.

Figure 19-3 shows an Access table named `SalesData` that is in an Access database file `SalesDB.accdb`.

ID	Date	Customer	State	Product	NumberSold	Price	Revenue
1	1/01/2006	Roberts	NSW	Oranges	903	15	13545
2	1/01/2006	Roberts	TAS	Oranges	331	15	4965
3	2/01/2006	Smith	QLD	Mangoes	299	20	5980
4	6/01/2006	Roberts	QLD	Oranges	612	15	9180
5	8/01/2006	Roberts	VIC	Apples	907	12.5	11337.5
6	8/01/2006	Pradesh	TAS	Pears	107	18	1926
7	10/01/2006	Roberts	VIC	Apples	770	12.5	9625
8	14/01/2006	Smith	NT	Apples	223	12.5	2787.5
9	14/01/2006	Smith	VIC	Oranges	132	15	1980
10	15/01/2006	Pradesh	QLD	Oranges	669	15	10035
11	17/01/2006	Roberts	NSW	Mangoes	881	20	17620
12	21/01/2006	Kee	SA	Pears	624	18	11232
13	22/01/2006	Roberts	QLD	Mangoes	193	20	3860
14	23/01/2006	Smith	SA	Mangoes	255	20	5100
15	27/01/2006	Kee	QLD	Mangoes	6	20	120
16	27/01/2006	Kee	VIC	Mangoes	311	20	6220

Figure 19-3

The following code uses ADO to open a recordset based on the `Sales` table. It uses early binding, so a reference to the ADO object library is required. You will need to create a reference to Microsoft ActiveX Data Objects. If you find multiple versions of this library, choose the one with the highest version number:

```
Sub GetSalesDataViaADO()
    'Example of ADO automation using early binding
    'Copies Sales table from Access database to new worksheet

    Dim con As ADODB.Connection
    Dim rsSales As ADODB.Recordset
    Dim i As Integer
    Dim wks As Worksheet
    Dim iCount As Integer

    'Establish connection to database
    Set con = New ADODB.Connection
    con.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _
```

```
"Data Source=C:\VBA_Prog_Ref\Chapter19\SalesDB.accdb"

'Open recordset based on Sales Table
Set rsSales = New ADODB.Recordset
Set rsSales.ActiveConnection = con
rsSales.Open "Select * From SalesData"

'Add new worksheet to active workbook
Set wks = Worksheets.Add

iCount = rsSales.Fields.Count
'Enter field names across row 1
For i = 0 To iCount - 1
    wks.Cells(1, i + 1).Value = rsSales.Fields(i).Name
Next

'Copy entire recordset data to worksheet, starting in A2
wks.Range("A2").CopyFromRecordset rsSales

'Format worksheet dates in A column
wks.Columns("B").NumberFormat = "mmm dd, yyyy"
'Bold row 1 and fit columns to largest entry
With wks.Range("A1").Resize(1, iCount)
    .Font.Bold = True
    .EntireColumn.AutoFit
End With

'Release object variables
Set rsSales = Nothing
Set con = Nothing

End Sub
```

The code creates a connection to the database and a recordset based on the `SalesData` table. A new worksheet is added to the Excel workbook, and the field names in `rsSales` are assigned to the first row of the new worksheet. The code uses the `CopyFromRecordset` method of the `Range` object to copy the records in `rsSales` to the worksheet, starting in cell A2. `CopyFromRecordset` is a very fast way to copy the data, compared to a looping procedure that copies record by record.

Access, Excel, and Outlook

As another example of integrating different Office applications, you will extract some data from Access, chart it using Excel, and e-mail the chart using Outlook. The code has been set up as four procedures. The first procedure is a sub procedure named `EmailChart` that establishes the operating parameters and executes the other three procedures. Note that the code uses early binding, and you need to create references to the ADO and Outlook object libraries:

```
Sub EmailChart()
'Gets data from Access using SQL statement
'Creates chart and emails chart file to recipient

    Dim sSQL As String
```

```
Dim rngData As Excel.Range
Dim sFileName As String
Dim sRecipient As String

sSQL = "SELECT Product, Sum(Revenue) "
sSQL = sSQL & " FROM SalesData"
sSQL = sSQL & " WHERE Date >= #1/1/2006# and Date<#1/1/2007#"
sSQL = sSQL & " GROUP BY Product;"

sFileName = "C:\VBA_Prog_Ref\Chapter19\Chart.xlsx"
' Replace the made up email address with a valid one (perhaps your own)
sRecipient = "somebody@foobar.com"

Set rngData = rngSalesData(sSQL)
ChartData rngData, sFileName
SendEmail sRecipient, sFileName

End Sub
```

sSQL is used to hold a string that is a SQL (Structured Query Language) command. SQL is covered in more detail in Chapter 20. In this case, the SQL specifies that you want to select the unique product names and the sum of the revenues for each product from your Access database `SalesData` table for all dates in the year 2006. `sFileName` defines the path and filename that will be used to hold the chart workbook. `sRecipient` holds the e-mail address of the person you are sending the chart to.

The code then executes the `rngSalesData` function that is listed as follows. The function accepts the SQL statement as an input parameter and returns a reference to the range containing the extracted data, which is assigned to `rngData`. The `ChartData` sub procedure is then executed, passing in the data range, as well as the path and filename for the chart workbook. Finally, the `SendEmail` sub procedure is executed, passing in the recipient's e-mail address and the location of the chart workbook to be attached to the e-mail:

```
Function rngSalesData(sSQL As String) As Excel.Range
'Function to extract data from database using
'SQL statement in sSQL
'Returns a reference to the range containing
'the data

Dim con As ADODB.Connection
Dim rsSales As ADODB.Recordset

'Establish connection to database
Set con = New ADODB.Connection
con.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=C:\VBA_Prog_Ref\Chapter19\SalesDB.accdb"

'Open recordset based on Sales Table
Set rsSales = New ADODB.Recordset
Set rsSales.ActiveConnection = con
rsSales.Open sSQL

'Clear sheet and bring in new data
```

Chapter 19: Interacting with Other Office Applications

```
With Worksheets("Data")
    .Cells.Clear

    With .Range("A1")
        'Copy entire recordset data to worksheet, starting in A1
        .CopyFromRecordset rsSales
        'Return reference to data range
        Set rngSalesData = .CurrentRegion
    End With
End With

'Release object variables
Set rsSales = Nothing
Set con = Nothing

End Function
```

The `rngSalesData` function is similar to the `GetSalesDataViaADO` sub procedure presented earlier. Instead of getting the entire `SalesData` table from the database, it uses SQL to be more selective. It clears the worksheet named `Data` and copies the selected data to a range starting in `A1`. It does not add the field names to the worksheet, just the product names and total revenue. It uses the `CurrentRegion` property to obtain a reference to all the extracted data and assigns the reference to the return value of the function:

```
Sub ChartData(rngData As Range, sFileName As String)
'Procedure to create chart based on data in rngData
'Bounds data to chart as arrays
'Saves chart to path and file in sFileName

'Create new workbook
With Workbooks.Add

    'Create new chart sheet
    With .Charts.Add

        'Create new data series and assign data
        With .SeriesCollection.NewSeries
            .XValues = rngData.Columns(1).Value
            .Values = rngData.Columns(2).Value
        End With

        'Format chart
        .HasLegend = False
        .HasTitle = True
        .ChartTitle.Text = "Year 2006 Revenue"

    End With

    'Save workbook and close it
    Application.DisplayAlerts = False
    .SaveAs sFileName
    Application.DisplayAlerts = True

End With
```



```
.Close  
  
End With  
  
End Sub
```

ChartData has input parameters to define the range containing the data to be charted and the destination for the file it creates. It creates a new workbook and adds a chart sheet to it. It creates a new series in the chart and assigns the values from the data range as arrays to the axes of the series. DisplayAlerts is set to False to prevent a warning if it overwrites an old file of the same name.

The following SendEmail sub sends the chart workbook as an attachment to an e-mail:

```
Sub SendEmail(sRecipient As String, sAttachment As String)  
'Send email to sRecipient  
'Attaching file in sAttachment  
  
    Dim olApp As Object  
    Dim olNameSpace As Object  
    Dim olFolder As Object  
    Dim olMail As Object  
  
    Set olApp = CreateObject("Outlook.Application")  
    Set olNameSpace = olApp.GetNamespace("MAPI")  
    'Might be necessary to Logon  
    'olNameSpace.Logon "UserName", "Password"  
    Set olFolder = olNameSpace.GetDefaultFolder(6)  
    Set olMail = olApp.CreateItem(0)  
    With olMail  
        .Subject = "Year 2006 Revenue Chart"  
        .Recipients.Add sRecipient  
        .Body = "Workbook with chart attached"  
  
        .Attachments.Add sAttachment  
        .Send  
    End With  
  
End Sub
```

SendEmail has input parameters for the e-mail address of the recipient and the filename of the attachment for the e-mail. If your Outlook configuration requires you to log on, you will need to uncomment the lines that get a reference to the Namespace and supply the username and password. A new mail item is created, using the CreateItem method. Text is added for the subject line and the body of the e-mail, and the recipient and attachment are specified. The Send method sends the e-mail.

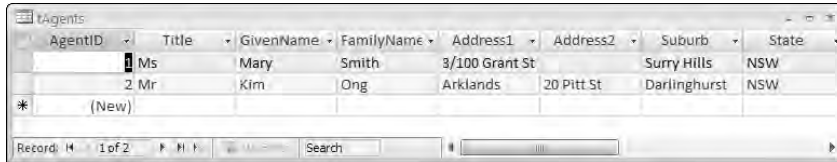
Better than Mail Merge

Using VBA, it is possible to emulate a Word mail merge. Mail merge is not capable of producing variable-length tables within a document, but you can do this using VBA. The following techniques and code show how.

Chapter 19: Interacting with Other Office Applications

Say you run an educational institution that provides courses to students, who are agents providing a specific service to the community. You have an Access database containing information on your students and the courses they attend. You want to produce a letter to go to each student, containing a summary of the courses they have attended.

The tAgents table holds information on the students, as shown in Figure 19-4.



AgentID	Title	GivenName	FamilyName	Address1	Address2	Suburb	State
1	Ms	Mary	Smith	3/100 Grant St		Surry Hills	NSW
2	Mr	Kim	Ong	Arklands	20 Pitt St	Darlinghurst	NSW
*		(New)					

Figure 19-4

The tCourses table holds information on the available courses, as shown in Figure 19-5.



CourseID	CourseName	Points
1	Migrant Law	1
2	Business Practices	2
4	Visa Requirements	1
*	(New)	0

Figure 19-5

The tCoursesAttended table holds information on the dates on which students attended courses, as shown in Figure 19-6.



CoursesAttendedID	CourseID	AgentID	CourseDate
1	1	1	2/03/2005
2	2	2	3/04/2005
3	4	1	4/05/2005
4	1	2	5/06/2005
*	(New)	0	0

Figure 19-6

There is also a query named qCoursesAttended, shown in Figure 19-7, that joins the tCoursesAttended table and the tCourses table to provide the details you need for the mail merge letter.

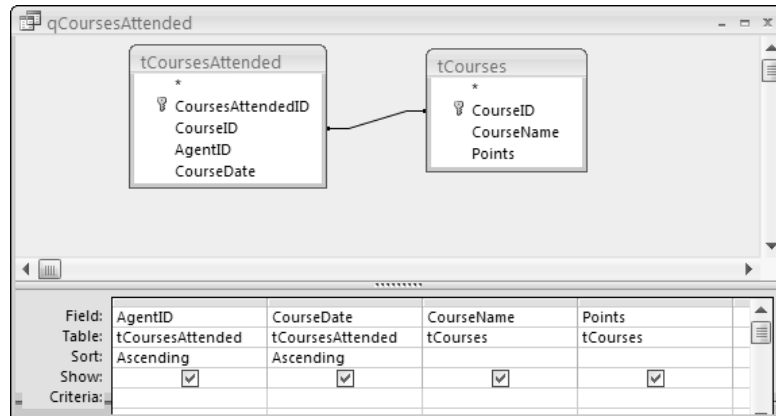


Figure 19-7

In Word, you have prepared a template letter, as shown in Figure 19-8.

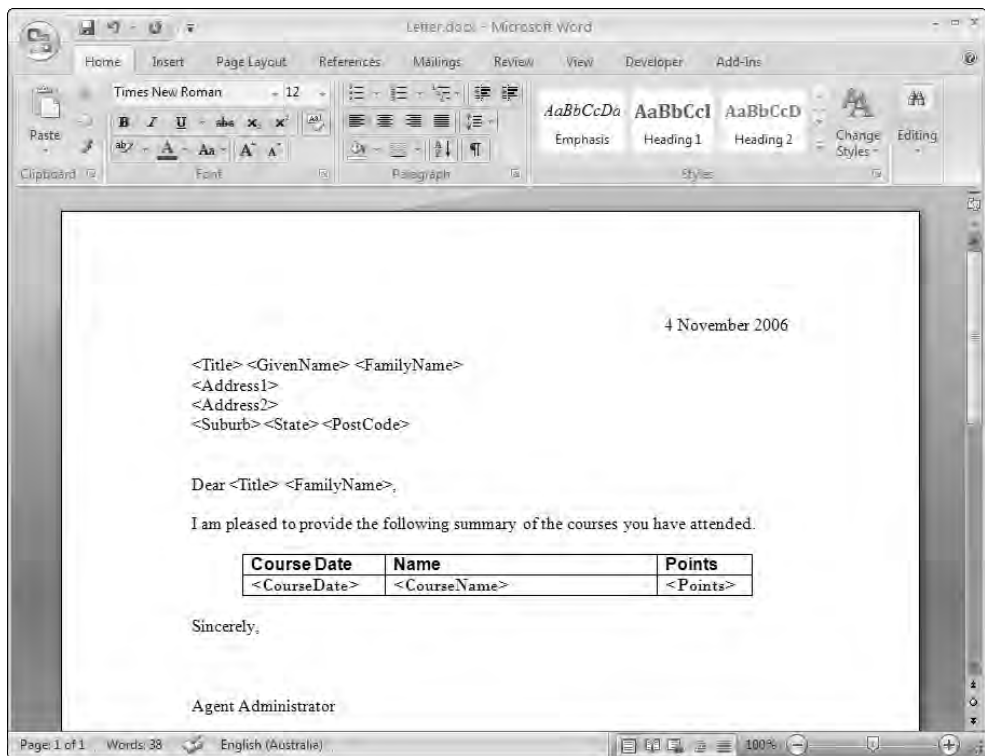


Figure 19-8

Chapter 19: Interacting with Other Office Applications

The items in angle brackets are actually field codes, as shown in Figure 19-9. You can toggle all the field codes in a document between the two views by using Alt+F9. Use Shift+F9 to toggle field codes you have selected. The field codes refer to document variables. You are going to assign values to these document variables using VBA. There is also a bookmark named `CourseTable` that has been inserted in the first line of the table. Its size and position are not relevant, as long as it is somewhere in the table. It serves as a way of identifying which table you are filling with data. The techniques used here will allow you to fill multiple tables in the one document.

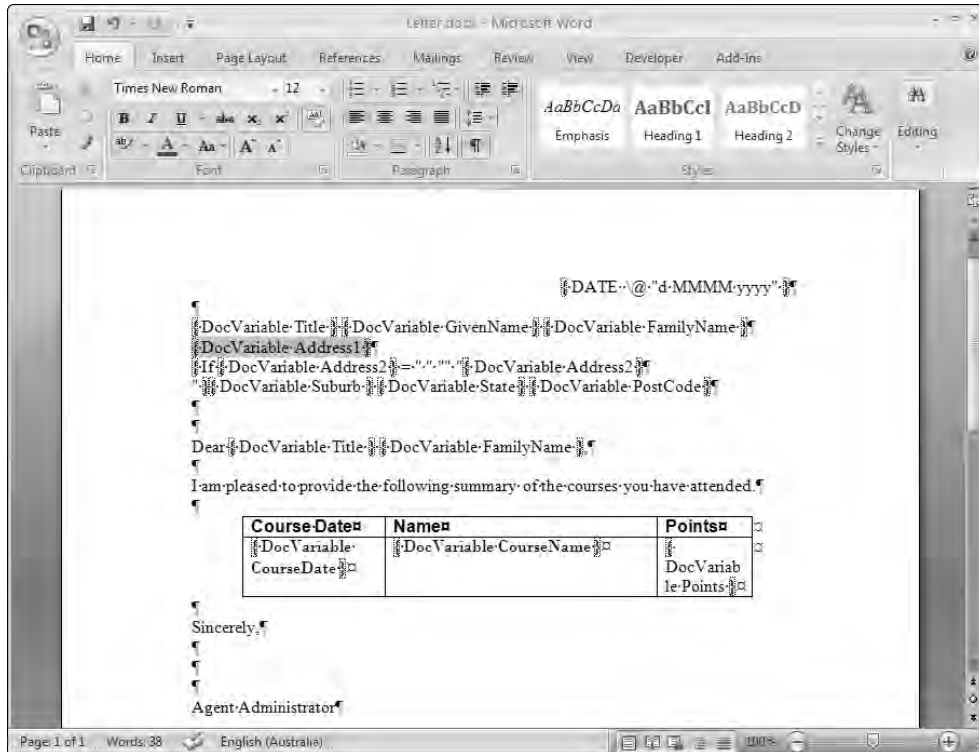


Figure 19-9

The code that extracts the information from the database and fills in the letters is as follows:

```
Sub CreateLetters()  
    Dim wdApp As Word.Application  
    Dim doc As Word.Document  
    Dim tbl As Word.Table  
    Dim wrgCopyRange As Word.Range  
    Dim wrgDestinationRange As Word.Range  
    Dim lPos As Long  
    Dim con As ADODB.Connection  
    Dim rs1 As ADODB.Recordset  
    Dim rs2 As ADODB.Recordset  
    Dim fld As ADODB.Field
```

```
Dim sFolder As String

sFolder = ThisWorkbook.Path
Set con = New ADODB.Connection
con.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=" & sFolder & "\MailMerge.accdb;"
Set rs1 = New ADODB.Recordset
rs1.Open "Select * FROM [tAgents];", con
```

The first part of the code is very similar to that already seen. The coding determines the path to the database by finding the location of the Excel workbook file containing the code, and assumes that the database is in the same location. It creates a recordset containing all the information for each student from `tAgents`. A new instance of Word is opened and made visible so you can watch the action:

```
Set wdApp = New Word.Application
wdApp.Visible = True 'deleting this line will improve performance

'Loop through all the agents
Do Until rs1.EOF
    'Open copy of document
    Set doc = wdApp.Documents.Add(Template:=sFolder & "\Letter.docx")

    'Assign field values to document variables
    For Each fld In rs1.Fields
        If IsNull(fld.Value) Then
            doc.Variables(fld.Name) = " "
        Else
            doc.Variables(fld.Name) = fld.Value
        End If
    Next fld
```

It is better not to make Word visible, because the code will run faster if Word remains hidden. You then loop through all the agents in the database, opening a copy of the form letter for each one. Once again, it is assumed that the template letter is in the same folder as the Excel workbook. For each agent, the code loops through all the fields in the current record of the recordset and feeds the data into the document variables with the same names. This covers all the data outside the table. Null values are converted to a single blank space. Word document variables can't hold a null or a zero-length string. Now locate the table in the document using the bookmark it contains:

```
'Locate the table containing the bookmark "CourseTable"
Set tbl = doc.Bookmarks("CourseTable").Range.Tables(1)
'Second row of table is to be copied
Set wrgCopyRange = tbl.Rows(2).Range

'Create recordset with data for current agent's table
Set rs2 = New ADODB.Recordset
rs2.Open "Select * FROM [qCoursesAttended] WHERE AgentID=" & _
        rs1.AgentID & ";", con

'Loop through the table records for current agent
Do Until rs2.EOF
    For Each fld In rs2.Fields
```

Chapter 19: Interacting with Other Office Applications

```
If IsNull(fld.Value) Then
    doc.Variables(fld.Name) = " "
Else
    doc.Variables(fld.Name) = fld.Value
End If
Next fld
```

Next, create an object variable `wrgCopyRange` that refers to the second row of the table, which you want to copy to the end of the table for each row of data. To get the appropriate data, open a second recordset that selects all the data for the current student and feeds the first record of that data into the document variables in the table. The second row of the table is then copied and pasted to the end of the table:

```
'Copy second row to end of table and lock the current field
'values in the copy
wrgCopyRange.Copy

lPos = tbl.Range.End
Set wrgDestinationRange = doc.Range(lPos, lPos)
wrgDestinationRange.Paste
wrgDestinationRange.Fields.Update
wrgDestinationRange.Fields.Unlink

rs2.MoveNext
Loop
```

The field codes in the copied row are updated with the latest values of the document variables and then unlinked from the variables, which converts them to plain text. The copy process is repeated for all the records in the current recordset. Next, the second row of the table, containing the field codes, is deleted, leaving only the table data:

```
'Delete the second row of the table and lock all
'variable values in document
tbl.Rows(2).Delete
doc.Fields.Update
doc.Fields.Unlink

rs1.MoveNext
Loop

End Sub
```

The fields in the remainder of the document are then updated and unlinked. The code then loops back to create the letter for the next student. You could print each letter and close it before looping back to the next letter, or it could be copied to a new document to form a continuous document with all the letters, as in Word mail merge. In this example, the letters are left as separate documents for viewing.

Readable Document Variables

Figure 19-8 displays each document variable using text that represents the name of the corresponding Access field name in angle brackets. When you enter references to document variables in field codes in a document, the variables have no value and do not display any text. You can assign text values to the document variables to make the document more readable and easier to edit, by running the following code:

```
Sub AddVariables()  
    Dim sFolder As String  
    Dim con As New ADODB.Connection  
    Dim rs As New ADODB.Recordset  
    Dim fld As ADODB.Field  
    Dim wdApp As Word.Application  
    Dim doc As Word.Document  
  
    'Open instance of Word and make it visible  
    Set wdApp = New Word.Application  
    wdApp.Visible = True  
  
    'Get directory of this file  
    sFolder = ThisWorkbook.Path  
  
    'Open the template letter in same directory  
    Set doc = wdApp.Documents.Open(FileName:=sFolder & "\Letter.docx")  
  
    With doc  
  
        'Connect to database in same directory  
        con.Open "Provider=Microsoft.ACE.OLEDB.12.0;" & _  
            "Data Source=" & sFolder & "\MailMerge.accdb;"  
  
        'Open agents table  
        rs.Open "tAgents", con  
  
        'Define document variables with name identical to field name  
        'and value equal to <field name>  
        For Each fld In rs.Fields  
            .Variables(fld.Name) = "<" & fld.Name & ">"  
        Next fld  
  
        rs.Close  
  
        'Repeat operation for courses attended query  
        rs.Open "qCoursesAttended", con  
  
        For Each fld In rs.Fields  
            .Variables(fld.Name) = "<" & fld.Name & ">"  
        Next fld  
  
        .Fields.Update  
  
    End With  
  
End Sub
```

The only substantial difference in this code, compared with `CreateLetters`, is that you open the document to edit it using the `Open` method instead of using the `Add` method to create a copy of the document. When creating the document variables, you assign them values that consist of the field names in angle brackets.

Summary

To automate the objects in another application, you create an object variable referring to the target application or an object in the application. You can use early binding or late binding to establish the link between VBA and the other application's objects. Early binding requires that you establish a reference to the target application's type library, and you must declare any object variables that refer to the target objects using their correct type. If you declare the object variables as the generic `Object` type, VBA uses late binding.

Early binding produces code that executes faster than late binding, and you can get information on the target application's objects using the Object Browser and the shortcut tips that automatically appear as you type your code. Syntax and type checking is also performed as you code, so you are less likely to get errors when the code executes than with late binding, where these checks cannot be done until the code is run.

You must use the `CreateObject` or `GetObject` function to create an object variable reference to the target application when using late binding. You can use the same functions when early binding, but it is more efficient to use the `New` keyword. However, if you want to test for an open instance of another application at run time, `GetObject` can be usefully employed with early binding.

The techniques presented in this chapter allow you to create powerful programs that seamlessly tap into the unique abilities of different products. The user remains in a familiar environment such as Excel, while the code ranges across any product that has a type library and exposes its objects to VBA.

You need to be aware that virus writers can use the information presented here to wreak havoc on unprotected systems. Make sure that your system is adequately covered.

Data Access with ADO

ActiveX Data Objects, or ADO for short, is Microsoft's technology of choice for performing client-server data access between any data consumer (the client) and any data source (the server). There are other data-access technologies you may have heard of in relation to Excel, including DAO and ODBC. However, these are not covered in this chapter because Microsoft intends for ADO to supersede these older technologies, and for the most part this has occurred.

ADO is a vast topic, easily the subject of its own book. This chapter necessarily presents only a small subset of ADO, covering the topics and situations that I've run across most frequently in my career as an Excel programmer. This chapter focuses on ADO 2.5. This version of ADO ships natively with Windows 2000 or Office 2000 and higher, so you can assume it will be present on any computer you distribute your application to.

An Introduction to Structured Query Language (SQL)

It's impossible to get very far into a discussion of data access without running into SQL, the querying language used to communicate with all databases commonly in use today. SQL is a standards-based language that has as many variations as there are databases. This chapter uses constructs compliant with the latest SQL standard, SQL-92, wherever possible.

There are four fundamental operations supported by SQL:

- ❑ `SELECT` — Used to retrieve data from a data source
- ❑ `INSERT` — Used to add new records to a data source
- ❑ `UPDATE` — Used to modify existing records in a data source
- ❑ `DELETE` — Used to remove records from a data source

The terms *record* and *field* are commonly used when describing data. The data sources you'll be concerned with in this chapter can all be thought of as being stored in a two-dimensional grid. A record represents a single row in that grid. A field represents a column in the grid. The intersection

Chapter 20: Data Access with ADO

of a record and a field is a specific value. A *resultset* is the term used to describe the set of data returned by a SQL `SELECT` statement.

You will notice that SQL keywords such as `SELECT` and `UPDATE` are shown in uppercase. This is considered good SQL programming practice. When viewing complex SQL statements, having SQL keywords in uppercase makes it significantly easier to distinguish between those keywords and their operands. The subsections of a SQL statement are called clauses. In all SQL statements, some clauses are required and others are optional. When describing the syntax of SQL statements, optional clauses and keywords will be surrounded by square brackets.

Use the `Customers` table from Microsoft's Northwind sample database, as shown in Figure 20-1, to illustrate the SQL syntax examples. Northwind must be installed with Access 2007 in order to follow many of the examples in this chapter.



ID	Company	First Name	Last Name
1	Company A	Anna	Bedecs
2	Company B	Antonio	Gratacos Solsona
3	Company C	Thomas	Axen
4	Company D	Christina	Lee
5	Company E	Martin	O'Donnell
6	Company F	Francisco	Pérez-Olaeta
7	Company G	Ming-Yang	Xie
8	Company H	Elizabeth	Andersen
9	Company I	Sven	Mortensen
10	Company J	Roland	Wacker

Figure 20-1

The SELECT Statement

The `SELECT` statement is by far the most commonly used statement in SQL. This is the statement that allows you to retrieve data from a data source. The following clauses of the `SELECT` statement are used in this chapter. Only the `SELECT` and `FROM` clauses are required to constitute a valid SQL statement:

```
SELECT [DISTINCT] column1, column2, ...
FROM table_name
[WHERE restriction_condition]
[ORDER BY column_name [ASC|DESC]]
```

The `SELECT` clause tells the data source what fields you wish to return. The field names in the `SELECT` clause are called the `SELECT` list. The `FROM` clause tells the data source which table the records should be retrieved from. For instance, a simple example statement could look like this:

```
SELECT Company, [First Name], [Last Name]
FROM Customers
```

This statement will notify the data source that you want to retrieve all of the values for the `Company`, `First Name`, and `Last Name` fields from the `Customers` table.

Note that the `First Name` and `Last Name` fields in the SQL statement are surrounded by square brackets. This is required for any field or table name that contains spaces or non-alphanumeric characters.

The `SELECT` statement also provides a shorthand method for indicating that you want to retrieve all fields from the specified table. This involves using a single asterisk as the `SELECT` list:

```
SELECT *  
FROM Customers
```

This SQL statement will return all fields and all records from the `Customers` table. It's generally not considered a good practice to use `*` in the `SELECT` list, because it leaves your code vulnerable to changes in field names or the order of fields in the table. It can also be very resource intensive with large tables, because all columns and rows will be returned whether or not they are actually needed by the client. However, there are times when it is a useful and time-saving shortcut.

Say that you want to see a list of countries where you have at least one customer located. Simply performing the following query would return one record for every customer in your table:

```
SELECT [Country/Region]  
FROM Customers
```

This resultset would contain many duplicate country names. The optional `DISTINCT` keyword allows you to return only unique values in your query:

```
SELECT DISTINCT [Country/Region]  
FROM Customers
```

If you only want to see the list of customers located in the U.S., you can use the `WHERE` clause to restrict the results to only those customers:

```
SELECT Company, [First Name], [Last Name]  
FROM Customers  
WHERE [Country/Region] = 'USA'
```

Note that the string literal `USA` must be surrounded by single quotes. This is also true of dates. Numeric expressions do not require any surrounding characters.

Finally, suppose you would like to have your USA customer list sorted by `Company`. This can be accomplished using the `ORDER BY` clause:

```
SELECT Company, [First Name], [Last Name]  
FROM Customers  
WHERE Country = 'USA'  
ORDER BY Company
```

Chapter 20: Data Access with ADO

The `ORDER BY` clause will order fields in ascending order by default. If instead you wanted to sort a field in descending order, you could use the optional `DESC` specifier immediately after the name of the column whose sort order you wanted to modify.

The INSERT Statement

The `INSERT` statement allows you to add new records to a table. The basic syntax of the `INSERT` statement is the following:

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...)
```

Use of the `INSERT` statement is very simple. You provide the name of the table and its columns that you'll be inserting data into, and then provide a list of values to be inserted. You must provide a value in the `VALUES` clause for each column named in the `INSERT` clause, and the values must appear in the same order as the column names they correspond to. Here's an example showing how to insert a new record into the `Customers` table:

```
INSERT INTO Customers (Company, [First Name], [Last Name], [Country/Region])
VALUES ('New Company', 'Rob', 'Bovey', 'USA')
```

Note that as with the `WHERE` clause of the `SELECT` statement, all of the string literals in the `VALUES` clause are surrounded by single quotes. This is the rule throughout SQL.

If you have provided values for every field in the table in your `VALUES` clause, the field list in the `INSERT` clause can be omitted. For example, if the four preceding fields were the only fields in the `Customers` table, you could simply use:

```
INSERT INTO Customers
VALUES ('New Company', 'Rob', 'Bovey', 'USA')
```

The UPDATE Statement

The `UPDATE` statement allows you to modify the values in one or more fields of an existing record or records in a table. The basic syntax of the `UPDATE` statement is the following:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
[WHERE restriction_condition]
```

Even though the `WHERE` clause of the `UPDATE` statement is optional, you must take care to specify it unless you are sure that you don't need it. Executing an `UPDATE` statement *without* a `WHERE` clause will modify the specified field(s) of every record in the specified table. Say, for example, you executed the following statement:

```
UPDATE Customers
SET [Country/Region] = 'USA'
```

Every record in the `Customers` table would have its `Country` field modified to contain the value `USA`. There are some cases where this mass update capability is useful, but it can also be very dangerous, because there is no way to undo the update if you execute it by mistake.

The more common use of the `UPDATE` statement is to modify the value of a specific record, identified by the use of the `WHERE` clause. Before examining an example of this usage, you need to understand a very important aspect of database design called the *primary key*. The primary key is a field or group of fields in a database table whose values can be used to uniquely identify each record in that table. There is no way to identify a specific record in a table that does not have a primary key. Without that capability, you cannot perform an update on a specific record.

The primary key in this sample `Customers` table is the `ID` field. Each customer record in the `Customers` table has a unique value for `ID`. In other words, a specific `ID` value occurs in one, and only one, customer record in the table.

Say that the `First Name` and `Last Name` fields have changed for the customer “Company A”, whose `ID` is 1. You could perform an `UPDATE` to record those changes in the following manner:

```
UPDATE Customers
SET [First Name] = 'First', [Last Name] = 'Last'
WHERE ID = 1
```

Because you used the primary key field to specify a single record in the `Customers` table, only this record will be updated.

The DELETE Statement

The `DELETE` statement allows you to remove one or more records from a table. The basic syntax of the `DELETE` statement is the following:

```
DELETE FROM table_name
[WHERE restriction_condition]
```

As with the `UPDATE` statement, notice that the `WHERE` clause is optional. This is probably more dangerous in the case of the `DELETE` statement, however, because executing a `DELETE` statement without a `WHERE` clause *will delete every single record in the specified table*. Once again, there is no way to undo this, so be very careful. You should always include a `WHERE` clause in your `DELETE` statements unless you have some very specific reason for wanting to remove all records from a table.

Assume that, for some reason, an entry was made into the `Customers` table with the `ID` value of 30 by mistake (maybe they were a supplier rather than a customer). To remove this record from the `Customers` table, you would use the following `DELETE` statement:

```
DELETE FROM Customers
WHERE ID = 30
```

Once again, because you used the record’s primary key in the `WHERE` clause, only that specific record will be affected by the `DELETE` statement.

An Overview of ADO

ADO is Microsoft's universal data-access technology. *Universal* means that ADO is designed to allow access to any kind of data source imaginable, from a SQL Server database to the Windows Active Directory to a text file saved on your local hard disk, and even to non-Microsoft products such as Oracle. All these things and many more can be accessed by ADO.

ADO doesn't actually access a data source directly. Instead, ADO is a data consumer that receives its data from a lower-level technology called *OLE DB*. OLE DB cannot be accessed directly using VBA, so ADO was designed to provide an interface that allows you to do so. ADO receives data from OLE DB providers. Most OLE DB providers are specific to a single type of data source. Each is designed to provide a common interface to whatever data its source may contain. One of the greatest strengths of ADO is that, regardless of the data source you are accessing, you use essentially the same set of commands. There's no need to learn different technologies or methods to access different data sources.

Microsoft also provides an OLE DB provider for ODBC. This general-purpose provider allows ADO to access any data source that understands ODBC, even if a specific OLE DB data provider is not available for that data source. Figure 20-2 shows the communication path between ADO and a data source.

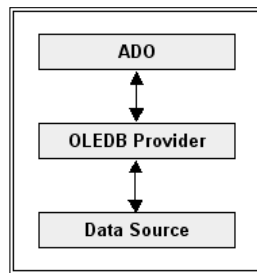


Figure 20-2

Unlike the deep, complex object models of the data access technologies that preceded it, the ADO object model is very flat and simple to understand. It achieves this simplicity without losing any of its power to access and manipulate data.

ADO consists of five top-level objects, all of which can be created independently. This chapter covers the `Connection` object, the `Command` object, and the `Recordset` object. ADO also exposes a `Record` object (not to be confused with the `Recordset` object), as well as a `Stream` object. These objects are not commonly used in Excel applications, so they are not covered in this chapter.

In addition to the five top-level objects, ADO contains four collections. That's it. Five objects and four collections are all you need to master to gain the power of ADO at your fingertips. Figure 20-3 shows the ADO object model.

The next three sections provide an introduction to each of the top-level ADO objects that you'll use in this chapter. These sections provide general information that will be applicable whenever you are using ADO. Specific examples of how to use ADO to accomplish a number of the most common data access tasks you'll encounter in Excel VBA are covered in the sections that follow.

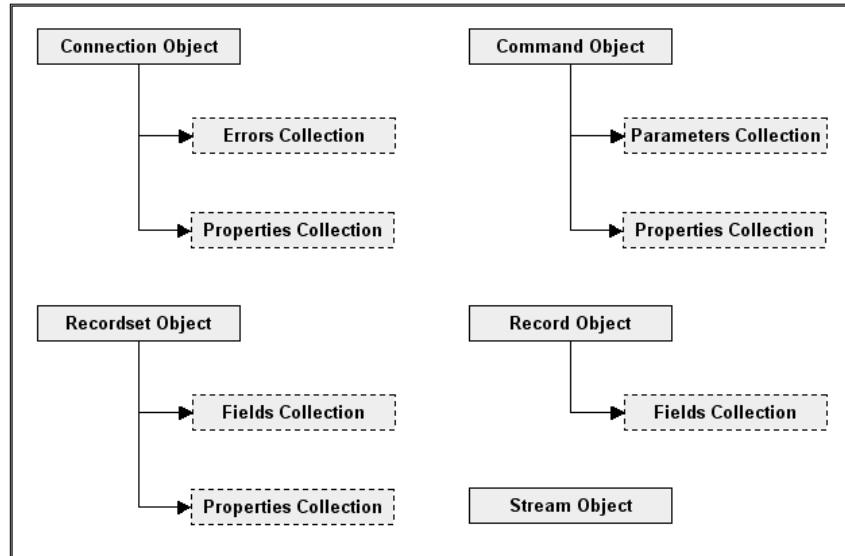


Figure 20-3

This is not intended to be an exhaustive reference to ADO. I will only be covering those items whose use will be demonstrated in this chapter, or those I consider particularly important to point out. ADO frequently gives you the flexibility to make the same setting in multiple ways, as both an object property and an argument to a method, for instance. In these cases, I will usually only cover the method I intend to demonstrate in the example sections.

The Connection Object

The `Connection` object is what provides the pipeline between your application and the data source you want to access. Like the other top-level ADO objects, the `Connection` object is extremely flexible. In some cases, this may be the only object you need to use. Simple commands can easily be executed directly through a `Connection` object. In other cases, you may not need to create a `Connection` object at all. The `Command` and `Recordset` objects can create a `Connection` object automatically if they need one.

Constructing and tearing down a data source connection can be a time-consuming process. If you will be executing multiple SQL statements over the course of your application, you should create a publicly scoped `Connection` object variable and use it for each query. This allows you to take advantage of *connection pooling*.

Connection pooling is a feature provided by ADO that will preserve and reuse connections to the data source rather than creating new connections for each query, which would be a waste of resources. Connections can be reused for different queries as long as their connection strings are identical. This is typically the case in Excel applications, so I recommend taking advantage of it.

Connection Object Properties

This section examines the important `Connection` object properties.

The *ConnectionString* Property

This property is used to provide ADO, and the OLE DB provider you are using, with the information required to connect to the data source. The connection string consists of a semicolon-delimited series of arguments in the form of "name=value;" pairs.

For the purposes of this chapter, the only ADO argument used is the `Provider` argument. The `Provider` argument tells ADO which OLE DB provider to use. All other arguments in connection strings presented in this chapter will be specific to the OLE DB provider being used. ADO will pass these arguments directly through to the provider. The following sample code demonstrates how to create a connection string to the Northwind database using the Access 2007 OLE DB provider:

```
objConn.ConnectionString = "Provider=Microsoft.ACE.OLEDB.12.0;" & _  
    "Data Source=C:\Files\Northwind 2007.accdb"
```

The only argument specific to ADO in the connection string is the `Provider` argument. All other arguments are passed directly through to the specified OLE DB provider. If a different provider were being used, the arguments would be different as well. You will see this when you begin to connect to various data sources in the example sections. The `Provider` argument to the connection string is optional. If no provider is specified, ADO uses the OLE DB provider for ODBC by default.

The *ConnectionTimeout* Property

This property specifies how many seconds ADO will wait for a connection to complete before canceling the attempt and raising an error. The default value is 15 seconds. If you have a situation where connections normally take a long time to complete, you can increase this number so ADO doesn't terminate the connection attempt prematurely. The following code sample changes the timeout value on the connection to 30 seconds:

```
objConn.ConnectionTimeout = 30
```

The *State* Property

The `State` property allows you to determine whether a connection is open, closed, connecting, or executing a command. The value will be a bit mask containing one or more of the following `ObjectStateEnum` constants:

- `adStateClosed`—The connection is closed
- `adStateOpen`—The connection is open
- `adStateConnecting`—The object is in the process of making a connection
- `adStateExecuting`—The connection is executing a command

If you attempt to close a `Connection` object that is already closed, you will cause an error. You can prevent this from occurring by testing the state of the `Connection` object before closing it:

```
If CBool(objConn.State And adStateOpen) Then objConn.Close
```


Connection Object Methods

This section examines the `Connection` object's more important methods, all of which have self-explanatory names.

The Open Method

This method opens a connection to the data source, and has the following syntax:

```
connection.Open connectionString, UserID, Password, Options
```

The `ConnectionString` argument serves the same purpose as the `ConnectionString` property discussed in the previous section. ADO allows you to set this property in advance or pass it in at the time you open the connection. The `UserID` and `Password` arguments can be passed separately from the connection string if you wish.

The `Options` argument is particularly interesting. This argument allows you to make your connection *asynchronously*. That is, you can tell your `Connection` object to go off and open the connection in the background while your code continues to run. You do this by setting the `Options` argument to the `ConnectOptionEnum` value `adAsyncConnect`. The following code sample demonstrates making an asynchronous connection:

```
objConn.Open Options:=adAsyncConnect
```

This is especially useful in situations where you have lengthy connection times, because it allows you to connect without freezing your application during the connection process.

The Execute Method

This method executes the command text provided to its `CommandText` argument. The `Execute` method has the following syntax for an action query (one that does not return a resultset):

```
connection.Execute CommandText, [RecordsAffected], [Options]
```

And for a select query:

```
Set Recordset = connection.Execute(CommandText, _
                                   [RecordsAffected], [Options])
```

The `CommandText` argument can contain any executable string recognized by the OLE DB provider. However, it will most commonly contain a SQL statement. The optional `RecordsAffected` argument is a return value that tells you how many records the `CommandText` operation has affected. It's a good idea to check this value against the number of records that you expected to be affected, so you can detect potential errors in your command text.

The `Options` argument is crucial to optimizing the execution efficiency of your command. Therefore, you should always use it even though it's nominally optional. The `Options` argument allows you to relay two different types of information to your OLE DB provider: what type of command is contained in the `CommandText` argument, and how the provider should execute the contents of the `CommandText` argument.

Chapter 20: Data Access with ADO

To execute the `CommandText`, the OLE DB provider must know what type of command it contains. If you don't specify the type, the provider will have to determine that information for itself. This will slow down the execution of your query. You can avoid this by specifying the `CommandText` type using one of the following `CommandTypeEnum` values:

- ❑ `adCmdText` — The `CommandText` is a raw SQL string.
- ❑ `adCmdTable` — The `CommandText` is the name of a table. This sends an internally generated SQL statement to the provider that looks something like "SELECT * FROM table_name".
- ❑ `adCmdStoredProc` — The `CommandText` is the name of a stored procedure (stored procedures are covered in the section "Using ADO with Microsoft SQL Server").
- ❑ `adCmdTableDirect` — The `CommandText` is the name of a table. However, unlike `adCmdTable`, this option does not generate a SQL statement and therefore returns the contents of the table more efficiently. Use this option if your provider supports it.

You can provide specific execution instructions to the provider by including one or more of the `ExecuteOptionEnum` constants:

- ❑ `adAsyncExecute` — Tells the provider to execute the command asynchronously, which returns execution to your code immediately.
- ❑ `adExecuteNoRecords` — Tells the provider not to construct a `Recordset` object. ADO will always construct a recordset in response to a command, even if your `CommandText` argument is not a row-returning query. To avoid the overhead required to create an unnecessary recordset, use this value in the `Options` argument whenever you execute a non-row-returning query.

The `CommandTypeEnum` and `ExecuteOptionEnum` values are bit masks that can be combined together in the `Options` argument using the logical `Or` operator. For example, to execute a plain text SQL command and tell ADO not to construct a `Recordset` object, you would use the following syntax:

```
szSQL = "DELETE FROM Customers WHERE CustomerID = 'XXXX'"
objConn.Execute szSQL, lNumAffected, adCmdText Or adExecuteNoRecords
If lNumAffected <> 1 Then MsgBox "Error executing SQL statement."
```

The Close Method

This method closes the connection to the data source. Simply closing the connection does not destroy the `Connection` object. To destroy the `Connection` object and free its memory, you need to set the `Connection` object variable to `Nothing`. For example, to ensure that a `Connection` object variable is closed and removed from memory, you would execute the following code:

```
If CBool(objConn.State And adStateOpen) Then objConn.Close
Set objConn = Nothing
```

Connection Object Events

Connection object events must be trapped by creating a `WithEvents` `Connection` object variable in a class module. Trapping these events is necessary whenever you are using a `Connection` object asynchronously, because these events are what notify your application that the `Connection` object has completed its task.

Covering asynchronous connections is beyond the scope of this chapter. However, they are important enough to deserve mention so you can pursue them further if you like. The two most commonly used `Connection` events are:

- ❑ `ConnectComplete`—Triggered when an asynchronous connection has been completed. You can examine the arguments passed to this event to determine if the connection was successful or not.
- ❑ `ExecuteComplete`—Triggered when an asynchronous command has finished executing.

Connection Object Collections

The `Connection` object has two collections, `Errors` and `Properties`.

Errors Collection

This collection contains a set of `Error` objects, each of which represents an OLE DB provider-specific error (ADO itself generates run-time errors). The `Errors` collection can contain not only errors, but also warnings and even messages (generated by the T-SQL `PRINT` statement, for instance). The `Errors` collection is very helpful in providing extra detail when something in your ADO code has malfunctioned. When debugging ADO problems, you can dump the contents of the `Errors` collection to the Immediate window with the following code:

```
For Each objError In objConn.Errors
    Debug.Print objError.Description
Next objError
```

The Properties Collection

This collection contains provider-specific, or *extended properties*, for the `Connection` object. Some providers add important settings that you will want to be aware of. Extended properties are beyond the scope of this chapter.

The Recordset Object

Just as the most commonly used SQL statement is the `SELECT` statement, the most commonly used ADO object is the `Recordset` object. The `Recordset` object serves as a container for the records and fields returned from a `SELECT` statement executed against a data source.

Recordset Object Properties

The examination of the `Recordset` object begins with a look at its important properties.

The ActiveConnection Property

Prior to opening the `Recordset` object, you can use the `ActiveConnection` property to assign an existing `Connection` object to the `Recordset` or a connection string for the recordset to use to connect to the database. If you assign a connection string, the recordset will create a `Connection` object for itself. Once the recordset has been opened, this property returns an object reference to the `Connection` object being used by the recordset.

Chapter 20: Data Access with ADO

The following code assigns a `Connection` object to the `ActiveConnection` property:

```
Set rsData.ActiveConnection = objConn
```

The following code assigns a connection string to the `ActiveConnection` property:

```
rsData.ActiveConnection = "Provider=Microsoft.ACE.OLEDB.12.0;" & _  
    "Data Source=C:\Files\Northwind 2007.accdb"
```

The **BOF** and **EOF** Properties

These properties indicate whether the record pointer of the `Recordset` object is positioned before the first record in the recordset (`BOF`, or beginning of file) or after the last record in the recordset (`EOF`, or end of file). If the recordset is empty, both `BOF` and `EOF` will be `True`. The following code example demonstrates using these properties to determine if there is data in a recordset:

```
If Not rsData.EOF Then  
    ' The Recordset contains data.  
Else  
    ' The Recordset is empty.  
End If
```

Note that there is a difference between an empty recordset and a closed recordset. If you execute a query that returns no data, ADO will present you with a perfectly valid open recordset, but one that contains no data. Therefore, you should always verify that a recordset contains data using the previous method prior to any attempt to access that data. Attempting to access data from an empty recordset will cause a run-time error.

The **CursorLocation** Property

This property allows you to specify whether the server-side cursor engine or the client-side cursor engine manages the records in the recordset. A cursor is the underlying object that manages the data in the recordset. Certain operations require the cursor engine to be on one side or the other. This is explored in more detail in the examples section.

This property must be set before the recordset is opened. If you do not specify a cursor location, the default is server-side. You can set this property to one of the two `CursorLocationEnum` values, either `adUseClient` or `adUseServer`. The following code sample demonstrates setting the cursor location to client-side:

```
rsData.CursorLocation = adUseClient
```

The **Filter** Property

This property allows you to filter an open recordset so that only records that meet the specified condition are visible. The records that cannot be seen are not deleted, removed, or changed in any way, but are simply hidden from normal recordset operations. This property can be set to a string that specifies the filter you want to place on the records, or to one of the `FilterGroupEnum` constants.

You can set multiple filters, and the records exposed by the filtered recordset will be only those records that meet all of the conditions. To remove the filter from the recordset, set the `Filter` property to an

empty string or the `adFilterNone` constant. The following code sample demonstrates filtering a recordset so that it displays only company names that begin with the letter B. Note that the use of wildcards is supported in a filter string:

```
rsData.Filter = "Company LIKE 'B*'"
```

You can use the logical AND, OR, and NOT operators to set additional `Filter` property values:

```
rsData.Filter = "Company LIKE 'B*' AND [Country/Region] = 'USA'"
```

The State Property

This is the same as the `State` property discussed in the section on the `Connection` object.

Recordset Object Methods

This section examines only five of the `Recordset` object's methods, because they are the ones that you are most likely to use.

The Open Method

This method opens the `Recordset` object and retrieves the data specified by the `Source` argument. The `Open` method has the following syntax:

```
recordset.Open Source, ActiveConnection, CursorType, LockType, Options
```

The `Source` argument tells the recordset what data it should retrieve. This is most commonly a SQL string or the name of a stored procedure, but can also be the name of a table or a `Command` object.

The `ActiveConnection` argument can be a connection string or a `Connection` object that identifies the connection to be used. If you assign a connection string to the `ActiveConnection` argument, the recordset will create a `Connection` object for itself.

The `CursorType` argument specifies the type of cursor to use when opening the recordset. This is set using one of the `CursorTypeEnum` values. This chapter uses only the `adOpenForwardOnly` and `adOpenStatic` cursor types. The first type will be used for normal queries, and using it means that the recordset can be navigated in only one direction, from beginning to end, which is the fastest method for accessing data. Note that the data in a forward-only recordset cannot be modified. The second type will be used for disconnected recordsets and allows complete navigation. If you do not specify a cursor type, `adOpenForwardOnly` is the default.

The `LockType` argument specifies what type of locks the provider should place on the underlying data source when opening the recordset. This is set using one of the `LockTypeEnum` values. This chapter uses only the following two lock types, corresponding to normal and disconnected recordsets, respectively: `adLockReadOnly` and `adLockBatchOptimistic`.

The `Options` argument here is the same as the `Options` argument covered in the `Connection` object's `Execute` method earlier in the chapter. It is used to tell the provider how to interpret and execute the contents of the `Source` argument.

The Close Method

This method closes the `Recordset` object. This does not free any memory used by the recordset. To free up the memory used by the `Recordset` object, you must set the `Recordset` object variable to `Nothing`.

The Move Methods

When a recordset is first opened, the *current record pointer* is positioned on the first record in the recordset. The `Move` methods are used to navigate through the records in an open recordset. They do this by repositioning the `Recordset` object's current record pointer. The following `Move` methods are used in this chapter:

- ❑ `MoveFirst` — Positions the current record pointer on the first record of the recordset.
- ❑ `MoveNext` — Positions the current record pointer to the next record in the recordset.

The following code sample demonstrates common recordset navigation handling:

```
' Verify that the Recordset contains data.
If Not rsData.EOF Then
    ' Loop until we reach the end of the Recordset.
    Do While Not rsData.EOF
        ' Perform some action on the current record's data.
        Debug.Print rsData.Fields(0).Value
        ' Move to the next record.
        rsData.MoveNext
    Loop
Else
    MsgBox "Error, no records returned.", vbCritical
End If
```

Pay particular attention to the use of the `MoveNext` method within the `Do While` loop. Omitting this is a very common error and will lead to an endless loop condition in your code. The very first line of code that you should place in the `Do While` loop is the call to `MoveNext`.

The NextRecordset Method

Some providers allow you to execute commands that return multiple recordsets. The `NextRecordset` method is used to move through these recordsets. The `NextRecordset` method clears the current recordset from the `Recordset` object, loads the next recordset into the `Recordset` object, and sets the current record pointer to the first record in that recordset. If the `NextRecordset` method is called and there are no more recordsets to retrieve, the `Recordset` object is set to `Nothing`. The following code sample demonstrates the use of the `NextRecordset` method:

```
' Verify that the Recordset contains more data.
Do While Not rsData Is Nothing
    ' Loop the records in the current recordset.
    Do While Not rsData.EOF
        ' Perform some action on the current record's data.
        Debug.Print rsData.Fields(0).Value
        ' Move to the next record.
        rsData.MoveNext
    Loop
```

```
' Return the next recordset
Set rsData = rsData.NextRecordset
Loop
```

Recordset Object Events

Recordset object events must be trapped by creating a `WithEvents Recordset` object variable in a class module. Trapping these events is necessary whenever you are using a `Recordset` object asynchronously, because these events are what notify your application that the `Recordset` object has completed its task.

Covering asynchronous recordset usage is beyond the scope of this chapter; however, the topic is important enough to deserve mention so that you can pursue it further if you like. The two most commonly used `Recordset` object events are:

- ❑ `FetchComplete` — This event is fired after all of the records have been retrieved when opening an asynchronous recordset.
- ❑ `FetchProgress` — The provider fires this event periodically to report the number of records retrieved so far during an asynchronous open operation. It is typically used to provide a visual progress indicator to the user.

Recordset Object Collections

Finish your look at the `Recordset` object by examining its collections.

The Fields Collection

The `Fields` collection contains the values, and information about those values, from the current record in a `Recordset` object. In Excel, the `Fields` collection is most commonly used to return the column names of each field in the recordset, prior to accessing the contents of the recordset using the `CopyFromRecordset` method of the `Range` object. The following example demonstrates how to read the field names from the `Fields` collection of a `Recordset` object:

```
With Sheet1.Range("A1")
  For Each objField In rsData.Fields
    .Offset(0, lOffset).Value = objField.Name
    lOffset = lOffset + 1
  Next objField
End With
```

The Properties Collection

This collection contains provider-specific or extended properties for the `Recordset` object. Some providers add important settings — called *extended properties* — that you will want to be aware of. The most important extended properties of each provider are covered in that provider's section.

The Command Object

The `Command` object is most commonly used for executing action queries. Action queries are queries that perform some action on the data source and do not return a resultset. Action queries include `INSERT`, `UPDATE`, and `DELETE` statements.

Command Object Properties

Begin by looking at the three most important `Command` object properties.

The `ActiveConnection` Property

This property is identical to the `ActiveConnection` property discussed in the section on the `Recordset` object.

The `CommandText` Property

This property is used to set the command that will be executed by the data provider. This property will normally be a SQL string or the name of a stored procedure. As you will see in the section on SQL Server, you must use the `CommandText` property along with the `Parameters` collection to take advantage of return values and output parameters in SQL Server stored procedures.

The `CommandType` Property

The `CommandType` property is identical to the `Options` argument to the `Connection` object's `Execute` method, covered earlier in the chapter. It is used to tell the provider how to interpret and execute the `Command` object's `CommandText`.

Command Object Methods

Only two of the `Command` object's methods are examined, because they are the most commonly used.

The `CreateParameter` Method

This method is used to manually create `Parameter` objects that can then be added to the `Command` object's `Parameters` collection. The `CreateParameter` object has the following syntax:

```
Set Parameter = command.CreateParameter([Name], [Type], [Direction], _  
                                         [Size], [Value])
```

`Name` is the name of the parameter object. You can use this name to reference the `Parameter` object through the `Command` object's `Parameters` collection. When working with SQL Server, the name of a `Parameter` should be the same as the name of the stored procedure argument that it corresponds to.

`Type` indicates the data type of the parameter. It is specified as one of the `DataTypeEnum` constants. There are several dozen possible data types, so I will not go into them in any detail here. You will see a few of them in the examples section. The rest can be located in the ADO help file.

`Direction` is a `ParameterDirectionEnum` value that indicates whether the parameter will be used to pass data to an input argument, receive data from an output argument, or accept a return value from a stored procedure. `Direction` can be one of the following values:

- `adParamInput` — The parameter represents an input argument
- `adParamInputOutput` — The parameter represents an input/output argument
- `adParamOutput` — The parameter represents an output argument
- `adParamReturnValue` — The parameter represents a return value

Size is used to specify the size of Parameter in bytes, and is dependent on Parameter's data type.

Value is used to provide an initial value for the Parameter.

The following code sample demonstrates how you can use the CreateParameter method in conjunction with the Parameters collection Append method to create a Parameter and append it to the Parameters collection with one line of code:

```
objCmd.Parameters.Append _  
    objCmd.CreateParameter("MyParam", adInteger, adParamInput, 0)
```

The Execute Method

This method executes the text contained in the Command object's CommandText property. The Execute method has the following syntax for an action query (one that does not return a resultset):

```
command.Execute [RecordsAffected], [Parameters], [Options]
```

And for a select query:

```
Set Recordset = command.Execute([RecordsAffected], [Parameters], [Options])
```

The RecordsAffected and Options arguments are identical to the corresponding arguments for the Connection object's Execute method, described in the "Connection Object Methods" section. If you are executing a SQL statement that requires one or more parameters to be passed, you can supply an array of values to the Parameters argument, one for each parameter required.

Command Object Collections

The final section before delving into ADO in Excel covers the Command object's two collections.

The Parameters Collection

This collection contains all of the Parameter objects associated with the Command object. Parameters are used to pass arguments to SQL statements and stored procedures, as well as to receive output and return values from stored procedures.

The Properties Collection

This collection contains provider-specific or extended properties for the Command object. Some providers add important settings that you will want to be aware of. The most important extended properties of each provider are covered in the provider-specific discussions later in the chapter.

Using ADO in Microsoft Excel Applications

Here's where it all comes together. This section combines the understanding of Excel programming that you've gained from previous chapters with the SQL and ADO techniques discussed so far in this chapter. Excel applications frequently require data from outside sources. The most common of these sources are Access and SQL Server databases. However, I've created applications that required source data from mainframe text file dumps and even Excel workbooks. As you'll see, ADO makes acquiring data from these various data sources easy.

Chapter 20: Data Access with ADO

To run the code examples shown in the sections that follow, you must set a reference from your Excel project to the ADO 2.5 Object Library. To do this, bring up the References dialog by selecting the Tools → References menu item from within the VBE. Scroll down until you locate the entry labeled Microsoft ActiveX Data Objects 2.5 Library. Place a check mark beside this entry and click OK (see Figure 20-4).

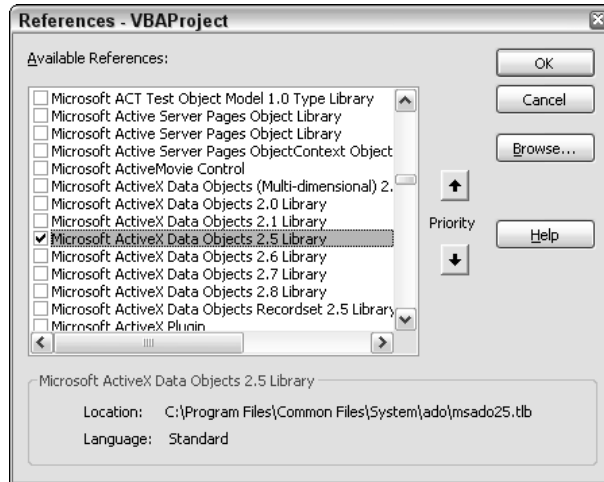


Figure 20-4

Note that it's perfectly normal to have multiple versions of the ADO object library available.

Using ADO with Microsoft Access

Because Excel applications that utilize data from Microsoft Access tend to have less complicated data access requirements than those using SQL Server, Access is used to introduce the basics of ADO.

In this section you'll utilize the Northwind database. This is a sample database provided with Microsoft Access 2007. If you don't have this database available, you will need to install it to run the example code.

Connecting to Microsoft Access

ADO connects to Microsoft Access databases through the use of the Microsoft Office 12 Access Database Engine OLE DB Provider. To connect to a Microsoft Access database, you simply specify this provider in the ADO connection string and then include any additional provider-specific arguments required. The following is a summary of the connection string arguments you will most frequently use when connecting to an Access database:

- ❑ `Provider=Microsoft.ACE.OLEDB.12.0` (required)
- ❑ `Data Source=[full path and filename to the Access database]` (required)

- ❑ `Mode=mode` (optional)—The three most commonly used settings for this property are:
 - ❑ `adModeShareDenyNone`—Opens the database and allows complete shared access to other users. This is the default setting if the `Mode` argument is not specified.
 - ❑ `adModeShareDenyWrite`—Opens the database and allows other users read access but prevents write access.
 - ❑ `adModeShareExclusive`—Opens the database in exclusive mode, which prevents any other users from connecting to the database.
- ❑ `User ID=username` (optional)—The username to use when connecting to the database. If the database requires a username and it is not supplied, the connection will fail.
- ❑ `Password=password` (optional)—If a password is required to connect to the database, this argument is used to supply it. Again, the connection will fail if it is required and not supplied, or is incorrect.

The following example shows a connection string that uses all of these arguments:

```
Public Const gsCONNECTION As String = _
    "Provider= Microsoft.ACE.OLEDB.12.0;" & _
    "Data Source=C:\Files\Northwind 2007.accdb;" & _
    "Mode=Share Exclusive;" & _
    "User ID=Admin;" & _
    "Password=password"
```

Retrieving Data from Microsoft Access Using a Plain Text Query

The following procedure demonstrates how to retrieve data from a Microsoft Access database using a plain text (sometimes referred to as ad hoc) query and place it on an Excel worksheet:

```
Public Sub PlainTextQuery()

    Dim rsData As ADODB.Recordset
    Dim sConnect As String
    Dim sSQL As String

    ' Create the connection string.
    sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=C:\Files\Northwind 2007.accdb"

    ' Create the SQL Statement.
    sSQL = "SELECT Company, [First Name] + ' ' + [Last Name] " & _
        "FROM Customers " & _
        "WHERE [Country/Region] = 'USA' " & _
        "ORDER BY Company;"

    ' Create the Recordset object and run the query.
    Set rsData = New ADODB.Recordset
    rsData.Open sSQL, sConnect, adOpenForwardOnly, _
        adLockReadOnly, adCmdText

    ' Make sure we got records back
```

```
If Not rsData.EOF Then
    ' Dump the contents of the recordset onto the worksheet.
    Sheet1.Range("A2").CopyFromRecordset rsData
    ' Close the Recordset object.
    rsData.Close
    ' Add headers to the worksheet.
    With Sheet1.Range("A1:B1")
        .Value = Array("Company", "Contact Name")
        .Font.Bold = True
    End With
    ' Fit the column widths to the data.
    Sheet1.UsedRange.EntireColumn.AutoFit
Else
    ' Close the Recordset object.
    rsData.Close
    MsgBox "Error: No records returned.", vbCritical
End If

' Destroy the Recordset object.
Set rsData = Nothing

End Sub
```

There are a number of things to note about this procedure:

- ❑ The only ADO object used was the `Recordset` object. As mentioned at the beginning of the ADO section, all of the top-level ADO objects can be created and used independently. If you were going to perform multiple queries over the course of your application, you would have created a separate, publicly scoped `Connection` object to take advantage of ADO's connection-pooling feature.
- ❑ The syntax of the `Recordset.Open` method has been optimized for maximum performance. You've told the provider what type of command is in the `Source` argument (`adCmdText`, a plain text query), and you've opened a forward-only, read-only, server-side cursor (server-side is the default if the `Recordset.CursorLocation` property is not specified). This type of cursor is often referred to as a *firehose cursor*, because it's the fastest way to retrieve data from a database.
- ❑ You do not make any modifications to the destination worksheet until you are sure you have successfully retrieved data from the database. This avoids having to undo anything if the query fails.
- ❑ You dump the data onto the destination worksheet and close the recordset as quickly as possible. In most data-access situations, you will be dealing with a multi-user environment, where multiple users can access the database simultaneously. Getting in and out of the database as quickly as possible is critical to preventing contention problems, or situations in which two users attempt to perform mutually incompatible actions on the same piece of data at the same time.
- ❑ Note the use of the `CopyFromRecordset` method of the Excel `Range` object. This is by far the fastest method for moving data out of a recordset and onto a worksheet. As you'll see, it doesn't fit every data-access situation, but it's the method of choice in any situation where its use is possible. Note that the `CopyFromRecordset` method does not copy the field names, only data.

Retrieving Data from Microsoft Access Using a Stored Query

Microsoft Access allows you to create and store SQL queries in the database. You can retrieve data from these stored queries just as easily as you can use a plain text SQL statement. The following procedure demonstrates this:

```
Public Sub SavedQuery()

    Dim objField As ADODB.Field
    Dim rsData As ADODB.Recordset
    Dim lOffset As Long
    Dim sConnect As String

    ' Create the connection string.
    sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=C:\Files\Northwind 2007.accdb"

    ' Create the Recordset object and run the query.
    Set rsData = New ADODB.Recordset
    rsData.Open "[Product Sales By Category]", sConnect, _
        adOpenForwardOnly, adLockReadOnly, adCmdTable

    ' Make sure we got records back
    If Not rsData.EOF Then
        ' Add headers to the worksheet.
        With Sheet1.Range("A1")
            For Each objField In rsData.Fields
                .Offset(0, lOffset).Value = objField.Name
                lOffset = lOffset + 1
            Next objField
            .Resize(1, rsData.Fields.Count).Font.Bold = True
        End With
        ' Dump the contents of the recordset onto the worksheet.
        Sheet1.Range("A2").CopyFromRecordset rsData
        ' Close the Recordset object.
        rsData.Close
        ' Fit the column widths to the data.
        Sheet1.UsedRange.EntireColumn.AutoFit
    Else
        ' Close the Recordset object.
        rsData.Close
        MsgBox "Error: No records returned.", vbCritical
    End If

    ' Destroy the Recordset object.
    Set rsData = Nothing

End Sub
```

There are two important points to note about this procedure:

- Examine the differences between the `Recordset.Open` method used in this procedure and the one used in the plain text query. In this case, rather than providing a SQL string, you specified the name of the stored query you wanted to execute. You also told the provider that the type of

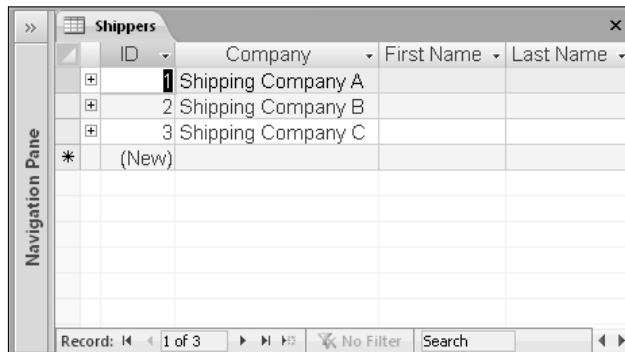
query being executed was a table query. The Access OLE DB provider treats stored queries and queries of entire database tables in the same manner.

- Because you did not create the SQL statement yourself, you did not know the names of the fields you were retrieving, or even how many fields there were. Therefore, to create the correct set of headers for each column in the destination worksheet, you needed to loop the `Fields` collection of the `Recordset` object and determine this information dynamically. To accomplish this, the recordset had to be open, so you added the fields to the worksheet prior to closing the recordset.

Inserting, Updating, and Deleting Records with Plain Text SQL in Microsoft Access

Executing plain text `INSERT`, `UPDATE`, and `DELETE` statements uses virtually identical methodology. Therefore, you'll examine these action queries by inserting a new record, updating that record, and then deleting it, all within the same procedure. This, of course, is not normally something you would do. You can take this generic procedure, however, and create a single-purpose insert, update, or delete procedure by simply removing the sections that you don't need.

Use the `Shippers` table from the Northwind database in the next procedure. The first few columns of this table are shown in Figure 20-5.



ID	Company	First Name	Last Name
1	Shipping Company A		
2	Shipping Company B		
3	Shipping Company C		
*	(New)		

Figure 20-5

Notice that the last row in the `ID` column contains the value `(New)`. This isn't really a value; rather, it's a prompt that alerts you to the fact that values for the `ID` column are automatically generated by the Access database. This column is the primary key for the `Shippers` table, and `AutoNumber` fields are a common method used to generate the unique value required for the primary key. You don't (and can't) set or change the value of an `AutoNumber` field. If you need to maintain a reference to a new record that you've inserted into the table, you'll need to retrieve the value that was assigned to that record by the `AutoNumber` field. You'll see how this is done in the example that follows:

```
Public Sub InsertUpdateDelete()  
  
    Dim objCommand As ADODB.Command  
    Dim rsData As ADODB.Recordset  
    Dim lRecordsAffected As Long  
    Dim lKey As Long
```

```

Dim sConnect As String

On Error GoTo ErrorHandler

' Create the connection string.
sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
    "Data Source=C:\Files\Northwind 2007.accdb;" & _
    "Mode=Share Exclusive"

' Create the Command object we'll use for all three queries.
Set objCommand = New ADODB.Command
objCommand.ActiveConnection = sConnect

'''' INSERT a new record into the database ''''''''''''''''''''''''''''''''''''''''
' Load the SQL string into the Command object.
objCommand.CommandText = "INSERT INTO Shippers" & _
    "(Company, [Business Phone]) " & _
    "VALUES('Air Carriers', '(205) 555-1212');"
' Execute the SQL statement.
objCommand.Execute RecordsAffected:=lRecordsAffected, _
    Options:=adCmdText Or adExecuteNoRecords
' Check for errors. Only one record should have been affected.
If lRecordsAffected <> 1 Then Err.Raise _
    Number:=vbObjectError + 1024, _
    Description:="Error executing INSERT statement."

' Retrieve the primary key generated for our new record.
objCommand.CommandText = "SELECT @@IDENTITY;"
Set rsData = objCommand.Execute(Options:=adCmdText)
' Check for errors. The recordset should contain data.
If rsData.EOF Then Err.Raise _
    Number:=vbObjectError + 1024, _
    Description:="Error retrieving primary key value."
' Store the primary key value for later use.
lKey = rsData.Fields(0).Value
rsData.Close

'''' UPDATE the record we just created ''''''''''''''''''''''''''''''''''''''''
' Load the SQL string into the Command object.
objCommand.CommandText = "UPDATE Shippers " & _
    "SET [Business Phone]='(206) 546-0086' " & _
    "WHERE ID=" & CStr(lKey) & ";"
' Execute the SQL statement.
objCommand.Execute RecordsAffected:=lRecordsAffected, _
    Options:=adCmdText Or adExecuteNoRecords
' Check for errors. Only one record should have been affected.
If lRecordsAffected <> 1 Then Err.Raise _
    Number:=vbObjectError + 1024, _
    Description:="Error executing UPDATE statement."

'''' DELETE our record from the database ''''''''''''''''''''''''''''''''''''''''
' Load the SQL string into the Command object.
objCommand.CommandText = "DELETE FROM Shippers " & _
    "WHERE ID = " & CStr(lKey) & ";"
' Execute the SQL statement.
objCommand.Execute RecordsAffected:=lRecordsAffected, _

```

```
Options:=adCmdText Or adExecuteNoRecords
' Check for errors. Only one record should have been affected.
If lRecordsAffected <> 1 Then Err.Raise _
    Number:=vbObjectError + 1024, _
    Description:="Error executing DELETE statement."

ErrorExit:

' Destroy our ADO objects.
Set objCommand = Nothing
Set rsData = Nothing

Exit Sub

ErrorHandler:
MsgBox Err.Description, vbCritical
Resume ErrorExit
End Sub
```

Quickly review what you've done in this procedure. First you inserted a new record into the `Shippers` table. Then you retrieved the primary key that had been assigned to that new record by the database (more on this in a moment). Next you used the primary key of your new record to locate it and modify the telephone number in its `Business Phone` field. Finally, you used the primary key of the record to locate it and delete it from the database.

Important things to note about this procedure:

- ❑ You used the same ADO `Command` object throughout the procedure. All that was required to execute different commands was to load new SQL statements into the `Command.CommandText` property.
- ❑ The process of preparing and executing the command and then checking for errors upon completion was identical for all three types of action query.
- ❑ After inserting a new record in the first part of the procedure, you needed to retrieve the primary key value that had been assigned to that record by the `ID AutoNumber` field. You did this by querying the value of the `@@IDENTITY` system variable. This is a variable maintained by the Access database that holds the value of the most recently assigned `AutoNumber` field. In most cases, you must be sure to query this value immediately after performing the insert. The `@@IDENTITY` variable is a database-wide variable, so your primary key value will be overwritten if any other user performs a similar insert before you query it. You prevented the possibility of this occurring in this example by opening the database in exclusive mode (note the `Mode` argument in the connection string).

Using ADO with Microsoft SQL Server

The previous section on Microsoft Access covered the basics of performing the various types of queries in ADO. Because ADO is designed to present a common gateway to different data sources, there isn't a lot of difference in these basic operations whether your database is in Access or in SQL Server. Therefore, after a brief introduction to the few important differences that arise when using ADO with SQL Server, this section covers more advanced topics, including stored procedures, multiple recordsets, and disconnected recordsets.

The examples in this section use the SQL Server version of the Northwind sample database. This database is similar to the Access 2007 version of Northwind used previously, but there will be some differences in the names and data types of various fields.

Connecting to Microsoft SQL Server

To connect to a Microsoft SQL Server database, you simply specify the OLE DB provider for SQL Server in the ADO connection string, and then include any additional provider-specific arguments required. The following is a summary of the connection string arguments you will most frequently use when connecting to a SQL Server database:

- ❑ `Provider=SQLOLEDB;`
- ❑ `Data Source=server name;` — This will typically be the NetBIOS name of the computer that SQL Server is installed on. If SQL Server is installed as a named instance, the server name will have the following syntax: `NetBIOS Name\SQL Server Instance Name`.
- ❑ `Initial Catalog=database name;` — Unlike Access, one instance of SQL Server can contain many databases. This argument will contain the name of the database you want to connect to.
- ❑ `User ID=username;` — The username for SQL Server authentication.
- ❑ `Password=password;` — The password for SQL Server authentication.
- ❑ `Network Library=netlib;` — By default, the SQL Server OLE DB provider will attempt to use *named pipes network protocol* to connect to SQL Server. This is required for using Windows integrated security (explained later). There are many instances, however, where it is not possible to use named pipes. These include accessing SQL Server from a Windows 9x operating system and accessing SQL Server over the Internet. In these cases, the preferred protocol for connecting to SQL Server is TCP/IP. This can be specified on each machine by using the SQL Server Client Network Utility, or you can simply use the Network Library connection string argument to specify the name of the TCP/IP network library, which is `dbmssocn`.
- ❑ `Integrated Security=SSPI;` — This connection string argument specifies that you want to use Windows integrated security rather than SQL Server authentication. The `User ID` and `Password` arguments will be ignored if this argument is present.

A Note About SQL Server Security

SQL Server can be set to use three types of security: SQL Server authentication, Windows integrated security, and mixed mode. SQL Server authentication means that separate user accounts must be added to SQL Server, and each user must supply a SQL Server username and password to connect.

This type of security is most commonly used when SQL Server must be accessed from outside the network. With Windows integrated security, SQL Server recognizes the same usernames and passwords that are used to log in to the Windows network. Mixed mode simply means you can use either one of the two.

Chapter 20: Data Access with ADO

Following are examples of two different SQL Server connection strings. The first example shows a connection string that uses SQL Server authentication and the TCP/IP connection protocol. The second example shows a connection string that uses Windows integrated security:

```
Public Const gsCONNECTION As String = _
    "Provider=SQLOLEDB;" & _
    "Data Source=ComputerName\SQLServerName;" & _
    "Initial Catalog=Northwind;" & _
    "User ID=User;Password=password;" & _
    "Network Library=dbmssocn"

Public Const gsCONNECTION As String = _
    "Provider=SQLOLEDB;" & _
    "Data Source=ComputerName\SQLServerName;" & _
    "Initial Catalog=Northwind;" & _
    "Integrated Security=SSPI"
```

Microsoft SQL Server Stored Procedures

The syntax for executing plain text (or ad hoc) queries against SQL Server is identical to that which you used in the example for Access. The only difference is the contents of the connection string. When programming with SQL Server, however, it is more common to call SQL Server *stored procedures*.

Stored procedures are simply precompiled SQL statements that can be accessed by name from the database. They are much like VBA procedures in that they can accept arguments and return values. An example of a simple stored procedure that queries the `Customers` table is shown here:

```
CREATE PROC spGetCustomerNames
    @Country nvarchar(24)
AS
    SELECT    CustomerID,
            CompanyName,
            ContactName
    FROM      Customers
    WHERE     Country = @Country
    ORDER BY CompanyName
```

This stored procedure takes one argument, `@Country`, and returns a recordset containing the values for the fields specified in the `SELECT` list for customers whose country matches the value passed to the `@Country` argument.

ADO provides a very quick and simple way to execute stored procedures using the `Connection` object. ADO treats all stored procedures in the currently connected database as dynamic methods of the `Connection` object. You can call a stored procedure exactly like any other `Connection` object method, passing any arguments to the stored procedure as method arguments, and optionally passing a `Recordset` object as the last argument if the stored procedure returns a recordset.

This method is best used for “one-off” procedures rather than those you will execute multiple times, because it isn’t the most efficient method. However, it is significantly easier to code. The following example demonstrates executing the previous stored procedure as a method of the `Connection` object:

```

Public Sub ExecuteStoredProcAsMethod()

    Dim objConn As ADODB.Connection
    Dim rsData As ADODB.Recordset
    Dim sConnect As String

    ' Create the connection string.
    sConnect = "Provider=SQLOLEDB;Data Source=P2800\P2800;" & _
        "Initial Catalog=Northwind;Integrated Security=SSPI"

    ' Create the Connection and Recordset objects.
    Set objConn = New ADODB.Connection
    Set rsData = New ADODB.Recordset

    ' Open the connection and execute the stored procedure.
    objConn.Open sConnect
    objConn.spGetCustomerNames "UK", rsData

    ' Make sure we got records back
    If Not rsData.EOF Then
        ' Dump the contents of the recordset onto the worksheet.
        Sheet1.Range("A1").CopyFromRecordset rsData
        ' Close the recordset
        rsData.Close
        ' Fit the column widths to the data.
        Sheet1.UsedRange.EntireColumn.AutoFit
    Else
        MsgBox "Error: No records returned.", vbCritical
    End If

    ' Clean up our ADO objects.
    If CBool(objConn.State And adStateOpen) Then objConn.Close
    Set objConn = Nothing
    Set rsData = Nothing

End Sub

```

In this procedure, you executed the `spGetCustomerNames` stored procedure and passed it the value "UK" for its `@Country` argument. This populated the `rsData` recordset with all of the customers located in the UK. Note that the `Connection` object must be opened before the dynamic methods are populated, and that you must instantiate the `Recordset` object prior to passing it as an argument.

The most efficient way to handle stored procedures that will be executed multiple times is to prepare a publicly scoped `Command` object to represent them. The `Connection` will be stored in the `Command` object's `ActiveConnection` property, the stored procedure's name will be stored in the `Command` object's `CommandText` property, and any arguments to the stored procedure will be used to populate the `Command` object's `Parameters` collection.

Once this `Command` object has been created, it can be executed as many times as you like over the course of your application, without incurring the overhead required to perform the tasks just described with each execution.

Chapter 20: Data Access with ADO

For this example, create a simple stored procedure that you can use to insert new records into the `Shippers` table:

```
CREATE PROC spInsertShippers
    @CompanyName nvarchar(40),
    @Phone       nvarchar(24)
AS
    INSERT INTO Shippers(CompanyName, Phone)
    VALUES(@CompanyName, @Phone)
    RETURN @@IDENTITY
```

As you can see, the stored procedure has two arguments, `@CompanyName` and `@Phone`, which are used to collect the values to insert into those respective fields in the `Shippers` table. Similar to the `ID` field in the Access version of the Northwind database, the `ShipperID` field in the SQL Server version of Northwind is populated automatically by the database any time a new record is inserted. You retrieve this automatically assigned value in a similar fashion, through the use of SQL Server's `@@IDENTITY` system function. In this case, however, you won't have to make a separate query to retrieve the Shipper ID value because it will be returned to you by the stored procedure.

To present a more realistic application scenario, the following example uses publicly scoped `Connection` and `Command` objects, procedures to create and destroy the connection, a procedure to prepare the `Command` object for use, and a procedure that demonstrates how to use the `Command` object:

```
Public Const gszCONNECTION As String = _
    "Provider=SQLOLEDB;Data Source=P2800\P2800;" & _
    "Initial Catalog=Northwind;Integrated Security=SSPI"

Public gobjCmd As ADODB.Command
Public gobjConn As ADODB.Connection

Private Sub CreateConnection()
    ' Create the Connection object.
    Set gobjConn = New ADODB.Connection
    gobjConn.Open gszCONNECTION
End Sub

Private Sub DestroyConnection()
    ' Check to see if connection is still open before attempting to close it.
    If CBool(gobjConn.State And adStateOpen) Then gobjConn.Close
    Set gobjConn = Nothing
End Sub

Private Sub PrepareCommandObject()

    ' Create the Command object.
    Set gobjCmd = New ADODB.Command
    Set gobjCmd.ActiveConnection = gobjConn
    gobjCmd.CommandText = "spInsertShippers"
    gobjCmd.CommandType = adCmdStoredProc

    ' Load the parameters collection. The first parameter
    ' is always the stored procedure return value.
    gobjCmd.Parameters.Append _
```

```

        gobjCmd.CreateParameter("@RETURN_VALUE", adInteger, _
            adParamReturnValue, 0)
    gobjCmd.Parameters.Append _
        gobjCmd.CreateParameter("@CompanyName", adVarChar, _
            adParamInput, 40)
    gobjCmd.Parameters.Append _
        gobjCmd.CreateParameter("@Phone", adVarChar, _
            adParamInput, 24)

End Sub

Public Sub UseCommandObject()

    Dim lKeyValue As Long
    Dim lNumAffected As Long

    On Error GoTo ErrorHandler

    ' Create the Connection and the reusable Command object.
    CreateConnection
    PrepareCommandObject

    ' Set the values of the input parameters.
    gobjCmd.Parameters("@CompanyName").Value = "Air Carriers"
    gobjCmd.Parameters("@Phone").Value = "(206) 555-1212"

    ' Execute the Command object and check for errors.
    gobjCmd.Execute RecordsAffected:=lNumAffected, _
        Options:=adExecuteNoRecords
    If lNumAffected <> 1 Then Err.Raise Number:=vbObjectError + 1024, _
        Description:="Error executing Command object."

    ' Retrieve the primary key value for the new record.
    lKeyValue = gobjCmd.Parameters("@RETURN_VALUE").Value
    Debug.Print "The key value of the new record is: " & CStr(lKeyValue)

ErrorExit:

    ' Destroy the Command and Connection objects.
    Set gobjCmd = Nothing
    DestroyConnection

    Exit Sub

ErrorHandler:
    MsgBox Err.Description, vbCritical
    Resume ErrorExit
End Sub

```

A few things to note about the mini application:

- ❑ In a normal application, you would not create and destroy the Connection and Command objects in the UseCommandObject procedure. These objects are intended for reuse and therefore typically would be created when your application first started and destroyed just before it ended.

- ❑ When constructing and using the `Command` object's `Parameters` collection, keep in mind that the first parameter is always reserved for the stored procedure return value, even if the stored procedure doesn't have a return value.
- ❑ Even though you didn't make any particular use of the `ShipperID` value returned from the stored procedure for the new record, in a normal application this value would be very important. The `CompanyName` and `Phone` fields are for human consumption; the primary key value is how the database identifies the record. For example, in the Northwind database, the `ShipperID` is a required field for entering new records into the `Orders` table. Therefore, if you planned on adding an order that was going to use the new shipper, you would have to know the `ShipperID`.

Multiple Recordsets

The SQL Server OLE DB provider is an example of a provider that allows you to execute a SQL statement that returns multiple recordsets. This feature comes in very handy when you need to populate multiple controls on a form with lookup-table information from the database. You can combine all of the lookup-table `SELECT` queries into a single stored procedure and then loop through the individual recordsets, assigning their contents to the corresponding controls.

For example, if you needed to create a user interface for entering information into the `Orders` table, you would need information from several related tables, including `Customers` and `Shippers`, as shown in Figure 20-6.

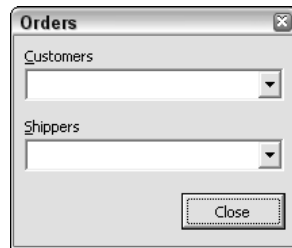


Figure 20-6

Create an abbreviated example of a stored procedure that returns the lookup information from these two tables, and then use the result to populate drop-downs on a UserForm:

```
CREATE PROC spGetLookupValues
AS
    -- Customers lookup table info.
    SELECT    CustomerID,
             CompanyName
    FROM      Customers

    -- Shippers lookup table info.
    SELECT    ShipperID,
             CompanyName
    FROM      Shippers
```

Note that this stored procedure contains two separate `SELECT` statements. These will populate two independent recordsets when the stored procedure is executed using ADO. The double dashes that you see at the beginning of the lines above each `SELECT` statement are T-SQL comment prefixes.

The following procedure is an example of a `UserForm_Initialize` event that populates drop-downs on the `UserForm` with the results of the `spGetLookupValues` stored procedure. For the purpose of this example, assume that the public `Connection` object `gobjConn` used in the previous example is still open and available for use:

```
Private Sub UserForm_Initialize()

    Dim rsData As ADODB.Recordset

    ' Create and open the Recordset object.
    Set rsData = New ADODB.Recordset
    rsData.Open "spGetLookupValues", gobjConn, _
                adOpenForwardOnly, adLockReadOnly, adCmdStoredProc

    ' The first recordset contains the customer list.
    Do While Not rsData.EOF
        ' Load the dropdown with the recordset values.
        ddCustomers.AddItem rsData.Fields(1).Value
        rsData.MoveNext
    Loop
    Set rsData = rsData.NextRecordset

    ' The second recordset contains the shippers list.
    Do While Not rsData.EOF
        ' Load the dropdown with the recordset values.
        ddShippers.AddItem rsData.Fields(1).Value
        rsData.MoveNext
    Loop
    Set rsData = rsData.NextRecordset

    ' No need to clean up the Recordset object at this point,
    ' it will be closed and set to nothing after the last
    ' call to the NextRecordset method.

End Sub
```

One thing to note about the method demonstrated here is that it requires prior knowledge of the number and order of recordsets returned by the call to the stored procedure. Also left out is any handling of the primary key values associated with the lookup table descriptions. In a real-world application, you would need to maintain these keys (I prefer using a hidden column in a multi-column drop-down for this purpose) so you could retrieve the primary key value that corresponded to the user's selection in each drop-down.

Disconnected Recordsets

The “Retrieving Data from Microsoft Access Using a Plain Text Query” section mentioned that getting in and out of the database as quickly as possible was an important goal. However, the `Recordset` object is a powerful tool that you would often like to hold onto and use without locking other users out of the database. The solution to this problem is the ADO disconnected recordset feature.

A disconnected recordset is a `Recordset` object whose connection to its data source has been severed, but that can still remain open. The result is a fully functional `Recordset` object that does not hold any locks in the database from which it was queried. Disconnected recordsets can remain open as long as

Chapter 20: Data Access with ADO

you need them, they can be reconnected to and resynchronized with the data source, and they can even be persisted to disk for later retrieval. A few of these capabilities are examined in the following example.

Imagine you wanted to implement a feature that would allow users to view any group of customers they chose. Running a query against the database each time the user specified a different criterion would be an inefficient way to accomplish this. A much better alternative would be to query the complete set of customers from the database and hold them in a disconnected recordset. You could then use the `Filter` property of the `Recordset` object to quickly extract the set of customers that your user requested.

The following example shows all of the elements required to create a disconnected recordset. Again, assume the availability of the public `gobjConn` `Connection` object:

```
Public grsData As ADODB.Recordset

Public Sub CreateDisconnectedRecordset ()

    Dim szSQL As String

    ' Create the SQL Statement.
    szSQL = "SELECT CustomerID, CompanyName, ContactName, Country " & _
        "FROM Customers"

    ' Steps to creating a disconnected recordset:
    ' 1) Create the Recordset object.
    Set grsData = New ADODB.Recordset
    ' 2) Set the cursor location to client side.
    grsData.CursorLocation = adUseClient
    ' 3) Set the cursor type to static.
    grsData.CursorType = adOpenStatic
    ' 4) Set the lock type to batch optimistic.
    grsData.LockType = adLockBatchOptimistic
    ' 5) Open the recordset.
    grsData.Open szSQL, gobjConn, , , adCmdText
    ' 6) Set the Recordset's Connection object to Nothing.
    Set grsData.ActiveConnection = Nothing

    ' grsData is now a disconnected recordset.
    Sheet1.Range("A1").CopyFromRecordset grsData

End Sub
```

Note that the `Recordset` object variable in the preceding example is declared with public scope. If you were to declare the `Recordset` object variable at the procedure level, VBA would automatically destroy it when the procedure ended and it would no longer be available for use.

Six crucial steps are required to successfully create a disconnected recordset. It's possible to combine several of them into one step during the `Recordset.Open` method, and it's more efficient to do so, but they are separated here for the sake of clarity:

- You must create a new, empty `Recordset` object to start with.
- You must set the cursor location to client-side. Because the recordset will be disconnected from the server, the cursor cannot be managed there. Note that this setting must be made *before* you open the recordset. It is not possible to change the cursor location once the recordset is open.

- ❑ The ADO client-side cursor engine supports only one type of cursor, the static cursor, so this is what the `CursorType` property must be set to.
- ❑ ADO has a lock type specifically designed for disconnected recordsets called Batch Optimistic. The Batch Optimistic lock type makes it possible to reconnect the disconnected recordset to the database and update the database with records that have been modified while the recordset was disconnected. This operation is beyond the scope of this chapter, so note that the Batch Optimistic lock type is required in order to create a disconnected recordset.
- ❑ Opening the recordset is the next step. This example used a plain text SQL query. This is not a requirement. You can create a disconnected recordset from almost any source that can be used to create a standard recordset. The client-side cursor engine lacks a few capabilities, however; multiple recordsets are one example.
- ❑ The final step is disconnecting the recordset from the data source. This is accomplished by setting the recordset's `Connection` object to `Nothing`. If you recall from the "Recordset Object Properties" section, the `Connection` object associated with a `Recordset` object is accessed through the `Recordset.ActiveConnection` property. Setting this property to `Nothing` severs the connection between the recordset and the data source.

Now that you have a disconnected recordset to work with, what kinds of things can you do with it? Just about any operation the `Recordset` object allows. Say that the user wanted to see a list of customers located in Germany, sorted by alphabetical order. This is how you'd accomplish that task:

```
' Set the Recordset filter to display only records
' whose Country field is Germany.
grsData.Filter = "Country = 'Germany'"
' Sort the records by CompanyName.
grsData.Sort = "CompanyName"
' Load the processed data onto Sheet1
Sheet1.Range("A1").CopyFromRecordset grsData
```

If you are working in a busy multi-user environment, the data in your disconnected recordset may become out-of-date during the course of your application due to other users inserting, updating, and deleting records. You can solve this problem by requerying the recordset. As demonstrated by the following example, this is a simple matter of reconnecting to the data source, executing the `Recordset.Requery` method, then disconnecting from the data source:

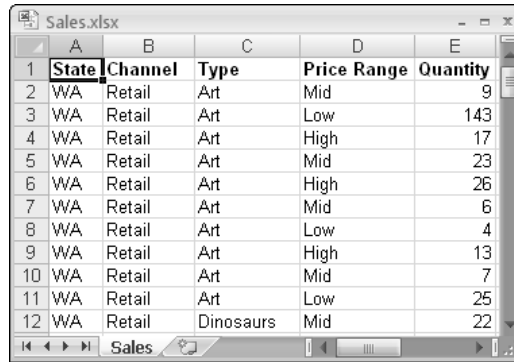
```
' Reconnect to the data source.
Set grsData.ActiveConnection = gobjConn
' Rerun the Recordset object's underlying query,
grsData.Requery Options:=adCmdText
' Disconnect from the data source.
Set grsData.ActiveConnection = Nothing
```

Using ADO with Non-Standard Data Sources

This section describes how you can use ADO to access data from two common non-standard data sources (data sources that are not strictly considered databases), Excel workbooks, and text files. Although the idea may seem somewhat counterintuitive, ADO is often the best choice for retrieving data from workbooks and text files because it eliminates the often lengthy process of opening them in Excel. Using ADO also allows you to take advantage of the power of SQL to do exactly what you want in the process.

Querying Microsoft Excel Workbooks

When using ADO to access data from Excel 2007 workbooks, you use the same OLE DB provider that you used earlier in this chapter to access data from Microsoft Access 2007. In addition to Access, this provider also supports most *ISAM data sources* (data sources that are laid out in a tabular, row and column format). You will use the `Sales.xlsx` workbook, shown in Figure 20-7, as the data source for the Excel examples.



	A	B	C	D	E
1	State	Channel	Type	Price Range	Quantity
2	WA	Retail	Art	Mid	9
3	WA	Retail	Art	Low	143
4	WA	Retail	Art	High	17
5	WA	Retail	Art	Mid	23
6	WA	Retail	Art	High	26
7	WA	Retail	Art	Mid	6
8	WA	Retail	Art	Low	4
9	WA	Retail	Art	High	13
10	WA	Retail	Art	Mid	7
11	WA	Retail	Art	Low	25
12	WA	Retail	Dinosaurs	Mid	22

Figure 20-7

When using ADO to work with Excel, the workbook file takes the place of the database, while worksheets within the workbook, as well as named ranges, serve as tables. Compare a connection string used to connect to an Access database with a connection string used to connect to an Excel workbook.

Connection string to an Access database:

```
sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _  
          "Data Source=C:\Files\Northwind 2007.accdb;"
```

Connection string to an Excel workbook:

```
sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _  
          "Data Source=C:\Files\Sales.xlsx;" & _  
          "Extended Properties=Excel 12.0;"
```

Note that the same provider is used, and that the full path and filename of the Excel workbook takes the place of the full path and filename of the Access database. The only difference is that you must specify the type name of the data source you want to connect to in the `Extended Properties` argument. When connecting to Excel 2007, you set the `Extended Properties` argument to `Excel 12.0`. For versions of Excel earlier than 2007, you set the `Extended Properties` argument to `Excel 8.0`.

You query data from an Excel worksheet using a plain text SQL statement exactly like you would query a database table. However, the format of the table name is different for Excel queries. You can specify the table that you want to query from an Excel workbook in one of four different ways:

- ❑ **Worksheet Name Alone**—When using the name of a specific worksheet as the table name in your SQL statement, the worksheet name must be suffixed with a \$ character and surrounded with square brackets. For example, [Sheet1\$] is a valid worksheet table name. If the worksheet name contains spaces or non-alphanumeric characters, you must surround it with single quotes. An example of this is ['My Sheet\$'].
- ❑ **Worksheet-level Range Name**—You can use a worksheet-level range name as a table name in your SQL statement. Simply prefix the range name with the worksheet name it belongs to, using the formatting conventions just described. An example of this would be [Sheet1\$SheetLevelName].
- ❑ **Specific Range Address**—You can specify the table in your SQL statement as a specific range address on the target worksheet. The syntax for this method is identical to that for a worksheet-level range name: [Sheet1\$A1:E20].
- ❑ **Workbook-level Range Name**—You can also use a workbook-level range name as the table in your SQL statement. In this case there is no special formatting required. You simply use the name directly, without brackets.

Although your sample workbook contains only one worksheet, this is not a requirement. The target workbook can contain as many worksheets and named ranges as you wish. You simply need to know which one to use in your query. The following procedure demonstrates all four table-specifying methods just discussed:

```
Public Sub QueryWorksheet()

    Dim rsData As ADODB.Recordset
    Dim sConnect As String
    Dim sSQL As String

    ' Create the connection string.
    sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=C:\Files\Sales.xlsx;" & _
        "Extended Properties=Excel 12.0;"

    ' Query based on the worksheet name.
    'sSQL = "SELECT * FROM [Sales$]"
    ' Query based on a sheet-level range name.
    'sSQL = "SELECT * FROM [Sales$SheetLevelName];"
    ' Query based on a specific range address.
    'sSQL = "SELECT * FROM [Sales$A1:E89];"
    ' Query based on a book-level range name.
    sSQL = "SELECT * FROM BookLevelName;"

    Set rsData = New ADODB.Recordset
    rsData.Open sSQL, sConnect, adOpenForwardOnly, _
        adLockReadOnly, adCmdText

    ' Check to make sure we received data.
    If Not rsData.EOF Then
        Sheet1.Range("A1").CopyFromRecordset rsData
    Else
        MsgBox "No records returned.", vbCritical
    End If
End Sub
```

```
End If

' Clean up our Recordset object.
rsData.Close
Set rsData = Nothing

End Sub
```

By default, the OLE DB provider for Microsoft Jet assumes that the first row in the table you specify with your SQL statement contains the field names for the data. If this is the case, you can perform more complex SQL queries, making use of the `WHERE` and `ORDER BY` clauses. If the first row of your data table does not contain field names, however, you must inform the provider of this fact or you will lose the first row of data. The way to accomplish this is by providing an additional setting, `HDR=No`, to the `Extended Properties` argument of the connection string:

```
sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
           "Data Source=C:\Files\Sales.xlsx;" & _
           "Extended Properties=Excel 12.0;HDR=No";"
```

Note that when you pass multiple settings to the `Extended Properties` argument, the entire setting string must be surrounded with double quotes and the individual settings must be delimited with semicolons. If your data table does not include column headers, you will be limited to `SELECT` queries.

Inserting and Updating Records in Microsoft Excel Workbooks

ADO can do more than just query data from an Excel workbook. You can also insert and update records in the workbook, just as you would with any other data source. Deleting records, however, is not supported. Updating records, although possible, is somewhat problematic when an Excel workbook is the data source, because Excel-based data tables rarely have anything that can be used as a primary key to uniquely identify a specific record. Therefore, you must specify the values of enough fields to uniquely identify the record concerned in the `WHERE` clause of your SQL statement when performing an update. If more than one record meets `WHERE` clause criteria, all such records will be updated.

Inserting is significantly less troublesome. All you do is construct a SQL statement that specifies values for each of the fields, and then execute it. Note once again that your data table must have column headers in order for it to be possible to execute action queries against it. The following example demonstrates how to insert a new record into the sales worksheet data table:

```
Public Sub WorksheetInsert()

    Dim objConn As ADODB.Connection
    Dim sConnect As String
    Dim sSQL As String

    ' Create the connection string.
    sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
              "Data Source=C:\Files\Sales.xlsx;" & _
              "Extended Properties=Excel 12.0;"

    ' Create the SQL statement.
    sSQL = "INSERT INTO [Sales$] " & _
```

```

        "VALUES('VA', 'On-Line', 'Computers', 'Mid', 30);"

' Create and open the Connection object.
Set objConn = New ADODB.Connection
objConn.Open sConnect

' Execute the insert statement.
objConn.Execute sSQL, , adCmdText Or adExecuteNoRecords

' Close and destroy the Connection object.
objConn.Close
Set objConn = Nothing

End Sub

```

Note that if you use ADO to insert a new record into an Excel worksheet, and you use a range name as the table in the `INSERT` statement, that range name will not be extended to include the new record. Therefore, you should only use ADO to insert records into worksheets in situations where the worksheet name can be used as the table in the `INSERT` statement.

Querying Text Files

The last data access technique to discuss in this chapter is querying text files using ADO. The need to query text files doesn't come up as often as some of the other situations addressed in this chapter. However, when you're faced with an extremely large text file, the result of a mainframe database data dump, for example, ADO can be a lifesaver.

Not only will it allow you to rapidly load large amounts of data into Excel, but using the power of SQL to limit the size of the resultset can also enable you to work with data from a text file that is simply too large to be opened directly in Excel. For the discussion on text file data access, use a comma-delimited text file, `Sales.csv`, whose contents are identical to the `Sales.xlsx` workbook used in the Excel examples in the previous section. The following example demonstrates how to construct a connection string to access a text file:

```

szConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
           "Data Source=C:\Files\" & _
           "Extended Properties=Text;"

```

Note that in the case of text files, the `Data Source` argument is set to the directory that contains the text file. Do not include the name of the file in this argument. Once again, the provider is informed of the format to be queried by using the `Extended Properties` argument. In this case, you simply set this argument to the value `"Text"`.

Querying a text file is virtually identical to querying an Excel workbook. The main difference is how the table name is specified in the SQL statement. When querying a text file, the filename itself is used as the table name in the query. This has the added benefit of allowing you to work with multiple text files in a single directory without having to modify your connection string.

As with Excel, you are limited to `SELECT` queries if the first row of your text file does not contain field names. You must also add the `HDR=No` setting to the `Extended Properties` argument if this is the case, in order to avoid losing the first row of data. The example text file has field names in the first row, and

Chapter 20: Data Access with ADO

you should assume that you need to limit the number of records you bring into Excel by adding a restriction in the form of a `WHERE` clause to your query. The following procedure demonstrates this:

```
Public Sub QueryTextFile()

    Dim rsData As ADODB.Recordset
    Dim sConnect As String
    Dim sSQL As String

    ' Create the connection string.
    sConnect = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=C:\Files\" & _
        "Extended Properties=Text;"

    ' Create the SQL statement.
    sSQL = "SELECT * FROM Sales.csv WHERE Type='Art';"

    Set rsData = New ADODB.Recordset
    rsData.Open sSQL, sConnect, adOpenForwardOnly, _
        adLockReadOnly, adCmdText

    ' Check to make sure we received data.
    If Not rsData.EOF Then
        ' Dump the returned data onto Sheet1.
        Sheet1.Range("A1").CopyFromRecordset rsData
    Else
        MsgBox "No records returned.", vbCritical
    End If

    ' Clean up our Recordset object.
    rsData.Close
    Set rsData = Nothing

End Sub
```

Summary

Many years ago you could concentrate on understanding Excel, and that would have been enough to get by. Today, however, it is becoming increasingly difficult to avoid coordinating your Excel activities with a back-end data source of some kind. This chapter provided you with a solid introduction to using SQL and ADO to access various data sources. Due to space constraints, this chapter could only scratch the surface of possibilities in each section. So if data access is or might become a significant part of your development effort, you are strongly recommended to obtain and read more specialized books on the subject.

The next chapter continues to look at managing external data with Excel by examining some of the built-in features provided by Excel for this purpose.

21

Managing External Data

Chapter 20 discussed how to access data with essentially unlimited flexibility using ADO. Excel also provides some built-in data management features, primarily through the `QueryTable`, `ListObject`, and `WorkbookConnection` objects. Excel's built-in data management features have less flexibility than custom ADO programming. For example, you can only use these features to retrieve data, not modify it. But they are simpler and offer a number of useful capabilities right out of the box. This chapter examines some of Excel's built-in data management capabilities.

The External Data User Interface

The built-in data management features in Excel 2007 are accessed from two groups on the Data tab of the Ribbon in the Excel 2007 user interface. The Get External Data and Manage Connections groups are shown in Figure 21-1.

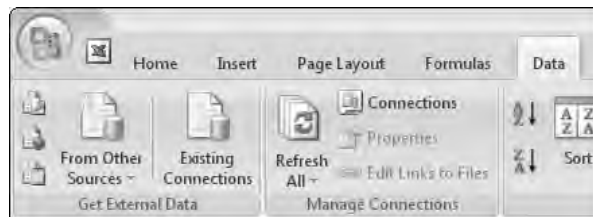


Figure 21-1

The controls on the Get External Data group are used to create new connections from your workbook to various data sources, and the controls on the Manage Connections group are used to manage data connections that already exist in your workbook.

Get External Data

The controls in the Get External Data group allow you to retrieve external data directly from various data sources or use predefined queries stored in various data connection files. The three buttons along the left side of the Get External Data group provide quick access for retrieving Microsoft Access data, data from the web, and data from text files, respectively. The From Other Sources button provides you with a drop-down list of all the other options available for retrieving data from external data sources, as shown in Figure 21-2.



Figure 21-2

This chapter is concerned with the external data features that create a `QueryTable` object in the background. These features include Web Queries and all queries performed on data sources that return uncomplicated tabular data, including text files and relational databases like Access and SQL Server.

The Existing Connections button displays the Existing Connections dialog, shown in Figure 21-3. This dialog displays a list of connections that currently exist in the workbook, as well as a list of stored data connection files that you can use to retrieve external data.

The controls in the Get External Data group will be disabled if you have selected a cell within the range of an existing external data connection. If these controls are disabled, try selecting an unused cell outside of any existing data tables.



Figure 21-3

Manage Connections

The controls on the Manage Connections group allow you to manipulate the individual connections in your workbook, as well as providing a central location for viewing and managing all connections in the currently active workbook. The buttons in this group that operate on a specific connection will be disabled until you select a cell within the range assigned to a connection. With a cell in a connection range selected, the Manage Connections group will look similar to Figure 21-4.

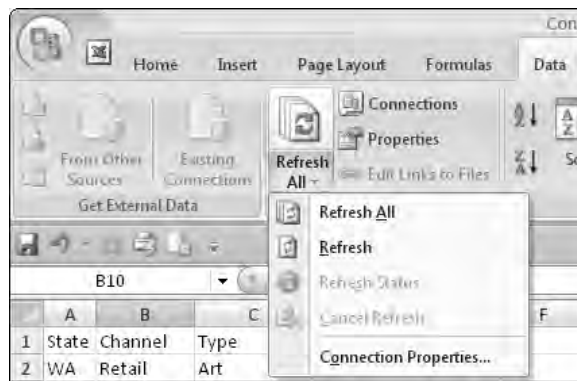


Figure 21-4

The most interesting control on the Manage Connections group is the new Connections button. This displays a dialog that lists all of the external data connections in the currently active workbook and allows you to see where they are used, as well as view and manipulate their properties. The Workbook Connections dialog is shown in Figure 21-5.

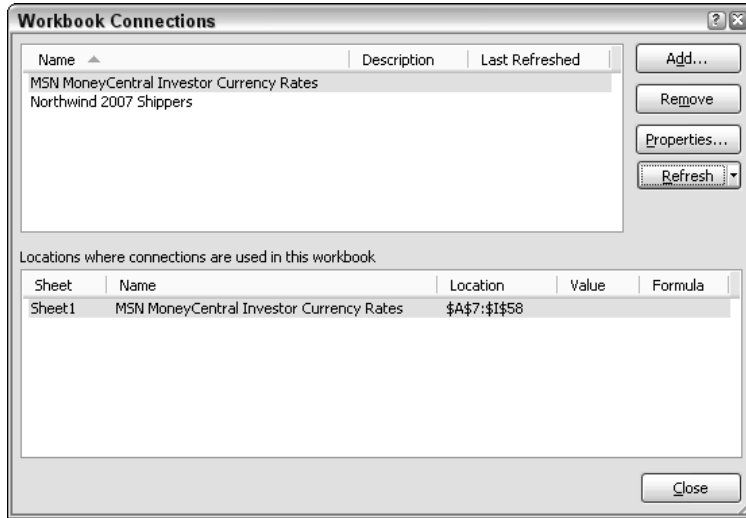


Figure 21-5

The Properties and Refresh buttons on this dialog expose exactly the same features as those shown on the Refresh All button in Figure 21-4.

The QueryTable and ListObject

When you use the Get External Data feature to create Web Queries or retrieve tabular data, you are creating a `QueryTable` to manage that data. This `QueryTable` can exist alone, or it can be associated with a `ListObject` (the `ListObject` is also covered in Chapter 6). Retrieving data using Web Queries or text files from the user interface will create a standalone `QueryTable`. Retrieving data from relational databases like Access or SQL Server will create a `ListObject` whose data source is a `QueryTable`. When you create your own `QueryTable` objects using VBA, you are free to create them either way. Both methods are demonstrated in this section.

This section utilizes the Northwind database, a sample database provided with Microsoft Access 2007. If you don't have this database available, you will need to install it to run the example code.

A QueryTable from a Relational Database

As a first introduction to the `QueryTable` object, create a simple `QueryTable` based on a table from the Access 2007 Northwind database used in Chapter 20. The code to create the `QueryTable` is as follows:

```

Sub CreateSimpleQueryTable()

    Dim qryTable As QueryTable
    Dim rngDestination As Range
    Dim strConnection As String

    ' Define the connection string and destination range.
    strConnection = "OLEDB;Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=C:\Files\Northwind 2007.accdb"
    Set rngDestination = Sheet1.Range("A1")

    ' Create the QueryTable.
    Set qryTable = Sheet1.QueryTables.Add(strConnection, rngDestination)

    ' Populate the QueryTable.
    qryTable.CommandText = "Customers"
    qryTable.CommandType = xlCmdTable
    qryTable.Refresh

End Sub

```

There are three steps involved in creating a `QueryTable`:

1. Define the connection string and destination range. If the connection string used here looks familiar, that's no coincidence. It's almost exactly the same connection string used to create an ADO connection to the Northwind 2007 Access database in Chapter 20. This is because both ADO and `QueryTables` use the same underlying OLE DB drivers to connect to relational databases. The only difference in this case is that the first item in the connection string must specify what type of connection string it is (in this case OLEDB). This is because a `QueryTable` can be based on several different connection types, so you need to tell it which type you're using. The destination range is a reference to the cell on the destination worksheet that you want to be the top-left cell in the resulting `QueryTable`.
2. Create the `QueryTable` object. This is a simple matter of calling the `QueryTables.Add` method, passing it the connection string and destination range defined in step 1, and assigning the result to a `QueryTable` object variable that will be used for further manipulation of the `QueryTable`.
3. Populate the `QueryTable` object. After step 2, the `QueryTable` exists, it knows what its data source is, and it knows where on the destination worksheet it will be located. However, it doesn't know what data to display. Remedy this by using the `CommandText` and `CommandType` properties of the `QueryTable` object to tell it to display the contents of the Customers table. Actual retrieval of data is accomplished by calling the `QueryTable.Refresh` method.

A section of the `QueryTable` created in the previous example is shown in Figure 21-6.

	A	B	C	D	E
1	ID	Company	First Name	Last Name	E-mail
2	1	Company A	First	Last	
3	2	Company B	Antonio	Gratacos Solsona	
4	3	Company C	Thomas	Axeh	
5	4	Company D	Christina	Lee	

Figure 21-6

Chapter 21: Managing External Data

A `QueryTable` that displays data from a relational database may need to be updated periodically—for example, in cases where other users may have added new records that need to be displayed. You can update the `QueryTable` data at any time by calling the `QueryTable.Refresh` method. The `QueryTable` will also perform this operation for you automatically, if you tell it to do so by adding the following line of code:

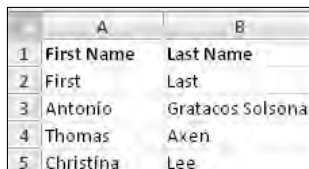
```
' Populate the QueryTable.  
qryTable.CommandText = "Customers"  
qryTable.CommandType = xlCmdTable  
qryTable.RefreshPeriod = 30  
qryTable.Refresh
```

This line of code tells the `QueryTable` to refresh itself automatically every 30 minutes, and it will continue to do so every 30 minutes for as long as the workbook is open. The value assigned to the `QueryTable.RefreshPeriod` property is persisted, so even after you close and reopen the workbook containing the `QueryTable`, it will continue to refresh automatically at the interval you've specified. The refresh period is specified in minutes, with valid values being 1 through 32,767. Setting the `QueryTable.RefreshPeriod` property to 0 disables automatic refreshing. This is also the default value if you don't specify this property.

You can think of a `QueryTable` as a container for whatever data you want to put in it. Just like you can refresh the `QueryTable` at any time to update it with the latest data from the data source, you can also change the data displayed by the `QueryTable` at any time by simply changing its `CommandText` and `CommandType` properties and calling its `Refresh` method:

```
With Sheet1.QueryTables(1)  
    .CommandText = "SELECT [First Name], [Last Name] FROM Customers"  
    .CommandType = xlCmdSql  
    .Refresh  
End With
```

This code changes the `QueryTable` from displaying the entire `Customers` table to deriving its data from a SQL statement that retrieves just the first and last names of each customer. See Chapter 20 for an introduction to SQL if you aren't familiar with it. As soon as this code is executed, the results displayed by the `QueryTable` will change accordingly, as shown in Figure 21-7.



	A	B
1	First Name	Last Name
2	First	Last
3	Antonio	Gratacos Solsona
4	Thomas	Axen
5	Christina	Lee

Figure 21-7

By default, a `QueryTable` will perform what is called a *background query*. This means that as soon as the query specified by the `QueryTable` has been submitted, VBA will return control of Excel back to the user. This may be a good choice if the `QueryTable` performs a long-running query and you don't want to require the user to sit and wait for it to finish. It may also be a bad choice if something critical in your application depends on the result of the query, and it therefore must be completed before your code continues.

In this case you may want to turn off background querying by setting the `QueryTable.BackgroundQuery` property to `False`:

```
' Populate the QueryTable.
qryTable.CommandText = "Customers"
qryTable.CommandType = xlCmdTable
qryTable.RefreshPeriod = 30
qryTable.BackgroundQuery = False
qryTable.Refresh
```

The `QueryTable.Refresh` method also has a `BackgroundQuery` argument that can be set to `False` to accomplish the same thing without requiring an additional line of code. The difference is that the `QueryTable.BackgroundQuery` property is persistent and applies to all future refreshes, whereas the `BackgroundQuery` argument to the `QueryTable.Refresh` method must be specified each time you refresh the `QueryTable` or it will simply default to `True`.

A Query Table Associated with a ListObject

Standalone `QueryTables` are good for retrieving data that will be used for background or display purposes only. If you want the user to be able to interact with the data after it has been retrieved, a better option is to create a `QueryTable` associated with a `ListObject`. This creates a table in the Excel user interface with all of the built-in ease-of-manipulation features that users need to work with the data.

Note that there is some overlap between the `QueryTable` and `ListObject` properties and methods. For example, both the `QueryTable` and `ListObject` have a `Refresh` method that updates their data. When there is duplication, which object's property or method you decide to use is a matter of preference. Because this chapter focuses on `QueryTables`, the `QueryTable` properties and methods are used wherever there is duplication between the two object models.

The code for creating a `QueryTable` associated with a `ListObject` is very similar to the code for creating a `QueryTable` alone. In fact, the preceding `QueryTable` example can be modified to use a `ListObject` by changing a single line of code:

```
Sub CreateQueryTableWithList()

    Dim qryTable As QueryTable
    Dim rngDestination As Range
    Dim strConnection As String

    ' Define the connection string and destination range.
    strConnection = "OLEDB;Provider=Microsoft.ACE.OLEDB.12.0;" & _
        "Data Source=C:\Files\Northwind 2007.accdb"
    Set rngDestination = Sheet2.Range("A1")

    ' Create the ListObject and get a reference to its QueryTable.
    Set qryTable = Sheet2.ListObjects.Add(SourceType:=xlSrcExternal, _
        Source:=strConnection, Destination:=rngDestination).QueryTable

    ' Populate the QueryTable.
```

Chapter 21: Managing External Data

```
qryTable.CommandText = "Customers"  
qryTable.CommandType = xlCmdTable  
qryTable.Refresh False
```

```
End Sub
```

The process for creating a standalone `QueryTable` and the process for creating a `QueryTable` associated with a `ListObject` are fundamentally the same, the only difference being that you use the arguments of the `ListObjects.Add` method to specify the connection string and destination. Because a `ListObject` can have a number of additional data sources besides the external data used in this example, you also need to specify what type of data source your connection string represents, using the `SourceType` argument of the `ListObjects.Add` method. See Chapter 6 for more details on `ListObjects`.

A portion of the table created by the previous code is shown in Figure 21-8.



	A	B	C	D	
1	ID	Company	First Name	Last Name	E-mail
2	1	Company A	First	Last	
3	2	Company B	Antonio	Gratacós Solsona	
4	3	Company C	Thomas	Axén	
5	4	Company D	Christina	Lee	

Figure 21-8

Any time a `ListObject` is created from an external data source, either using VBA or through the Excel UI, an associated `QueryTable` object is created. You can use this `QueryTable` to manipulate the source data for any `ListObject` in a workbook.

Note that a `QueryTable` associated with a `ListObject` is not part of the `Worksheet.QueryTables` collection. It can only be accessed through the `ListObject.QueryTable` property of its associated `ListObject`. If you are using VBA to examine worksheets for the existence of query tables, you will need to look for them both directly in the `QueryTables` collection and indirectly in the `ListObjects` collection.

QueryTables and Parameter Queries

It is often useful to base your `QueryTable` on a parameter query rather than a fixed SQL statement. This allows you to determine which subset of the data you display, and even allows you to provide your users with the ability to modify the parameters when the `QueryTable` is refreshed.

One notable quirk of parameter queries is that `QueryTables` will not support them if you use the OLE DB provider used in the previous two sections. Instead, you must switch to the ODBC driver. This is a simple matter of changing the first argument of the connection string from OLEDB to ODBC and providing ODBC connection information:

```
Sub CreateQueryTableWithParameters()  
  
    Dim qryTable As QueryTable  
    Dim rngDestination As Range
```

```

Dim strConnection As String
Dim strSQL As String

' Define the connection string and destination range.
strConnection = "ODBC;DSN=MS Access Database;" & _
               "DBQ=C:\Files\Northwind 2007.accdb;"
Set rngDestination = Sheet3.Range("A1")

' Create a parameter query.
strSQL = "SELECT [Product Name], [List Price], [Quantity Per Unit]" & _
        " FROM Products" & _
        " WHERE Category = ?;" ' This is the parameter.

' Create the QueryTable.
Set qryTable = Sheet3.QueryTables.Add(strConnection, rngDestination)

' Populate the QueryTable.
qryTable.CommandText = strSQL
qryTable.CommandType = xlCmdSql
qryTable.Refresh False

End Sub

```

In this example, you set the `CommandText` property of the `QueryTable` to a SQL statement that selects the Product Name, List Price, and Quantity Per Unit from the Products table of the Northwind 2007 database. The `WHERE` clause of the SQL statement contains a parameter. A parameter is created by placing a question mark character (?) where an actual value would normally go. A SQL statement may contain one or more parameters at any point where a value from the database would normally go. You cannot use parameters for table names, column names, or SQL keywords.

When this code is executed, VBA automatically recognizes that the SQL contains a parameter, and it will prompt you to enter a value for the parameter by displaying the dialog box shown in Figure 21-9.

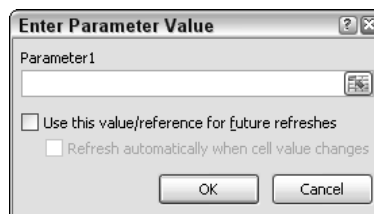


Figure 21-9

For each parameter in the `CommandText` property of the `QueryTable`, VBA creates a `Parameter` object that it adds to the `QueryTable.Parameters` collection. You can gain some additional control over how parameters are handled by creating these `Parameter` objects and adding them to the `Parameters` collection yourself.

You create parameters using the `QueryTable.Parameters.Add` method. This method adds a `Parameter` object to the `QueryTable` and returns a reference to it. You can then use that reference to modify the behavior of the `Parameter` object. The next example uses this capability to prepopulate the

Chapter 21: Managing External Data

parameter with an initial value when the `QueryTable` is first created, and then to have it prompt the user for a new value each time the `QueryTable` is refreshed after that.

If you create your own `Parameter` objects, they must be created in the same order that the corresponding parameters appear in the SQL statement.

```
Sub CreateQueryTableWithParameters()  
  
    Dim objParam As Parameter  
    Dim qryTable As QueryTable  
    Dim rngDestination As Range  
    Dim strConnection As String  
    Dim strSQL As String  
  
    ' Define the connection string and destination range.  
    strConnection = "ODBC;DSN=MS Access Database;" & _  
        "DBQ=C:\Files\Northwind 2007.accdb;"  
    Set rngDestination = Sheet3.Range("A1")  
  
    ' Create a parameter query.  
    strSQL = "SELECT [Product Name], [List Price], [Quantity Per Unit]" & _  
        " FROM Products" & _  
        " WHERE Category = ?;" ' This is the parameter.  
  
    ' Create the QueryTable.  
    Set qryTable = Sheet3.QueryTables.Add(strConnection, rngDestination)  
  
    ' Create the parameter and give it an initial value.  
    Set objParam = qryTable.Parameters.Add("Select Category", _  
        xlParamTypeVarChar)  
    objParam.SetParam xlConstant, "Beverages"  
  
    ' Populate the QueryTable.  
    qryTable.CommandText = strSQL  
    qryTable.CommandType = xlCmdSql  
    qryTable.Refresh False  
  
    ' Configure the parameter to prompt the user for a  
    ' new value the next time the QueryTable is refreshed.  
    objParam.SetParam xlPrompt, "Select Category"  
  
End Sub
```

You can also configure a `Parameter` to retrieve its value from a cell on the worksheet, and to automatically refresh the `QueryTable` whenever the value in that cell changes. This allows you to, for example, provide the users with a data validation list of choices for the parameter, rather than forcing them to remember all the potentially valid values. This feature is also very useful when your query requires multiple parameters. Rather than having to contend with a prompt dialog for every parameter, the user can simply make the appropriate entries in the cells that specify the parameters.

If you replace the final line of code in the preceding procedure with the following two lines of code, the `Parameter` will retrieve its value from cell F1 and refresh the `QueryTable` automatically whenever the value in cell F1 changes:

```
' Configure the parameter to retrieve its value from cell F1
' and refresh the QueryTable whenever that value changes.
objParam.SetParam xlRange, Sheet3.Range("F1")
objParam.RefreshOnChange = True
```

In the previous examples, you used all three options provided by the `Parameter.SetParam` method. The first argument of the `SetParam` method specifies the option to be used, and the second argument provides additional data for that option. The three `SetParam` options are as follows:

- ❑ `xlConstant` — Tells the parameter to use the value specified by the second argument. The second argument can be anything that returns a value of the correct data type for the parameter, including a variable, a cell reference, or a hard-coded value.
- ❑ `xlPrompt` — Tells the parameter to prompt the user for its value. The second argument is the prompt string that will appear in the dialog box.
- ❑ `xlRange` — Tells the parameter to retrieve its value from the cell specified by the second argument. The second argument must be a valid `Range` object.

QueryTables from Web Queries

`QueryTables` are not limited to retrieving data from traditional databases. They can also retrieve data from web sites. Creating a `QueryTable` based on a web site is known as performing a `Web Query`. A simple example of a `QueryTable` based on a `Web Query` follows. It pulls in the most recent data on major U.S. financial indexes from the Wall Street Journal web site:

```
Sub CreateWebQuery()

    Dim qryTable As QueryTable
    Dim rngDestination As Range
    Dim strConnection As String

    ' Define the connection string and destination range.
    strConnection = "URL;http://online.wsj.com/public/page/" & _
        "markets.html?mod=hpp_us_indexes"
    Set rngDestination = Sheet4.Range("A1")

    ' Create the QueryTable.
    Set qryTable = Sheet4.QueryTables.Add(strConnection, rngDestination)

    ' Populate the QueryTable.
    qryTable.WebSelectionType = xlSpecifiedTables
    qryTable.WebTables = "19"
    qryTable.WebFormatting = xlWebFormattingAll
    qryTable.Refresh False

End Sub
```

Chapter 21: Managing External Data

Note the similarity between this code and the code used to create a `QueryTable` from a relational database. The differences are the content of the connection string and a different set of properties that must be set prior to refreshing the `QueryTable`.

You inform the `QueryTable` that it will be performing a Web Query by specifying `URL` as the first argument in the connection string. The rest of the connection string is a URL that specifies the web page from which you want to retrieve the data.

You could retrieve an entire web page with your Web Query, but this is rarely what you want to do. Most web pages are structured as a group of HTML tables nested and arranged to produce the visual layout you see in your browser. The Web Querying functionality of the `QueryTable` object lets you take advantage of this fact by allowing you to specify just the table or tables on the web site that you want to retrieve.

This is accomplished by setting the `WebSelectionType` property to `xlSpecifiedTables` and then listing the index numbers of the tables you want to retrieve in the `WebTables` property. If you want to retrieve more than one table, simply separate the list of index numbers with commas. There is no tried and true method for determining which table index number holds the data you want. The easiest way to determine this value is to start with code similar to that shown earlier, and increment the `WebTables` property beginning with the number 1 until you locate the index that corresponds to the data you want. A section of the result of the Web Query is shown in Figure 21-10.

	A	B	C	D
1	*at close			9:48 am EDT
2				
3	DJIA	11028.63	17.21	
4				
5	Nasdaq	2075.77	-4.94	
6				
7	S&P 500	1260.23	0.42	

Figure 21-10

Although the ability to specify a table index gives you significant flexibility in a Web Query, it also makes your code vulnerable to design changes on the target web site that might cause a change in the table index number of the table you want to retrieve.

You have three formatting options for the results returned by the Web Query, controlled by the value of the `WebFormatting` property:

- `xlWebFormattingAll` — Imports the HTML format of the results exactly as they appear on the source web page, including functioning hyperlinks.
- `xlWebFormattingRTF` — Imports the data with the HTML formatting converted to rich-text format. This will produce results similar to `xlWebFormattingAll`, but without hyperlinks or merged cells.
- `xlWebFormattingNone` — Imports the data as plain text.

You will note the obvious thick gray bars separating the rows of data in Figure 21-10. These appear because that is how the table was structured on the source web page. You will often get artifacts like this when performing web queries. You can simply hide these rows and they will remain hidden, even when the `QueryTable` is refreshed. Just don't forget to unhide any rows and/or columns you've hidden if you do something that changes the structure of the Web Query such that the data you want to display ends up hidden.

Some web sites provide parameterized URLs specifically designed for use in web queries. The Yahoo Finance web site is one example. It provides a URL to which you can append a comma-delimited list of stock symbols to retrieve the information for. The following example creates a parameterized Web Query procedure that wraps this Yahoo Finance URL. Just call this procedure and pass it a comma-delimited list of stock ticker symbols, and it will return a table with the latest data for those symbols. You can pass as few or as many symbols as you like:

```
Sub ParameterizedWebQuery(strQuoteList As String, wksSheet As Worksheet)

    Dim qryTable As QueryTable
    Dim rngDestination As Range
    Dim strConnection As String

    ' Define the connection string and destination range.
    strConnection = "URL;http://finance.yahoo.com/q/cq?d=v1&s=" & strQuoteList
    Set rngDestination = wksSheet.Range("A1")

    ' Create the QueryTable.
    Set qryTable = wksSheet.QueryTables.Add(strConnection, rngDestination)

    ' Populate the QueryTable.
    qryTable.WebSelectionType = xlSpecifiedTables
    qryTable.WebTables = "9"
    qryTable.WebFormatting = xlWebFormattingNone
    qryTable.Refresh False

End Sub

Sub CallParameterizedWebQuery()
    ParameterizedWebQuery "MSFT, DELL, IBM", Sheet6
End Sub
```

In this example, you call the parameterized Web Query procedure, pass it the stock symbols for Microsoft, Dell, and IBM, and tell it to place the resulting `QueryTable` on Sheet6. The results of this Web Query are shown in Figure 21-11.

	A	B	C	D	E	F
1	Symbol	Time	Trade	Change	% Chg	Volume
2	MSFT	9:35AM ET	23.93	Up 0.06	Up 0.25%	3,077,336
3	DELL	9:35AM ET	20.32	Up 0.41	Up 2.06%	2,343,948
4	IBM	9:30AM ET	75	Up 0.14	Up 0.19%	88,000
5						

Figure 21-11

Chapter 21: Managing External Data

In a real-world application, you would typically let the user specify the list of stock symbols—for example, by allowing them to pick the symbols from a list in a `UserForm` or allowing them to enter the symbols on a worksheet. You would then read the selected symbols, create a comma-delimited list from them, and pass this list to the `ParameterizedWebQuery` procedure.

There is no tried and true method for determining how to structure the URL used in a Web Query. If the format is not documented on the web site (and it rarely will be), you can use the macro recorder and experiment with the site directly in the web browser to determine the appropriate format for the Web Query URL. This is how both of the URLs in the previous examples were created.

Like all `QueryTables`, you can modify the data displayed by a `QueryTable` based on a Web Query at any time. Unlike `QueryTables` based on relational databases, however, you cannot modify the `CommandText` property of a `QueryTable` based on a Web Query. In fact, attempting to do so will render the `QueryTable` inoperable.

Instead, you change the data displayed in a `QueryTable` based on a Web Query by modifying its `Connection` property. The new connection string must be in exactly the same format as a connection string used to create a Web Query initially, and if the table index you want to retrieve is different from the current `WebTables` property value, you will have to update that property as well prior to refreshing the `QueryTable`.

A QueryTable from a Text File

You can also use a `QueryTable` to extract data from a text file. The advantage of using a `QueryTable` as opposed to the `Workbooks.OpenText` method is that the text file data can be loaded directly into the workbook you specify, as opposed to being opened in a new workbook.

This example uses the same `Sales.csv` source file used in the ADO text file example in Chapter 20. The code to load data from this text file using a `QueryTable` is as follows:

```
Sub QueryTableFromTextFile()  
  
    Dim qryTable As QueryTable  
    Dim rngDestination As Range  
    Dim strConnection As String  
  
    ' Define the connection string and destination range.  
    strConnection = "TEXT;C:\Files\Sales.csv"  
    Set rngDestination = Sheet6.Range("A1")  
  
    ' Create the QueryTable.  
    Set qryTable = Sheet6.QueryTables.Add(strConnection, rngDestination)  
  
    ' Populate the QueryTable.  
    qryTable.TextFileStartRow = 1  
    qryTable.TextFileParseType = xlDelimited  
    qryTable.TextFileCommaDelimiter = True
```

```

qryTable.TextFileTextQualifier = xlTextQualifierNone
qryTable.TextFileColumnDataTypes = Array(2, 2, 2, 2, 1)
qryTable.Refresh False

```

End Sub

You inform the `QueryTable` that it will be extracting data from a text file by specifying `TEXT` as the first argument in the connection string. The second argument of the connection string is the full path and filename of the text file.

The `QueryTable` object has a series of text file-specific properties that allow you to control how the text file data is loaded. The following list describes the five of these properties that are most commonly used:

- ❑ `TextFileStartRow`— This property tells the query table which row of the text file to start with when it loads the data. A text file may have one or more initial rows that are not part of the data set. You can use this property to skip these initial rows by specifying some number greater than 1. A value of 1 tells the `QueryTable` to import the entire text file.
- ❑ `TextFileParseType`— This property tells the `QueryTable` whether the columns of data in the text file are separated by some delimiting character (`xlDelimited`) or are fixed width (`xlFixedWidth`). The value of this property will determine which additional properties you specify. Your text file is comma-delimited, so the options for a delimited text file are discussed next. If your text file has fixed-width columns, you would also set the `TextFileFixedColumnWidths` property to an array of column width values, one for each column in your text file.
- ❑ `TextFileCommaDelimiter`— When loading a delimited text file, you need to tell the `QueryTable` what delimiter to look for. You're loading a comma-delimited text file, so you set the `TextFileCommaDelimiter` property to `True`. The following additional properties are available to specify other delimiters: `TextFileTabDelimiter`, `TextFileSemicolonDelimiter`, `TextFileSpaceDelimiter`, and `TextFileOtherDelimiter`.
- ❑ `TextFileTextQualifier`— You will often encounter text files where values in text data type fields are surrounded by single or double quotes. This property is used to inform the `QueryTable` if this is the case, and if so, which of these characters it should treat as a qualifier and ignore when loading the data. The options are `xlTextQualifierSingleQuote`, `xlTextQualifierDoubleQuote`, and `xlTextQualifierNone`. You use the last option because your text file does not use any text field qualifiers.
- ❑ `TextFileColumnDataTypes`— This property is used to tell the `QueryTable` how to interpret each column of data in the text file. The property takes an array of integer values, one for each column in the text file, that specify what type of data is contained in those columns. For uncomplicated text and numeric data, you can normally pass the value 1 for every column. This tells the `QueryTable` to automatically determine what type of data is being loaded. If the text file contains data that may be misinterpreted, you can use this property to tell the `QueryTable` what type of data is contained in each column. A value of 2 means the column contains text data. A value of 9 tells the `QueryTable` to skip the column entirely. The additional seven allowable numeric values are used to handle various arrangements of date data types. See the VBA help for more details on these.

Creating and Using Connection Files

The information used to create query tables can be stored to a file on disk, called a connection file, and reused. Connection files provide several advantages when used in conjunction with `QueryTables`. They allow you to create a variety of query tables, store them to disk, and distribute them with your applications. Users can also use connection files directly by selecting them from the Existing Connections dialog on the Data tab of the Ribbon. Finally, if some detail of the data source changes — for example, the server name where a relational database is located, you can simply edit the connection file to reflect the new information and redistribute it. `QueryTables` based on the connection file will then be updated automatically with the new information.

Unfortunately, connection files do not present a very consistent story. There are more than half a dozen potential file types available, and most of them can be used for multiple types of source data. Each of the three data sources covered in this chapter (databases, web queries, and text files) requires a different type of connection file. This section focuses on Office Data Connect (.odc) and Web Query (.iqy) files. Office Data Connect files store connection information for databases, and Web Query files store connection information for web queries.

Connection files are simply text files with a file extension that identifies their type and contents, formatted according to what that type of connection requires. Connection files can be stored and used from any accessible directory on a computer, but connection files that you want to be visible automatically in the Get External Data ⇨ Existing Connections list in the Excel user interface must be located in one of the following places:

- ❑ The `C:\...\My Documents\My Data Sources\` folder under the profile of the currently logged-in user.
- ❑ The `C:\Program Files\Common Files\ODBC\Data Sources\` folder.
- ❑ The `C:\Program Files\Microsoft Office\Office12\Queries\` folder.
- ❑ A custom location that has been defined by your network administrator through the use of Office policy settings.

Office Data Connect Files

An Office Data Connect (ODC) file contains XML data that specifies all of the information required to recreate an external connection to a database. See Chapter 12 for an introduction to XML if you aren't familiar with it. Although an ODC file contains XML data, it is not a well-formed XML file, so if you need to edit its contents, you may find it easier to use a text editor as opposed to an XML editor.

You can save the connection information for a `QueryTable` or `ListObject` to an ODC file by using the `SaveAsODC` method. The following line of code demonstrates how to save an ODC file that represents the first `QueryTable` created earlier in this chapter:

```
Sheet1.QueryTables(1).SaveAsODC "C:\Files\CustomersTable.odc"
```

ODC files can only be saved from `QueryTables` or `ListObjects` that were created through OLEDB. Because utilizing parameter queries requires the use of ODBC, you cannot save an ODC file for a `QueryTable` or `ListObject` that utilizes a parameter query.

Upon opening the `CustomersTable.odc` file in a text editor, you will see a lot of content, most of which is beyond the scope of this chapter. All the data you would potentially need to edit is contained within the two XML elements located near the top of the file. An abbreviated version of these two elements is as follows:

```
<xml id=docprops>
  <o:DocumentProperties....>
    <o:Name>Connection</o:Name>
  </o:DocumentProperties>
</xml>
<xml id=msodc>
  <odc:OfficeDataConnection....>
    <odc:Connection odc:Type="OLEDB">
      <odc:ConnectionString>
        Provider=Microsoft.ACE.OLEDB.12.0;Data Source=C:\Files\Northwind 2007.accdb;...
      </odc:ConnectionString>
      <odc:CommandType>Table</odc:CommandType>
      <odc:CommandText>Customers</odc:CommandText>
    </odc:Connection>
  </odc:OfficeDataConnection>
</xml>
```

The first XML element, called the `docprops` element, contains the name that will be displayed to the user in the Existing Connections dialog. This is automatically given the same name as the connection from which the ODC file was derived. Because you did not explicitly name your connection when you created it, VBA gave it the default name `Connection`. This is what you see stored in the `Name` element within the XML `docprops` element. You can simply edit this value to give the ODC file a more meaningful description.

The second XML element, called the `msodc` element, contains the connection string, command text, and command type used to create the `QueryTable`. The connection string that you see in the preceding example shows just the arguments you specified when you created the original `QueryTable`. If you open the ODC file yourself, you will see more than a dozen additional connection string arguments. These are simply the default values that OLEDB uses for any arguments you haven't specified.

Again, however, a connection string is simply plain text, so if you needed to modify the path to the database for all users of your application, you could simply type a new path into the `ConnectionString` element, save the file, and distribute it to everyone using your application. Any `QueryTables` built from the connection file would automatically update themselves with the new information.

To use the `CustomersTable.odc` file, you need to attach it to your `QueryTable` in the following manner:

```
qryTable.SourceConnectionFile = "C:\Files\CustomersTable.odc"
qryTable.RobustConnect = xlAlways
```

Be sure the `QueryTable` to which you are attaching the ODC file is the same `QueryTable` from which it was created. Once you have done this, the `QueryTable` will use the information contained in the specified ODC file to obtain its connection information whenever it needs to be refreshed.

It would be nice if you could simply create a new `QueryTable` using the ODC file as the data source in the first place. However, because the `Connection` argument of the `QueryTables.Add` method is required and does not accept an ODC file as its value, you must create the `QueryTable` initially in the manner shown in the previous section, and then attach the `QueryTable` to the ODC file afterward.

Web Query Files

Web Query (IQY) files are far simpler and easier to understand than ODC files. The process of generating an IQY file from an existing Web Query is a bit convoluted. There's no automated method of creating them like there is with ODC files. Creating an IQY file from an existing Web Query requires the following steps:

1. Open the Workbook Connections dialog using the Data ⇄ Manage Connections ⇄ Connections button.
2. In the Workbook Connections dialog, select the connection that corresponds to your Web Query and click the Properties button. This will display the Connection Properties dialog.
3. In the Connection Properties dialog, select the Definition tab and click the Edit Query button. This will display the Edit Web Query dialog.
4. You will see a toolbar across the top of the Edit Web Query dialog. The second button from the left on this toolbar is the Save Query button. Click this button and you will be prompted by a Save Workspace dialog to save your Web Query as an IQY file.

If you perform these steps on the connection for the Wall Street Journal Web Query created in the previous section, an IQY file with the following contents will be generated:

```
WEB
1
http://online.wsj.com/public/page/markets.html?mod=hpp_us_indexes

Selection=19
Formatting=All
PreFormattedTextToColumns=False
ConsecutiveDelimitersAsOne=False
SingleBlockTextImport=False
DisableDateRecognition=False
DisableRedirections=True
```

Only three entries in this file are important for your purposes. These are the URL, `Selection`, and `Formatting` entries. In fact, you could delete everything in the IQY file other than these three entries and the query would perform exactly as expected.

The URL is self-explanatory. It's the same URL used in the connection string when you originally created the Web Query. The `Selection` line specifies what the query should retrieve. This will be either the string value `EntirePage` or a number indicating the specific table you wish to retrieve. In this case the number 19 was generated, because that is the index number of the table specified when the Web Query was created.

The `Formatting` line specifies how the query should be formatted on the worksheet. Its potential values are `All`, `RTF`, or `None`, which correspond to the similarly named settings for the `WebFormatting` argument described in the previous section.

In addition to simplicity, IQY files have one additional advantage over ODC files for VBA programmers: you can create a new `QueryTable` directly from an IQY file using VBA. You could re-create the Wall Street Journal `QueryTable` example using the preceding IQY file in the following manner:

```
Sub CreateWebQueryFromIQY()  
  
    Dim qryTable As QueryTable  
    Dim rngDestination As Range  
    Dim strConnection As String  
  
    ' Define the connection string and destination range.  
    strConnection = "FINDER;C:\Files\Wall Street Journal Query.iqy"  
    Set rngDestination = Sheet7.Range("A1")  
  
    ' Create the QueryTable.  
    Set qryTable = Sheet7.QueryTables.Add(strConnection, rngDestination)  
  
    ' Populate the QueryTable.  
    qryTable.Refresh False  
  
End Sub
```

When creating a `QueryTable` from an IQY file, you pass `FINDER` as the first argument to the connection string and the full path and filename of the IQY file as the second argument. Notice that you don't need to set any additional properties of the `QueryTable` after creating it and prior to refreshing it. This is because all this information is contained in the IQY file.

The WorkbookConnection Object and the Connections Collection

New to Excel 2007 is an object and collection designed to manage all external data connections in a workbook. Each time you create any one of the built-in objects that Excel uses to manage external data, including `QueryTables`, `ListObjects`, and `PivotCaches`, you are also creating a new instance of a `WorkbookConnection` object. All of the `WorkbookConnection` objects in a given workbook are contained in the `Workbook.Connections` collection for that workbook.

You can also create a standalone `WorkbookConnection` object, one that is not associated with any external data container. The eventual intent for this feature is to allow you to create query tables, list objects, and all other external data containers using a `WorkbookConnection` object as the data source. However, as of Excel 2007, this feature has only been implemented for `PivotCache` objects. See Chapter 7 for more details on `PivotCache` objects.

Chapter 21: Managing External Data

You can create a new `WorkbookConnection` object using the `Add` or `AddFromFile` methods of the `Workbook.Connections` collection. The following example creates a new `WorkbookConnection` using the `Add` method:

```
Sub CreateNewConnection()  
  
    Dim objWBConnect As WorkbookConnection  
  
    Set objWBConnect = ThisWorkbook.Connections.Add( _  
        Name:="New Connection", _  
        Description:="My New Connection Demo", _  
        ConnectionString:="OLEDB;Provider=Microsoft.ACE.OLEDB.12.0;" & _  
            "Data Source=C:\Files\Northwind 2007.accdb", _  
        CommandText:="SELECT [First Name], [Last Name] FROM Customers", _  
        lCmdtype:=xlCmdSql)  
  
End Sub
```

After a `WorkbookConnection` object has been created, it is persisted when the workbook is saved. It will then be available any time the workbook is open, so there is no need to re-create it.

You can also use the `Workbook.Connections` collection to iterate through all the `WorkbookConnection` objects in a workbook and examine or modify their properties. The next example populates a worksheet with a list of all `WorkbookConnection` objects in the current workbook, and their type and their connection string if applicable:

```
Sub ExamineWorkbookConnections()  
  
    Dim lOffset As Long  
    Dim objWBConnect As WorkbookConnection  
  
    Sheet8.UsedRange.Clear  
    With Sheet8.Range("A1:C1")  
        .Value = Array("Connection Name", "Connection Type", "Connection String")  
        .EntireColumn.AutoFit  
    End With  
  
    For Each objWBConnect In ThisWorkbook.Connections  
        lOffset = lOffset + 1  
        Sheet8.Range("A1").Offset(lOffset, 0).Value = objWBConnect.Name  
        Sheet8.Range("A1").Offset(lOffset, 1).Value = objWBConnect.Type  
        If objWBConnect.Type = xlConnectionTypeODBC Then  
            Sheet8.Range("A1").Offset(lOffset, 2).Value = _  
                objWBConnect.ODBCConnection.Connection  
        ElseIf objWBConnect.Type = xlConnectionTypeOLEDB Then  
            Sheet8.Range("A1").Offset(lOffset, 2).Value = _  
                objWBConnect.OLEDBConnection.Connection  
        Else  
            Sheet8.Range("A1").Offset(lOffset, 2).Value = "Not Applicable"  
        End If  
    Next objWBConnect  
  
End Sub
```

Note that `WorkbookConnection` objects based on ODBC and OLE DB have additional child connection objects, the `ODBCConnection` object and `OLEDBConnection` object. These objects maintain the connection information required by ODBC and OLE DB.

External Data Security Settings

When you open an Excel 2007 workbook that contains connections to external data, you will encounter a security prompt like the one displayed in Figure 21-12.



Figure 21-12

You can enable your external connections by clicking the Enable Content button and choosing the option to enable the content. If you don't do this, any automatic refreshing of your `QueryTables`, `ListObject`, or other data containers attached to external data will be disabled without further warning. If you attempt to manually refresh any of the data in your workbook, you will be prompted by another security warning, shown in Figure 21-13.



Figure 21-13

If you click OK in this dialog, all of your external data connections will be enabled. If you click Cancel, all of your external data connections will remain disabled and the refresh will not take place.

You can avoid these security issues by placing your file in a trusted location, or by selecting the Enable All Data Connections option in the Trust Center ⇄ External Content section. Trusted locations are beyond the scope of this chapter, but you can enable all data connections in all workbooks you open in Excel 2007 using the following steps:

1. Click the Microsoft Office Button, and then click the Excel Options button. This will open the Excel Options dialog.
2. In the Excel Options dialog, select Trust Center from the list on the left side.
3. Next, click the Trust Center Settings button on the right side of the Excel Options dialog. This will open the Trust Center dialog.
4. In the Trust Center dialog, select External Content from the list on the left side.

Chapter 21: Managing External Data

You should now see at the dialog as it appears in Figure 21-14.

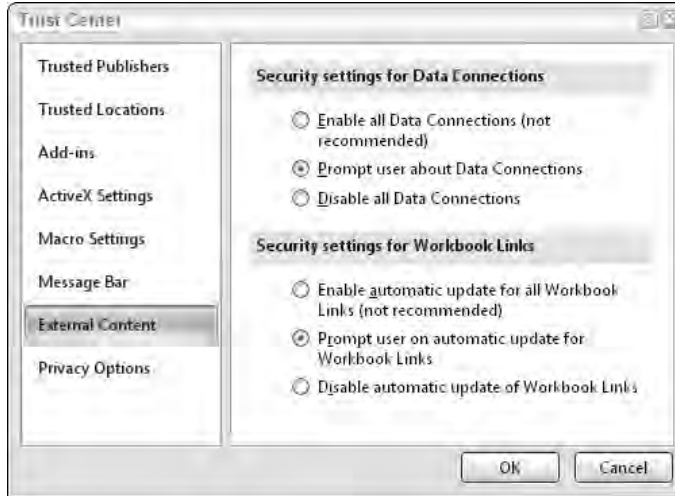


Figure 21-14

If you select the Enable all Data Connections option from the Security Settings for Data Connections section on the left side of the dialog, your external connections will be enabled automatically. Note that this setting can only be made manually; there's no way to use VBA to dynamically enable content on a user's computer. Also, if you're working on a corporate network, your network administrator may set security policies that do not allow you to make this change at all.

Summary

This chapter examined the built-in features Excel provides for dealing with external data, as exposed through the `QueryTable` object. It showed how to use query tables to retrieve data from relational databases, web sites, and text files. The chapter examined how connection files allow you to persist `QueryTable` connection information to an easily modified and distributed text file, and you took a brief look at the two new objects used to manage connection information in Excel 2007: the `Workbook.Connections` collection and the `WorkbookConnection` object.

The Trust Center and Document Security

Document security issues have become inescapable for Excel VBA developers. Even if you intend to develop macros only for use on your own computer, you still need to understand at least a few document security settings in order for them to run properly. This chapter covers the security settings in the Trust Center, as well as automating the removal of personal information from Excel workbooks.

The Trust Center

The Trust Center is the new user interface for all settings related to document security in Office 2007. Many of these settings have been with Office for a long time and were simply relocated to the Trust Center user interface. There are also some new and enhanced document security features in Office 2007. Many of the Trust Center settings are specific to Excel, but some settings will affect all Office applications.

The Trust Center user interface is buried a bit deeply within the Excel user interface. You access the Trust Center dialog in the following manner:

1. Click the Microsoft Office Button and then click the Excel Options button. This will open the Excel Options dialog.
2. In the Excel Options dialog, select Trust Center from the list on the left side.
3. Next, click the Trust Center Settings button on the right side of the Excel Options dialog. This will open the Trust Center dialog, shown in Figure 22-1.

The Trust Center organizes all Excel 2007 document security options under eight categories. The following sections review the settings contained in each of these categories.



Figure 22-1

Trusted Publishers

The Trusted Publishers category, shown in Figure 22-1, lists the digital certificates the user has chosen to trust. A digital certificate is a software signature that can be used to “sign” an Excel add-in or other type of application. The certificate ensures that the application actually originates from where it claims to, and that it has not been tampered with since it was signed. Digital certificates are also called digital signatures.

Digital certificates must be obtained from a certification authority such as VeriSign (www.verisign.com). They are relatively expensive, costing approximately \$500 per year to maintain as of this writing. Because of this, the use of digital certificates is rare in Excel VBA development and is not covered in detail in this chapter.

Trusted Locations

Trusted locations are a new concept in Office 2007. A trusted location is a folder on your computer or on your network that has been designated as containing only safe documents. Excel workbooks, add-ins, and other Office documents that are placed in a trusted location will not be subject to any security restrictions. The Trusted Locations category is shown in Figure 22-2.

The Trusted Locations list shows two types of trusted locations. User Locations are trusted locations that are either added by the user or trusted by default. All Office 2007 template and startup folders, as well as the Office library folder, are trusted locations by default. Additional trusted folders can be added by the user. Policy Locations are trusted locations that have been defined by the network administrator for all users.

You can add new trusted locations using the Add New Location button. This button displays the dialog shown in Figure 22-3.

The Browse button is used to select a folder to trust. You can choose to trust all subfolders of the selected folder by placing a check mark in the Subfolders of this location Are Also Trusted checkbox. You can also add a description to the trusted location that will be displayed when this location is selected in the Trusted Locations list.

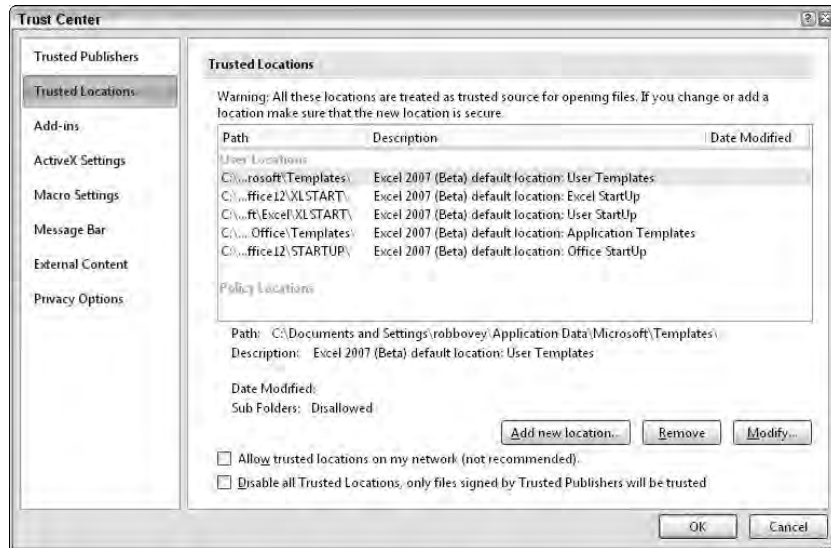


Figure 22-2



Figure 22-3

You cannot trust the root folder (for example, C: \) of any disk on your computer. By default, you cannot designate a network folder as a trusted location. If you want to trust a network folder, you must place a check mark in the Allow Trusted Locations on My Network checkbox shown at the bottom of Figure 22-2.

Selecting a location from the Trusted Locations list and clicking the Remove button will remove that folder from the list and prevent it from being a trusted location. Selecting a location from the Trusted Locations list and clicking the Modify button will redisplay the dialog shown in Figure 22-3 and allow you to make modifications to the selected location. These two buttons are disabled if no location is currently selected in the Trusted Locations list.

You can disable the trusted locations feature entirely by placing a check mark in the Disable All Trusted Locations checkbox shown at the bottom of Figure 22-2. If this option is selected, no folders of any type will be implicitly trusted, and only documents from trusted publishers will bypass security settings.

Add-ins

The Add-ins category, shown in Figure 22-4, contains some of the settings that control whether or not Excel 2007 trusts the code contained in Excel add-ins. The key word here is “some,” because even if none of the settings in this category are selected, there are overlapping settings in other categories covered later in this chapter that can prevent an Excel add-in from running. However, an incorrect setting in the Add-ins category can prevent *any* Excel add-in from functioning properly, so it is important to understand the settings in this category in order to rule out problems with them when you are troubleshooting problems with your Excel applications.

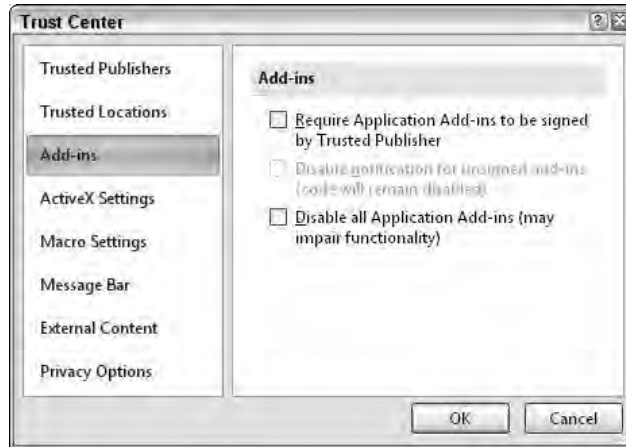


Figure 22-4

The Add-ins category contains the following settings:

- Require Application Add-ins to be signed by Trusted Publisher — If this checkbox is checked, only add-ins signed by a trusted publisher, as described in the “Trusted Publishers” section, will be allowed to run. This setting overrides the fact that an add-in may be located in a trusted folder. If this setting has been selected and you attempt to load an add-in that has not been signed by a trusted publisher, the add-in will be opened but its code will not run. You will receive a notification in the message bar that the add-in was disabled. The message bar will be covered in the Message Bar category later in this chapter.
- Disable notification for unsigned add-ins (code will remain disabled) — This setting is only enabled when the preceding checkbox has been checked. If this setting is selected, add-ins not signed by trusted publishers will be disabled without any notification.
- Disable all Application Add-ins (may impair functionality) — If this checkbox is checked, all Excel add-ins will be disabled without notification. This setting overrides all other Trust Center security settings, including trusted publishers, trusted locations, and macro settings (covered later in this chapter). If this setting is selected, the other two settings in this category will be disabled.

Note that if you modify any of the settings in this category, you must close and restart Excel for the changes to take effect.

ActiveX Settings

The ActiveX Settings category contains settings that control how Excel treats ActiveX controls embedded in documents. The most common example of ActiveX controls embedded in an Excel worksheet are Excel VBA applications that have used ActiveX controls from the Controls group in the Developer tab of the Ribbon, as shown in Figure 22-5.

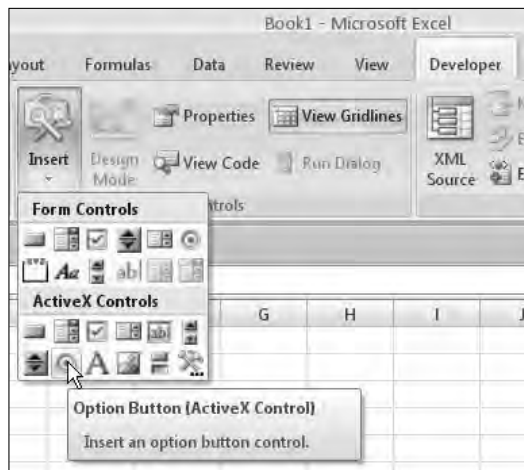


Figure 22-5

Keep in mind that the ActiveX Settings category applies only to ActiveX controls embedded in worksheets. It has no effect on ActiveX controls on UserForms in a VBA project. The settings contained in the ActiveX Settings category are shown in Figure 22-6.

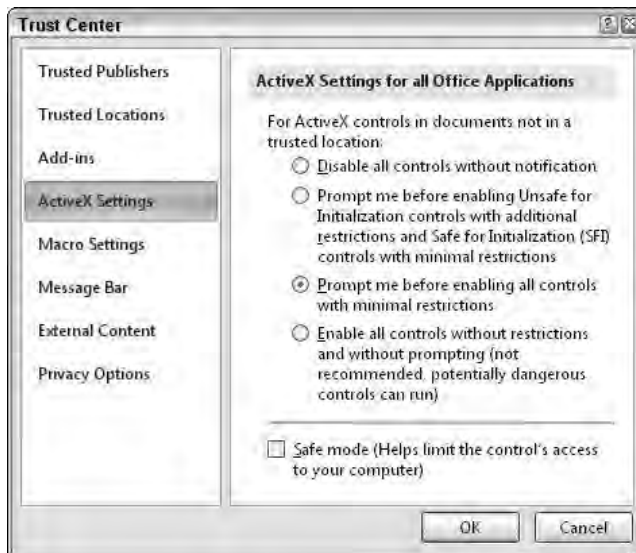


Figure 22-6

Chapter 22: The Trust Center and Document Security

Note that the settings in this category will apply to all Office applications, not just Excel. The settings in the ActiveX control category are also overridden by the Trusted Locations settings. That is, if a workbook is located in a trusted folder, the settings in the ActiveX category will not apply to it. The one exception to this is an ActiveX control that has a *kill bit* set for it in the registry. This can occur if a particular ActiveX control is known to be malicious. ActiveX controls with a kill bit set will not run under any circumstances.

ActiveX Settings can be a bit confusing, because their effect can vary depending on the specific type of controls a document contains. There are two types of ActiveX controls:

- SFI—Safe for initialization
- UFI—Unsafe for initialization

How a control is configured to be SFI or UFI is a detail that is beyond the scope of this discussion, and there is no simple way to determine whether a given ActiveX control is considered SFI or UFI. Just remember that all ActiveX controls from the Controls group in the Developer tab of the Ribbon are SFI, whereas other ActiveX controls may be either. For example, the `Frame` and `MultiPage` controls that appear in the Control Toolbox in the Visual Basic Editor are both UFI controls. In general, UFI controls are subject to more security restrictions than SFI controls.

The following list describes the settings in the ActiveX Settings category and explains their effect on ActiveX controls embedded in Excel worksheets. The descriptions assume the Excel workbook containing the ActiveX control is not located in a trusted folder:

- Disable all controls without notification— All ActiveX controls in all workbooks will be disabled. You will not be notified that the controls have been disabled.
- Prompt me before enabling Unsafe for Initialization controls with additional restrictions and Safe for Initialization (SFI) controls with minimal restrictions— If all ActiveX controls in the workbook are SFI controls, then all controls are loaded normally. If the workbook contains at least one UFI control, all controls are disabled and a Message Bar notification is displayed. If you use the Message Bar to enable content, SFI controls are loaded normally, but UFI controls are loaded without any persisted values. This means that enabling ActiveX controls with this option selected will cause you to lose all customizations you have made to any UFI controls in the workbook.
- Prompt me before enabling all controls with minimal restrictions— This is the default option. If all ActiveX controls in the workbook are SFI controls, then all controls are loaded normally. If the workbook contains at least one UFI control, all ActiveX controls are disabled and a Message Bar notification is displayed. If you use the Message Bar to enable content, all controls are loaded normally.
- Enable all controls without restrictions and without prompting (not recommended, potentially dangerous controls can run)— All ActiveX controls in all workbooks are allowed to run, with no notification that they are present.
- Safe mode (Helps limit the control's access to your computer)— Applies only to SFI controls. Some SFI controls have more extensive (and possibly more dangerous) capabilities when loaded in unsafe mode than when loaded in safe mode. Selecting this checkbox causes Excel to always load SFI controls in safe mode. UFI controls by definition are always in unsafe mode, so you must use the options just described to control how they are managed. This setting is disabled and unavailable if you've selected the Disable All Controls without Notification option.

Macro Settings

The Macro Settings category contains settings that control how Excel responds to workbooks and add-ins that contain macros. These settings do not apply to workbooks or add-ins that have been opened from a trusted location. See the “Trusted Locations” section for more details. Unlike ActiveX Settings, changes to Macro Settings only affect Excel. The settings contained in the Macro Settings category are shown in Figure 22-7.

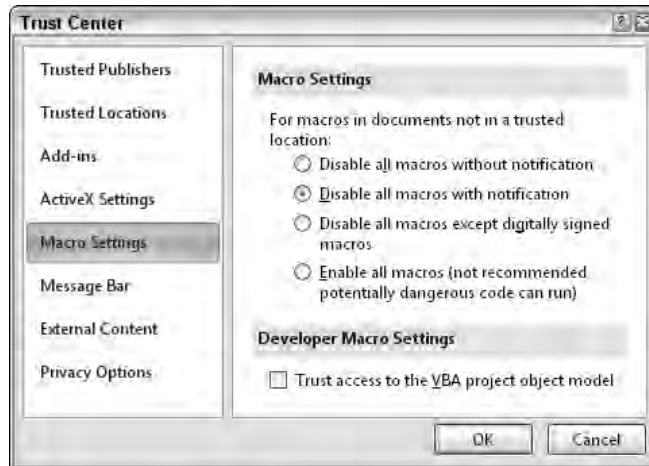


Figure 22-7

The following list describes the settings in the Macro Settings category and explains their effect on workbooks and add-ins that contain macros. Note that changes to these settings will not affect currently open workbooks. You will need to close and reopen any currently open workbook in order for the new settings to take effect.

- ❑ **Disable all macros without notification** — If this option is selected, all macros in all workbooks and add-ins are disabled without Message Bar notification (see the following “Message Bar” section for more information on the Message Bar). Note that you will still be able to view and edit macros in workbooks opened with this setting selected, but you will not be able to run those macros.
- ❑ **Disable all macros with notification** — If this option is selected, you will be presented with a security alert dialog, shown in Figure 22-8, each time you open a workbook or add-in that contains macros. Using the security alert dialog, you can choose to enable or disable macros on an individual basis for each file you open. This is the default option for the Macro Settings category.
- ❑ **Disable all macros except digitally signed macros** — This option operates exactly like the Disable All Macros without Notification option on workbooks containing VBA code that have not been digitally signed. If a workbook containing VBA code has been digitally signed by a trusted publisher, the workbook is opened with macros enabled. If a workbook containing VBA code has been digitally signed but you have not yet trusted the signer, you are presented with a security alert dialog, similar to the one shown in Figure 22-8, that allows you to either enable or disable the VBA code in the signed workbook. For more information on trusted publishers, see the “Trusted Publishers” section.

Chapter 22: The Trust Center and Document Security

- ❑ Enable all macros (not recommended, potentially dangerous code can run) — If this option is selected, all macros in all workbooks and add-ins will be enabled and you will not be notified when you open a workbook or add-in that contains VBA code.
- ❑ Trust access to the VBA project object model — Some VBA programs are designed to operate on the VBA project of a workbook or add-in. These programs are usually tools designed for use by VBA programmers. This checkbox must be checked in order for tools of this nature to function. This checkbox is unchecked by default, and VBA code is not allowed to access the VBA project of any workbook or add-in.



Figure 22-8

Message Bar

The Message Bar is a new feature in Office 2007. It alerts you to various document security-related actions taken by Office and allows you to decide what to do about them. The difference between the Message Bar and the dialog-based alerts from previous versions of Office that it replaced is that the Message Bar is not modal and therefore does not interfere with your use of the application. You can respond to the Message Bar immediately or choose to ignore it while you continue to work. If you ignore the Message Bar alert, the security action it is warning you about will remain in effect by default until you choose otherwise.

For example, in the Add-ins category, discussed previously, there is a setting that allows you to Require Application Add-ins to be Signed by Trusted Publisher. If you choose this setting and attempt to open an unsigned add-in, you will receive the Message Bar notification shown in Figure 22-9.

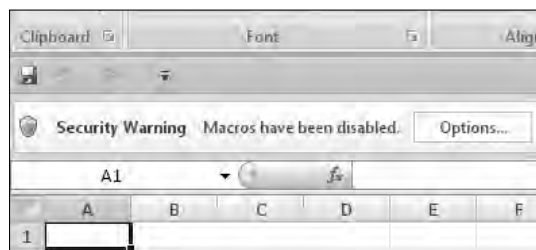


Figure 22-9

The Message Bar notifies you that macros have been disabled in the unsigned add-in. The Message Bar will remain visible, but will not interfere with your use of Excel until you click the Options button and direct Excel to leave the unsigned add-in disabled or enable its macros.

The Message Bar category contains settings that control the display of the Message Bar, as well as a setting related to Trust Center activity logging. The settings contained in the Message Bar category are shown in Figure 22-10.

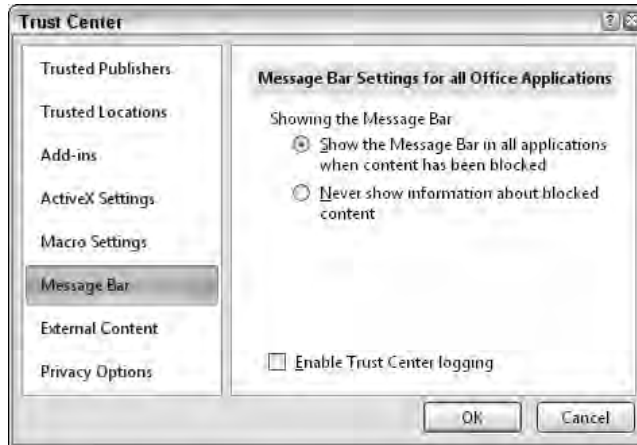


Figure 22-10

Note that the settings in the Message Bar category apply to all Office applications, not just Excel. The following list describes each of the settings in the Message Bar category:

- ❑ Show the Message Bar in all applications when content has been blocked — This is the default option. It directs the Trust Center to display a Message Bar notification when it has blocked some content in your document, as specified by your current security settings.
- ❑ Never show information about blocked content — Selecting this option will turn off the Message Bar. The Trust Center will continue to block everything specified by your security settings, but it will not notify you that it has done so. If something is not functioning in your application and no Message Bar notification is being displayed, check to see if this option has been selected.
- ❑ Enable Trust Center logging — Place a check mark in this box if you want the Trust Center to record all security-related activities in a log file. The log file will be named `XLTC.D.LOG` and will be located in the directory `C:\Documents and Settings\\Local Settings\Application Data\Microsoft\Office\TCDiag`.

External Content

The External Content category controls how Excel updates content external to a workbook. There are two types of external content regulated by the External Content category settings. The first is content from databases and other non-Excel sources, which are usually accessed through PivotTables, ListObjects, and QueryTables. The second are external Excel workbooks, which are most commonly linked to from worksheet formulas. The two types of content are controlled by separate groups of settings under the External Content category. These settings are shown in Figure 22-11.



Figure 22-11

- ❑ Security settings for Data Connections — These options apply to external connections to non-Excel data sources.
 - ❑ Enable all Data Connections (not recommended) — If this option is selected, you will not receive any warning when opening a workbook that contains connections to external data. All external data connection features, including automatic data refreshing of data, will be enabled.
 - ❑ Prompt user about Data Connections — If this option is selected, you will receive a Message Bar notification when opening a workbook that contains connections to external data. All external data connection features will be disabled until you click the Options button on the Message Bar and choose to enable the connections. This is the default option for this section.
 - ❑ Disable all Data Connections — If this option is selected, all external data connection features will be disabled automatically, with no Message Bar notification.
- ❑ Security settings for Workbook Links — These options apply to links to other Excel workbooks.
 - ❑ Enable automatic update for all Workbook Links (not recommended) — If this option is selected, all links to external Excel workbooks will be updated automatically, and you will not receive any Message Bar notification.
 - ❑ Prompt user on automatic update for Workbook Links — If this option is selected, you will receive a Message Bar notification when opening a workbook that contains links to external workbooks. The links will not be updated unless you click the Options button on the Message Bar and choose to enable the links.
 - ❑ Disable automatic update of Workbook Links — If this option is selected, all links to external Excel workbooks will be disabled, with no Message Bar notification given.

Privacy Options

The Privacy Options category is a catchall category for a number of settings loosely related to privacy and/or security that didn't fit into any of the other Trust Center categories. The settings in the Privacy Options category are shown in Figure 22-12.

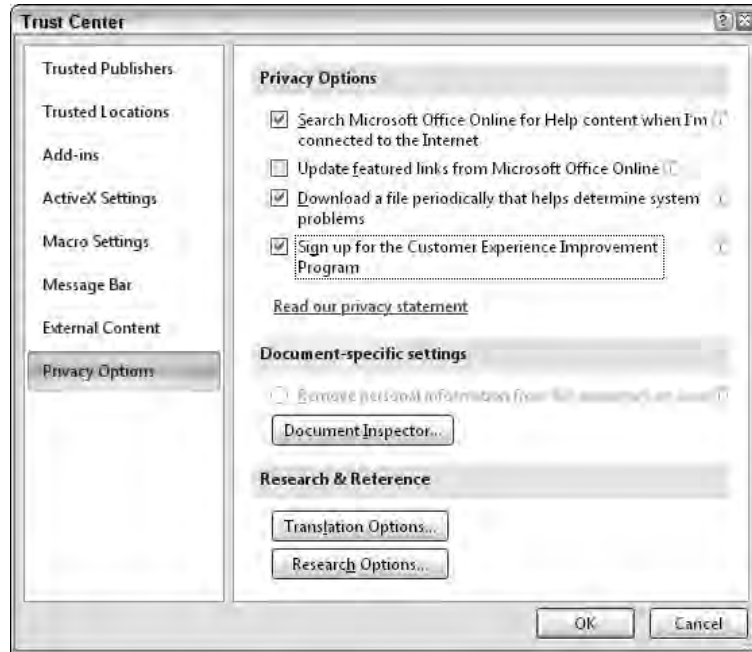


Figure 22-12

The settings in the Privacy Options category have no use in or effect on VBA programs, so they are not covered in this section. What are covered in this section are the new `RemoveDocumentInformation` method and `DocumentInspectors` collection. Because the Excel user interface corresponding to these new VBA capabilities is located in the Privacy Options category, it is examined here briefly.

The Document Inspector is a new feature of Office that allows you to search all the places in your document where items of personal information might be stored and then remove those items. In the Trust Center, if you click the Document Inspector button, the dialog in Figure 22-13 will be displayed.

You can choose which areas of your workbook you want to inspect and then click the Inspect button to begin the process. The Document Inspector will analyze your workbook and produce a report of its findings, using the dialog shown in Figure 22-14.

If the Document Inspector locates data in any of the places you directed it to analyze, it will give you the option to remove that data. When using the Document Inspector, keep in mind that it is not smart. It cannot tell the difference between data critical to your document and unwanted personal information. The Document Inspector simply removes all data from the specified place if you click the Remove All button.

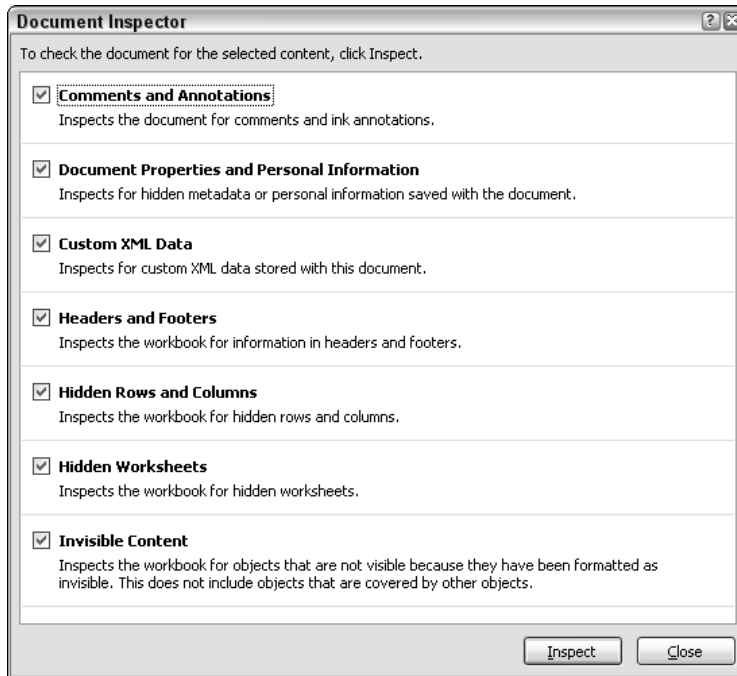


Figure 22-13

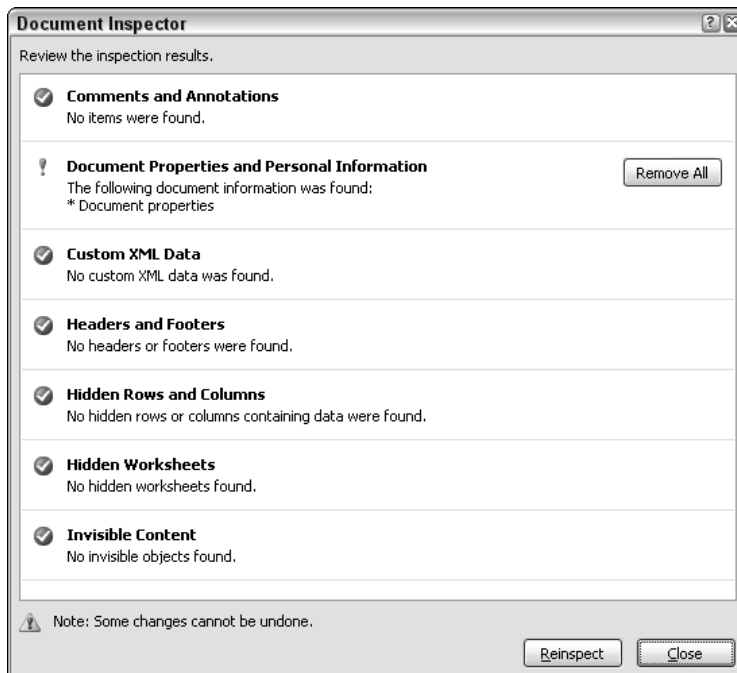


Figure 22-14

You can probably see that, from the perspective of an Excel developer, the Document Inspector looks like a very dangerous thing. If a user has unfettered access to your application workbook and that workbook contains critical information in hidden rows, columns, or worksheets, the Document Inspector can easily strip that data out, leaving the workbook crippled.

Fortunately, there is a simple way to prevent the Document Inspector from destroying your workbook. If there are any protected worksheets in a workbook, it will prevent the Document Inspector from running on the workbook. Therefore, to protect your workbooks from the Document Inspector, make sure they contain at least one protected worksheet. This worksheet doesn't need to contain any critical data, and it can even be hidden from the user.

Automating Document Inspection

The new document inspection capabilities in Excel 2007 can also be a good thing. In previous versions of Office, it was very easy to release sensitive information accidentally hidden in a workbook with other data intended for release.

The Document Inspector, covered in the previous section, provides a manual method to remove this type of information. If you are faced with processing a large number of workbooks, however, you need an automated solution. This solution is provided by the new `RemoveDocumentInformation` method and `DocumentInspectors` collection of the `Workbook` object in Excel 2007. These two features somewhat arbitrarily divide the Document Inspector sections into two groups with different capabilities for each, as described in the following sections.

Note that like the Document Inspector user interface, the `RemoveDocumentInformation` method, and the `DocumentInspectors` collection can only operate on open workbooks with all worksheets unprotected.

The RemoveDocumentInformation Method

The `RemoveDocumentInformation` method is a method of the `Workbook` object that provides a VBA interface to the features exposed by the Comments and Annotations and Document Properties and Personal Information sections of the Document Inspector user interface. This method provides very granular control over how the specific pieces of information that fall into these two sections are handled.

The `RemoveDocumentInformation` method takes a single argument that specifies the type of information you want to remove from the workbook. If you want to remove more than one type of information, you can call the method multiple times with different arguments each time. There is also an argument that will remove all types of information covered by this method. This is the `xLRDIA11` argument shown in the list that follows.

The following list shows the complete list of arguments to the `RemoveDocumentInformation` method:

- `xLRDIA11`
- `xLRDIComment`
- `xLRDIContentType`

- xlRDIDefinedNameComments
- xlRDIDocumentManagementPolicy
- xlRDIDocumentProperties
- xlRDIDocumentServerProperties
- xlRDIDocumentWorkspace
- xlRDIEmailHeader
- xlRDIInactiveDataConnections
- xlRDIInkAnnotations
- xlRDIPublishInfo
- xlRDIRemovePersonalInformation
- xlRDIPrinterPath
- xlRDIRoutingSlip
- xlRDIScenarioComments
- xlRDISendForReview

The type of information affected by most of these arguments is easy to determine based on the argument name. If you wanted to remove document properties, you would obviously use the `xlRDIDocumentProperties` argument, for example.

The following sample code shows how you would loop through all .xlsx format workbooks in the `C:\Files` folder, and use the `RemoveDocumentProperties` method to remove all cell comments and document properties from those workbooks:

```
Sub RemoveComments()  
  
    Dim strPath As String  
    Dim strFileName As String  
    Dim wkbBook As Workbook  
  
    Application.ScreenUpdating = False  
  
    strPath = "C:\Files\  
    strFileName = Dir$(strPath & "*.xlsx")  
  
    Do While Len(strFileName) > 0  
        Set wkbBook = Workbooks.Open(strPath & strFileName)  
        wkbBook.RemoveDocumentInformation xlRDIComments  
        wkbBook.RemoveDocumentInformation xlRDIDocumentProperties  
        wkbBook.Save  
        wkbBook.Close False  
        strFileName = Dir$()  
    Loop  
  
    Application.ScreenUpdating = True  
  
End Sub
```

Attempting to invoke the `RemoveDocumentInformation` method of a workbook that contains any protected worksheets will cause a VBA run-time error, so be sure all worksheets in the documents you want to process are unprotected.

The DocumentInspectors Collection

The `DocumentInspectors` collection of the `Workbook` object contains a group of `DocumentInspector` objects that provide a VBA interface to the features exposed by the last five sections of the Document Inspector user interface. These features and their corresponding index numbers in the `DocumentInspectors` collection are shown here:

- ❑ 1—Custom XML Data
- ❑ 2—Headers and Footers
- ❑ 3—Hidden Rows and Columns
- ❑ 4—Hidden Worksheets
- ❑ 5—Invisible Content

Individual `DocumentInspector` objects can only be accessed by index number from the `DocumentInspectors` collection, or by iterating the entire collection with a `For . . . Each` loop.

There are a number of differences between capabilities of the `RemoveDocumentInformation` method and the `DocumentInspector` object. These differences are summarized as follows:

- ❑ The `DocumentInspector` object has an `Inspect` method that can be used to passively determine whether a workbook contains any information that would be removed by the object, without actually doing so.
- ❑ Using the `Fix` method of the `DocumentInspector` object to remove affected information is an all-or-nothing process. For example, if you choose to remove headers and footers using the `DocumentInspector` object designed for that purpose, all headers and footers in all worksheets in the workbook will be removed. There is no way to limit the removal to headers or footers alone.
- ❑ If the `Fix` method of the `DocumentInspector` object fails as a result of the workbook containing protected worksheets, or for any other reason, it will simply return an error code and description of the problem rather than causing a VBA run-time error.

The following sample code shows how you would loop through all `.xlsx` format workbooks in the `C:\Files` folder and use a `DocumentInspector` object to remove all hidden rows and columns from those workbooks. If the `DocumentInspector` object is unable to perform its task on a workbook, the code will display a message box with the reason for the failure:

```
Sub RemoveHiddenRowsAndColumns()  
  
    Dim objDI As DocumentInspector  
    Dim uStatus As MsoDocInspectorStatus  
    Dim strResult As String  
    Dim strPath As String
```

Chapter 22: The Trust Center and Document Security

```
Dim strFileName As String
Dim wkbBook As Workbook

Application.ScreenUpdating = False

strPath = "C:\Files\"
strFileName = Dir$(strPath & "*.xlsx")

Do While Len(strFileName) > 0

    Set wkbBook = Workbooks.Open(strPath & strFileName)
    Set objDI = wkbBook.DocumentInspectors(3)

    objDI.Fix uStatus, strResult
    If uStatus = msoDocInspectorStatusError Then
        ' If the Fix method could not complete, display the error.
        MsgBox wkbBook.Name & ": " & strResult, vbExclamation
    Else
        ' Otherwise save the changes made to the workbook.
        wkbBook.Save
    End If

    wkbBook.Close False
    strFileName = Dir$()

Loop

Application.ScreenUpdating = True

End Sub
```

As demonstrated in this code sample, the two arguments to the `DocumentInspector.Fix` method are actually return values. After the method has attempted to run, these arguments tell you what happened and why.

Summary

All things considered, from a security standpoint Excel 2007 has become much less friendly to VBA code than previous versions of Excel. The good news is that all of the settings that control how Excel responds to VBA code have been gathered in one location, the new Trust Center. When you are having problems getting your VBA applications to run, you will need an excellent understanding of the settings in the Trust Center to distinguish between bugs in your code and problems caused by Excel security features.

If you or your company needs to send workbooks to someone who may try to extract personal information from them, the `RemoveDocumentInformation` method and the `DocumentInspector` object can help you ensure that your workbooks don't contain any such information.

Browsing OLAP Data Sources with Excel

The dominant database type in most organizations is the OLTP (On-line Transaction Processing) database. Indeed, most of you are probably working with some form of an OLTP database as you read this. The main characteristics of this type of database are: they typically contain many tables, each table usually contains multiple relationships with other tables, and records within any given table can be routinely added, deleted, or updated.

Although OLTP databases are effective in gathering and managing data, they typically don't make for effective data sources for reporting, for three main reasons:

- ❑ **Complexity:** The large number of tables and relationships that can exist in an OLTP database can leave you wondering exactly which tables to join and how the tables relate to each other.
- ❑ **Volume:** OLTP databases normally contain individual records. Lots of them. To create any number of aggregate reports and views, you would have to run views that group, aggregate, and sort records on the fly. The sheer volume of data in the database could very well inundate you with painfully slow reporting.
- ❑ **Consistency:** By its very nature, the records in a transactional database are ever-changing. Building a reporting solution on top of this type of database will inevitably lead to inconsistent results from month to month, or even from day to day.

Some organizations avoid these woes by building their reporting solutions on top of OLAP (On-Line Analytical Processing) databases. OLAP databases are data islands that are isolated from the hustle and bustle of transactional databases. An OLAP database can help alleviate these problems in the following ways:

- ❑ **Structured Data:** In an OLAP database, all of the relationships between the various data points have been predefined and stored in what are known as *cubes*. These cubes contain the hierarchical structures that allow for the easy navigation of available data dimensions and measures. With this configuration, you no longer have to create joins yourself or try to guess how one data table relates to another. All of that complexity is taken care of behind the scenes, leaving you free to develop the reports you need.
- ❑ **Predefined Aggregations:** The data in an OLAP database is not only organized, but it is aggregated. This means that grouping, sorting, and aggregations are all predefined in OLAP databases. In addition, OLAP databases make heavy use of indexes, a technique that allows a database to search for records more efficiently. All of this amounts to reporting solutions that are optimized to provide the reports you need as fast as possible.
- ❑ **Consistent Results:** OLAP databases only contain snapshots of data. That is to say, the data in an OLAP database is typically historical data that is read-only, stored solely for reporting purposes. New data is typically appended to the OLAP database on a regular basis, but the existing data is rarely edited or deleted. This allows you to retrieve consistent results when building your reporting solutions.

Excel has some effective built-in tools that allow for the exploration and reporting of data from OLAP databases. In this chapter, you will discover some of the ways you can browse the OLAP data sources in your organization via Excel and VBA.

You cannot create an OLAP database using Excel. OLAP databases are typically created with SQL Server Analysis Services. If your organization does not utilize OLAP databases, you may want to speak with your SQL Server DBA to discuss the possibility of some OLAP reporting solutions.

Analyzing OLAP Data via Pivot Tables

Every so often, you will run into some functionality where the user interface provided in Excel is the most effective way to perform a task. The process of browsing an OLAP data source is just such a task. In Excel, the most effective way to browse an OLAP data source is indeed through a pivot table. This section walks through the process of connecting to and browsing an OLAP data source via a pivot table.

Connecting to an OLAP Data Source

The process of connecting to an OLAP data source is similar to that of any other external data source. In the Data tab, select the From Other Data Sources option and choose From Analysis Services, as demonstrated in Figure 23-1.

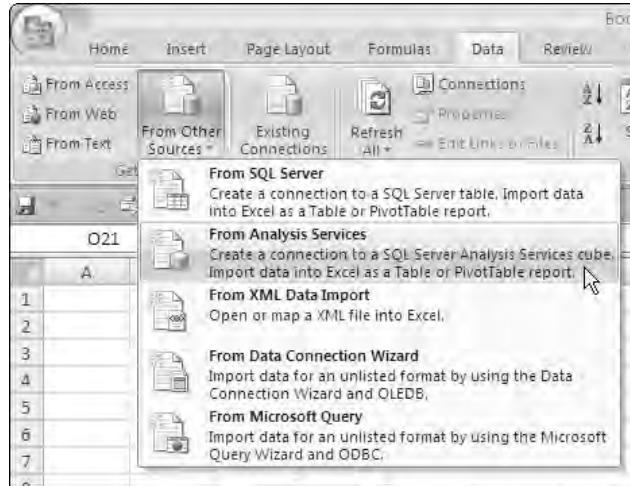


Figure 23-1

From here, you will be taken through the Data Connection Wizard, where you will provide all the information necessary to connect to the appropriate cube. Once you have gone through the entire dialog for the Data Connection Wizard, you will be presented with the options illustrated in Figure 23-2. Select the option of viewing the data in a PivotTable Report.

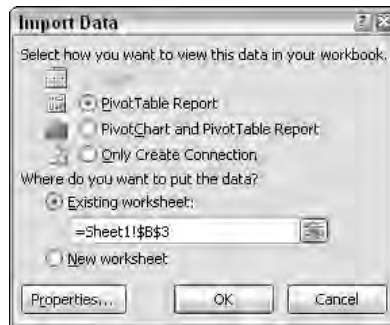


Figure 23-2

Clicking the OK button will create a pivot table whose pivot cache is connected to the OLAP data source to which you just connected. You are now ready to browse your OLAP data source.

The examples in this chapter use the Analysis Services Tutorial cube that comes with SQL Server Analysis Services 2005. To follow along using this sample data source, simply load the Analysis Services Tutorial OLAP cube found in the supporting files for SQL Server Analysis Services 2005.

Browsing the OLAP Data Source

The PivotTable Field List for a pivot table connected to an OLAP data source will look somewhat different from that of a standard pivot table. The PivotTable Field List for an OLAP pivot table will represent the structure of the OLAP cube you are connected to.

To effectively browse an OLAP cube, you need to understand the components that make up the structure of a cube. Figure 23-3 illustrates the basic structure of a typical OLAP cube.

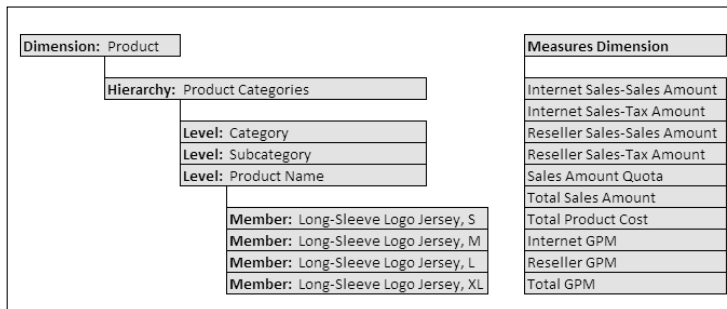


Figure 23-3

- ❑ **Dimension:** A dimension is a high-level classification of data that contains items that are used as criteria by which measures can be sliced. An example of a dimension would be a Product dimension. A single OLAP cube can have multiple dimensions.
- ❑ **Hierarchy:** A hierarchy is the predefined aggregation of levels within a particular dimension. Hierarchies allow a user to work with multiple levels without a previous knowledge of the parent-child relationships between the levels.
- ❑ **Level:** A level is a category of aggregation within a hierarchy. For dimensions with multiple layers of information, each layer is a level.
- ❑ **Member:** A member is an individual data item within a dimension. Members are typically referenced in a structured fashion through the hierarchy and level. In the example shown in Figure 23-3, the members you see belong to the Product Name level. The other levels have their own members that are not shown here.
- ❑ **Measures Dimension:** The measures dimension contains the aggregated data that can be sliced by any of the dimensions, hierarchies, levels, and members in the cube.

Once you understand how the data in an OLAP cube is structured, the structure represented in the PivotTable Field List starts to make sense. You will see Measures (represented by the sigma icon), Dimensions (represented by a table icon), Hierarchies, and Levels. You will find that you can navigate through the OLAP cube with ease using your pivot table's field list.

In Figure 23-4, you will see that the Internet Sales-Sales Amount measure has been added, and that measure is sliced by the Product Categories hierarchy found in the Product dimension.

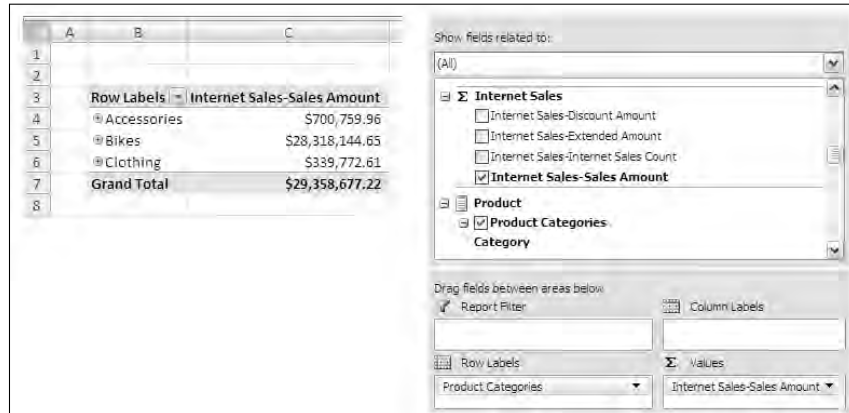


Figure 23-4

With just a few steps, you've built a pivot table report that allows you to drill into the levels of the pre-defined Product Categories hierarchy, as demonstrated in Figure 23-5.

Note that the layout of your resulting pivot table may look slightly different based on the default report layout. Your layout options are tabular, compact, or outline.

Category	Subcategory	Product Name	Internet Sales-Sales Amount
Accessories			\$700,759.96
Bikes	Mountain Bikes	Mountain-100 Silver, 38	\$197,199.42
		Mountain-100 Silver, 42	\$142,799.58
		Mountain-100 Silver, 44	\$166,599.51
		Mountain-100 Silver, 48	\$122,399.64
		Mountain-100 Black, 38	\$165,374.51

Figure 23-5

You can imagine a reporting solution that consists entirely of pivot table reports on top of an OLAP cube. This would allow your users to have a robust data source at their fingertips without leaving the familiar Excel environment. And because the data contained in the OLAP pivot tables does not reside in a pivot cache on the local machine, your reporting solutions are inherently streamlined and efficient. Furthermore, because the reporting solution is designed in Excel with Excel pivot tables, you would be able to enhance your reporting solution using the PivotTable VBA techniques covered in Chapter 7.

Limitations of OLAP-based PivotTables

For the most part, pivot tables that are based on OLAP data sources look, feel, and act like a standard pivot table. However, you must remember that an OLAP data source is ultimately controlled by the Database Administrator responsible for maintaining the Analysis Services server. That control encompasses every aspect of the OLAP cube's behavior, from the dimensions and measures included in the cube to the ability to drill into the details of a dimension. You, as the consumer of the OLAP data source, have limited control over how the OLAP cube ultimately looks and feels.

This limited control translates into some limitations to the actions you can take with your OLAP-based pivot tables. You should take these limitations into account before moving forward with an OLAP-based reporting solution.

When your PivotTable report is based on an OLAP data source:

- You cannot create a calculated field.
- You cannot create a calculated item.
- The page field settings are not available.
- You cannot change the function used to summarize a data field.
- The Show Pages command is disabled.
- The Show items with no data option is disabled.
- Any changes made to field names will be lost when you remove the field from the pivot table.
- The Subtotal hidden page items setting is disabled.
- The Background query option is not available.
- The Optimize memory checkbox in the PivotTable Options dialog box is disabled.

Understanding the MDX behind OLAP-based Pivot Tables

You may not know it, but when you are using a pivot table with an OLAP cube, you are sending the OLAP server MDX (Multidimensional Expression) queries. MDX is an expression language that is used to return data from multidimensional data sources (such as OLAP cubes).

To see the MDX query behind your OLAP-based pivot table, simply run the following procedure, which uses the MDX property of the PivotTable object:

```
Sub GetMDX()  
MsgBox ActiveSheet.PivotTables("PivotTable1").MDX  
End Sub
```

You will get a message box that looks similar to the one shown in Figure 23-6.

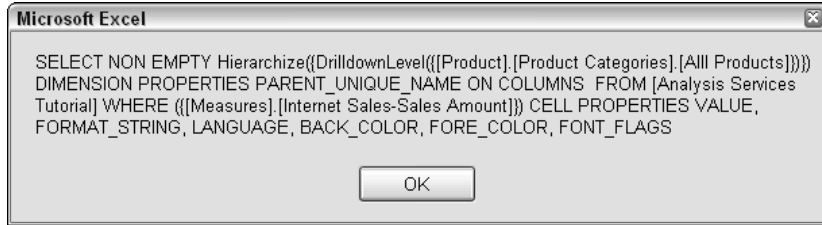


Figure 23-6

What you see in the resulting message box is the actual MDX query that was used to fill the pivot table with which you are working. Because the pivot table is refreshed or changed, subsequent MDX queries are passed to the OLAP database. The results of the query are sent back to Excel and displayed through the pivot table. This is how you are able to work with OLAP data without a local copy of a pivot cache.

Don't be fooled by the seeming complexity of this MDX statement. Excel tends to play it safe by throwing in superfluous syntax. Later you will learn how to decipher Excel's MDX. In the meantime, take a moment to cover the fundamentals of MDX. A basic understanding of MDX is not only beneficial when working with OLAP data sources, but necessary to use the techniques outlined later in this chapter.

MDX is a robust topic that is rich in scope and complexity. In that light, this chapter only touches on the fundamentals of MDX. If, after reading this chapter, you have a desire to learn more about MDX, consider picking up *MDX Solutions*, an excellent guide to MDX that is both easy to understand and comprehensive.

The Basics of MDX

Those of you who are familiar with SQL will have relatively little trouble picking up the basic concepts of MDX. As you look at the general syntax for an MDX statement, you will see the familiar `SELECT` and `FROM` clauses:

```
SELECT {member selection} ON COLUMNS,  
       {member selection} ON ROWS  
FROM   [cube name]
```

A member selection can be any combination of dimensions or members. These selections are given an axis designation. In MDX, a member selection can actually be placed in any one of up to 64 axes. To keep things simple, look at the most common axes: columns and rows. When a member selection is placed `On Columns`, that member selection will be column-oriented. Place a member selection `On Rows`, and that selection will be row-oriented. Finally, the cube name identifies the name of the cube with which you are working.

Chapter 23: Browsing OLAP Data Sources with Excel

Take a look at the following MDX query. Here you are requesting Internet sales and tax amounts as columns, and the product category as rows. As you can see in the FROM clause, this data is coming from the Analysis Services Tutorial cube.

```
SELECT {[Measures].[Internet Sales-Sales Amount],
       [Measures].[Internet Sales-Tax Amount]} ON COLUMNS,
       {[Product].[Product Categories].[Category].Members} ON ROWS
FROM   [Analysis Services Tutorial]
```

You will note that you are identifying the member selections by explicitly walking through the cube structure that gets you to that member. For instance, the query selects the Category members using [Product].[Product Categories].[Category].Members. This expression explicitly walks through the Product dimension, the Product Categories hierarchy, and the Category level, and finally ends with the members of the Category level. Refer back to Figure 23-3 to get a graphical view of how the cube structure works.

Notice that the resulting dataset, shown in Figure 23-7, is in a cross tab structure. This is the power of MDX and OLAP cubes. With MDX, you can create any number of datasets that are structured in any number of ways. Although this type of result could be achieved via a SQL statement, it would not be as straightforward and as easy as an MDX query.

	Sales Amount	Tax Amount
Accessories	\$700,759.96	\$56,060.80
Bikes	\$28,318,144.65	\$2,265,451.62
Clothing	\$339,772.61	\$27,181.81

Figure 23-7

You can also limit the results of your MDX query by slicing your selections using a WHERE clause. This works similarly to a WHERE clause in a SQL statement. The following MDX query limits the results to sales for the United States only. The resulting dataset is shown in Figure 23-8.

```
SELECT {[Date].[Calendar Quarter].Members} ON COLUMNS,
       {[Product].[Product Categories].[Category].Members} ON ROWS
FROM   [Analysis Services Tutorial]
WHERE  ([Customer].[Country-Region].[United States],
       [Measures].[Internet Sales-Sales Amount])
```

	2003 1	2003 2	2003 3	2003 4
Accessories			\$42,245	\$66,664
Bikes	\$277,589	\$393,458	\$765,330	\$1,270,211
Clothing			\$20,703	\$32,555

Figure 23-8

You may be wondering why the Internet Sales Amount measure is included in the WHERE clause of the previous MDX query. In every OLAP cube, one measure is designated as the default measure. If an MDX query is passed without explicitly asking for a particular measure, the default measure is returned. The default measure is typically designated by the administrator of your Analysis Services database.

To get the exact measure you are looking for, you must explicitly call for it in your MDX query. The most common way a measure is passed in an MDX query is through the WHERE clause. For instance:

```
SELECT {[Customer].[Country-Region].Members} ON COLUMNS
      {[Product].[Product Categories].[Category].Members} ON ROWS
FROM [Analysis Services Tutorial]
WHERE ([Measures].[Internet Sales-Sales Amount])
```

Deciphering Excel's MDX Queries

In the next section of this chapter, you discover how you can pass your own MDX queries to an OLAP server using ADO and a little VBA in order to retrieve a flat dataset. One of the benefits of being able to see the MDX query behind an OLAP-based pivot table is that you can see how the MDX for a particular view should be set up. This gives you a kind of built-in MDX tutor that you can leverage when you need it.

Create an MDX Log

You can set up an MDX log that documents every MDX query that is passed to the OLAP server. The procedure shown here does just that. As you can see, this procedure is entered into the PivotTableUpdate event of the worksheet. This ensures that each time there is a change in the pivot table, the MDX query is captured. You then simply trap the MDX query in a string variable and append it to a text file:

```
Private Sub Worksheet_PivotTableUpdate(ByVal Target As PivotTable)
Dim StrMDX As String

'Trap the MDX statement
  StrMDX = ActiveSheet.PivotTables("PivotTable1").MDX

'Append the MDX to a specified text file
  Open "C:\MDX_Log.txt" For Append As #1
  Print #1, StrMDX & Chr(13) & Chr(10)
  Close #1

End Sub
```

After looking at a few of the MDX queries that Excel outputs, you will soon realize that the way Excel uses MDX is different from the way you would. There are two reasons for this. First, Excel adds some syntax that it feels is necessary to make the MDX query run properly. This is analogous to the way that Excel records macros; it errs on the side of creating more syntax than is actually necessary in order to play it safe. Second, Excel imposes automatic rules when building its MDX queries.

Chapter 23: Browsing OLAP Data Sources with Excel

In the following example, the text in bold is the syntax that is actually needed to run this MDX query. The rest is syntax Excel uses as safeguards and rules to ensure that the query runs the way Excel expects it to run.

```
SELECT NON EMPTY Hierarchize({DrilldownLevel({[Product].[Product  
Categories].Members [All Products]}})) DIMENSION PROPERTIES  
PARENT_UNIQUE_NAME ON COLUMNS FROM [Analysis Services Tutorial] WHERE  
({[Measures].[Internet Sales-Sales Amount]}) CELL PROPERTIES VALUE,  
FORMAT_STRING, LANGUAGE, BACK_COLOR, FORE_COLOR, FONT_FLAGS
```

The following sections outline some of the most common syntactical expressions you will find in Excel-generated MDX queries that can be safely eliminated.

NON EMPTY

By default, Excel will inhibit the display of any members that are empty. It does this by using the NON EMPTY keyword. You don't need this keyword unless you are looking for only nonempty members.

You can force Excel to include empty members by adjusting the Display properties of the pivot table. To do so, right-click the pivot table and select Table Options ⇄ Display. Then place a check in Show Items with No Data on Rows and Show Items with No Data on Columns. Any subsequent MDX queries will not include the NON EMPTY keyword. You can also change these settings by using the `DisplayEmptyRow` and `DisplayEmptyColumn` properties of the `PivotTable` object:

```
ActiveSheet.PivotTables("PivotTable1").DisplayEmptyRow = True  
ActiveSheet.PivotTables("PivotTable1").DisplayEmptyColumn = True
```

Hierarchize

The `Hierarchize` function sorts all members in hierarchical order. Most OLAP cubes are sorted properly by default, so you do not need to include this in your MDX statements. Excel plays it safe by employing this function.

DrilldownLevel

By default, Excel will always pull the ALL level of any hierarchy and then drill down to the needed level by employing the `DrilldownLevel` function. This function drills downs one level below the specified level.

Various Server-Side Settings

Excel will automatically call for all of the server-side settings. These settings allow the OLAP administrators to push down formatting, language, and other properties to the consumers of their OLAP data. When using MDX queries via VBA, you will rarely need these settings. You can, for the most part, ignore any of the formatting-related keywords such as `CELL PROPERTIES VALUE`, `FORMAT_STRING`, `LANGUAGE`, `BACK_COLOR`, `FORE_COLOR`, `FONT_FLAGS`, and `DIMENSION PROPERTIES PARENT_UNIQUE_NAME`.

With some practice, you will be able to spot the necessary MDX that you can leverage in building your own queries. From the sample MDX created earlier in this section, you can whittle the MDX down to the following query that can be used in VBA to retrieve OLAP data:

```
SELECT {[Product].[Product Categories].Members} ON COLUMNS
FROM [Analysis Services Tutorial]
WHERE ({[Measures].[Internet Sales-Sales Amount]})
```

You would logically think that if an OLAP-based pivot table uses MDX queries to retrieve data, then you would be able to feed a pivot table with custom MDX queries. Unfortunately, that is not the case. There is currently no way to make a pivot table accept custom MDX queries as arguments for its data.

Browsing OLAP Data Sources without Pivot Tables

At the start of this chapter, you saw that pivot tables were the most effective way to browse and analyze cube data. So why would you want to browse an OLAP data source without a pivot table? The reason is that you can do some interesting things with your OLAP data sources by using procedures outside of the pivot table environment. In this section, you discover a few techniques that allow you to extend your OLAP experience past pivot tables.

Using ADO to Return Flattened Recordsets

You may find that you don't necessarily need to create pivot table processes for your reporting. You may only need to get a few tabular datasets from your OLAP data sources. You can automate the creation of flattened datasets by using a combination of MDX and ADO. Once you understand the basics of MDX queries, you can use them just as you would SQL statements.

The following example first sets and opens a connection to the OLAP server by using the MSOLAP provider. The Data Source is the name of the Analysis Services server, and the Initial Catalog is the name of the database you are querying.

Next, you define the MDX query and execute the query into a recordset. You then enumerate through the recordset to output the column headings. The column headings from an OLAP data source do not come out very clean, so take some time to clean up most of the special characters and other strange naming conventions. Finally, use the CopyFromRecordset method to display the data:

```
Sub Return_Flat_Recordset()
    Dim ObjConnection As New ADODB.Connection
    Dim ObjRecordset As New ADODB.Recordset
    Dim ObjWorkSheet As Object
    Dim StrCellValue As String
    Dim StrMDX As String
    Dim c As Integer
```

```
Dim i As Integer

'Set and open a connection to the OLAP Server
ObjConnection.Open "Provider=MSOLAP;Data Source=D4SBY981;Initial
Catalog=AdventureWorks;"

'Create the MDX query
StrMDX = "SELECT" & _
        "[[Customer].[Country-Region].Members} ON COLUMNS," & _
        "[[Product].[Product Categories].[SubCategory].Members} ON ROWS"& _
        "FROM [Analysis Services Tutorial] " & _
        "WHERE ([Measures].[Internet Sales-Sales Amount])"

'Execute the MDX query and open in a recordset
ObjRecordset.Open StrMDX, ObjConnection

'Create fresh worksheet
Set ObjWorkSheet = Worksheets.Add
ObjWorkSheet.Activate

'Enumerate through the fields in the recordset and add column headings to the
spreadsheet
c = 1
For i = 0 To ObjRecordset.Fields.Count - 1
    ActiveSheet.Cells(1, c).Value = ObjRecordset.Fields(i).Name

    'Take a moment to clean up column headings
    StrCellValue = ActiveSheet.Cells(1, c).Value
    StrCellValue = Replace(StrCellValue, "[", " ")
    StrCellValue = Replace(StrCellValue, "]", " ")
    StrCellValue = Replace(StrCellValue, ".", " ")
    StrCellValue = Replace(StrCellValue, "&", " ")
    StrCellValue = Replace(StrCellValue, "MEMBER_CAPTION", " ")
    ActiveSheet.Cells(1, c).Value = StrCellValue

    c = c + 1
Next i

'Start at first row of the recordset and output the dataset
ObjRecordset.MoveFirst
ObjWorkSheet.Cells(2, 1).CopyFromRecordset ObjRecordset

'Clean up
ObjRecordset.Close
ObjConnection.Close

End Sub
```

Using ADO MD to Get Cube Schema Information

Some of the most useful information you can have when starting to work with an OLAP data source is information about the data source itself. How many cubes are on the server? How many dimensions are in each cube? What are the levels available? Answers to these types of questions can help you get

acquainted with your OLAP cubes, allowing you to fully understand how the data is organized and structured. These answers can most easily be gained through the use of ADO MD (Microsoft ActiveX Data Objects Multidimensional).

ADO MD allows you to access both data and metadata from an OLAP or multidimensional data provider. Although ADO and ADO MD are related, they have separate object models. Figure 23-9 illustrates the ADO MD object model.

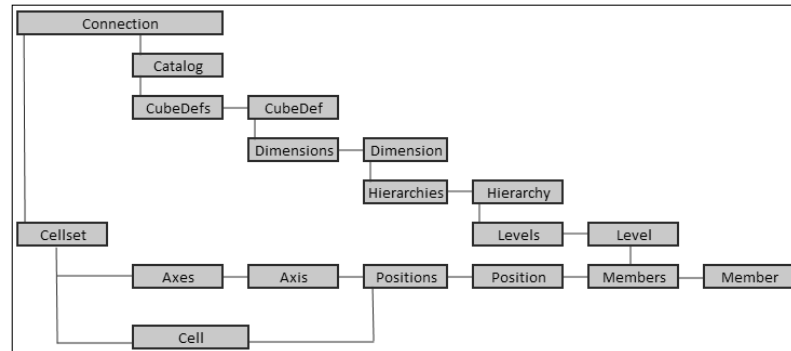


Figure 23-9

Notice that there are two main branches that stem from the `Connection` object: the `Catalog` and `Cellset` branches. The `Catalog` branch is used to query the metadata or structure of the OLAP data source, and the `Cellset` branch is used to query the data in the data source.

To get information about the cube schema, you will utilize the `Catalog` object. You will note that the object hierarchy in the `Catalog` branch looks very similar to the hierarchy typically found in an OLAP data source. Note how the `Catalog/CubeDef/Dimension/Hierarchy/Level/Member` structure within the object model can be related to the `Database/Cube/Dimension/Hierarchy/Level/Member` in an actual OLAP server. This helps when you start thinking about the schema information you need.

Creating an Inventory of Dimensions, Hierarchies, and Levels

The procedure shown in this section allows you to create an inventory of the major structures in a particular OLAP cube. This type of inventory allows you to quickly gain familiarity with the new OLAP cubes and allows you see the hierarchical structure of the cube in way that is not limited to the PivotTable Field List.

To use the ADO MD object model, you must add a reference to the Microsoft ActiveX Data Objects (Multidimensional) reference library.

Chapter 23: Browsing OLAP Data Sources with Excel

In this procedure, first instantiate an ADO MD `Catalog` Object, and then establish a connection to the data source using this object. Once you have a connection to a valid ADO MD Catalog, you have access to all of the other objects within that catalog. In this case you specifically need the `CubeDef`, `Dimension`, `Hierarchy`, and `Level` objects. Identify the cube you are querying, and then you enumerate through the cube in a structured fashion, logging the name of each `Dimension`, `Hierarchy`, and `Level` into a text file. This procedure is just one of many ways you can choose to utilize the ADO MD `Catalog` object to retrieve schema information for your OLAP cubes:

```
Sub Create_Cube_Inventory()
Dim ObjCatalog As New ADOMD.Catalog
Dim ObjCube As ADOMD.CubeDef
Dim ObjDimension As ADOMD.Dimension
Dim ObjHierarchy As ADOMD.Hierarchy
Dim ObjLevel As ADOMD.Level
Dim StrConnection As String

'Create connection string
StrConnection = "Provider=MSOLAP;Data Source=D4SBY981;" & _
"Initial Catalog=AdventureWorks;"

'Point catalog object to the data source using your connection string
ObjCatalog.ActiveConnection = StrConnection

'Identify the cube you are working with
Set ObjCube = ObjCatalog.CubeDefs("Analysis Services Tutorial")

'Create an empty text file
Open "C:\CubeInventory.txt" For Append As #1

'Enumerate through each dimension, hierarchy and level and log them into the text
file
For Each ObjDimension In ObjCube.Dimensions
Print #1, Chr(13) & Chr(10)
Print #1, "Dimension: " & ObjDimension.Name
Print #1, "*****"

    For Each ObjHierarchy In ObjDimension.Hierarchies
Print #1, "    Hierarchy Name: " & ObjHierarchy.Name

        For Each ObjLevel In ObjHierarchy.Levels
Print #1, "        Level: " & ObjLevel.Name
Next ObjLevel

    Next ObjHierarchy

Next ObjDimension

Next ObjDimension

'Clean up
Close #1
Set ObjCatalog = Nothing

End Sub
```

Creating Offline Cubes

Offline cubes are files that locally store portions of the source data found in an OLAP data source for browsing while disconnected from the network. These types of cubes are useful when you need to distribute reporting solutions to clients that do not have access to your network, or clients for whom network access is extremely slow. This section explores the various ways to create Offline cubes.

Creating an Offline Cube Manually

To create an offline cube by hand, start with an OLAP-based pivot table. Place your cursor anywhere inside the pivot table, and then go up to the Options tab in the Ribbon and select OLAP Tools ⇨ Offline OLAP. This will activate the Offline OLAP Settings dialog box, where you will select Create Offline Data File.

From here, you will step through a wizard that allows you to select the dimensions, levels, and measures you want to make available in the offline cube. Keep in mind that your selections through this wizard will determine the data that will be available to you while disconnected from the server. The last screen of the offline cube wizard allows you to select the location and name of your cube file.

Once the process is complete, your reward will be a .cub file that can be used as the data source for your pivot table reports. You can either double-click the cube file or open the cube file from Excel.

Any attempt to refresh an offline cube file will cause the cube file to try to connect to the original OLAP data source. The idea is that you can use the data within the cube file while you are disconnected from the network, and you can refresh the cube file while a data connection is available.

Using the CreateCubeFile Method

If you are in need of something a bit more automated, you can use the `CreateCubeFile` method. This method creates a cube file from a PivotTable report that is connected to an OLAP data source:

```
Sub CreateCubeFile()  
ActiveSheet.PivotTables("PivotTable1").CreateCubeFile File:="C:\CustomCubeFile.cub"  
End Sub
```

The benefit of using this method is that the data in the offline cube file will consist of the exact data that existed in the pivot table at the time you executed the procedure. For example, if your pivot table contains a page field that is filtered to show data for only the United States, then the offline cube that is created by the `CreateCubeFile` method will contain data only for the United States. This is because the `CreateCubeFile` method essentially runs the MDX query behind the pivot table and outputs the results to a local .cub file. So if you have the need to create several offline cubes, each containing a different set of data, you can simply automate the rearranging of the data fields with the pivot table (using the VBA techniques covered in Chapter 7), and then employ the `CreateCubeFile` method.

Note that you may get a security warning when trying to open a cube file. This is a normal security feature that Excel employs in order to protect you from malicious datasets. Simply click the Enable button on the Security notice to allow connection to the cube file.

Creating an Offline Cube Using ADO MD and VBA

With a combination of ADO MD and VBA, there is no need to create an OLAP-based pivot table in order to generate offline cube files. You can simply utilize the `CREATE GLOBAL CUBE MDX` statement.

The `CREATE GLOBAL CUBE` statement is a bit like an action query, telling the MDX to perform a particular task with the results that are retrieved. That task, in this case, is to output the results of the MDX query into a `.cub` file.

The basic structure of a `CREATE GLOBAL CUBE` statement is relatively simple:

```
CREATE GLOBAL CUBE [Name_Given_To_Cube]
STORAGE 'C:\Location.cub'
FROM [Source Cube]
(MEASURES, DIMENSIONS)
```

First, you specify a name for your offline cube. Next, you use the `STORAGE` clause to specify the name and location of the `.cub` file that will be output. Then use the `FROM` clause to identify the source cube. Finally, you list all of the Measures and Dimensions you would like to be included in the `.cub` file.

The example demonstrated here will create an offline cube called `MyCustomCube.cub`. It will contain Internet Sales Amount measures for the `Customer` and `Product` Dimensions, including all hierarchies and levels beneath the selected dimensions:

```
CREATE GLOBAL CUBE [MyCustomCube]
STORAGE 'C:\MyCustomCube.cub'
FROM [Analysis Services Tutorial]
(MEASURE [Analysis Services Tutorial].[Internet Sales-Sales Amount],
DIMENSION [Analysis Services Tutorial].[Customer],
DIMENSION [Analysis Services Tutorial].[Product])
```

To utilize this statement via VBA, you would execute the query through an ADO MD Cellset, as demonstrated in the following procedure:

```
Public Sub Create_Local_Cube()
Dim ObjCatalog As New ADOMD.Catalog
Dim ObjCellset As New ADOMD.Cellset
Dim StrConnection As String
Dim StrMDX As String

'Create connection string
StrConnection = "Provider=MSOLAP;Data Source=D4SBY981;" & _
```

```
"Initial Catalog=AdventureWorks;"

'Create the MDX Query
StrMDX = "CREATE GLOBAL CUBE [MyCustomCube]" & _
        "STORAGE 'C:\MyCustomCube.cub' " & _
        "FROM [Analysis Services Tutorial] " & _
        "(" & _
        "MEASURE [Analysis Services Tutorial].[Internet Sales-Sales Amount], " & _
        "DIMENSION [Analysis Services Tutorial].[Customer], " & _
        "DIMENSION [Analysis Services Tutorial].[Product]" & _
        ")"

'Feed the cellset object your MDX Query
ObjCellset.Source = StrMDX

'Point catalog and cellset objects to the data source using your connection string
ObjCatalog.ActiveConnection = StrConnection
ObjCellset.ActiveConnection = StrConnection

'Open Cellset and fill the local cube
ObjCellset.Open

'Clean up
Set ObjCatalog = Nothing
Set ObjCellset = Nothing

End Sub
```

Summary

Excel pivot tables are extremely powerful OLAP browsers. They allow you to analyze impossible amounts of data in a familiar environment, the most powerful analytical tool in Excel. Reporting solutions built on OLAP data sources are easy to build with pivot tables, and are typically more efficient than those that are built using transactional databases. OLAP-based pivot tables run on MDX queries. It's worth your time to understand MDX and find creative ways to use the MDX behind pivot tables to improve your processes. ADO MD offers some automated ways to do things that are difficult to do using the pivot table interface in Excel. ADO MD allows you to investigate your OLAP cube's schema, as well as automate the creation of offline OLAP cubes.

Excel and the Internet

Until a few years ago, a typical Excel-based application was almost entirely contained within Excel itself; the only external interaction would be with the user, from whom you obtained data and to whom you presented results. If you needed to store data, you'd use separate workbooks and try to mimic a relational database as best you could.

As data access technologies developed, from ODBC drivers through DAO to the current versions of ADO (documented in Chapter 20), it became more commonplace to store data in external databases and retrieve data from (and update data in) other systems across the network. It is now quite common to see Excel used as a front-end querying and analysis tool for large corporate databases, using QueryTables and PivotTables to retrieve the data. The data available to Excel applications was, however, limited to what was available across the company network, and to those databases you could get permission to access.

Starting with the release of Office 97, Microsoft has slowly extended Excel's reach to include the Internet and associated technologies, either by adding native functionality directly to Excel (such as Web Queries), or by ensuring that Excel developers can easily use standard external objects (such as the Internet Transfer Control, the Web Browser control, and the MSXML parser) and including those objects within the Office installation.

In Excel 2007, you have sufficient functionality to think outside of the pure Excel/ADO environment in terms of obtaining data, publishing results, monitoring applications, and sharing data with many disparate systems across the Internet.

This chapter introduces the functionality available in Excel 2007 and demonstrates how to use some of it to exploit the Internet within your applications. A complete discussion of all of Excel's Internet-related functionality is beyond the scope of this book.

Note that throughout this chapter, the term *Internet* is used in its broadest sense, covering both internal and external networks. The chapter assumes a basic understanding of the Internet and how it works. The examples use a web server running on a local PC. However, these techniques are equally applicable to applications running on a remote server.

What Can the Internet Do for You?

In a nutshell, the Internet is all about sharing information.

It's about publishing information to unknown consumers, using standard formats and protocols to enable them to access the information you provide in a consistent, reliable, and secure manner.

It's about making that information available globally, both inside and outside the organization's networks, while maintaining control over security and access to potentially sensitive information.

It's about retrieving the information that other individuals or organizations provide, from multiple disparate sources, to use as inputs to your application.

It's about sharing information between producers and consumers, suppliers and customers, using standard formats for that exchange.

It's about looking outside of the classic Excel application, and adding value to that application by sharing its results with a wider audience than simply the user sitting at the PC.

It's about using Excel as a key component of a larger business process that may span multiple organizations.

Using the Internet for Storing Workbooks

The simplest way of sharing information is to store workbooks on a web server. Though Excel 97 introduced the ability to download workbooks from web sites, Excel 2000 and 2007 extended that to allow you to save workbooks as well. They do this by using the FrontPage Server Extensions, which must be running on the server. To open and save a workbook from or to a web site, use the URL instead of the filename:

```
Sub OpenFromWebSiteAndSaveBack()  
  
    Dim oBk As Workbook  
  
    'Open a workbook from a web site  
    Set oBk = Workbooks.Open("http://www.MySite.com/book1.xlsx")  
  
    'Save the workbook to the web site with a new name  
    oBk.SaveAs "http://www.MySite.com/Book2.xlsx"  
  
End Sub
```

If the server requires you to log on, you have the option of letting Excel prompt for the ID and password each time (as in the preceding example), or include the ID and password as part of the URL:

```
Sub OpenFromSecureWebSiteAndSaveBack()  
  
    Dim oBk As Workbook  
  
    'Open a workbook from a web site
```



```
Set oBk = Workbooks.Open("http://UserID:Pwd@www.MySite.com/book1.xlsx")
```

```
'Save the workbook to the web site with a new name
oBk.SaveAs "http://UserID:Pwd@www.MySite.com/Book2.xlsx"
```

```
End Sub
```

The URLs can, of course, also be used in Excel's Office Menu ⇨ Open and Save As dialogs.

Using the Internet as a Data Source

The classic Excel application has two sources of data — databases on the network, and the user. If an item of data was not available in a database, the user was required to type it in and maintain it. To enable this, the application had to include a number of sheets and dialogs to store the information and provide a mechanism for the data entry.

A typical example of this would be maintaining exchange rate information in a financial model; it is usually the user's responsibility to obtain the latest rates and type them into the model. You can add value to the application by automating the retrieval of up-to-date exchange rate information from one of many web sites.

The following sections demonstrate different techniques for retrieving information from the web, using the USD exchange rates available from www.x-rates.com/d/USD/table.html (see Figure 24-1) as an example.

The screenshot shows a web browser window displaying an exchange rates table. The table is titled 'American Dollar' and lists various currencies with their respective exchange rates against 1 USD. The table is organized into columns for '1 USD' and 'in USD'. The currencies listed include Australian Dollar, Brazilian Real, British Pound, Canadian Dollar, Chinese Yuan, Danish Krone, Euro, Hong Kong Dollar, Indian Rupee, Japanese Yen, Malaysian Ringgit, Mexican Peso, New Zealand Dollar, Norwegian Kroner, Singapore Dollar, South African Rand, South Korean Won, Sri Lanka Rupee, Swedish Krona, Swiss Franc, Taiwan Dollar, Thai Baht, and Venezuelan Bolivar. The exchange rates are provided for each currency.

American Dollar	1 USD	in USD
Australian Dollar	1.29752	0.770701
Brazilian Real	2.1428	0.466679
British Pound	0.524742	1.9057
Canadian Dollar	1.1275	0.886918
Chinese Yuan	7.8651	0.127144
Danish Krone	5.8362	0.171344
Euro	0.782779	1.2775
Hong Kong Dollar	7.7838	0.128472
Indian Rupee	44.59	0.0224266
Japanese Yen	117.74	0.00849329
Malaysian Ringgit	3.64	0.274725
Mexican Peso	10.866	0.0920302
New Zealand Dollar	1.49678	0.668101
Norwegian Kroner	6.4518	0.154996
Singapore Dollar	1.5603	0.640902
South African Rand	7.2925	0.137127
South Korean Won	934.7	0.00106986
Sri Lanka Rupee	107.15	0.00933271
Swedish Krona	7.1451	0.139956
Swiss Franc	1.2494	0.800384
Taiwan Dollar	32.85	0.0304414
Thai Baht	36.68	0.0272628
Venezuelan Bolivar	2144.6	0.000466287

Figure 24-1

Opening Web Pages as Workbooks

The simplest solution is to open the entire web page as if it were a workbook, then scan the sheet for the required information, such the USD/GBP exchange rate:

```
Sub OpenUSDRatesPage()  
  
    Dim oBk As Workbook  
    Dim oRng As Range  
  
    'Open the rates pages as a workbook  
    Set oBk = Workbooks.Open("http://www.x-rates.com/d/USD/table.html")  
  
    'Find the British Pounds entry  
    Set oRng = oBk.Worksheets(1).Cells.Find("British Pound")  
  
    'Read off the exchange rate  
    MsgBox "The USD/GBP exchange rate is " & oRng.Offset(0, 1).Value  
  
End Sub
```

The problem with using this approach is that you have to load the entire web page (including graphics, banners, and so on), which may have much more information than you want. The irrelevant data can greatly slow down the speed of data retrieval.

Using Web Queries

Web Queries were introduced in Excel 97 and have been enhanced in each subsequent version of Excel. They enable you to retrieve a single table of information from a web page, with options to automatically refresh the data each time the workbook is opened, or at frequent intervals.

One of the problems with Web Queries is that Excel uses the thousands and decimal separators specified in the Windows Regional Settings when attempting to recognize numbers in the page. If the exchange rate web page were retrieved in many European countries, the period would be treated as a thousands separator, not a decimal separator, resulting in exchange rates that are many times too large. In Excel 2002, Microsoft added three properties to the `Application` object to temporarily override the settings used when recognizing numbers:

- `Application.DecimalSeparator` — The character to use for the decimal separator
- `Application.ThousandsSeparator` — The same for the thousands separator
- `Application.UseSystemSeparators` — Whether to use the Windows separators or Excel's

Using these properties, you can set Excel's separators to match those on the web page, perform the query, and then set them back again. Web Queries could not be used reliably in versions prior to Excel 2002 in countries that used non-U.S. decimal and thousands separators. If you want to use the Web Query's automatic refreshing options, you have to set these separators in the `BeforeRefresh` event, and set them back in the `AfterRefresh` event. This requires advanced VBA techniques, using class modules to trap events, as discussed in Chapter 16.

In this case, you can retrieve just the table of exchange rates, using the following code to create and execute a new Web Query. In practice, it's easiest to use the macro recorder to ensure the selections are correct:

```
'Retrieve USD exchange rates using a Web Query
Sub GetRatesWithWebQuery()

    Dim oBk As Workbook
    Dim oQT As QueryTable

    'Store the current settings of Excel's number formatting
    Dim sDecimal As String
    Dim sThousand As String
    Dim bUseSystem As Boolean

    'Create a new workbook
    Set oBk = Workbooks.Add

    'Create a query table to download USD rates
    With oBk.Worksheets(1)
        Set oQT = .QueryTables.Add( _
            Connection:="URL;http://www.x-rates.com/d/USD/table.html", _
            Destination:=.Range("A1"))
    End With

    'Set the QueryTable's properties
    With oQT
        .Name = "USD"

        'State that we're selecting a specific table
        .WebSelectionType = xlSpecifiedTables

        'Import the 14th table on the page
        .WebTables = "14"

        'Ignore the web page's formatting
        .WebFormatting = xlWebFormattingNone

        'Do not try to recognize dates
        .WebDisableDateRecognition = True

        'Don't automatically refresh the query each time the file is opened
        .RefreshOnFileOpen = False

        'Waiting for the query to complete before continuing
        .BackgroundQuery = True

        'Save the query data with the workbook
        .SaveData = True

        'Adjust column widths to autofit new data
```

```
.AdjustColumnWidth = True
End With

With Application
    'Remember Excel's current number format settings
    sDecimal = .DecimalSeparator
    sThousand = .ThousandsSeparator
    bUseSystem = .UseSystemSeparators

    'Set Excel's separators to match those of the web site
    .DecimalSeparator = "."
    .ThousandsSeparator = ","
    .UseSystemSeparators = True

    'Ignore any errors raised by the query failing
    On Error Resume Next

    'Perform the query, waiting for it to complete
    oQT.Refresh BackgroundQuery:=False

    'Reset Excel's number format settings
    .DecimalSeparator = sDecimal
    .ThousandsSeparator = sThousand
    .UseSystemSeparators = bUseSystem
End With

End Sub
```

The `.WebTables = "14"` line in this example tells Excel that you want the 14th table on the page. Literally, this is the 14th occurrence of a `<TABLE>` tag in the source HTML for the page.

Parsing Web Pages for Specific Information

Web Queries are an excellent way of retrieving tables of information from web pages, but they are a little cumbersome if you are only interested in one or two items of information and are extremely susceptible to minor changes in the web page layout (such as adding an extra `<table>` tag at the top of the page). Another way is to read the page using a hidden instance of Internet Explorer, search within the page for the required information (using either the HTML or plain text representations), and then return the result. The following code requires a reference to the Microsoft Internet Controls object library:

```
Sub GetUSDtoGBPRateUsingIE()

    Dim oIE As SHDocVw.InternetExplorer
    Dim sPage As String
    Dim iGBP As Long, iDec As Long
    Dim iStart As Long, iEnd As Long
    Dim dRate As Double

    'Create a new (hidden) instance of IE
    Set oIE = New SHDocVw.InternetExplorer

    'Open the web page
```

```

oIE.Navigate "http://www.x-rates.com/d/USD/table.html"

'Wait for the page to complete loading
Do Until oIE.readyState = READYSTATE_COMPLETE
    DoEvents
Loop

'Retrieve the text of the web page into a variable
sPage = oIE.Document.body.InnerText

'*****
'   Brazilian Real    2.1397  0.467355
'   British Pound    0.524934  1.905
'   Canadian Dollar   1.1056  0.904486
'*****

'To find the exchange rate, we have to find the entry for British
'Pounds, then work forwards to find the exchange rate

'Find the entry for British Pounds in the web page text.
iGBP = InStr(1, sPage, "British Pound")

'Find the next decimal, which will be in the middle of the
'exchange rate number
iDec = InStr(iGBP, sPage, ".")

'Find the start and end of the number
iStart = InStrRev(sPage, " ", iDec) + 1
iEnd = InStr(iDec, sPage, " ")

'Evaluate the number, knowing that it's in US format
dRate = Val(Mid$(sPage, iStart, iEnd - iStart))

'Display the rate
MsgBox "The USD/GBP exchange rate is " & dRate

End Sub

```

The most appropriate method to use will depend on the precise circumstances, and how much data is required. For single items, it is probably easier to use the last approach. For more than a few items, it will be easier to use a Web Query to read the page or table into a workbook, and then find the required items on the sheet.

Using the Internet to Publish Results

A web server can be used as a repository of information, storing your application's results and presenting them to a wider audience than can be achieved with printed reports. By presenting results as web pages, the reader of those pages can easily use the results as sources of data for their own analysis, and easily pass those results to other interested parties.

Setting Up a Web Server

For all the examples from now on, you will require write access to a web server. Because later examples use Active Server Pages (ASP), here you will use Microsoft's IIS 5.0. Open IIS, and right-click the Default Web Site node. Select Properties and click the Home Directory tab. You will be presented with various configuration options for the default web site. Make sure that the Read and Write checkboxes are selected (see Figure 24-2) and click OK.



Figure 24-2

Notice the Local Path: box. This is where the root of your web server is located. By default, it is C:\inetpub\wwwroot\. Any web pages placed in this directory are published at the URL `http://localhost/PageName.html`.

Saving Worksheets as Web Pages

The easiest way to present results as a web page is to create a template workbook that has all the formatting and links that you'd like to show. When your application produces its results, it is then a simple task to copy the relevant numbers to the template, then save the template direct to the web server (assuming the server is configured to allow this—if not, you may have to save the HTML file to a network location):

```
Sub PublishResultsToWeb()  
  
    Dim oBk As Workbook  
    Dim oSht As Worksheet  
  
    'Create a new copy of the Web Template workbook
```

```

Set oBk = Workbooks.Add("c:\mydir\WebTemplate.xlsx")

'Get the first sheet in the workbook
Set oSht = oBk.Worksheets(1)

'Populate the results
oSht.Range("Profits").Value = Workbooks("Results.xlsx") _
    .Worksheets("Financials").Range("Profits").Value

'Save as a web page, direct to the server
oSht.SaveAs "http://localhost/ResultsJuly2001.htm", xlHtml

'Close the workbook
oBk.Close False

End Sub

```

Prior to Excel 2007, the resulting HTML was horrible to modify, because it contained a large amount of extraneous information that could be used to “round-trip” a worksheet through the HTML format. Round-tripping through HTML never worked well, always lost data or formatting, and was almost never used. With the introduction of the XML file formats in Excel 2007, the requirement to round-trip through HTML no longer exists, and the Save to HTML feature has been redesigned as a simple publishing mechanism — resulting in much cleaner HTML.

Creating Interactive Web Pages

The previous example saved a static rendition of the worksheet in HTML format to the web server. In Excel 2000, Microsoft introduced the Office Web Components to create interactive web pages. When saving a worksheet in interactive form, the worksheet was converted to a set of objects collectively known as the Office Web Components. In Excel 2007, the Office Web Components have been dropped and replaced by a dedicated server-side Excel component. Known as Excel Services, the component runs on top of the SharePoint architecture and is able to open Excel workbooks, update data, perform calculations, and render the results as plain HTML for display in a SharePoint portal. Unfortunately, Excel Services is beyond the scope of this book.

Using the Internet as a Communication Channel

Retrieving data from web pages and publishing results as web pages is in many ways a passive use of the Internet; the web server is being used primarily as a storage medium. Web servers are also able to host applications, with which you can interact in a more dynamic manner. The server application acts as a single point of contact for all the client workbooks, to perform the following functions:

- ❑ A centralized data store
- ❑ Collation of data from multiple clients
- ❑ Presentation of that data back to other clients

- ❑ Workflow management
- ❑ Calculation engines

As an example, consider a timesheet reporting system, where each member of staff has an Excel workbook to enter their time on a daily basis. At the end of each month, they connect to the Internet and send their timesheets to an application running on a web server. That application stores the submitted data in a central database. Some time later, a manager connects to the server and is sent the submitted hours for her staff. She checks the numbers and authorizes payment, sending her authorization code back to the server. The payroll department retrieves the authorized timesheet data from the same web server directly into its accounting system and processes the payments.

In this business process, Excel is used for the front-end client, providing a rich and powerful user interface, yet it only fulfils a specific part of the overall process. The server application maintains the data (the completed timesheets) and presents it in whichever format is appropriate for the specific part of the process.

By using the Internet and standard data formats for this two-way communication, you can easily integrate Excel clients with completely separate systems, as in the payroll system in the example, and allow the business process to operate outside of the corporate network.

Communicating with a Web Server

Within a corporate network, nearly all data transfer takes place using proprietary binary formats, ranging from transferring files to performing remote database queries. Due primarily to security considerations, communication across the Internet has evolved to use textual formats, such as HTML and more recently XML. XML is covered in detail in Chapter 12.

To be able to communicate with an application running on a web server, you need to be able to pass information to, and receive information from, that application.

In Excel 2007, the `Workbook` object's `FollowHyperlink` method can be used to communicate with a web server. There are a few problems with using this, including:

- ❑ If an error occurs during the connection, Excel will freeze.
- ❑ Any data returned from the hyperlink is automatically displayed as a new workbook.
- ❑ You have very little control over the communication.

A much more flexible alternative is provided by the Microsoft Internet Transfer Control, `msinet.ocx`. This ActiveX control, often referred to as the ITC, is an easy-to-use wrapper for the `wininet.dll` file, which provides low-level Internet-related services for the Windows platform.

Sending Data from the Client to the Server Application

Two mechanisms can be used to send information to a web server. You can either include the information as part of the URL string or send it as a separate section of the HTTP request.

URL Encoding

Parameters can be included within the URL string by appending them to the end of the URL, with a question mark (?) between the URL and the first parameter and an ampersand (&) between each parameter:

```
http://www.MySite.com/MyPage.asp?param1=value1&param2=value2&param2=value3
```

This has the advantage that the parameters form part of the URL and hence can be stored in the user's Favorites list or typed directly into the browser. It has the disadvantage that there is a limit to the total length of a URL (2,083 characters in Internet Explorer), restricting the amount of information that can be passed in this way.

POSTing Data

Whenever a request for a web page is sent to a web server, the request contains a large amount of information in various header records. This includes things like the client application type and version, the communication protocol and version, and user IDs and passwords. It also contains a `POST` field that can be used to send information to the server application.

Because there is virtually no limit to the amount of data that can be put in a `POST` field, it is the preferred way of transferring information to the server, and is the method used in most applications.

Most web page forms use the `POST` mechanism to send the completed form to the server for processing. Excel can mimic the web-based form by sending the same data to the same server in the same `POST` format, with the result being sent back as the new web page. The names of the parameters can be found by examining the source of the real web page form. The form controls will be surrounded by a pair of `<form>` and `</form>` tags, and each control within the form has a name attribute; those names are used as the parameter names when the form's data is sent to the server:

```
Sub SendDataUsingPOST()

    Dim oInet As Inet
    Dim lContent As Long
    Dim sData As String
    Dim sHeader As String
    Dim sResult As String

    'Create a new instance of the Internet Transfer Control
    Set oInet = New Inet

    'Build the POST string with the data to submit
    sData = "Param1=Value1" & "&" & _
           "Param2=Value2" & "&" & _
           "Param3=Value3"

    'Spaces must be replaced with + signs
    sData = Replace(sData, " ", "+")

    'Tell the POST that we're sending an encoded parameter list
    sHeader = "Content-Type: application/x-www-form-urlencoded"

    'Send the error information to the server
```

```
oInet.AccessType = icDirect
oInet.Execute "http://www.mysite.com/mypage.asp", "POST", _
    sData, sHeader

'Wait for the server to complete its work
Do While oInet.StillExecuting
    DoEvents
Loop

'Retrieve the returned text
lContent = oInet.GetHeader("content-length")
sResult = oInet.GetChunk(lContent + 100)

End Sub
```

In this example, you're simply sending a simple list of name/value pairs to the server. If you need to send more complex data, you can send it as XML by providing the XML in the `sData` variable, with a header set to `"Content-Type: text/xml"`.

Summary

In Excel 2007, Microsoft has enabled the Excel developer to use the Internet as an integral part of an application solution in the following ways:

- ❑ Workbooks can be opened from and saved to web servers running the FrontPage Server Extensions.
- ❑ Excel can open HTML pages as though they were workbooks.
- ❑ Web Queries can be used to extract tables of data from web pages.
- ❑ The Internet Explorer object library can be automated to retrieve individual items of data from a web page, without the overhead of using a workbook.
- ❑ Excel workbooks can be saved as content-rich web pages, using either HTML or XML.
- ❑ The ability to publish interactive web pages using the Office Web Components has been removed from Excel 2007 and replaced with the Excel Services server-side component.
- ❑ The Microsoft Internet Transfer Control can be used to exchange data between Excel applications and web servers, using either a simple parameter list or more structured XML.

Together, these tools enable you to develop new types of business solutions, where Excel is one key part of a larger business process that may span multiple organizations and geographical locations.

International Issues

If you think that your application may be used internationally, it has to work with any choice of Windows Regional Setting, on any language version of Windows, and with any language choice for the Excel user interface.

If you are very lucky, all your potential users will have exactly the same settings as your development machine and you won't need to worry about international issues. However, a more likely scenario is that you will not even know who all your users are going to be, let alone where in the world they will live or the settings they will use.

Any bugs in your application that arise from the disregarding or ignoring of international issues will not occur on your development machine unless you explicitly test for them. However, they will be found immediately by your clients.

The combination of Regional Settings and Excel language is called the user's *locale*, and the aim of this chapter is to show you how to write locale-independent VBA applications. To do this, we include an explanation of the features in Excel that deal with locale-related issues, and highlight areas within Excel where locale support is absent or limited. Workarounds are provided for most of these limitations, but some are so problematic that the only solution is to not use the feature at all.

The rules provided in this chapter should be included in your coding standards and used by you and your colleagues. It is easy to write locale-independent code from scratch; it is much more difficult to make existing code compatible with the many different locales in the world today.

Changing Windows Regional Settings and the Office 2007 UI Language

Throughout this chapter, the potential errors will be demonstrated by using the three locales outlined in the following table.

Setting	U.S.	UK	Norway
Decimal Separator	.	.	,
Thousand Separator	,	,	.
Date Order	mm/dd/yyyy	dd/mm/yyyy	dd.mm.yyyy
Date Separator	/	/	.
Example Number: 1234.56	1,234.56	1,234.56	1.234,56
Example Date: February 10, 2007	02/10/2007	10/02/2007	10.02.2007
Windows and Excel Language	English	English	Norwegian
Text for Boolean True	True	True	Sann

The regional settings are changed using the Regional Settings applet (Regional Options in Windows 2000) in Windows Control Panel, and the Office 2007 language is changed using the Microsoft Office 2007 Language Settings program. Unfortunately, the only way to change the Windows language is to install a new version from scratch.

When testing your application, it is a very good idea to use some fictional regional settings, such as having a hash mark (#) for the thousands separator, an exclamation point (!) for the decimal separator, and a year/month/day date order. It is then very easy to determine if your application is using your settings or some internal default. For completeness, you should also have a machine in your office with a different language version of Windows from the one you normally use.

Responding to Regional Settings and the Windows Language

This section explains how to write applications that work with different regional settings and Windows language versions, which should be considered the absolute minimum requirement.

Identifying the User's Regional Settings and Windows Language

Everything you need to know about your user's Windows Regional Settings and Windows language version is found in the `Application.International` property. The online help lists all of the items that can be accessed, though you are unlikely to use more than a few of them. The most notable are:

- ❑ `XlCountryCode`: The language version of Excel (or of the currently active Office language)
- ❑ `XlCountrySetting`: The Windows regional settings location
- ❑ `XlDateOrder`: The choice of month-day-year, day-month-year, or year-month-day order to display dates

Note that there is no constant that enables you to identify which language version of Windows is installed (but you can get that information from the Windows API if required).

Windows Regional Settings is abbreviated to WRS in the rest of this chapter, and is also described as local settings.

VBA Conversion Functions from an International Perspective

The online help files explain the use of VBA's conversion functions in terms of converting between different data types. This section explains their behavior when converting to and from strings in different locales.

Implicit Conversion

This is the most common form of type conversion used in VBA code and forces the VBA interpreter to convert the data using whichever format it thinks is most appropriate. A typical example of this code is:

```
Dim dtMyDate As Date
dtMyDate = DateValue("Jan 1, 2007")
MsgBox "This first day of this year is " & dtMyDate
```

When converting a number to a string in Office 2007, VBA uses the WRS to supply either a date string in the user's `ShortDate` format, the number formatted according to the WRS, or the text for `True` or `False` in the WRS language. This is fine, if you want the output as a locally formatted string. If, however, your code assumes you've got a U.S.-formatted string, it will fail. Of course, if you develop using U.S. formats, you won't notice the difference (though your client will).

There is a much bigger problem with using implicit conversion if you are writing code for multiple versions of Excel. In previous versions, the number formats used in the conversion were those appropriate for the Excel language being used at run time (buried within the Excel object library), which might be different from both U.S. and local formats, and were not affected by changing the WRS.

Be very careful with the data types returned from, and used by, Excel and VBA functions. For example, `Application.GetOpenFilename` returns a `Variant` containing the Boolean value `False` if the user cancels, or a string containing the text of the selected file. If you store this result in a `String` variable, the Boolean `False` will be converted to a string in the user's WRS language, and it may not equal the string `"False"` that you may be comparing it to.

To avoid these problems, use the Object Browser to check the function's return type and parameter types, and then make sure to match them, or explicitly convert them to your variable's data type. Applying this recommendation gives you (at least) three solutions to using `Application.GetOpenFilename`.

Typical code running in Norway:

```
Dim stFile As String
stFile = Application.GetOpenFilename()
If stFile = "False" Then
    ...
```

Chapter 25: International Issues

If the user cancels, `GetOpenFilename` returns a variable containing the Boolean value `False`. Excel converts it to a string to put in your variable, using the Windows language. In Norway, the string will contain "Usann". If this is compared to the string "False", it doesn't match, so the program thinks it is a valid filename and subsequently crashes.

Solution 1:

```
Dim vaFile As Variant
vaFile = Application.GetOpenFileName()
If vaFile = False Then      'Compare using the same data types
    ...
```

Solution 2:

```
Dim vaFile As Variant
vaFile = Application.GetOpenFileName()
If CStr(vaFile) = "False" Then      'Explicit conversion with CStr() always
                                     'gives a US Boolean string
    ...
```

Solution 3:

```
Dim vaFile As Variant
vaFile = Application.GetOpenFileName()
If TypeName(vaFile) = "Boolean" Then      'Got a Boolean, so must have
                                           'cancelled
    ...
```

Note that in all three cases, the key point is that you are matching the data type returned by `GetOpenFilename` (a `Variant`) with your variable. If you use the `MultiSelect:=True` parameter within the `GetOpenFileName` function, the last of the preceding solutions should be used. This is because the `vaFile` variable will contain an array of filenames, or the Boolean `False`. Attempting to compare an array with `False`, or trying to convert it to a string, will result in a run-time error.

Date Literals

When coding in VBA, you can write dates using a format of `#01/01/2007#`, which is obviously January 1, 2007. But what is `#02/01/2007#`? Is it January 2 or February 1? It's actually February 1, 2007. This is because when coding in Excel, you do so in American English, regardless of any other settings you may have, and hence you must use U.S.-formatted date literals (mm/dd/yyyy format). If other formats are typed in (such as `#yyyy-mm-dd#`), Excel will convert them to the `#mm/dd/yyyy#` order.

What happens if you happen to be Norwegian or British and try typing in your local date format (which you *will* do at some time, usually near a deadline)? If you type in a Norwegian-formatted date literal, `#02.01.2007#`, you get a syntax error, which at least alerts you to the mistake you made. However, if you type in dates in a UK format (dd/mm/yyyy format) things get a little more interesting. VBA recognizes the date and so doesn't give an error, but "sees" that you have the day and month the wrong way around; it swaps them for you. So, typing in dates from January 10, 2007 to January 15, 2007 results in:

You Typed	VBA Shows	Meaning
10/1/2007	10/1/2007	October 1, 2007
11/1/2007	11/1/2007	November 1, 2007
12/1/2007	12/1/2007	December 1, 2007
13/1/2007	1/13/2007	January 13, 2007
14/1/2007	1/14/2007	January 14, 2007
15/1/2007	1/15/2007	January 15, 2007

If these literals are sprinkled through your code, you will not notice the errors.

It is much safer to avoid using date literals and use the VBA functions `DateSerial (Year, Month, Day)` or `DateValue (DateString)`, where `DateString` is a non-ambiguous string such as "January 1, 2007". Both of these functions return the corresponding `Date` number.

The IsNumeric and IsDate Functions

These two functions test if a string can be evaluated as a number or date according to the WRS and Windows language version. You should always use these functions before trying to convert a string to another data type. There are no `IsBoolean` functions to check if a string is a U.S.-formatted number or date. Note that `IsNumeric` does not recognize a percent sign (%) character on the end of a number, and `IsDate` does not recognize days of the week. `IsDate` is also extremely generous in its recognition; if there's any possible way that a string could represent a date, it will return `True`.

The CStr Function

This is the function most used by VBA in implicit data type conversions. It converts a `Variant` to a `String`, formatted according to the WRS. When converting a `Date` type, the `ShortDate` format is used, as defined in the WRS. Note that when converting `Booleans`, the resulting text is the English "True" or "False" and is not dependent on any Windows settings. Compare this with the implicit conversion of `Booleans`, whereby `MsgBox "I am " & True` results in the `True` being displayed in the WRS language ("I am Sann" in Norwegian Regional Settings).

The CDbI, CSng, CLng, CInt, CByte, CCur, and CDec Functions

All of these can convert a string representation of a number into a numeric data type (as well as converting different numeric data types into each other). The string must be formatted according to WRS. These functions do not recognize date strings or percent sign (%) characters.

The CDate and DateValue Functions

These methods can convert a string to a `Date` data type (`CDate` can also convert other data types to the `Date` type). The string must be formatted according to WRS and use the Windows language for month names. It does not recognize the names for the days of the week, giving a `Type Mismatch` error. If the year is not specified in the string, it uses the current year.

The CBool

CBool converts a string (or a number) to a Boolean value. Contrary to all the other Cxxx conversion functions, the string must be the English "True" or "False".

The Format Function

The Format function converts a number or date to a string, using a number format supplied in code. The number format must use U.S. symbols (m, d, s, and so on), but results in a string formatted according to WRS (with the correct decimal, thousands, and date separators) and the WRS language (for the weekday and month names). For example, the following code will result in Friday 05/01/2007 in the UK, but Fredag 05.01.2007 when used with Norwegian settings:

```
MsgBox Format(DateSerial(2007, 1, 5), "dddd dd/mm/yyyy")
```

If you omit the number format string, it behaves in exactly the same way as the CStr function (even though online help says it behaves like Str), including the strange handling of Boolean values, where Format(True) always results in the English "True". Note that it does not change the date order returned to agree with the WRS, so your code has to determine the date order in use before creating the number format string, using Application.International(xlDateOrder). If the preceding code is run in the U.S., it returns the same as in the UK — Friday 05/01/2007 — but your users will interpret that to mean May 1, not the correct January 5.

The FormatCurrency, FormatDateTime, FormatNumber, and FormatPercent Functions

These functions added in Excel 2000 provide the same functionality as the Format function, but use parameters to define the specific resulting format instead of a custom format string. They correspond to standard options in Excel's Number Format dialog, whereas the Format function corresponds to the Custom option. They have the same international behavior as the Format function from the previous section.

The Str Function

The Str function converts a number, date, or Boolean to a U.S.-formatted string, regardless of the WRS, Windows language, or Office language version. When converting a positive number, it adds a space on the left. When converting a decimal fraction, it does not add a leading zero. The following custom function is an extension of Str, which removes the leading space and adds the zero.

The sNumToUS Function

This function converts a number, date, or Boolean variable to a U.S.-formatted string. There is an additional parameter that can be used to return a string using Excel's DATE function, which would typically be used when constructing .Formula strings:

```
Function sNumToUS(vValue As Variant, Optional bUseDATEFunction) As String

    Dim sTmp As String

    'Don't accept strings or arrays as input
    If TypeName(vValue) = "String" Then Exit Function
```



```

If Right(TypeName(vValue), 2) = "()" Then Exit Function

If IsMissing(bUseDATEFunction) Then bUseDATEFunction = False

'Do we want it returned as Excel's DATE function
'(which we can't do with strings)?
If bUseDATEFunction Then

    'We do, so build the Excel DATE() function string
    sTmp = "DATE(" & Year(vValue) & "," & Month(vValue) & "," & _
        Day(vValue) & ")"
Else
    'Is it a date type?
    If TypeName(vValue) = "Date" Then
        sTmp = Format(vValue, "mm"/"dd"/"yyyy")
    Else
        'Convert number to string in US format and remove leading space
        sTmp = Trim(Str(vValue))

        'If we have fractions, we don't get a leading zero, so add one.
        If Left(sTmp, 1) = "." Then sTmp = "0" & sTmp
        If Left(sTmp, 2) = "-." Then sTmp = "-0" & Mid(sTmp, 2)
    End If
End If

'Return the US formatted string
sNumToUS = sTmp
End Function

```

`vValue` is a variant containing the number to convert, which can be:

- A number to be converted to a string with U.S. formats
- A date to be converted to a string in mm/dd/yyyy format
- A Boolean converted to the strings "True" or "False"

`bUseDATEFunction` is an optional Boolean for handling dates. When it is set to `False`, `sNumToUS` returns a date string in mm/dd/yyyy format. When it is set to `True`, `sNumToUS` returns a date as `DATE (yyyy, mm, dd)`.

The Val Function

This is the most commonly used function to convert from strings to numbers. It actually only converts a U.S.-formatted numerical string to a number. All the other string-to-number conversion functions try to convert the entire string to a number and raise an error if they can't. `Val`, however, works from left to right until it finds a character that it doesn't recognize as part of a number. Many characters typically found in numbers, such as dollar signs (\$) and commas, are enough to stop it from recognizing the number. `Val` does not recognize U.S.-formatted date strings.

`Val` also has the dubious distinction of being the only one of VBA's conversion functions to take a specific data type for its input. Whereas all the others use `Variants`, `Val` accepts only a string. This means that anything you pass to `Val` is converted to a string (implicitly, therefore according to the WRS and Windows language) before being evaluated according to U.S. formats.

The use of `Val` can have unwanted side effects (otherwise known as bugs), which are very difficult to detect in code that is running fine on your own machine, but which would fail on another machine with different WRS.

In the following table, `myDate` is a `Date` variable containing February 10, 2007 and `myDbl` is a `Double` containing 1.234.

Expression	U.S.	UK	Norway
<code>Val(myDate)</code>	2	10	10.02 (or 10.2)
<code>Val(myDbl)</code>	1.234	1.234	1
<code>Val(True)</code>	0 (=False)	0 (=False)	0 (=False)
<code>Val("SomeText")</code>	0	0	0
<code>Val("6 My St.")</code>	6	6	6

For clarity of comparison, the results are all displayed using U.S./UK number formats, though `Val(myDate)` would appear as 10,02 with Norwegian settings.

Application.Evaluate

Though not normally considered to be a conversion function, `Application.Evaluate` is the only way to convert a U.S.-formatted date string to a date number. The following two functions, `IsDateUS` and `DateValueUS`, are wrapper functions that use this method.

The `IsDateUS` Function

The built-in `IsDate` function validates a string against the Windows Regional Settings. This function provides you with a way to check if a string contains a U.S.-formatted date:

```
Function IsDateUS(sDate As String) As Boolean
    IsDateUS = Not IsError(Application.Evaluate("DATEVALUE("'" & _
                                                sDate & "'")"))
End Function
```

`sDate` is a string containing a U.S.-formatted date. `IsDateUS` returns `True` if the string contains a valid U.S. date, and `False` if not.

The `DateValueUS` Function

The VBA `DateValue` function converts a string formatted according to the Windows Regional Settings to a `Date` type. This function converts a string containing a U.S.-formatted date to a `Date` type. If the string cannot be recognized as a U.S.-formatted date, it returns an `Error` value that can be tested for, using the `IsError` function:

```
Function DateValueUS(sDate As String) As Variant

    DateValueUS = Application.Evaluate("DATEVALUE("'" & sDate & "'")")

End Function
```

`sDate` is a string containing a U.S.-formatted date. `DateValueUS` returns the date value of the given string, in a `Variant` (because it may contain an error value if `sDate` is not a valid date string).

Interacting with Excel

VBA and Excel are two different programs that have had very different upbringings. VBA speaks American. Excel also speaks American. However, Excel can also speak in its users' language if they have the appropriate Windows settings and Office language pack installed. On the other hand, VBA knows only a little about Windows settings, and even less about Office 2007 language packs. So, either you can do some awkward coding to teach VBA how to speak to Excel in the user's language, or you can just let them converse in American. I very much recommend the latter.

Unfortunately, most of the newer features in Excel are not multilingual. Some only speak American, and others only speak in the user's language. You can use the American-only features if you understand their limitations; the others are best avoided. All of them are documented later in the chapter.

Sending Data to Excel

By far the best way to get numbers, dates, Booleans, and strings into Excel cells is to do so in their native format. Hence, the following code works perfectly, regardless of locale:

```
Sub SendToExcel()
    Dim dtDate As Date, dNumber As Double, bBool As Boolean, _
        stString As String

    dtDate = DateSerial(2007, 2, 13)
    dNumber = 1234.567
    bBool = True
    stString = "Hello World"

    Range("A1").Value = dtDate
    Range("A2").Value = dNumber
    Range("A3").Value = bBool
    Range("A4").Value = stString
End Sub
```

There is a boundary layer between VBA and Excel. When VBA passes a variable through the boundary, Excel does its best to interpret it according to its own rules. If the VBA and Excel data types are mutually compatible, the variable passes straight through unhindered.

The problems start when Excel forces you to pass it numbers, dates, or Booleans within strings, or when you choose to do so yourself. The answer to the latter situation is easy — don't do it. Whenever you have a string representation of some other data type, if it is possible, always explicitly convert it to the data type you want Excel to store before passing it to Excel.

Chapter 25: International Issues

Excel requires string input in the following circumstances:

- Setting the formula for a cell, chart series, conditional format, data validation rule, or pivot table calculated field
- Specifying the `RefersTo` formula for a defined name
- Specifying `AutoFilter` criteria
- Passing a formula to `ExecuteExcel4Macro`
- Setting the number format of a cell, style, chart axis, or pivot table field
- Setting the number format in the VBA `Format` function

In these cases, you have to ensure that the string that VBA sends to Excel is in U.S.-formatted text—you must use English language formulas and U.S. regional settings. If the string is built within the code, you must be very careful to explicitly convert all your variables to U.S.-formatted strings.

Take this simple example:

```
Sub SetLimit(dLimit As Double)
    ActiveCell.Formula = "=IF(A1<" & dLimit & ",1,0)"
End Sub
```

You are setting a cell's formula based on a parameter supplied by another routine. Note that the formula is being constructed in the code and you are using U.S. language and regional settings (that is, using the English `IF` and a comma for the list separator). When used with different values for `dLimit` in different locales, you get the results shown in the following table.

dLimit	U.S.	UK	Norway
100	Works fine	Works fine	Works fine
100.23	Works fine	Works fine	Run-time error 1004

It fails when run in Norway with any non-integer value for `dLimit`. This is because you are implicitly converting the variable to a string, which you'll recall uses the Windows Regional Settings number formats. The resulting string that you're passing to Excel is:

```
=IF(A1<100,23,1,0)
```

This fails because the `IF` function does not have four parameters. If you change the function to read:

```
Sub SetLimit(dLimit As Double)
    ActiveCell.Formula = "=IF(A1<" & Str(dLimit) & ",1,0)"
End Sub
```

The function will work correctly, because `Str` forces a conversion to a U.S.-formatted string.

If you try the same routine with a `Date` instead of a `Double`, you come across another problem. The text that is passed to Excel (for example, for February 13, 2007) is:

```
=IF(A1<02/13/2007,1,0)
```

While this is a valid formula, Excel interprets the date as a set of divisions, so the formula is equivalent to:

```
=IF(A1<0.000077,1,0)
```

This is unlikely to ever be true. To avoid this, you have to convert the `Date` data type to a `Double`, and from that to a string:

```
Sub SetDateLimit(dtLimit As Date)
    ActiveCell.Formula = "=IF(A1<" & Str(CDbl(dtLimit)) & ",1,0)"
End Sub
```

The function is then the correct (but less readable):

```
=IF(A1<36935,1,0)
```

To maintain readability, you should convert dates to Excel's `DATE` function, to give:

```
=IF(A1<DATE(2007,2,13),1,0)
```

This is also achieved by the `sNumToUS` function presented earlier in this chapter, when the `bUseDateFunction` parameter is set to `True`:

```
Sub SetDateLimit(dLimit As Date)
    ActiveCell.Formula = "=IF(A1<" & sNumToUS(dLimit, True) & ",1,0)"
End Sub
```

If you call the revised `SetLimit` procedure with a value of 100.23 and look at the cell that the formula was put into, you'll see that Excel has converted the U.S. string into the local language and regional settings. In Norway, for example, the cell actually shows:

```
=HVIS(A1<100,23;1;0)
```

This translation also applies to number formats. Whenever you set a number format within VBA, you can give Excel a format string that uses U.S. characters (d for day, m for month, and y for year). When applied to the cell (or style or chart axis), or used in the `Format` function, Excel translates these characters to the local versions. For example, the following code results in a number format of dd/mm/åååå when you check it in the Number Format dialog in Norwegian Windows:

```
ActiveCell.NumberFormat = "dd/mm/yyyy"
```

This capability of Excel to translate U.S. strings into the local language and formats makes it easy for developers to create locale-independent applications. All you have to do is code in American and ensure that you explicitly convert your variables to U.S.-formatted strings before passing them to Excel.

Reading Data from Excel

When reading a cell's value, using its `Value` property, the data type that Excel provides to VBA is determined by a combination of the cell's value and its formatting. For example, the number 3000 could reach VBA as a `Double`, a `Currency`, or a `Date` (March 18, 1908). The only international issue of concern here is if the cell's value is read directly into a string variable—the conversion will then be done implicitly, and you may not get what you expect (particularly if the cell contains a Boolean value).

As is the case when sending data to Excel, the translation between U.S. and local functions and formats occurs when reading data from Excel. This means that a cell's `.Formula` or `.NumberFormat` property is given in English, and with U.S. number and date formatting, regardless of the user's choice of language or regional settings.

Although for most applications it is much simpler to read and write using U.S. formulas and formats, you will sometimes need to read exactly what the user is seeing (in their choice of language and regional settings). This is done by using the `xxxLocal` versions of many properties, which return (and interpret) strings according to the user's settings. They are typically used when displaying a formula or number format on a `UserForm`, and are discussed in the following section.

The Rules for Working with Excel

- ❑ Pass values to Excel in their natural format if possible (don't convert dates, numbers, or Booleans to strings if you don't have to). If you have strings, convert them yourself before passing them to Excel.
- ❑ When you have to convert numbers and dates to strings for passing to Excel (such as in criteria for `AutoFilter` or `Formula` strings), *always explicitly convert the data* to a U.S.-formatted string, using `Trim(Str(MyNumber))`, or the `sNumToUS` function shown previously, for all number and date types. Excel will then use it correctly and convert it to the local number and date formats.
- ❑ Avoid using `Date` literals (such as `#1/3/2007#`) in your code. It is better to use the VBA `DateSerial` or Excel `DATE` functions, which are not ambiguous.
- ❑ If possible, use the date number instead of a string representation of a date. Numbers are much less prone to ambiguity (though not immune).
- ❑ When writing formulas in code to be put into a cell (using the `.Formula` property), create the string using English functions. Excel will translate them to the local Office language for you.
- ❑ When setting number formats or using the `Format` function, use U.S. formatting characters—for example, `ActiveCell.NumberFormat = "dd mmm yyyy"`. Excel will translate these to the local number format for you.
- ❑ When reading information from a worksheet, using `.Formula`, `.NumberFormat`, and so forth, Excel will supply it using English formulas and U.S. format codes, regardless of the local Excel language.

Interacting with Users

The golden rule when displaying data to your users, or getting data from them, is to always respect their choice of Windows Regional Settings and Office UI Language. They should not be forced to enter numbers, dates, formulas, or number formats according to U.S. settings, just because it's easier for you to develop.

Paper Sizes

One of the most annoying things for users is discovering that their printer does not recognize the paper sizes used in your templates. If you use templates for your reports, you should always change the paper size to the user's default size. This can easily be determined by creating a new workbook and reading off the paper size from the `PageSetup` object.

Excel 2002 added the `Application.MapPaperSize` property to automatically switch between the equivalent common paper sizes of different countries (for example, between Letter in the U.S. and A4 in the UK). If this is property is set to `True`, Excel should take care of paper sizes for you.

Displaying Data

Excel does a very good job of displaying worksheets according to the user's selection of regional settings and language. When displaying data in UserForms or dialog sheets, however, you have to do all the formatting yourself.

As discussed previously, Excel converts numbers and dates to strings according to the WRS by default. This means that you can write code like the following and be safe in the knowledge that Excel will display it correctly:

```
tbNumber.Text = dNumber
```

There are two problems with this approach:

- ❑ Dates will get the default `ShortDate` format, which may not include four digits for the year, and will not include a time component. To force a four-digit year and include a time, use the `sFormatDate` function shown later in this chapter. It may be better, though, to use a less ambiguous date format on UserForms, such as the "mmm dd, yyyy" format used throughout this book.
- ❑ Versions of Excel prior to Excel 97 did not use the Windows Regional Settings for their default formats. If you are creating applications for use in older versions of Excel, you can't rely on the correct behavior.

The solution is simple—just use the `Format` function. This tells VBA to convert the number to a locally formatted string and works in all versions of Excel:

```
tbNumber.Text = Format(dNumber)
```

Interpreting Data

Your users will want to type in dates and numbers according to their choice of regional settings, and your code must validate those entries accordingly and maybe display meaningful error messages back to the user. This means that you have to use the `Cxxx` conversion functions, and the `IsNumeric` and `IsDate` validation functions.

Unfortunately, these functions all have their problems (such as not recognizing the `%` sign at the end of a number), which require some working around. An easy solution is to use the `bWinToNum` and `bWinToDate` functions shown at the end of this chapter to perform the validation, conversion, and error prompting for you. The validation code for a UserForm will typically be done in the OK button's `Click` event, and will be something like this:

```
Private Sub bnOK_Click()  
    Dim dResult As Double  
  
    'Validate the number or display an error  
    If bWinToNum(tbNumber.Text, dResult, True) Then  
        'It was valid, so store the number  
        Sheet1.Range("A1").Value = dResult  
    Else  
        'An error, so set the focus back and quit the routine  
        tbNumber.SetFocus  
    End If  
    Exit Sub  
End Sub  
  
'All OK and stored, so hide the userform  
Me.Hide  
End Sub
```

The xxxLocal Properties

Up until now, you have had to interact with Excel using English-language functions and the default U.S. formats. Presented now is an alternative situation, where your code interacts with the user in his or her own language using the appropriate regional settings. How, then, can your program take something typed in by the user (such as a number format or formula) and send it straight to Excel, or display an Excel formula in a message box in the user's own language?

Microsoft has anticipated this requirement and has provided us with local versions of most of the functions we need. They have the same names as their U.S. equivalents, with the word "Local" on the end (such as `FormulaLocal`, `NumberFormatLocal`, and so on). When you use these functions, Excel does not perform any language or format coercion for you. The text you read and write is exactly how it appears to the user. Nearly all of the older functions that return strings, or have string arguments, have local equivalents; newer objects do not. The following table lists all of the `xxxLocal` properties and the objects to which they apply.

Applies to	Return strings according to U.S. number and date formats and English text	Use and return locally formatted strings, and in the language used for the Office UI (or Windows version)
Number/string conversion	<code>Str</code>	<code>CStr</code>
Number/string conversion	<code>Val</code>	<code>Cdbl</code> , and so on
Name, Style, CommandBar	<code>.Name</code>	<code>.NameLocal</code>
Range, Chart Series	<code>.Formula</code>	<code>.FormulaLocal</code>
Range, Chart Series	<code>.FormulaR1C1</code>	<code>.FormulaR1C1Local</code>
Range, Style, Chart Data Label, Chart Axes Label	<code>.NumberFormat</code>	<code>.NumberFormatLocal</code>
Range	<code>.Address</code>	<code>.AddressLocal</code>
Range	<code>.AddressR1C1</code>	<code>.AddressR1C1Local</code>
Defined Name	<code>.RefersTo</code>	<code>.RefersToLocal</code>
Defined Name	<code>.RefersToR1C1</code>	<code>.RefertToR1C1Local</code>
Defined Name	<code>.Category</code>	<code>.CategoryLocal</code>

The Rules for Working with Your Users

- ❑ When converting a number or date to a text string for displaying to your users, or setting it as the `.Caption` or `.Text` property of a control, explicitly convert numbers and dates to text according to the WRS, using `Format(myNum)`, or `CStr(MyNum)`.
- ❑ When converting dates to strings, Excel does *not* rearrange the date part order, so `Format(MyDate, "dd/mm/yyyy")` will always give a DMY date order (but will show the correct date separator). Use `Application.International(xlDateOrder)` to determine the correct date order — as used in the `sFormatDate` function shown at the end of this chapter — or use one of the standard date formats (for example, `ShortDate`).
- ❑ If possible, use locale-independent date formats, such as `Format(MyDate, "mmm dd, yyyy")`. Excel will display month names according to the user's WRS language.
- ❑ When evaluating date or number strings that have been entered by the user, use `CDate` or `Cdbl` to convert the string to a date or number. These will use the WRS to interpret the string. Note that `Cdbl` does not handle the percent sign (%) character if the user has put one at the end of the number.
- ❑ Always validate numbers and dates entered by the user before trying to convert them. See the `bWinToNum` and `bWinToDate` functions at the end of this chapter for examples.
- ❑ When displaying information about Excel objects, use the `xxxLocal` properties (where they exist) to display it in your user's language and formats.
- ❑ Use the `xxxLocal` properties when setting the properties of Excel objects with text provided by the user (which you must assume is in their native language and format).

Excel 2007's International Options

In the Office Menu ⇨ Excel Options dialog, the Advanced section contains Editing Options that allow the user to specify the characters that Excel uses for the thousands and decimal separators, overriding the Windows Regional Settings. These options can be read and changed in code, using `Application.ThousandsSeparator`, `Application.DecimalSeparator`, and `Application.UseSystemSeparators`.

Using these properties you could, for example, print, save (as text), or publish a workbook using local number formats, change the separators being used, print, save (as text), or publish another version for a different target country, and then change them back to their original settings. It is a great pity, though, that Microsoft didn't add the ability to override the rest of the Windows Regional Settings attributes (such as date order, date separator, whether to use (10) or -10, and so on), and it's an omission that makes this feature virtually useless in practice.

One problem with using this feature is that it does not change the number format strings used in the `=TEXT` worksheet function. So as soon as the option is changed (either in code or through the UI), all cells that use the `=TEXT` function will no longer be formatted correctly. See later in this chapter for a workaround.

There is a big problem with this feature, in that while these options affect all of Excel's `xxxLocal` properties and functions (including the `Application.International` settings), *they are ignored by VBA*.

A couple examples highlight the scale of the problem:

- ❑ The VBA `Format` function—used almost every time a number is displayed to the user—ignores these options, resulting in text formatted according to the Windows Regional Settings, not those used by Excel.
- ❑ If the user types numbers into your UserForms or InputBoxes using the override separators, they will not be recognized as numbers by `IsNumeric`, `CDbl`, and so on, resulting in `TypeMismatch` errors.

The only way to work around this problem is to perform your own switching between WRS and Override separators before displaying numbers to the users, and immediately after receiving numbers from them, using the following two functions:

```
Function WRSToOverride(ByVal sNumber As String) As String

    Dim sWRS As String, sWRSThousand As String, sWRSDecimal As String
    Dim sXLThousand As String, sXLDecimal As String

    'Only do for Excel 2002 and greater
    If Val(Application.Version) >= 10 Then

        'Only do if the user is not using System Separators
        If Not Application.UseSystemSeparators Then

            'Get the separators used by the Windows Regional Settings
            sWRS = Format(1000, "#,##0.00")
            sWRSThousand = Mid(sWRS, 2, 1)
```

```

        sWRSDecimal = Mid(sWRS, 6, 1)

        'Get the override separators used by Excel
        sXLThousand = Application.ThousandsSeparator
        sXLDecimal = Application.DecimalSeparator

        'Swap from WRS' to Excel's separators
        sNumber = Replace(sNumber, sWRSThousand, vbTab)
        sNumber = Replace(sNumber, sWRSDecimal, sXLDecimal)
        sNumber = Replace(sNumber, vbTab, sXLThousand)
    End If
End If

'Return the converted string
WRSToOverride = sNumber

End Function

```

WRSToOverride converts between WRS and Excel's number formats, and returns a string using Excel's Override formatting. sNumber is a string containing a WRS-formatted number:

```

Function OverrideToWRS(ByVal sNumber As String) As String

    Dim sWRS As String, sWRSThousand As String, sWRSDecimal As String
    Dim sXLThousand As String, sXLDecimal As String

    'Only do for Excel 2002 and greater
    If Val(Application.Version) >= 10 Then

        'Only do if the user is not using System Separators
        If Not Application.UseSystemSeparators Then

            'Get the separators used by the Windows Regional Settings
            sWRS = Format$(1000, "#,##0.00")
            sWRSThousand = Mid$(sWRS, 2, 1)
            sWRSDecimal = Mid$(sWRS, 6, 1)

            'Get the override separators used by Excel
            sXLThousand = Application.ThousandsSeparator
            sXLDecimal = Application.DecimalSeparator

            'Swap from Excel's to WRS' separators
            sNumber = Replace(sNumber, sXLThousand, vbTab)
            sNumber = Replace(sNumber, sXLDecimal, sWRSDecimal)
            sNumber = Replace(sNumber, vbTab, sWRSThousand)
        End If
    End If

    'Return the converted string
    OverrideToWRS = sNumber

End Function

```

Chapter 25: International Issues

`OverrideToWRS` converts between WRS and Excel's number formats, and returns the string using WRS' formatting. `sNumber` is a string containing an Excel Override formatted number.

The final problem is that when you are interacting with users, you should be doing so using the number formats that they are familiar with. By adding the ability to override the Windows Regional Settings, Excel is introducing a third set of separators for you, and your users, to contend with. You are therefore completely reliant on the users remembering that override separators have been set, and that they may not be the separators that the users are used to seeing (that is, according to the WRS).

I strongly recommend that your application checks if `Application.UseSystemSeparators` is `False` and displays a warning message to the user, suggesting that it be turned on, so number formatting is set using Control Panel rather than Excel's overrides:

```
If Not Application.UseSystemSeparators Then
    MsgBox "Please set the required number formatting using Control Panel"
    Application.UseSystemSeparators = True
End If
```

Features That Don't Play by the Rules

The `xxxLocal` functions discussed in the previous section were all introduced during the original move from XLM functions to VBA in Excel 5.0. They cover most of the more common functions that a developer is likely to use. There were, however, a number of significant omissions in the original conversion, and new features have been added to Excel since then, with almost complete disregard for international issues.

This section guides you through the maze of inconsistency, poor design, and omission that you'll find hidden within the following Excel 2007 features. This table shows the methods, properties, and functions in Excel that are sensitive to the user's locale, but that do not behave according to the rules stated in previous sections.

Applies to	U.S. Version	Local Version
Opening a text file	<code>OpenText</code>	<code>OpenText</code>
Saving as a text file	<code>SaveAs</code>	<code>SaveAs</code>
Application	<code>.ShowDataForm</code>	<code>.ShowDataForm</code>
Worksheet, Range		<code>.Paste</code> , <code>.PasteSpecial</code>
PivotTable calculated fields and items	<code>.Formula</code>	
Conditional formats		<code>.Formula</code>
QueryTables (Web Queries)		<code>.Refresh</code>
Worksheet functions		<code>=TEXT</code>

Applies to	U.S. Version	Local Version
Range	.Value, .Formula	
Range	.FormulaArray	
Range	.AutoFilter	.AutoFilter
Range		.AdvancedFilter
Application	.Evaluate	
Application	.ConvertFormula	
Application	.ExecuteExcel4Macro	

Fortunately, workarounds are available for most of these issues. There are a few, however, that should be completely avoided.

The OpenText Function

`Workbooks.OpenText` is the VBA equivalent of opening a text file in Excel by using Office Menu ⇨ Open. It opens the text file, parses it to identify numbers, dates, Booleans, and strings, and stores the results in worksheet cells. Of relevance to this chapter is the method Excel uses to parse the data file (and how it has changed over the past few versions).

In Excel 5, the text file was parsed according to your Windows Regional Settings when opened from the user interface, but according to U.S. formats when opened in code. In Excel 97, this was changed to always use these settings from both the UI and code. Unfortunately, this meant that there was no way to open a U.S.-formatted text file with any confidence that the resulting numbers were correct. Since Excel 5, you have been able to specify the date order to be recognized, on a column-by-column basis, which works very well for numeric dates (for example, 01/02/2007).

Excel 2000 introduced the Advanced button on the Text Import Wizard, and the associated `DecimalSeparator` and `ThousandsSeparator` parameters of the `OpenText` method. These parameters allow you to specify the separators that Excel should use to identify numbers, and they are welcome additions. It is slightly disappointing to see that you cannot specify the general date order in the same way:

```
Workbooks.OpenText filename:="DATA.TXT", _
    dataType:=xlDelimited, tab:=True, _
    DecimalSeparator:=",", ThousandsSeparator:=."
```

While Microsoft is to be congratulated for fixing the number format problems in Excel 2000, further congratulations are due for fixing the problem of month and day names in Excel 2002, and for providing a much tidier alternative for distinguishing between U.S.-formatted and locally formatted text files.

Prior to Excel 2002, the `OpenText` method would only recognize month and day names according to the Windows Regional Settings, and date orders had to be specified for every date field that wasn't in MDY order. In Excel 2002, the `OpenText` method was given a `Local` parameter, with which you can specify whether the text file being imported uses U.S. English formatting throughout, or whether it uses locally formatted dates, numbers, and so on:

Chapter 25: International Issues

- ❑ If `Local:=True`, Excel will recognize numbers, dates, and month and day names according to the Windows Regional Settings (and the `OverrideDecimalSeparator` and `ThousandsSeparator` separators, if set).
- ❑ If `Local:=False`, Excel will recognize numbers, dates, and month and day names according to standard U.S. English settings.

In either case, the extra parameters of `DecimalSeparator`, `ThousandsSeparator`, and `FieldInfo` can be used to further refine the specification (overriding the `Local` parameter's defaults).

The SaveAs Function

`Workbook.SaveAs` is the VBA equivalent of saving a text file in Excel by using Office Menu ⇨ Save As and choosing a format of Text.

In all versions of Excel prior to Excel 2002, this resulted in a U.S.-formatted text file, with a DMY date order, English month and day names, and so on.

In Excel 2002, the `SaveAs` method was given the same `Local` parameter described in the `OpenText` method in the previous section, resulting in a U.S.-formatted or locally formatted text file, as appropriate. Note that if a cell has been given a locale-specific date format (that is, the number format begins with a locale specifier, such as `[$-814]` for Norwegian), that formatting will be retained in the text file, regardless of whether it is saved in U.S. or local format:

```
ActiveWorkbook.SaveAs "Data.Txt", xlText, local:=True
```

The ShowDataForm Sub Procedure

Using `ActiveSheet.ShowDataForm` means exposing yourself to one of the most dangerous of Excel's international issues. `ShowDataForm` is the VBA equivalent of the pre-2007 Data ⇨ Form menu item (which is not available by default in the Excel 2007 Ribbon, but can be added to the QAT by selecting the Form command from the All Commands list). It displays a standard dialog that allows the user to enter and change data in an Excel list or database. When run from Excel, the dates and numbers are displayed according to the WRS, and changes made by the user are interpreted according to the WRS, which fully complies with the user interaction rules stated previously.

When used in code, `ActiveSheet.ShowDataForm` displays dates and numbers according to U.S. formats but interprets them according to WRS. Hence, if you have a date of February 10, 2007, shown in the worksheet in the dd/mm/yyyy order of 10/02/2007, Excel will display it on the data form as 2/10/2007. If you change this to the 11th (2/11/2007), Excel will store November 2, 2007 in the sheet. Similarly, if you are using Norwegian number formats, a number of 1-decimal-234 will be displayed on the form as 1.234. Change that to read 1.235 and Excel stores 1235, one thousand times too big.

Because this is a rarely used feature, our suggestion is to leave it buried in the Ribbon command well and write your own data-entry UserForm.

Pasting Text

When pasting text from other applications into Excel, it is parsed according to the WRS. There is no way to tell Excel the number and date formats and language to recognize. The only workaround is to use a `DataObject` to retrieve the text from the clipboard, parse it yourself in VBA, then write the result to the sheet. For clarity, the following example assumes that the clipboard contains a single U.S.-formatted number and that it should be enhanced to check for U.S.-formatted dates as well:

```
Sub ParsePastedNumber()

    Dim oDO As DataObject
    Dim sText As String

    'Create a new data object
    Set oDO = New DataObject

    'Read the contents of the clipboard into the DataObject
    oDO.GetFromClipboard

    'Get the text from the DataObject
    sText = oDO.GetText

    'If we know the text is in a US format,
    'use Val() to convert it to a number
    ActiveCell.Value = Val(sText)

End Sub
```

PivotTable Calculated Fields and Items, and Conditional Format and Data Validation Formulas

If you are used to using the `.Formula` property of a range or chart series, you'll know that it returns and accepts formula strings that use English functions and U.S. number formats. There is an equivalent `.FormulaLocal` property that returns and accepts formula strings as they appear on the sheet (using the Office UI language and WRS number formats).

PivotTable calculated fields and items and conditional formats also have a `.Formula` property, but for these objects, it returns and accepts formula strings as they appear to the user — that is, it behaves in the same way as the `.FormulaLocal` property of a `Range` object. This means that to set the formula for one of these objects, you need to construct it in the Office UI language, and according to the WRS.

A workaround for this is to use the cell's own `.Formula` and `.FormulaLocal` properties to convert between the formats, as shown in the following `ConvertFormulaLocale` function.

This function converts a formula string between U.S. and local formats and languages:

```
Function ConvertFormulaLocale(sFormula As String, bUSToLocal As Boolean) _
    As String

    On Error GoTo ERR_BAD_FORMULA

    'Use a cell that is likely to be empty!
```

```
'This should be changed to suit your own situation
With ThisWorkbook.Worksheets(1).Range("IU1")
    If bUSToLocal Then
        .Formula = sFormula
        ConvertFormulaLocale = .FormulaLocal
    Else
        .FormulaLocal = sFormula
        ConvertFormulaLocale = .Formula
    End If

    .ClearContents
End With

ERR_BAD_FORMULA:

End Function
```

`sFormula` is the text of the formula to convert from, and `bUSToLocal` should be set to `True` to convert U.S. to local, and `False` to convert local to U.S.

Web Queries

Although the concept behind Web Queries is an excellent one, they have been implemented with complete disregard for international issues. When the text of the web page is parsed by Excel, all the numbers and dates are interpreted according to your Windows Regional Settings. This means that if a European web page is opened in the U.S., or a U.S. page is opened in Europe, it is likely that the numbers will be wrong. For example, if the web page contains the text *1.1*, it will appear as January 1 on a computer running Norwegian Windows.

The `WebDisableDateRecognition` option for the `QueryTable` can be used to prevent numbers from being recognized as dates. Setting Excel's override number and decimal separators can ensure that numbers are recognized correctly, if the web page is displayed in a known format.

Web Queries must be used with great care in a multinational application, using the following approach:

- Set `Application.UseSystemSeparators` to `False`.
- Set `Application.DecimalSeparator` and `Application.ThousandsSeparator` to those used on the web page.
- Perform the query, ensuring `WebDisableDateRecognition` is set to `True`.
- Reset `Application.DecimalSeparator`, `Application.ThousandsSeparator`, and `Application.UseSystemSeparators` to their original values.

=TEXT() Worksheet Function

The `=TEXT()` worksheet function converts a number to a string, according to a specified format. The format string has to use formatting characters defined by the Windows Regional Settings (or Excel's International Options override). Hence, if you use `=TEXT(NOW(), "dd/mm/yyyy")`, you will get 01/02/yyyy on Norwegian Windows, because Excel will only recognize å as the Norwegian number-format character used for years. Excel does not translate the number-format characters when it opens the

file on a different platform. The easiest way to work around this is to include a range of formatted cells within your workbook, and some VBA code that writes their number format to the cell when the workbook is opened. Any `=TEXT()` functions in the sheet can then refer to those cells for their formatting codes.

The Range.Value, Range.Formula, and Range.FormulaArray Properties

These three properties of a range break the rules by not having local equivalents. The strings passed to (and returned by) them are in U.S. format. Use the `ConvertFormulaLocale` function shown previously to convert between U.S. and local versions of formulas.

The Range.AutoFilter Method

The `AutoFilter` method of a `Range` object is a very curious beast. You are forced to pass it strings for its filter criteria, and hence you must be aware of its string handling behavior. The criteria string consists of an operator (`=`, `>`, `<`, `>=`, and so on) followed by a value. If no operator is specified, the equals operator (`=`) is assumed.

The key issue is that when using the equals operator, `AutoFilter` performs a textual match, whereas using any other operator results in a match by value. This gives you problems when trying to locate exact matches for dates and numbers. If you use equals, Excel matches on the text that is displayed in the cell — that is, the formatted number. Because the text displayed in a cell will change with different regional settings and Windows language versions, it is impossible for you to create a criteria string that will locate an exact match in all locales.

There is a workaround for this problem. When using any of the other filter criteria, Excel plays by the rules and interprets the criteria string according to U.S. formats. Hence, a search criterion of `">=02/01/2007"` will find all dates on or after February 1, 2007, in all locales. You can use this to match an exact date by using two `AutoFilter` criteria. The following code will give an exact match on February 1, 2007 and will work in any locale:

```
Range("A1:D200").AutoFilter 2, ">=02/01/2007", xlAnd, "<=02/01/2007"
```

The Range.AdvancedFilter Method

The `AdvancedFilter` method does play by the rules, but in a way that may be undesirable. The criteria used for filtering are entered on the worksheet in the criteria range. In a similar way to `AutoFilter`, the criteria string includes an operator and a value. Note that when using the equals operator, `AdvancedFilter` correctly matches by value and hence differs from `AutoFilter` in this respect.

Because this is entirely within the Excel domain, the string must be formatted according to the Windows Regional Settings to work, which poses a problem when matching on dates and numbers. An advanced filter search criterion of `">1.234"` will find all numbers greater than 1.234 in the U.S., but all numbers greater than 1234 when run in Norway. A criterion of `">02/03/2007"` will find all dates after February 3 in the U.S., but after March 2 in Europe.

Chapter 25: International Issues

The only workarounds are to populate the criteria strings from code, before running the `AdvancedFilter` method, or to use a calculated criteria string, using the `=TEXT` trick mentioned previously. Instead of a criterion of `">=02/03/2007"`, to find all dates on or after February 3, 2007, you could use this formula:

```
= ">=" & TEXT ( DATE ( 2007 , 2 , 3 ) , DateFormat )
```

Here, `DateFormat` is a reference to a cell that has been set to a local date format. If the date is an integer (does not contain a time component), you could also just use the criteria string `">=39116"`, and hope that the user realizes that 39116 is actually February 3, 2007.

The `Application.Evaluate`, `Application.ConvertFormula`, and `Application.ExecuteExcel4Macro` Functions

These functions all play by the rules, in that you must use U.S.-formatted strings. They do not, however, have local equivalents. To evaluate a formula that the user may have typed into a `UserForm` (or convert it from using relative to absolute cell ranges), you need to convert it to U.S. format before passing it to `Application.Evaluate` or `Application.ConvertFormula`.

The `Application.ExecuteExcel4Macro` function is used to execute XLM-style functions. One of the most common uses of it is to call the `XLM PAGE.SETUP` function, which is much faster than the VBA equivalent. This takes many parameters, including strings, numbers, and Booleans. Be very careful to explicitly convert all these parameters to U.S.-formatted strings, and avoid the temptation to shorten the code by omitting the `Str` around each one.

Responding to Office 2007 Language Settings

One major advance, starting with the release of Office 2000, is that there is a single set of executables, with a set of plug-in language packs (whereas in prior versions, each language was a different executable, with its own set of bugs). This makes it very easy for users of Office to have their own choice of language for the user interface, help files, and so on. In fact, if a number of people share the same computer, each person can run the Office applications in a different language.

As a developer of Excel applications, you must respect the user's language selection and do as much as you can to present your own user interface in their choice of language.

Where Does the Text Come From?

The following sections outline the three factors that together determine the text seen by the Office user.

Regional Settings Location

The Regional Settings location is chosen on the first tab (called Regional Settings) of the Control Panel's Regional Settings applet, and it defines:

- ❑ The day and month names shown in Excel cells for long date formats
- ❑ The day and month names returned by the VBA `Format` function
- ❑ The month names recognized by the VBA `CDate` function and when typing dates into Excel directly
- ❑ The month names recognized by the Text Import Wizard and the VBA `OpenText` method (when the `Local` parameter is `True`)
- ❑ The number format characters used in the `=TEXT` worksheet function
- ❑ The text resulting from the implicit conversion of Boolean values to strings, such as:
"I am " & True

Office UI Language Settings

The Office User Interface language can be selected by using the Microsoft Office Language Settings applet, installed with Office 2007, and it defines:

- ❑ The text displayed on Excel's menus and dialog boxes
- ❑ The text for the standard buttons on Excel's message boxes
- ❑ The text for Excel's built-in worksheet functions
- ❑ The text displayed in Excel's cells for Boolean values
- ❑ The text for Boolean values recognized by the Text Import Wizard, the VBA `OpenText` method, and when typing directly into Excel
- ❑ The default names for worksheets in a new workbook
- ❑ The local names for command bars

Language Version of Windows

By this, I mean the basic language version of Windows itself. This choice defines the text for the standard buttons in the VBA `MsgBox` function (when using the `vbMsgBoxStyles` constants). Hence, whereas the text of the buttons on Excel's built-in messages responds to the Office UI language, the text of the buttons on your own messages responds to the Windows language. Note that the only way to discover the Windows language is with a Windows API call.

Some things in Office 2007 are 100% (U.S.) English and don't respond to any changes in Windows language, regional settings, or Office UI language:

- ❑ The text resulting from the explicit conversion of Boolean values to strings — that is, all of `Str(True)`, `CStr(True)`, and `Format(True)` result in "True". Hence, the only way to convert a Boolean variable to the same string that Excel displays for it is to enter it into a cell and then read the cell's `.FormulaLocal` property.
- ❑ The text of Boolean strings recognized by `CBool`.

Identifying the Office UI Language Settings

The first step in creating a multilingual application is to identify the user's settings. You can identify the language chosen in Windows Regional Settings by using `Application.International(xlCountrySetting)`, which returns a number that corresponds approximately to the country codes used by the telephone system (1 is the USA, 44 is the UK, 47 is Norway, and so forth).

You can also use `Application.International(xlCountryCode)` to retrieve the user interface language using the same numbering system. This method has worked well in previous versions of Excel, where there were only 30 or so languages from which to choose your copy of Office.

Beginning with Office 2000, things have changed a little. By moving all the language configuration into separate language packs, Microsoft can support many more languages with relative ease. If you use the Object Browser to look at the `msoLanguageID` constants defined in the Office object library, you'll see that there are more than 180 languages and dialects listed.

You can use the following code to find out the exact Office UI language, and then decide whether you can display your application in that language or a similar language, or revert to a default language (as shown in the following section):

```
lLanguageID = Application.LanguageSettings.LanguageID(msoLanguageIDUI)
```

Creating a Multilingual Application

When developing a multilingual application, you have to balance a number of factors, including:

- The time and cost spent developing, translating, and testing the translated application
- The increased sales from having a translated version
- Improved ease of use, and hence reduced support costs
- The requirement for multilingual support
- Should you create language-specific versions, or use add-on language packs?

You also have to decide how much of the application to translate, and which languages to support:

- Translate nothing
- Translate only the packaging and promotional documentation
- Enable the code to work in a multilingual environment (month names and so on)
- Translate the user interface (menus, dialogs, screens, and messages)
- Translate the help files, examples, and tutorials
- Customize the application for each location (for example, to use local data feeds)
- Support left-to-right languages only
- Support right-to-left languages (and hence redesign your UserForms)
- Support Double-Byte-Character-Set languages (for example, Japanese)

The decision of how far to go will depend to a large extent on your users, your budget, and the availability of translators.

A Suggested Approach

It is doubtful that creating a single Excel application to support all 180-plus Office languages will make economic sense, but the time spent making your application support a few of the more common languages will often be a wise investment. This will, of course, depend on your users, and whether support for a new language is preferable to new features.

The approach that I take is to write the application to support multiple languages, and to provide users with the ability to switch between the installed languages or conform to their choice of Office UI Language. I develop the application in English, and then have it translated into one or two other languages depending on my target users. I will only translate it into other languages if there is sufficient demand.

How to Store String Resources

When creating multilingual applications, you cannot hard code *any* text strings that will be displayed to the user; you must look them up in a *string resource*. The easiest form of string resource is a simple worksheet table. Give all your text items a unique identifier and store them in a worksheet, one row per identifier and one column for each supported language. You can then look up the ID and return the string in the appropriate language using a simple `VLOOKUP` function.

You will need to do the same for all your menu items, worksheet contents, and UserForm controls. The following code is a simple example, which assumes you have a worksheet called `shLanguage` that contains a lookup table that has been given a name of `rgTranslation`. It also assumes you have a public variable to identify which column to read the text from. The variable typically would be set in an Options type screen.

Note that the code shown here is not particularly fast and is shown as an example. A faster (and more complex) routine would read the entire column of IDs and selected language texts into two static VBA arrays, then work from those, only reading in a new array when the language selection was changed:

```
Public iLanguageCol As Integer

Sub Test()
    iLanguageCol = 2
    MsgBox GetText(1001)
End Sub

' lTextID - The string ID to look up
Function GetText(lTextID As Long) As String

    Dim vaTest As Variant
    Static rgLangTable As Range

    'Set an object to point to the string resource table (once)
    If rgLangTable Is Nothing Then
        Set rgLangTable = ThisWorkbook.Worksheets("shLanguage") _
            .Range("rgTranslation")
    End If

    vaTest = Application.VLookup(lTextID, rgLangTable, iLanguageCol, False)
    GetText = vaTest
End Function
```

```
End If

'If the language choice is not set, assume the first language in our table
If iLanguageCol < 2 Then iLanguageCol = 2

'Try to locate and read off the required text
vaTest = Application.VLookup(lTextID, rgLangTable, iLanguageCol)

'If we got some text, return it
If Not IsError(vaTest) Then GetText = vaTest
End Function
```

Many of your messages will be constructed at run time. For example, you may have code to check that a number is within certain boundaries:

```
If iValue <= iMin Or iValue >= iMax Then
    MsgBox "The number must be greater than " & CStr(iMin) & _
        " and less than " & CStr(iMax) & "."
End If
```

This would mean that you have to store two text strings with different IDs in your resource sheet, which is both inefficient and much harder to translate. In the example given, you probably would not have a separate translation string for the full stop. Hence, the maximum value would always come at the end of the sentence, which may not be appropriate for many languages. A better approach is to store the combined string with placeholders for the two numbers, and substitute the numbers at run time (using the custom `ReplaceHolders` function, shown at the end of the chapter):

```
If iValue < iMin Or iValue > iMax Then
    MsgBox ReplaceHolders( _
        "The number must be greater than %0 and less than %1.", _
        CStr(iMin), CStr(iMax))
End If
```

The translator (who may not understand your program) can construct a correct sentence, inserting the values at the appropriate points.

Working in a Multilingual Environment

These sections provide some tips on how to work in a multilingual environment.

Allow Extra Space

In general, most other languages use longer words than the English equivalents. When designing your UserForms and worksheets, you must allow extra room for the non-English text to fit in the controls and cells. A good rule of thumb is to make your controls 1.5 times the width of the English text.

Using Excel's Objects

The names that Excel gives to its objects when they are created often depend on the user's choice of Office UI Language. For example, when creating a blank workbook using `Workbooks.Add`, it will not always be called `BookN`, and the first worksheet in it will not always be called `Sheet1`. With the German UI, for example, they are called `MappeN` and `Tabelle1`, respectively. Instead of referring to these objects

by name, you should create an object reference as they are created, then use that object elsewhere in your code:

```
Dim Wkb As Workbook, Wks As Worksheet

Set Wbk = Workbooks.Add
Set Wks = Wkb.Worksheets(1)
```

Using SendKeys

In the best of cases, the use of `SendKeys` should be avoided if at all possible. It is most often used to send key combinations to Excel to activate a menu item or navigate a dialog box. It works by matching the menu item or dialog control accelerator keys, in the same way that you can use `Alt+key` combinations to navigate Excel using the keyboard. When used in a non-English version of Excel, it is highly unlikely that the key combinations in the `SendKeys` string will match up with the menus and dialogs, having potentially disastrous results.

Using RibbonX

Chapter 14 explained how you can customize the Ribbon by storing a RibbonX XML stream in your workbooks. You have the choice of either specifying the control captions directly in the XML (using the `label` attribute) or using a `getLabel` or similar callback to specify them at run time. In a multilingual environment, you have to use the callbacks for all captions, `ToolTips`, and so forth, so you can provide the appropriate string for the chosen language.

The Rules for Developing a Multilingual Application

- ❑ Decide early in the analysis phase the level of multilingual support that you are going to provide, then stick to it.
- ❑ Do not include any text strings within your code. Always look them up in a table.
- ❑ Never construct sentences by concatenating separate text strings, because the foreign language version is unlikely to use the same word order. Instead, use placeholders in your text and replace the placeholder at run time.
- ❑ When constructing `UserForms`, always make the controls bigger than you need for the English text; most other languages use longer words.
- ❑ Do not try to guess the name that Excel gives to objects that you create in code. For example, when creating a new workbook, the first sheet will not always be `"Sheet1"`.
- ❑ Do not use `SendKeys`.

Some Helpful Functions

In addition to some of the custom functions already presented, such as `IsDateUS`, here are some more functions that are very useful when creating multinational applications. Note that the code has been written to be compatible with all versions of Excel, from 5.0 to 2007, and hence avoids the use of newer VBA constructs (such as giving optional parameters specific data types).

The bWinToNum Function

This function checks if a string contains a number formatted according to the Windows Regional Settings and converts it to a `Double`. The function returns `True` or `False` to indicate the success of the validation, and optionally displays an error message to the user. It is best used as a wrapper function when validating numbers entered by a user, as shown in the “Interacting with Users” section.

Note that if the user has used Excel’s International Options to override the WRS decimal and thousands separators, the `OverrideToWRS` function must be used to ensure you send a WRS-formatted string to this function:

```
Function bWinToNum(ByVal sWinString As String, _
                  ByRef dResult As Double, _
                  Optional bShowMsg) As Boolean

    Dim dFrac As Double

    ' Take a copy of the string to play with
    sWinString = Trim(sWinString)
    dFrac = 1

    If IsMissing(bShowMsg) Then bShowMsg = True
    If sWinString = "-" Then sWinString = "0"
    If sWinString = "" Then sWinString = "0"

    ' Check for percentage, strip it out and remember to divide by 100
    If InStr(1, sWinString, "%") > 0 Then
        dFrac = dFrac / 100
        sWinString = Application.Substitute(sWinString, "%", "")
    End If

    ' Are we left with a number string in windows format?
    If IsNumeric(sWinString) Then
        ' If so, convert it to a number and return success
        dResult = CDbl(sWinString) * dFrac
        bWinToNum = True
    Else
        ' If not, display a message, return zero and failure
        If bShowMsg Then MsgBox "This entry was not recognized as a number," _
            & Chr(10) & "according to your Windows Regional Settings.", vbOKOnly
        dResult = 0
        bWinToNum = False
    End If

End Function
```

`sWinString` is the string to be converted, and `dResult` is the converted number, set to 0 if the number is not valid or empty. `bShowMsg` is optional and should be set to `True` (or missing) to show an error message, or `False` to suppress the error message.

The bWinToDate Function

This provides the same functionality as `bWinToNum`, but for dates instead of numbers:


```

Function bWinToDate(ByVal sWinString As String, _
                   ByRef dResult As Double, _
                   Optional bShowMsg) As Boolean

    If IsMissing(bShowMsg) Then bShowMsg = True

    If sWinString = "" Then
        ' An empty string gives a valid date of zero
        dResult = 0
        bWinToDate = True

    ElseIf IsDate(sWinString) Then
        ' We got a proper date, so convert it to a Double
        ' (i.e. the internal date number)
        dResult = Cdbl(CDate(sWinString))
        bWinToDate = True

    Else
        ' If not, display a message, return zero and failure
        If bShowMsg Then MsgBox "This entry was not recognized as a date," & _
            Chr(10) & "according to your Windows Regional Settings.", vbOKOnly
        dResult = 0
        bWinToDate = False
    End If

End Function

```

`sWinString` is the string to be converted. `dResult` is the converted number, set to 0 if the number is not valid, or empty. `bShowMsg` is optional and should be set to `True` (or missing) to show an error message, or `False` to suppress the error message.

The `sFormatDate` Function

This function formats a date according to the Windows Regional Settings, using a four-digit year, and optionally including a time string in the result:

```

Function sFormatDate(dDate As Date, Optional bTimeReq) As String

    Dim sDate As String

    'Default bTimeReq to False if not supplied
    If IsMissing(bTimeReq) Then bTimeReq = False

    Select Case Application.International(xlDateOrder)
        Case 0 'month-day-year
            sDate = Format$(dDate, "mm/dd/yyyy")
        Case 1 'day-month-year
            sDate = Format$(dDate, "dd/mm/yyyy")
        Case 2 'year-month-day
            sDate = Format$(dDate, "yyyy/mm/dd")
    End Select

```

```
End Select

If bTimeReq Then sDate = sDate & " " & Format$(dDate, "hh:mm:ss")

sFormatDate = sDate

End Function
```

`dDate` is the Excel date number, and `bTimeReq` is an optional argument that should be set to `True` to include the time string in the result.

The ReplaceHolders Function

This function replaces the placeholders in a string with values provided to it:

```
Function ReplaceHolders(ByVal sString As String, ParamArray avReplace()) As String

    Dim i As Integer

    'Work backwards, so we don't replace %10 with our %1 text
    For i = UBound(avReplace) To LBound(avReplace) Step -1
        sString = Application.Substitute(sString, "%" & i, _
            avReplace(i - LBound(avReplace)))
    Next

    ReplaceHolders = sString

End Function
```

`sString` is the text to replace the placeholders in, and `avReplace` is a list of items to replace the placeholders.

Summary

It is possible to create an Excel application that will work on every installation of Excel in the world and support all 180-plus Office languages, but it is unlikely to be economically viable.

If you have a limited set of users and you are able to dictate their language and Windows Regional Settings, you can create your application without worrying about international issues. Even if this is the case, you should get into the habit of creating locale-independent code. The requirement for locale-independence should be included in your analysis, design, and coding standards. It is much, much easier and cheaper to write locale-independent code at the onset than to rework an existing application.

At a minimum, your application should work regardless of the users' choice of Windows Regional Settings or Windows or Office UI Language, or whether they have set non-standard thousands and decimal separators using Office Menu ⇨ Excel Options ⇨ Advanced. You should be able to achieve this by following the rules listed in this chapter.

The following Excel features don't play by the rules and have to be treated very carefully:

- `OpenText`
- `SaveAs` to a text file
- `ShowDataForm`
- Pasting text from other applications
- The `.Formula` property in all its guises
- `<range>.Value`
- `<range>.FormulaArray`
- `<range>.AutoFilter`
- `<range>.AdvancedFilter`
- The `=TEXT()` worksheet function
- `Application.Evaluate`
- `Application.ConvertFormula`
- `Application.ExecuteExcel4Macro`
- Web Queries

You may have to avoid some features in Excel completely:

- `SendKeys`
- Using `True` and `False` in imported text files

Programming the VBE

Up until now, the book has focused on writing VBA procedures to automate Excel. While writing the code, you have been working in the Visual Basic Editor (VBE), otherwise known as the Visual Basic Integrated Design Environment (VBIDE).

An object library is provided with Office 2007 that is shown as Microsoft Visual Basic for Applications Extensibility 5.3 in the VBE's Tools ⇄ References list. The objects in this library and their methods, properties, and events enable you to:

- ❑ Programmatically create, delete, and modify the code, UserForms, and references in your own and other workbooks
- ❑ Program the VBE itself to create useful Add-ins to assist you in your development efforts and automate many of your development tasks

There have been no significant changes to the Visual Basic for Applications Extensibility library between Office 2000 and Office 2007, so all the examples in this chapter apply equally to all versions.

The only responsible way to start this chapter is with a warning. Macro viruses work by using the methods shown in this chapter to modify the target file's code, thus infecting it. To prevent this, Microsoft has made it possible to disable access to all workbooks' `VBProjects`. By default the access is disabled, so none of the code in this chapter will work. To enable access to the `VBProjects`, place a check mark next to the Trust Access to the VBA Project Object Model checkbox in Excel 2007's Office Menu ⇄ Excel Options ⇄ Trust Center ⇄ Trust Center Settings ⇄ Macro Settings dialog.

This chapter explains how to write code to automate the VBE by walking you through the development of an Excel-related VBE Toolkit to speed up your application development. You will then add a few utilities to the toolkit that demonstrate how to programmatically manipulate code, UserForms, and references. For simplicity, most of the code examples in this chapter have not been provided with error handling. You can find the completed toolkit Add-in at www.wrox.com.

Identifying VBE Objects in Code

All the objects that form the VBE, and their properties and methods, are contained in their own object library. You need to create a reference to this library before you can use the objects; do this by switching to the VBE, selecting the menu item Tools ⇨ References, checking the Microsoft Visual Basic for Applications Extensibility 5.3 library, and clicking OK (see Figure 26-1).

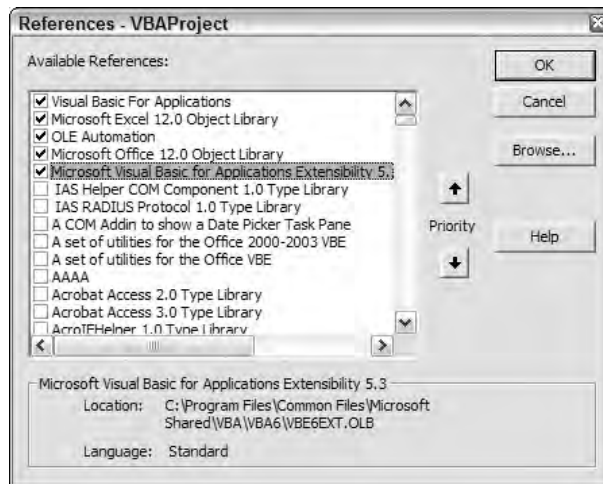


Figure 26-1

In code, this library is referred to as the VBIDE object library.

The full VBIDE object model is documented in Appendix B. The more important objects are summarized in the next sections.

The VBE Object

The top-level object of the Visual Basic Editor is known as the VBE object and is itself a property of the Excel Application object. Hence, to create an object variable to refer to the VBE, you need code similar to this:

```
Dim oVBE As VBIDE.VBE
Set oVBE = Application.VBE
```

The VBProject Object

This object is the container for all the programming aspects of a workbook, including UserForms, standard modules, class modules, and the code behind each worksheet and the workbook itself. Each VBProject corresponds to one of the top-level items in the Project Explorer. A specific VBProject object can be located either by iterating through the VBE's VBProjects collection or through the VBProject property of a workbook.

To find the `VBProject` that corresponds to the workbook `Book1.xlsm`, the following code can be used:

```
Dim oVBP As VBIDE.VBProject
Set oVBP = Workbooks("Book1.xlsm").VBProject
```

When creating Add-ins for the VBE itself, you often need to know which project is currently highlighted in the Project Explorer. This is given by the `ActiveVBProject` property of the VBE:

```
Dim obVBP As VBIDE.VBProject
Set obVBP = Application.VBE.ActiveVBProject
```

Note that the `ActiveVBProject` is the project that the user is editing within the VBE. It is not related in any way to the `ActiveWorkbook` given by Excel. In fact, with the Developer Editions of Office 2000 and Office XP, it was possible to create self-contained VB Projects that are not part of an Excel workbook. That capability should still be available if you upgraded from one of those versions.

The VBComponent Object

The UserForms, standard modules, class modules, and code modules behind the worksheets and workbook are all `VBComponent` objects. Each `VBComponent` object corresponds to one of the lower-level items in the Project Explorer tree. A specific `VBComponent` can be located through the `VBComponents` collection of a `VBProject`. Hence, to find the `VBComponent` that represents the `UserForm1` form in `Book1.xls`, code like this can be used:

```
Dim oVBC As VBIDE.VBComponent
Set oVBC = Workbooks("Book1.xls").VBProject.VBComponents("UserForm1")
```

The name of the `VBComponent` that contains the code behind the workbook, worksheets, and charts is given by the `CodeName` property of the related Excel object (the workbook, worksheet, or chart object). Hence, to find the `VBComponent` for the code behind the workbook (where code can be written to hook into workbook events), this code can be used:

```
Dim oVBC As VBIDE.VBComponent

With Workbooks("Book1.xlsm")
    Set oVBC = .VBProject.VBComponents(.CodeName)
End With
```

And for a specific worksheet:

```
Dim oVBC As VBIDE.VBComponent

With Workbooks("Book1.xlsm")
    Set oVBC = .VBProject.VBComponents(.Worksheets("Sheet1").CodeName)
End With
```

Note that the name of the workbook's `VBComponent` is usually `ThisWorkbook` in the Project Explorer. Do not be tempted to rely on this name. If your user has chosen a different language for the Office User Interface, it will be different. The name can also be easily changed by the user in the VBE. For this reason, *do not* use code like this:

Chapter 26: Programming the VBE

```
Dim oVBC As VBIDE.VBComponent

With Workbooks("Book1.xlsm")
    Set oVBC = .VBProject.VBComponents("ThisWorkbook")
End With
```

When developing Add-ins for the VBE, you often need to know the `VBComponent` that the user is editing (the one highlighted in the Project Explorer). This is given by the `SelectedVBComponent` property of the VBE:

```
Dim oVBC As VBIDE.VBComponent
Set oVBC = Application.VBE.SelectedVBComponent
```

Each `VBComponent` has a `Properties` collection, corresponding approximately to the list shown in the Properties window of the VBE when a `VBComponent` is selected in the Project Explorer. One of these is the `Name` property, shown in the following test routine:

```
Sub ShowNames()
    With Application.VBE.SelectedVBComponent
        Debug.Print .Name & ": " & .Properties("Name")
    End With
End Sub
```

For most `VBComponent` objects, the text returned by `.Name` and `.Properties("Name")` is the same. However, for the `VBComponent` objects that contain the code behind workbooks, worksheets, and charts, the `.Properties` collection includes all the properties of the native Excel object, so `.Properties("Name")` gives the name of the workbook, worksheet, or chart. You can use this to find the Excel object that corresponds to the item that the user is working on in the VBE, or the Excel workbook that corresponds to the `ActiveVBProject`. The code for doing this is shown later in this chapter.

The CodeModule Object

All of the VBA code for a `VBComponent` is contained within its `CodeModule` object. Through this object, you can programmatically read, add, change, and delete lines of code. There is only one `CodeModule` for each `VBComponent`. In the Office VBE, every type of `VBComponent` has a `CodeModule`.

The CodePane Object

This object provides access to the user's view of a `CodeModule`. Through this object, you can identify such items as the section of a `CodeModule` that is visible on the screen and the text that the user has selected. You can identify which `CodePane` is currently being edited by using the VBE's `ActiveCodePane` property:

```
Dim oCP As VBIDE.CodePane
Set oCP = Application.VBE.ActiveCodePane
```

The Designer Object

Some `VBComponents` (such as `UserForms`) present both code and a graphical interface to the developer. Whereas the code is accessed through the `CodeModule` and `CodePane` objects, the `Designer` object gives

you access to the graphical part. In the standard versions of Office, UserForms are the only components with a graphical interface for you to control. However, the Developer Editions included in Office 2000 and XP included a number of other items (such as the Data Connection Designer) that have graphical interfaces; these too are exposed to us through the `Designer` object and may be available in Office 2007 if you upgraded from one of those earlier versions.

These are the main objects that you'll be using throughout the rest of this chapter, as you create the VBE Toolkit Add-in.

Starting Up

There is very little difference in Excel 2007 between a normal workbook and an Add-in. The code and UserForms can be modified in the same manner, and they both offer the same level of protection (locking the Project from view). The two advantages of using an Add-in to hold your tools are that it is invisible within the Excel User Interface, and that it can be loaded using Excel's Add-ins dialog (Office Menu ⇨ Excel Options ⇨ Add-Ins ⇨ Manage: Excel Add-Ins ⇨ Go). This chapter uses the term *Add-in* to mean a container for tools that you're adding to Excel or the VBE. In fact, during the development of the Add-in, you will actually keep the file as a standard workbook, only converting it to an Add-in at the end.

Most Add-ins have a common structure, and the one you develop in this chapter will be no exception:

- ❑ A startup module to trap the opening and closing of the Add-in
- ❑ Some code to add the custom menu items to the command bars on opening and remove them on closing
- ❑ For the VBE, a class module to handle the menu items' `Click` events
- ❑ Some code to perform the menus' actions

Start with a new workbook and delete all of the worksheets, apart from the first. Press Alt+F11 to switch to the VBE, and find your workbook in the Project Explorer. Select the `VBProject` entry for it. In the Properties window, change the project's name to `aaVBETools2007`. The name starts with the prefix `aa`, so it always appears at the top of the Project Explorer, nicely out of the way of any other projects you may be developing.

Double-click the `ThisWorkbook` `VBComponent` to bring up its code pane and type in the following code:

```
Option Explicit

Dim moMenuHandler As CMenuHandler

' .....
' Subroutine: Workbook_Open
'
' Purpose:      Create a new instance of the menu-handling class.
'               The class's Initialize event sets up the menus.
'
Private Sub Workbook_Open()
    Set moMenuHandler = Nothing
    Set moMenuHandler = New CMenuHandler
End Sub
```

This code is run when the workbook is opened, and it just creates a new instance of a class module (which you'll create next) and stores a reference to it in a module-level variable. Using this technique, the `Class_Initialize` event is run when the workbook is opened. The class is kept alive while the workbook is open and is only destroyed (with `Class_Terminate` called) when the workbook is actually closed — crucially, *after* the user has been given the opportunity to cancel the close (whereas the `Workbook_BeforeClose` event is called before the user's opportunity to cancel the close).

Adding Menu Items to the VBE

The VBE uses the `CommandBar` object model rather than the Ribbon, so the procedure for adding menus to the VBE is almost the same as that documented in Chapter 15 for creating popup toolbars. There is one major difference, which is how to run your routine when the menu item is clicked. When adding menu items to Excel's popup toolbars, you set the `OnAction` property of the `CommandBarButton` to the name of the procedure to run. In the VBE, the `CommandBarButton` still has an `OnAction` property, but it is ignored.

Instead, Microsoft added the `Click` event to the `CommandBarButton` (and the `Change` event to the `CommandBarComboBox`). To use these events, you have to use a class module containing a variable of the correct type declared `WithEvents`. So add a class module to the project, give it the name of `CMenuHandler`, and type in the following code:

```
Option Explicit

'A variable to hook the click event for all our menus.
Private WithEvents mbtnEvents As Office.CommandBarButton

.....
' Subroutine: mbtnEvents_Click
'
' Purpose:   Handle the click event of all our menus, by running the procedure
'           stored in the button's OnAction property
'
Private Sub mbtnEvents_Click(ByVal Ctrl As Office.CommandBarButton, _
                             CancelDefault As Boolean)

    On Error Resume Next      'In case the routine is wrong/doesn't exist

    'Run the routine given by the commandbar control's OnAction property
    Application.Run Ctrl.OnAction

    'We handled it OK
    CancelDefault = True

End Sub
```

The key things to note here are:

- ❑ A variable, `mBtnEvents`, is declared to receive the `Click` event for the menu items.
- ❑ The `Click` event is raised by the object referred to by the `mBtnEvents` variable (the only one exposed by it).

- ❑ The `Click` event passes the `Ctrl` object (the menu item or toolbar button) that was clicked.
- ❑ The code runs the routine specified in the control's `OnAction` property. The code is simulating the behavior that occurs when adding menu items to Excel's popup toolbars.

To use this procedure, set the `mbtnEvents` variable to refer to one of your custom menu items. The `CommandBars` event model is designed in such a way that when setting the reference, you're also creating an association between the `mbtnEvents` variable and the menu's `Tag` property. This means that *all* menu items that share the same `Tag` property will also raise `Click` events against that variable. You can now add as many menus as you like, and all their `Click` events will be handled by that one procedure (as long as you give them all the same `Tag` property).

Now that you can respond to menus being clicked, all you need to do to build the Add-in is add some custom menus and the procedures to be called from the `Click` event. The easiest place to create your menus is from within the `Class_Initialize` event of the `CMenuHandler` class, which is called when the class is created in the `Workbook_Open` procedure. You can also include code to tidy up after yourself, by removing the custom menus in the `Class_Terminate` event. Because you'll be adding lots of menus in this chapter, it makes sense to factor out the menu-creation code into a separate procedure. The entire `CMenuHandler` class is shown here:

```
Option Explicit

'A variable to hook the click event for all our menus.
Private WithEvents mbtnEvents As Office.CommandBarButton

'A unique tag to identify our menus
Private Const msTAG As String = "Excel2007VBEWorkbookTools"

.....
' Subroutine: Class_Initialize
'
' Purpose:      Called when the class is created in the Workbook_Open event,
'               this procedure creates the menus for the Add-In
'
Private Sub Class_Initialize()

    'Just in case some of our menus got left behind, remove any previous remnants
    DeleteMenus

    'We'll add our menus here, later in the chapter

    'Associate our event-hook variable with any one of our menus.
    On Error Resume Next 'In case we don't find any
    Set mbtnEvents = Application.VBE.CommandBars.FindControl( _
        Type:=msoControlButton, Tag:=msTAG)

End Sub

.....
' Subroutine: DeleteMenus
'
' Purpose:      Find all the menus with our unique Tag and delete them
```

```
'
Private Sub DeleteMenus()

    Dim oCtl As CommandBarControl
    On Error Resume Next
    For Each oCtl In Application.VBE.CommandBars.FindControls(Tag:=msTAG)
        oCtl.Delete
    Next

End Sub

' .....
' Subroutine: AddMenu
'
' Purpose:    Add a menu to a menu bar, storing the procedure to run
'            in the control's OnAction
'
Private Sub AddMenu(ByRef oBar As CommandBar, ByVal sCaption As String, _
                   ByVal sProcedure As String, _
                   Optional ByVal lPosition As Long, _
                   Optional ByVal lFaceID As Long, _
                   Optional ByVal lStyle As MsoButtonStyle = msoButtonCaption, _
                   Optional ByVal sTooltip As String)

    Dim oBtn As CommandBarButton

    'If we were given a position, add it there. If not, add it at the end
    If lPosition > 0 Then
        Set oBtn = oBar.Controls.Add(msoControlButton, , , lPosition, True)
    Else
        Set oBtn = oBar.Controls.Add(msoControlButton, , , , True)
    End If

    With oBtn
        .Tag = msTAG
        .Caption = sCaption
        .FaceId = lFaceID
        .Style = lStyle
        .TooltipText = sTooltip
        .OnAction = "'" & ThisWorkbook.Name & "'" & sProcedure
    End With

End Sub

' .....
' Subroutine: mbtnEvents_Click
'
' Purpose:    Handle the click event of all our menus, by running the procedure
'            stored in the button's OnAction property
'
Private Sub mbtnEvents_Click(ByVal Ctrl As Office.CommandBarButton, _
                             CancelDefault As Boolean)

    On Error Resume Next    'In case the routine is wrong/doesn't exist
```

```

'Run the routine given by the commandbar control's OnAction property
Application.Run Ctrl.OnAction

'We handled it OK
CancelDefault = True

End Sub

.....
' Subroutine: Class_Terminate
'
' Purpose:      Remove our menus when the class is destroyed,
'              i.e. as the workbook is unloaded
'
Private Sub Class_Terminate()
    DeleteMenus
End Sub

```

Throughout the rest of this chapter, you'll be creating menus by adding code in `Class_Initialize` to call the `AddMenu` procedure, passing in the command bar for the menu to be added to and the properties for the menu.

The names for each of the top-level `CommandBars` in the VBE are shown in the following table. Note that Excel should always recognize these names, regardless of the user's choice of language for the Office User Interface (apart from a few rare exceptions, such as the Dutch menus, in which case you'll get a run-time error). The same is not true for the menu items placed on these toolbars. The only language-independent way to locate specific built-in menu items is to use their ID numbers. A routine to list the ID numbers of built-in menu items is provided in Chapter 15.

Name	Description
Menu Bar	The normal VBE menu bar
Standard	The normal VBE toolbar
Edit	The VBE edit toolbar, containing useful code-editing tools
Debug	The VBE debug toolbar, containing typical debugging tools
UserForm	The VBE UserForm toolbar, containing useful form-editing tools
MSForms	The popup menu for a UserForm (shown when you right-click the UserForm background)
MSForms Control	The popup menu for a normal control on a UserForm
MSForms Control Group	The popup menu that appears when you right-click a group of controls on a UserForm
MSForms MPC	The popup menu for the Multi-Page control
MSForms Palette	The popup menu that appears when you right-click a tool in the Control Toolbox

Table continued on following page

Name	Description
MSForms Toolbox	The popup menu that appears when you right-click one of the tabs at the top of the Control Toolbox
MSForms DragDrop	The popup menu that appears when you use the right mouse button to drag a control between tabs in the Control Toolbox, or onto a UserForm
Code Window	The popup menu for a code window
Code Window (Break)	The popup menu for a code window, when in Break (debug) mode
Watch Window	The popup menu for the Watch window
Immediate Window	The popup menu for the Immediate window
Locals Window	The popup menu for the Locals window
Project Window	The popup menu for the Project Explorer
Project Window (Break)	The popup menu for the Project Explorer, when in Break mode
Object Browser	The popup menu for the Object Browser
Property Browser	The popup menu for the Properties window
Docked Window	The popup menu that appears when you right-click the title bar of a docked window

Working with Workbooks

The ability to save a workbook from the VBE is built into Office 2007, but for a full complement of file operations, you need to add routines to create, open, and close workbooks and display the standard workbook properties form. Adding a Most Recently Used list to the VBE is left as an exercise for the reader.

Start with the simple code to create a new workbook, and then test the Add-in to prove that all the setup you've done so far works as expected. Then add code to the `CMenuHandler Class_Initialize` procedure to create the menus. Here you're adding a menu item to the File menu and a button to the Standard toolbar:

```
Private Sub Class_Initialize()  
  
    'Just in case some of our menus got left behind, remove any previous remnants  
    DeleteMenus  
  
    'Create the menus  
    AddMenu Application.VBE.CommandBars("File"), "&New Workbook", _  
            "FileNewBook", 1, 18, msoButtonIconAndCaption, "Create new workbook"  
  
    AddMenu Application.VBE.CommandBars("Standard"), "&New Workbook", _  
            "FileNewBook", 3, 18, msoButtonIcon, "Create new workbook"
```

```
'Associate our event-hook variable with any one of our menus.
On Error Resume Next 'In case we don't find any
Set mbtnEvents = Application.VBE.CommandBars.FindControl( _
    Type:=msoControlButton, Tag:=msTAG)
```

```
End Sub
```

Add a new module called `modMenuFile` and copy in the following code. The rest of the file-related routines will be added to this module later:

```
Option Explicit

'.....
' Subroutine: FileNewBook
'
' Purpose:    Create a new workbook
'
Sub FileNewBook()
    'Just ignore any errors
    On Error Resume Next

    'Create a new workbook
    Application.Workbooks.Add

    'Refresh the VBE display
    With Application.VBE.MainWindow
        .Visible = False
        .Visible = True
    End With
End Sub
```

This just adds a new blank workbook and refreshes the VBE display. Note that the VBE Project Explorer does not always update correctly when workbooks are added and removed through code. The easiest way to refresh the VBE display is to hide and then reshown the main VBE window.

First, check that the Add-in compiles, using the Debug ⇄ Compile menu. If any compile errors are highlighted, check your code against the previous three listings. To run the Add-in, place the cursor within the `Workbook_Open` procedure and press F5 to run it. If all goes well, a new menu item will be added to the VBE File menu and a standard New icon will appear on the VBE toolbar, just to the left of the Save icon, as shown in Figures 26-2 and 26-3.

When you click the button, a new workbook will be created in Excel and you will see its `VBProject` added to the Project Explorer. Congratulations, you have programmed the VBE.

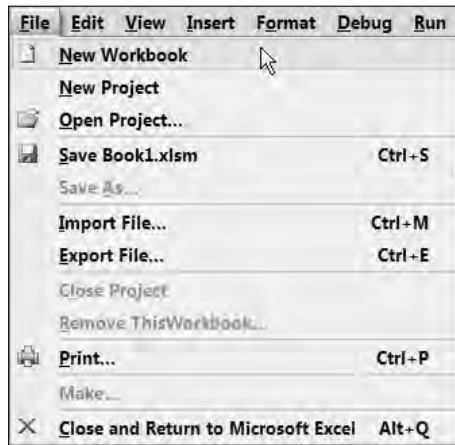


Figure 26-2



Figure 26-3

By the end of this chapter, you'll have functionality within the VBE to:

- Create a new workbook
- Open an existing workbook
- Save a workbook (this is built into the VBE)
- Close a workbook
- Display a workbook's *Properties* dialog

For the *Open* routine, another menu item will be added to the File menu, and another standard button will be added to the toolbar. For the *Close* and *Properties* routines, an item will once again be added to the File menu, but it will also be added to the Project Explorer popup menu, allowing you to close a *VBProject* by right-clicking it in the Project Explorer. The following additions to the *Class_Initialize* procedure will achieve this:

```
AddMenu Application.VBE.CommandBars("File"), "&Open Workbook", _
    "FileOpenBook", 2, 23, msoButtonIconAndCaption, "Open a workbook"
AddMenu Application.VBE.CommandBars("Standard"), "&Open Workbook", _
    "FileOpenBook", 4, 23, msoButtonIcon, "Open a workbook"

AddMenu Application.VBE.CommandBars("File"), "Close &Workbook", _
    "FileCloseBook", 3, , , _
    "Close the workbook containing the active project"
AddMenu Application.VBE.CommandBars("Project Window"), "Close &Workbook", _
    "FileCloseBook", , , , , _
```



```
"Close the workbook containing the active project"
```

```
AddMenu Application.VBE.CommandBars("File"), "Workbook Proper&ties", _
    "FileBookProps", 4, , , "Workbook Properties"
AddMenu Application.VBE.CommandBars("Project Window"), "Workbook Proper&ties", _
    "FileBookProps", , , , "Workbook Properties"
```

Note that the Close menu does not have a standard image, so the `lFaceID` parameter has been left out, and because you haven't specified a position, it is added to the bottom of the Project Explorer popup menu.

When opening a workbook in Excel, you can hold down the Shift key to prevent any macros from running. To accurately simulate that functionality, a test should be made to see if the Shift key is held down when the menu button is clicked, and to turn off any events if this is the case. Unfortunately, if the user holds down the Shift key when a workbook is opened in the VBE, the routine will stop dead (see Microsoft KnowledgeBase article Q175223 for the gory details at <http://support.microsoft.com/kb/q175223/>, which lists it as a bug in Excel 97, but it's still there in Excel 2007). Two workarounds are to use the Ctrl key for the same effect or wait until the Shift key is released before opening the workbook. The problem with waiting for the Shift key to be released is that the users will usually keep the key pressed until they see the file opened, so waiting for it to be released will just appear as though Excel has crashed.

You'll need a module in your Add-in for common utility functions, so add a standard module called `modCommon`, with the following code that uses Windows API calls to check the state of the Shift key:

```
Option Explicit

'The Add-In title is shown on various message boxes
Public Const psAddinTitle As String = "Excel 2007 VBA Prog Ref VBE Tools"

'Windows API call to see if the Shift, Ctrl and/or Alt keys are pressed
Private Declare Function GetAsyncKeyState Lib "user32" ( _
    ByVal vKey As Long) As Integer
.....
' Subroutine: fnGetShiftCtrlAlt
'
' Purpose:    Uses a Windows API call to detect if the shift, ctrl and/or
'            alt keys are pressed
'
Function fnGetShiftCtrlAlt() As Integer
    Dim iKeys As Integer

    Const VK_SHIFT As Long = &H10
    Const VK_CONTROL As Long = &H11
    Const VK_ALT As Long = &H12

    'Check to see if the Shift, Ctrl and Alt keys are pressed
    If GetAsyncKeyState(VK_SHIFT) <> 0 Then iKeys = iKeys + 1
    If GetAsyncKeyState(VK_CONTROL) <> 0 Then iKeys = iKeys + 2
    If GetAsyncKeyState(VK_ALT) <> 0 Then iKeys = iKeys + 4

    fnGetShiftCtrlAlt = iKeys
End Function
```

Chapter 26: Programming the VBE

The `Public Declare Function` line tells VBA about a function that's available in Windows to return whether the Shift, Ctrl, or Alt keys are held down — see Chapter 27 for more information about these types of calls. For the `Open` routine, Excel's `GetOpenFilename` method will be used to retrieve the name of a file, and then open it. If the user holds down the Ctrl key, the application events will be turned off, so the user can open the workbook without triggering any other code — within either the workbook being opened or Excel's application-level `WorkbookOpen` event. If the user is not holding down the Ctrl key, an attempt is made to run any `Auto_Open` routines in the workbook.

Add the following routine to the `modMenuFile` module:

```
.....
' Subroutine: FileOpenBook
'
' Purpose:   Opens an existing workbook
'
Sub FileOpenBook()
    Dim vFile As Variant, bCtrl As Boolean
    Dim Wbk As Workbook

    'Use our error handler to display a message if something goes wrong
    On Error GoTo ERR_HANDLER

    'Check if the Ctrl key is held down (1=Shift, 2=Ctrl, 4=Alt)
    bCtrl = (fnGetShiftCtrlAlt And 2) = 2

    'Hide Excel, so the Open dialog appears in the VBE
    Application.Visible = False

    'Get the filename to open (or False if cancelled)
    vFile = Application.GetOpenFilename

    'Make Excel visible again
    Application.Visible = True

    'If the user didn't cancel, open the workbook, adding it to Excel's MRU
    If Not (vFile = False) Then
        If bCtrl Then
            'If the user held the Ctrl key down, we should disable events and
            'not update links
            Application.EnableEvents = False

            Set Wbk = Workbooks.Open(Filename:=vFile, updatelinks:=0, AddToMru:=True)

            'Enable events again
            Application.EnableEvents = True
        Else
            'Shift not held down, so open the book normally and run Auto_Open
            Set Wbk = Workbooks.Open(Filename:=vFile, AddToMru:=True)

            Wbk.RunAutoMacros xlAutoOpen
        End If
    End If
End Sub
```

```

'Refresh the VBE display
With Application.VBE.MainWindow
    .Visible = False
    .Visible = True
End With

'No error, so Exit routine
Exit Sub

ERR_HANDLER:

'Display the error message (in the VBE Window) and end the routine.
Application.Visible = False

MsgBox "An Error has occurred." & vbCrLf & _
    Err.Number & ": " & Err.Description, vbOKOnly, psAddinTitle

Application.Visible = True
End Sub

```

Whenever a dialog is used that would normally be shown in the Excel window (including the built-in dialogs, any UserForms, and even `MsgBox` and `InputBox` calls), Excel automatically switches to its own window to show the dialog. When developing applications for the VBE, however, you really want the dialog to appear within the VBE window, not Excel's. The easiest way to achieve this effect is to hide the Excel window before showing the dialog, and then unhide it afterward.

The `Close` routine presents a new challenge. You are adding a `Close Workbook` menu item to the popup menu for the Project Explorer, and hence you need to determine which `VBProject` was clicked. The `ActiveVBProject` property of the VBE provides this, but a way is needed to get from the `VBProject` object to the workbook containing it. The method for doing this was described in the "Identifying VBE Objects in Code" section at the start of this chapter, and the code is shown in the following listing. Add it to the `modCommon` module, because it will be used in most of your functions:

```

Function fnActiveProjectBook() As Workbook
    Dim oVBP As VBIDE.VBProject, oVBC As VBIDE.VBComponent
    Dim sName As String

    'Get the VBProject that is active in the VBE
    Set oVBP = Application.VBE.ActiveVBProject

    On Error Resume Next

    'Try just reading the file name directly from the project
    sName = oVBP.FileName

    'If any errors occur (e.g. the project is locked),
    'assume we can't find the workbook
    On Error GoTo ERR_CANT_FIND_WORKBOOK

    If sName <> "" Then

        'Strip off the path
        If InStrRev(sName, "\") <> 0 Then

```

```
sName = Mid(sName, InStrRev(sName, "\") + 1)

'If it's the name of a workbook, we found it! (it could be the name of a
'VBE project)
If fnIsWorkbook(sName) Then
    Set fnActiveProjectBook = Workbooks(sName)
    Exit Function
End If
End If
Else
'Loop through all the VB Components in the project.
'The 'ThisWorkbook' component exposes the name of the workbook, but the
'component may not be called "ThisWorkbook"!

For Each oVBC In oVBP.VBComponents
    'Only need to check Document types (i.e. Excel objects)
    If oVBC.Type = vbext_ct_Document Then

        'Get the underlying name of the component
        sName = oVBC.Properties("Name")

        'Is it the name of an open workbook
        If fnIsWorkbook(sName) Then

            'Yes it is, but is it the correct one?
            If Workbooks(sName).VBProject Is oVBP Then

                'We found it!
                Set fnActiveProjectBook = Workbooks(sName)
                Exit Function
            End If
        End If
    End If
End If
Next
End If

PTR_CANT_FIND:

'We didn't find the workbook, so display an error message in the VBE
Application.Visible = False
MsgBox "Unable to identify the workbook for this project.", vbOKOnly, _
    psAddinTitle
Application.Visible = True

Set fnActiveProjectBook = Nothing
Exit Function

ERR_CANT_FIND_WORKBOOK:

'We had an error, so we can't find the workbook.
'Continue with the clean-up code.
Resume PTR_CANT_FIND
End Function
```

```

Function fnIsWorkbook(sBook As String) As Boolean

    'Use inline error handling to check for a workbook
    On Error Resume Next
    fnIsWorkbook = Len(Workbooks(sBook).Name) > 0
End Function

```

Now that you can identify which workbook corresponds to a VB Project, you can add the procedure to close a workbook in the `modMenuFile` module:

```

' .....
' Subroutine: FileCloseBook
'
' Purpose:    Closes the workbook containing the active VB Project
'
Sub FileCloseBook()
    Dim Wbk As Workbook, bCtrl As Boolean

    'Use our error handler to display a message if something goes wrong
    On Error GoTo ERR_HANDLER

    'Try to get the workbook containing the active VB Project
    Set Wbk = fnActiveProjectBook

    'If we didn't find it, just quit
    If Wbk Is Nothing Then Exit Sub

    'Check if the Ctrl key is held down (1=Shift, 2=Ctrl, 4=Alt)
    bCtrl = (fnGetShiftCtrlAlt And 2) = 2

    If bCtrl Then
        'Ctrl key is held down, so disable events and don't run Auto_Close
        'Disable events, so we don't run anything in the workbook being closed
        Application.EnableEvents = False

        'Close the workbook - Excel will ask to save changes etc.
        Wbk.Close

        'Enable events again
        Application.EnableEvents = True
    Else
        'Normal close so run any Auto_Close macros and close the workbook
        Wbk.RunAutoMacros xlAutoClose
        Wbk.Close
    End If

    'Exit procedure, bypass error handling routine
    Exit Sub

ERR_HANDLER:

    'Display the error message (in the VBE Window) and end the routine.
    Application.Visible = False

```

```
MsgBox "An Error has occurred." & vbCrLf & _
    Err.Number & ": " & Err.Description, vbOKOnly, psAddinTitle

Application.Visible = True
End Sub
```

The last workbook-related tool to be defined displays the File Properties dialog for the active VB Project's workbook. One of the main uses for the workbook properties is to provide the information shown in the Add-Ins dialog. The list box shows the Add-in's title from its Properties dialog, and the description shown when an Add-in is selected is obtained from its Comments box.

Excel's built-in Properties dialog can be used for this, but you cannot tell it which workbook to show the properties for—the active workbook is used. Therefore, any Add-ins need to be temporarily converted to normal workbooks and “unhidden” if they are hidden. After showing the Properties dialog, the workbooks must be converted back to Add-ins. The following code does that, and should also be put in the `modMenuFile` module:

```
.....
' Subroutine: FileBookProps
'
' Purpose:    Displays the workbook properties dialog for the active VB Project
'
Sub FileBookProps()
    Dim Wbk As Workbook, bAddin As Boolean, bVis As Boolean

    'Just ignore any errors
    On Error Resume Next

    'Try to get the workbook containing the active VB Project
    Set Wbk = fnActiveProjectBook

    'If we didn't find it, just quit
    If Wbk Is Nothing Then Exit Sub

    'Hide the Excel window, so the dialog seems to appear within the VBE environment
    Application.Visible = False

    'Using the workbook...
    With Wbk
        'If it is an addin, convert it to a normal workbook temporarily
        bAddin = .IsAddin
        .IsAddin = False

        'Make sure its window is visible
        bVis = .Windows(1).Visible
        .Windows(1).Visible = True

        'Display the Workbook Properties dialog
        Application.Dialogs(xlDialogProperties).Show

        'Restore the workbook's visibility and addin Status
        .Windows(1).Visible = bVis
        .IsAddin = bAddin
    End With
End Sub
```

```

End With

'Make Excel visible again
Application.Visible = True
End Sub

```

To test the Add-in so far, just rerun the `Workbook_Open` routine to re-create the menu items, and then check that each item works as intended.

Attempting to close the Add-in itself using the menu might cause the computer to lock up.

Working with Code

So far in this chapter, you have been working at a fairly high level in the VBIDE and Excel object models (limiting yourself to the `VBProject` and `Workbook` objects) to add typical file operations to the Visual Basic environment. You now have the ability to create new workbooks (and hence their VB Projects), open existing workbooks, change a workbook's properties, and save and close workbooks from within the VBE.

This section plunges to the lowest level of the VBE object model and explains how to work with the user's code. It shows how to detect the line of code the user is editing (and even identify the selected characters within that line), and get information about the procedure, module, and project containing that line of code. Adding and changing code is left until the next section, where you'll be creating a `UserForm`, adding some buttons to it, and adding code to handle the buttons' events.

To demonstrate how to identify the code that the user is working on, right-click access will be added to provide a print routine, with individual buttons to print the current selection, current procedure, module, or project. First add some code in `CMenuHandler Class_Initialize` to create a cascading menu to the Code Window popup menu, and then add four menu items to the cascading menu, each of which has its own face ID:

```

With Application.VBE.CommandBars("Code Window").Controls.Add(msoControlPopup, _
    before:=4, temporary:=True)

    .Caption = "P&rint"
    .Tag = msTAG

    AddMenu .CommandBar, "&Selected Text", "CodePrintSel", , 3518, _
        msoButtonIconAndCaption, "Print selected text"
    AddMenu .CommandBar, "&Procedure", "CodePrintProc", , 2564, _
        msoButtonIconAndCaption, "Print current procedure"
    AddMenu .CommandBar, "&Module", "CodePrintMod", , 472, _
        msoButtonIconAndCaption, "Print current module"
    AddMenu .CommandBar, "Pro&ject", "CodePrintProj", , 2557, _
        msoButtonIconAndCaption, "Print all modules in the project"

End With

```

Chapter 26: Programming the VBE

The result is shown in Figure 26-4.

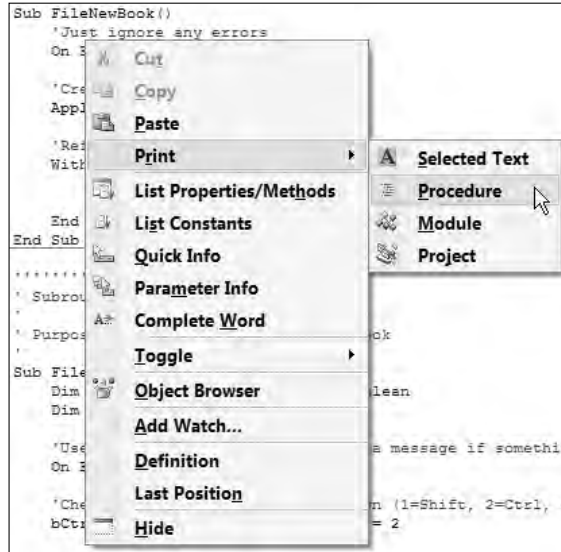


Figure 26-4

The code for the four printing routines will be placed into their own module, so add a new module to the project called `modMenuCode`.

Unfortunately, the VBIDE object model does not include a `Print` method for any of its objects. To provide right-click printing, there are three options:

- Show the VBE's `Print` dialog and operate it using `SendKeys`.
- Copy the code to a worksheet range and print it from there.
- Copy the code to a private instance of `Word`, reformat to show the Excel reserved words and so on in their correct colors, and then print it from `Word`.

For the sake of simplicity, the first option will be implemented. The main problem that this presents is how to select the `Selected Text`, `Module`, or `Project` option buttons on the `Print` dialog, using `SendKeys`, especially as the `Selected Text` option is only enabled when some text is actually selected.

The answer is to identify if any text is selected, then send the appropriate number of down arrow keys to the dialog to select either the `Module` or `Project` options. If you could rely on your users only ever having an English user interface language, you could send `Alt+M` or `Alt+J` keystrokes — sending down arrows works with any choice of user interface language.

The code for the `Selected Text` menu item is the simplest and is presented in the following listing. All that is required is to identify if the user has actually selected anything and, if so, to send some keystrokes to the `Print` dialog to print it:


```

Option Explicit

' .....
' Subroutine: CodePrintSel
'
' Purpose:    Print the current selection
'
Sub CodePrintSel()
    Dim lStartLine As Long, lStartCol As Long, lEndLine As Long, lEndCol As Long

    'Get the current selected text
    Application.VBE.ActiveCodePane.GetSelection lStartLine, lStartCol, _
                                                lEndLine, lEndCol

    'If there's something selected, print it
    If lStartLine <> lEndLine Or lStartCol <> lEndCol Then
        Application.SendKeys "{ENTER}"
        Application.VBE.CommandBars.FindControl(ID:=4).Execute
    End If
End Sub

```

The main items to note are:

- ❑ The `ActiveCodePane` property of the VBE is being used to identify which module the user is editing.
- ❑ The variables sent to the `GetSelection` method are sent `ByRef` and actually get filled by the method. After the call to `GetSelection`, they contain the start and ending line numbers and start and ending columns of the currently selected text.
- ❑ A simple Enter keystroke is sent to the keyboard buffer, then the VBE Print dialog is immediately shown by running the File ⇨ Print menu item (ID = 4) directly. By default (if some text is selected), when the VBE Print dialog is shown the Selected Text option is selected, so this does not need to be changed.

It should be noted that while no method in the Excel object model changes the value of the variables passed to it, this technique is quite common in the VBIDE object model and is getting more common in Windows applications generally.

To print the current module and project, very similar code can be used. The only difference is that it checks whether any text is selected (that is, if the Selected Text option in the Print dialog is enabled) and then sends a number of down keystrokes to the dialog to select the correct option. Both of these routines can be added to the `modMenuCode` module:

```

' .....
' Subroutine: CodePrintMod
'
' Purpose:    Print the current module
'
Sub CodePrintMod()

```

```
Dim lStartLine As Long, lStartCol As Long, lEndLine As Long, lEndCol As Long

'Get the current selection
Application.VBE.ActiveCodePane.GetSelection lStartLine, lStartCol, _
                                         lEndLine, lEndCol

If lStartLine <> lEndLine Or lStartCol <> lEndCol Then
    'If there's something selected, make sure the 'Module' item is selected
    Application.SendKeys "{DOWN}{ENTER}"
Else
    'If there's nothing selected, the 'Module' item is selected by default
    Application.SendKeys "{ENTER}"
End If

Application.VBE.CommandBars.FindControl(ID:=4).Execute
End Sub

.....
' Subroutine: CodePrintProj
'
' Purpose:    Print the current project

Sub CodePrintProj()
    Dim lStartLine As Long, lStartCol As Long, lEndLine As Long, lEndCol As Long

    'Get the current selection
    Application.VBE.ActiveCodePane.GetSelection lStartLine, lStartCol, _
                                         lEndLine, lEndCol

    'Make sure the 'Project' item is selected
    If lStartLine <> lEndLine Or lStartCol <> lEndCol Then
        Application.SendKeys "{DOWN}{DOWN}{ENTER}"
    Else
        Application.SendKeys "{DOWN}{ENTER}"
    End If

    Application.VBE.CommandBars.FindControl(ID:=4).Execute
End Sub
```

The code to print the current procedure is slightly more complex, because the Print dialog does not have a Current Procedure option. The steps you need to perform are as follows:

1. Identify and store away the user's current selection.
2. Identify the procedure (or declaration lines) containing the user's selection.
3. Expand the selection to encompass the full procedure (or all the declaration lines).
4. Show the Print dialog to print this expanded selection.
5. Restore the user's original selections.

Doing this on some PCs raises an interesting issue—the final step of restoring the user's original selection sometimes gets run before the Print dialog has been shown. This is presumably because the printing is

done on a separate thread of execution, and Excel 2007 is having a concurrency problem. The easy fix is to include a `DoEvents` statement immediately after showing the Print dialog, to let the print routine carry out its task. This will also yield control to the operating system, allowing it to process any pending or queued events. The code to print the current procedure should be added to the `modMenuCode` module:

```

.....
' Subroutine: CodePrintProc
'
' Purpose:      Print the current procedure

Sub CodePrintProc()
    Dim lStartLine As Long, lStartCol As Long, lEndLine As Long, lEndCol As Long
    Dim lProcType As Long, sProcName As String, lProcStart As Long, lProcEnd As Long

    With Application.VBE.ActiveCodePane
        'Get the current selection, so we know what to print and can restore it later
        .GetSelection lStartLine, lStartCol, lEndLine, lEndCol

        With .CodeModule
            If lStartLine <= .CountOfDeclarationLines Then
                'We're in the declarations section
                lProcStart = 1
                lProcEnd = .CountOfDeclarationLines
            Else
                'We're in a procedure, so find its start and end
                sProcName = .ProcOfLine(lStartLine, lProcType)
                lProcStart = .ProcStartLine(sProcName, lProcType)
                lProcEnd = lProcStart + .ProcCountLines(sProcName, lProcType)
            End If
        End With

        'Select the text to print
        .SetSelection lProcStart, 1, lProcEnd, 255

        'Print it
        Application.SendKeys "{ENTER}"
        Application.VBE.CommandBars.FindControl(ID:=4).Execute

        'The VBE Printing code is on another thread, so we need to let it do its stuff
        'before setting the selection back.
        DoEvents

        'And select the original text again
        .SetSelection lStartLine, lStartCol, lEndLine, lEndCol
    End With
End Sub

```

The main item to note in this code is that the `ProcOfLine` method accepts the start line as input, fills the `lProcType` variable with a number to identify the procedure type (Sub, Function, Property Let, Property Get, and so on), and returns the name of the procedure. The procedure type and name are used to find the start of the procedure (using `ProcStartLine`) and the number of lines within the procedure (`ProcCountLines`), which are then selected and printed.

Working with UserForms

The code examples presented in this chapter so far have been extending the VBE to provide additional tools for the developer. This section shifts its attention to programmatically creating and manipulating UserForms, adding controls, and adding procedures to the UserForm's code module to handle the controls' events. Though the example provided in this section continues to extend the VBE, the same code and techniques can be applied in end-user applications, including:

- ❑ Adding UserForms to workbooks created by the application
- ❑ Sizing the UserForm and moving and sizing its controls to make the best use of the available screen space
- ❑ Adding code to handle events in UserForms created by the application
- ❑ Changing the controls shown on an existing UserForm in response to user input
- ❑ Creating UserForms on the fly, as they are needed (for example, when the number and type of controls on the UserForm will vary significantly depending on the data to be shown)

These techniques will be demonstrated by writing code to add a UserForm to the active project, complete with standard-sized OK and Cancel buttons, as well as code to handle the buttons' `Click` events and the UserForm's `QueryClose` event. The UserForm's size will be set to two-thirds of the width and height of the Excel window, and the OK and Cancel buttons' position will be adjusted accordingly.

The example shown here is the difficult way to achieve the desired result, and is intended to be an educational, rather than a practical, example. The easy way to add a standardized UserForm is to create it manually and export it to disk as a `.frm` file, then import it using the following code (do not type this in):

```
Dim oVBC As VBComponent
Set oVBC = Application.VBE.ActiveVBProject.VBComponents.Import("MyForm.frm")
```

When you need to include it in another project, just import it again. The only advantage to doing it through code is that the UserForm can be given a size appropriate to the user's screen resolution and size, and its controls are positioned correctly.

Start by adding code in `CMenuHandler Class_Initialize` to create another menu:

```
AddMenu Application.VBE.CommandBars("Standard").FindControl(ID:=32806) _
.CommandBar, "&Standard Form", "FormNewUserform", 2, 581, _
msoButtonIconAndCaption, "Insert standardized UserForm"
```

The result of this addition will be the Standard Form menu, shown in Figure 26-5.

You'll be using objects from the `MSForms` object library to create the form and controls, so add a reference to the Microsoft Forms 2.0 Object Library by using the Tools ⇄ References dialog, or just adding and removing a UserForm. It takes quite a lot of code to create an entire form, so in this section the code listing is shown and explained piecemeal; all the highlighted lines of code should be typed in.

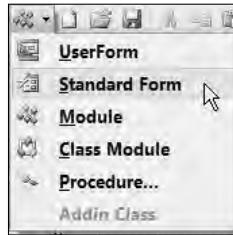


Figure 26-5

Add a new module for this routine, call it `modMenuForm`, and copy in the following code:

```
Option Explicit

'Window API call to freeze a window
'It does the same as Application.ScreenUpdating, but for the VBE
Private Declare Function LockWindowUpdate Lib "user32" (ByVal hwndLock As Long) _
    As Long
```

`Application.ScreenUpdating` does not affect the VBE, and the following `FormNewUserform` procedure to create a form results in quite a lot of screen activity as the form is sized and the controls are drawn. A simple Windows API call can be used to freeze the VBE window at the start of the routine and unfreeze it at the end (see Chapter 27 for more information about using this and other API functions):

```
.....
' Subroutine: FormNewUserform
'
' Purpose:    Creates a new userform, Add-Ing standard OK and Cancel buttons
'             and code to handle their events
'
Sub FormNewUserform()
    Dim oVBC As VBIDE.VBComponent, fmFrmDesign As UserForm, lLine As Long
```

Microsoft's Windows design guidelines recommend a gap of 6 points (approximately 4 pixels) between buttons, and between a button and the edge of a form.

```
Const dGap As Double = 6
```

This is one of the more complex routines in the Add-in, so some error-handling code will be added to it. Every routine in this chapter should really be given similar error-handling code.

```
'Use our error handler to display a message if something goes wrong
On Error GoTo ERR_HANDLER
```

Use the Windows API call to freeze the VBE's window. Note that `HWND` is a hidden property of the `MainWindow` object. To display the hidden properties of an object, open the Object Browser, right-click in its window, and click the Show Hidden Members item.

```
'Freeze the VBE window - same as Application.ScreenUpdating = False
LockWindowUpdate Application.VBE.MainWindow.HWND
```

Chapter 26: Programming the VBE

The `VBCOMponent` object (`oVBC` in the code) provides the “canvas” (background) of the UserForm, its `Properties` collection, and its `CodeModule`. When a new UserForm is added to a project, a `VBCOMponent` object is passed back that contains the form. The `VBCOMponent`’s `Properties` collection can be used to change the size, color, font, caption, and so forth of the form’s background.

```
'Add a new userform to the active VB Project
Set oVBC = Application.VBE.ActiveVBProject.VBComponents.Add(vbext_ct_MSForm)

'Set the form's height and width to 2/3 that of the Excel application.
oVBC.Properties("Width") = Application.UsableWidth * 2 / 3
oVBC.Properties("Height") = Application.UsableHeight * 2 / 3
```

The `VBCOMponent`’s `Designer` object provides access to the *content* of the UserForm, and is responsible for the area inside the form’s borders and below its title bar. In this code, two controls are added to the normal blank UserForm to provide standard OK and Close buttons. The name to use for the control (`Forms.CommandButton.1` in this case) can be found by adding the control to a worksheet, then examining the resulting `=EMBED` function. The appropriate controls can be found on the Developer tab of the Ribbon by clicking `Controls` ⇨ `Insert` ⇨ `ActiveX Controls`.

```
'Get the UserForm's Designer
Set fmFrmDesign = oVBC.Designer

'Use the designer to add the standard controls
With fmFrmDesign
  'Add an OK button, according to standard Windows size
  With .Controls.Add("Forms.CommandButton.1", "bnOK")
    .Caption = "OK"
    .Default = True
    .Height = 18
    .Width = 54
  End With

  'Add a Cancel button, according to standard Windows size
  With .Controls.Add("Forms.CommandButton.1", "bnCancel")
    .Caption = "Cancel"
    .Cancel = True
    .Height = 18
    .Width = 54
  End With

  'Move the OK and Cancel buttons to the bottom-right of the UserForm,
  'with a standard-width gap around and between them
  With .Controls("bnOK")
    .Top = fmFrmDesign.InsideHeight - .Height - dGap
    .Left = fmFrmDesign.InsideWidth - .Width * 2 - dGap * 2
  End With

  With .Controls("bnCancel")
    .Top = fmFrmDesign.InsideHeight - .Height - dGap
    .Left = fmFrmDesign.InsideWidth - .Width - dGap
  End With
End With
```

This could be extended to add list boxes, labels, checkboxes, and so on. From this point on, you could just as easily be working with an existing UserForm, changing its size and the position and size of its controls to make the best use of the available screen resolution. The preceding code simply moves the OK and Cancel buttons to the bottom-right corner of the UserForm, without adjusting their size. The same technique can be used to move and size all of a UserForm's controls.

Now that buttons have been added to the UserForm at the correct place (the bottom-right corner), code can be added to the UserForm's module to handle the buttons' and UserForm's events. To add a procedure to a code module, you can use the `CreateEventProc`, `InsertLines`, or `AddFromString` methods. In this example, the code is being added from strings. Alternatively, the code could be kept in a separate text file and imported into the UserForm's module. If `CreateEventProc` is used, all of the procedure's parameters are filled in on your behalf, and you get the `Private Sub...` line, the `End Sub` line, and a blank line between them. `CreateEventProc` returns the number of the line in the module where the `Private Sub...` was added, which is then used to insert a comment line and to replace the default blank line with the code:

```
' Now add some code to the userform's code module
With oVBC.CodeModule
    'Add the code for the OK button's Click event
    lLine = .CreateEventProc("Click", "bnOK")
    .InsertLines lLine, "'Standard OK button handler"
    .ReplaceLine lLine + 2, "    mbOK = True" & vbCrLf & "    Me.Hide"
```

If you use `AddFromString` or `InsertLines`, you have to supply the full text, such as this for the Cancel button:

```
'Add the code for the Cancel button's Click event
.AddFromString vbCrLf & _
    "'Standard Cancel button handler" & vbCrLf & _
    "Private Sub bnCancel_Click()" & vbCrLf & _
    "    mbOK = False" & vbCrLf & _
    "    Me.Hide" & vbCrLf & _
    "End Sub"
```

The code for the UserForm's `QueryClose` event is the same as that of the Cancel button, so some code will be added just to call the `bnCancel_Click` routine:

```
'Add the code for the UserForm's Close event - just call the Cancel code
lLine = .CreateEventProc("QueryClose", "UserForm")
.InsertLines lLine, "'Standard Close handler, treat same as Cancel"
.ReplaceLine lLine + 2, "    bnCancel_Click"

'And close the code window that was automatically opened by Excel
'when we created the event procedures
.CodePane.Window.Close
End With

'Unfreeze the VBE window - same as Application.ScreenUpdating = True
LockWindowUpdate 0&

Exit Sub
```

Chapter 26: Programming the VBE

The standard error handler unfreezes the window, displays the error message, and closes. Such error handling should be added to all the routines in the Add-in:

```
ERR_HANDLER:

'Unfreeze the VBE window - same as Application.ScreenUpdating = True
LockWindowUpdate 0&

'Display the error message (in the VBE Window) and end the routine.
Application.Visible = False

MsgBox "An Error occurred while creating the standard userform." & vbCrLf & _
    Err.Number & ": " & Err.Description, vbOKOnly, psAddinTitle

Application.Visible = True
End Sub
```

The Add-in is now complete. Switch back to Excel, use Office Menu ⇨ Prepare ⇨ Properties to give it a title and comment, then save the workbook as an Add-in (near the bottom of the list of available file types) with a .xlam extension. Then use the Add-Ins dialog box (Office Menu ⇨ Excel Options ⇨ Add-Ins ⇨ Manage: Excel Add-Ins ⇨ Go) to install it.

Working with References

One of the major enhancements in recent versions of VBA is the ability to declare a reference to an external object library (using the Tools ⇨ References dialog), and then use the objects defined in that library as if they were built into Excel. In this chapter, for example, you have been using the objects defined in the VBA Extensibility library without thinking about where they came from.

The term for this is *early binding*, so named because you are binding the external object library to your application at design time. Using early binding gives the following benefits:

- ❑ The code is much faster, because all the links between the libraries have been checked and compiled.
- ❑ The `New` operator can be used to create instances of the external objects.
- ❑ All of the constants defined in the object library can be utilized, thus avoiding numerous “magic numbers” throughout the code.
- ❑ Excel displays the Auto List Members, Auto Quick Info, and Auto Data Tips information for the objects while the application is being developed.

This is explained in more detail in Chapter 19.

There is, however, one major disadvantage. If you try to run your application on a computer that does not have the external object library installed, you will get a compile-time error that cannot be trapped using standard error-handling techniques — usually showing a perfectly valid line of code as being the culprit. Excel will display the error when it runs some code in a module, which contains:

- ❑ An undeclared variable in a procedure—and you didn't use `Option Explicit`
- ❑ A declaration of a type defined in the missing object library
- ❑ A constant defined in the missing object library
- ❑ A call to a routine, object, method, or property defined in the missing object library

The `VBEReferences` collection provides a method of checking that all the application's references are functioning correctly, and that all the required external object libraries are installed and are the correct versions. The code to check this should be put in your `Auto_Open` (or `Workbook_Open`) routine, and the module that contains the `Auto_Open` must not contain any code that uses the external object libraries. If there is a broken reference, it is unlikely that any other code will run, so the routine simply stops after displaying which references are missing. Typical `Auto_Open` code is:

```
Sub Auto_Open()
    Dim oRef As Object, bBroken As Boolean, sDescn As String

    For Each oRef In ThisWorkbook.VBProject.References
        'Is the link broken?
        If oRef.IsBroken Then
            'Some broken links don't have descriptions, so ignore the error
            On Error Resume Next
            sDescn = "<Not known>"
            sDescn = oRef.Description
            On Error GoTo 0

            'Display a message, asking the user to install the missing item
            MsgBox "Missing reference to:" & vbCrLf & _
                "    Name: " & sDescn & vbCrLf & _
                "    Path: " & oRef.FullPath & vbCrLf & _
                "Please reinstall this file."

            bBroken = True
        End If
    Next

    'If everything present and correct, carry on with the initializing code
    If Not bBroken Then
        '...Continue to open
    End If
End Sub
```

COM Add-ins

The VBE is common to all the Office 2007 applications, so it would be convenient to be able to create Add-ins for the VBE in a way that is not specific to a single Office application. In this chapter, you have created the VBE Toolkit as an Excel Add-in, with the result that the added functionality will only be available when using the VBE from within Excel. This was an appropriate choice, because the Add-in provides some Excel-specific functionality (such as opening and closing workbooks), but is a poor choice for the other functions you've added.

Chapter 26: Programming the VBE

COM Add-ins provide a different way of extending the VBE and other Office applications, in a way that is not necessarily specific to a single application. By using a COM Add-in, you can create a single Add-in that targets the VBE specifically, and hence works across all the Office applications.

Chapter 18 explains how to create COM Add-ins.

Summary

The Microsoft Visual Basic for Applications Extensibility 5.3 object library provides a rich set of objects, properties, methods, and events for controlling the VBE itself. Using these objects, developers can create their own labor-saving Add-ins to help in their daily development tasks.

Many end-user applications can also utilize these objects to manipulate their own code modules, UserForms, and references to provide a feature-rich, flexible, and robust set of functionality.

The example Add-in developed in this chapter can be downloaded from www.wrox.com.

Programming with the Windows API

Visual Basic for Applications is a high-level language that provides you with a rich, powerful, yet quite simple set of functionality for controlling the Office suite of products, as well as many other applications. You are insulated — some would say protected — from the “mundane minutiae” of Windows programming that, say, a C++ programmer has to contend with.

The price you pay for this protection is an inability to investigate and control many elements of the Windows platform. You can, for example, use `Application.International` to read most of the Windows Regional Settings and read the screen dimensions from `Application.UsableWidth` and `Application.UsableHeight`, but that’s about it. All the Windows-related items available to you are properties of the `Application` object and are listed in Appendix A.

The Windows platform includes a vast amount of low-level functionality that is not normally accessible from VBA, from identifying the system colors to creating a temporary file. Some of the functionality has been exposed in VBA, but only to a limited extent, such as creating and using an Internet connection (for example, you can open a page from the Internet using `Workbooks.Open "<URL>"`, but you can’t just download it to disk). There are also a number of other object libraries typically available on Windows computers that provide high-level, VBA-friendly access to the underlying Windows functionality. Examples of these are the Windows Scripting Runtime and the Internet Transfer Control.

There are times, though, when you need to go beyond the limits of VBA and the other object libraries, and delve into the files that contain the low-level procedures provided and used by Windows. The Windows Operating System is made up of a large number of separate files, mostly dynamic link libraries (DLLs), each containing code to perform a discrete set of interrelated functions. DLLs are files that contain functions that can be called by other Windows programs or other DLLs. They cannot be run like programs themselves.

These files are collectively known as the Windows Application Programming Interface, or Windows API. Some of the most common files you’ll use in the Windows API are detailed in the following table.

File	Function Group(s)
USER32.EXE	User interface functions (such as managing windows, the keyboard, clipboard, and so on)
KERNEL32.DLL	File- and system-related functions (such as managing programs)
GDI32.DLL	Graphics and display functions
SHELL32.DLL	Windows shell functions (such as handling icons and launching programs)
COMDLG32.DLL	Standard Windows dialog functions
ADVAPI32.DLL	Registry and NT Security functions
MPR.DLL and NETAPI32.DLL	Network functions
WININET.DLL	Internet functions
WINMM.DLL	Multimedia functions
WINSPOOL.DRV	Printing functions

This chapter explains how to use the functions contained in these files in your VBA applications and includes a number of useful examples. All of the Windows API functions are documented in the Platform SDK section of the MSDN Library at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winprog/winprog/windows_api_start_page.asp, which can be thought of as the online help for the Windows API. Because Microsoft regularly changes its URLs, that page can be found in the MSDN Library menu under Win32 and COM Development ⇄ Development Guides ⇄ Windows API ⇄ Windows API.

Anatomy of an API Call

Before you can use the procedures contained in the Windows DLLs, you need to tell the VBA interpreter where they can be found, the parameters they take, and what they return. Do this using the `Declare` statement, which VBA help shows as:

```
[Public | Private] Declare Sub name Lib "libname" [Alias "aliasname"]  
[[arglist]]  
[Public | Private] Declare Function name Lib "libname" [Alias "aliasname"]  
[[arglist]] [As type]
```

The following is the declaration used to find the Windows `TEMP` directory:

```
Private Declare Function GetTempPath Lib "kernel32" _  
    Alias "GetTempPathA" ( _  
    ByVal nBufferLength As Long, _  
    ByVal lpBuffer As String) As Long
```

This tells VBA that:

- ❑ The function is going to be referred to in the code as `GetTempPath`
- ❑ The procedure can be found in `kernel32.dll`
- ❑ It goes by the name of `GetTempPathA` in the DLL (case sensitive)
- ❑ It takes two parameters, a `Long` and a `String` (more about these later)
- ❑ It returns a `Long`

Microsoft used to include a simple API declaration viewer with the Developer Editions of Office, but that is now only available by installing Visual Studio and hasn't been updated to include the more recent versions of Windows. At the time of this writing, a great alternative is available for free download from www.activevb.de/rubriken/apiviewer/index-apiviewereng.html.

Interpreting C-Style Declarations

The MSDN library is the best source for information about the functions in the Windows API, but it's primarily targeted toward C and C++ programmers and displays the function declarations using C notation. The API viewer mentioned in the previous section contains many of the declarations for the core Windows functions in VBA notation, but if you encounter a function that it does not include, it is usually possible to convert the C notation to a VBA `Declare` statement, using the following method.

The declaration shown in MSDN for the `GetTempPath` function (at <http://msdn.microsoft.com/library/en-us/fileio/fs/gettemppath.asp>) is:

```
DWORD GetTempPath(  
    DWORD nBufferLength,  
    LPTSTR lpBuffer  
);
```

This should be read as:

```
<Return data type> <Function name>(  
    <Parameter data type> <Parameter name>,  
    <Parameter data type> <Parameter name>  
);
```

Rearranging the C-style declaration to a VBA `Declare` statement gives the following (where the C-style `DWORD` and `LPSTR` are converted to VBA data types later):

```
Declare Function <Our Name> Lib "???" Alias "GetTempPath" ( _  
    nBufferLength As DWORD, _  
    lpBuffer As LPTSTR _  
) As DWORD
```

On the Windows platform, there are two types of character sets. The ANSI character set has been the standard for many years and uses one byte to represent one character, which only gives 255 characters

Chapter 27: Programming with the Windows API

available at any time. To provide simultaneous access to a much wider range of characters (such as Far Eastern alphabets), the Unicode character set was introduced. This allocates two bytes for each character, allowing for 65,535 characters.

To provide the same functionality for both character sets, the Windows API includes two versions of all the functions that involve strings, denoted by the *A* suffix for the ANSI version and *w* for the Unicode (or Wide) version. VBA always uses ANSI strings, so you will always use the *A* version of the functions—in this case `GetTempPathA`. The C-style declarations also use different names for their data types, which you need to convert. Though not an exhaustive list, the following table shows the most common data types.

C Data Type	VBA Declaration
BOOL	ByVal <Name> As Long
BYTE	ByVal <Name> As Byte
BYTE *	ByRef <Name> As Byte
Char	ByVal <Name> As Byte
char huge *	ByVal <Name> As String
char FAR *	ByVal <Name> As String
char NEAR *	ByVal <Name> As String
DWORD	ByVal <Name> As Long
HANDLE	ByVal <Name> As Long
HBITMAP	ByVal <Name> As Long
HBRUSH	ByVal <Name> As Long
HCURSOR	ByVal <Name> As Long
HDC	ByVal <Name> As Long
HFONT	ByVal <Name> As Long
HICON	ByVal <Name> As Long
HINSTANCE	ByVal <Name> As Long
HLOCAL	ByVal <Name> As Long
HMENU	ByVal <Name> As Long
HMETAFILE	ByVal <Name> As Long
HMODULE	ByVal <Name> As Long
HPALETTE	ByVal <Name> As Long
HPEN	ByVal <Name> As Long
HRGN	ByVal <Name> As Long
HTASK	ByVal <Name> As Long

C Data Type	VBA Declaration
HWND	ByVal <Name> As Long
Int	ByVal <Name> As Long
int FAR *	ByVal <Name> As Long
LARGE INTEGER	ByVal <Name> As Currency
LONG	ByVal <Name> As Long
LPARAM	ByVal <Name> As Long
LPCSTR	ByVal <Name> As String
LPCTSTR	ByVal <Name> As String
LPSTR	ByVal <Name> As String
LPTSTR	ByVal <Name> As String
LPVOID	ByRef <Name> As Any
LRESULT	ByVal <Name> As Long
UINT	ByVal <Name> As Integer
UINT FAR *	ByVal <Name> As Integer
WORD	ByVal <Name> As Integer
WPARAM	ByVal <Name> As Integer
Other	Probably a user-defined type, which you need to define.

Some API definitions on the MSDN also include the `IN` and `OUT` identifiers. If the VBA type is shown in the table as `ByVal <Name> As Long`, it should be changed to `ByRef . . .` for the `OUT` parameters.

Strings are always passed `ByVal` (by value) to API functions. This is because VBA uses its own storage mechanism for strings, which the C DLLs do not understand. By passing the string `ByVal`, VBA converts its own storage structure into one that the DLLs can use.

Putting these into the declaration gives:

```
Declare Function GetTempPath Lib "???" _
    Alias "GetTempPathA" ( _
        ByVal nBufferLength As Long, _
        ByVal lpBuffer As String _
    ) As Long
```

Chapter 27: Programming with the Windows API

The only thing that the declaration doesn't tell you is the DLL that contains the function. Looking at the bottom of the MSDN page, the Requirements section includes the lines:

```
DLL: Requires kernel32.dll
```

This tells you that the function is in the file `kernel32.dll`, giving the final declaration of:

```
Declare Function GetTempPath Lib "kernel32.dll" _
    Alias "GetTempPathA" ( _
        ByVal nBufferLength As Long, _
        ByVal lpBuffer As String _
    ) As Long
```

This is the same as that shown in the API viewer, which should be your first reference point for all API function definitions. Note that the `Alias` clause is not required when the function name is the same as the alias (typically when there are no `String` parameters), and is automatically removed when the function is copied into a code module.

Warning: Using an incorrect function declaration is likely to crash Excel. When developing with API calls, save your work regularly.

Constants, Structures, Handles, and Classes

Most of the API functions include arguments that accept a limited set of predefined constants. For example, to get information about the operating system's capabilities, you can use the `GetSystemMetrics` function:

```
Declare Function GetSystemMetrics Lib "user32" ( _
    ByVal nIndex As Long) As Long
```

The value that you pass in the `nIndex` argument tells the function which metric you want to be given, and must be one of a specific set of constants that the function knows about. The applicable constants are listed in the MSDN documentation, with their corresponding values in many cases. The API Viewer also contains most of the constants that you are likely to need. There are more than 80 constants for `GetSystemMetrics`, including `SM_CXSCREEN` and `SM_CYSCREEN` to retrieve the screen's dimensions:

```
Const SM_CXSCREEN As Long = 0    'Screen width
Const SM_CYSCREEN As Long = 1    'Screen height

Private Declare Function GetSystemMetrics Lib "user32" _
    (ByVal nIndex As Long) As Long

Sub ShowScreenDimensions()
    Dim lScreenX As Long, lScreenY As Long

    'Get the screen's dimensions
    lScreenX = GetSystemMetrics(SM_CXSCREEN)
```



```
lScreenY = GetSystemMetrics(SM_CYSCREEN)

MsgBox "Screen resolution is " & lScreenX & "x" & lScreenY
End Sub
```

Many of the Windows API functions pass information using *structures*, which is the C term for a user-defined type (UDT). For example, the `GetWindowRect` function is used to return the size of a window, and is defined as:

```
Declare Function GetWindowRect Lib "user32" ( _
    ByVal hwnd As Long, _
    lpRect As RECT) As Long
```

The `lpRect` parameter is a `RECT` structure that is filled in by the `GetWindowRect` function with the window's dimensions. The `RECT` structure is defined on MSDN (at http://msdn.microsoft.com/library/en-us/gdi/rectangl_6cqa.asp) as:

```
typedef struct tagRECT {
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
} RECT;
```

This can be converted to a VBA UDT using the same data-type conversion shown in the previous section, giving:

```
Type RECT
    Left As Long
    Top As Long
    Right As Long
    Bottom As Long
End Type
```

The UDT definitions for most of the common structures are also included in the API Viewer.

The first parameter of the `GetWindowRect` function is shown as `hwnd`, and it represents a handle to a window. A *handle* is simply a pointer to an area of memory that contains information about the object being pointed to (in this case, a window). Handles are allocated dynamically by Windows and are unlikely to be the same between sessions. You cannot, therefore, hard code the handle number in your code, but must use other API functions to give you the handle you need. For example, to obtain the dimensions of a window, you need to get the window's `hwnd`. The API function `FindWindow` gives it to you:

```
'API call to find a window
Public Declare Function FindWindow Lib "user32" _
    Alias "FindWindowA" ( _
    ByVal lpClassName As String, _
    ByVal lpWindowName As String) As Long
```

This function looks through all the open windows until it finds one with the class name and caption that you ask for. The `Hwnd` property for Excel's main window was added to the `Application` object in Excel

Chapter 27: Programming with the Windows API

2002, so you only need to use `FindWindow` for the main Excel window if you want to be compatible with Excel 2000 or earlier, or if you want to find the window handle for any other type of window (such as `UserForms`). All of the code examples in this chapter use `FindWindow`, to be compatible with as many versions of Excel as possible.

There are many different types of windows in Windows applications, ranging from Excel's application window to the windows used for dialog sheets, `UserForms`, `ListBoxes`, and buttons. Each type of window has a unique identifier, known as its *class*. Some common class names in Excel are outlined in the following table.

Window	Class Name
Excel's main window	XLMAIN
Excel desktop	XLDESK
Excel worksheet	EXCEL7
Excel UserForm	ThunderDFrame (since Excel 2000) ThunderRT6DFrame (since Excel 2000, when running as a COM Add-In) ThunderXFrame (in Excel 97)
Excel status bar	EXCEL4
Excel chart window (prior to Excel 2007)	EXCELE

The `FindWindow` function uses this class name and the window's caption to find the window.

Note that the class names for some of Excel's standard items have changed with every release of Excel (but very few between Excel 2000 and 2007). You therefore need to include version checking in your code to determine which class name to use:

```
Select Case Val(Application.Version)
  Case Is >= 9   'Use Excel 2000/2002/2003/2007 class names
  Case Is >= 8   'Use Excel 97 class names
  Case Else      'Use Excel 5/95 class names
End Select
```

This results in a potential forward-compatibility problem: You don't know what the class names are going to be in future versions. Fortunately, Microsoft tries to retain compatibility as much as possible and has kept the same class names in Excel 2007 as prior versions. One of the more important changes is the loss of the `EXCELE` window. That class was officially used for the Chart Window in previous versions, but was often hijacked as a convenient way of locating a cell's on-screen position (by creating a chart at that cell and reading the position of the `EXCELE` window).

Putting these items together, you can use the following code to find the location and size of the Excel main window (in pixels):

```
'UDT to hold window dimensions
Type RECT
  Left As Long
```

```

    Top As Long
    Right As Long
    Bottom As Long
End Type

'API function to locate a window
Declare Function FindWindow Lib "user32" _
    Alias "FindWindowA" ( _
        ByVal lpClassName As String, _
        ByVal lpWindowName As String) As Long

'API function to retrieve a window's dimensions
Declare Function GetWindowRect Lib "user32" ( _
    ByVal hWnd As Long, _
    lpRect As RECT) As Long

Sub ShowExcelWindowSize()
    Dim hWnd As Long, uRect As RECT

    'Get the handle on Excel's main window
    'Could also use hWnd = Application.Hwnd in Excel 2002+
    hWnd = FindWindow("XLMAIN", Application.Caption)

    'Get the window's dimensions into the RECT structure
    GetWindowRect hWnd, uRect

    'Display the result
    MsgBox "The Excel window has the following dimensions:" & _
        vbCrLf & " Left: " & uRect.Left & _
        vbCrLf & " Right: " & uRect.Right & _
        vbCrLf & " Top: " & uRect.Top & _
        vbCrLf & " Bottom: " & uRect.Bottom & _
        vbCrLf & " Width: " & (uRect.Right - uRect.Left) & _
        vbCrLf & " Height: " & (uRect.Bottom - uRect.Top)

End Sub

```

Resize the Excel window to cover a portion of the screen, and run the `ShowExcelWindowSize` routine. You should be given a message box showing the window's dimensions. Now try it with Excel maximized—you may get negative values for the top and left. This is because the `GetWindowRect` function returns the size of the Excel window, measuring around the edge of its borders. When maximized, the borders are off the screen, but still part of the window.

What If Something Goes Wrong?

One of the hardest parts of working with the Windows API functions is identifying the cause of any errors. If an API call fails for any reason, it *should* return some indication of failure (usually a zero result from the function) and register the error with Windows. You should then be able to use the VBA function `Err.LastDLLError` to retrieve the error code, and use the `FormatMessage` API function to retrieve the descriptive text for the error:

Chapter 27: Programming with the Windows API

```
'Windows API declaration to get the API error text
Private Declare Function FormatMessage Lib "kernel32" _
    Alias "FormatMessageA" ( _
        ByVal dwFlags As Long, _
        ByVal lpSource As Long, _
        ByVal dwMessageId As Long, _
        ByVal dwLanguageId As Long, _
        ByVal lpBuffer As String, _
        ByVal nSize As Long, _
        Arguments As Long) As Long

'Constant for use in the FormatMessage API function
Private Const FORMAT_MESSAGE_FROM_SYSTEM As Long = &H1000

Sub ShowExcelWindowSize()
    'Define some variables to use in the API calls
    Dim hWnd As Long, uRect As RECT

    'Get the handle on Excel's main window
    hWnd = FindWindow("XLMAIN", Application.Caption)

    If hWnd = 0 Then
        'An error occurred, so get the text of the error
        MsgBox LastDLLerrText(Err.LastDllError)
    Else
        'Etc.
    End If
End Sub

Function LastDLLerrText(ByVal lErrorCode As Long) As String
    ' *****
    ' *
    ' * Function Name:    LastDLLerrText
    ' *
    ' * Input:           lErrorCode - a Windows error number
    ' *
    ' * Output:          Returns the description corresponding to the error
    ' *
    ' * Purpose:         Retrieve a Windows error description
    ' *
    ' *****

    Dim sBuff As String * 255, iAPIResult As Long

    'Get the text of the error and return it
    iAPIResult = FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, 0&, lErrorCode, _
        0, sBuff, 255, 0)

    LastDLLerrText = Left(sBuff, iAPIResult)
End Function
```

The full code for this example can be found in the module 'm1_ExcelWindowSize' in the API Examples.xlsm workbook, available at www.wrox.com.

Unfortunately, this technique does not always work. For example, if you change the class name to `XLMAINTTEST` in the `FindWindow` function call, you may expect to get an error message of `Unable to find window`. In Windows XP, the error information is populated with the cryptic text `The system cannot find the file specified`. In most cases, you do get more useful error information, as shown in the next section.

Wrapping API Calls in Class Modules

If you need to use lots of API calls in your application, your code can get very messy, very quickly. Most developers prefer to encapsulate the API calls within class modules, which provide a number of benefits:

- ❑ The API declarations and calls are removed from your core application code.
- ❑ The class module can perform a number of initialization and clean-up tasks, improving your system's robustness.
- ❑ Many of the API functions take a large number of parameters, most of which are not used in your situation. The class module needs to expose only those properties that need to be changed by your calling routine.
- ❑ Class modules can be stored as text, providing a self-contained set of functionality that is easy to reuse in future projects.

The following code is an example of a class module for working with temporary files, allowing the calling code to:

- ❑ Create a temporary file in the Windows default `TEMP` directory
- ❑ Create a temporary file in a user-specified directory
- ❑ Retrieve the path and filename of the temporary file
- ❑ Retrieve the text of any errors that may have occurred while creating the temporary file
- ❑ Delete the temporary file after use

Create a class module called `CTempFile` and copy in the following code (this class can also be found in the `API Examples.xlsm` file at www.wrox.com):

```
'*****  
'*  
'* MODULE NAME:      EXCEL 2007 PROG REF - TEMP FILE CLASS  
'* AUTHOR:          STEPHEN BULLEN, Office Automation Ltd.  
'*  
'* CONTACT:         stephen@oaltd.co.uk  
'* WEB SITE:        http://www.oaltd.co.uk  
'*  
'* DESCRIPTION:     Encapsulates API calls for handling temporary files  
'*  
'*****  
Option Explicit
```

Chapter 27: Programming with the Windows API

```
'Windows API declaration to find the Windows Temporary directory
Private Declare Function GetTempPath Lib "kernel32" _
    Alias "GetTempPathA" ( _
        ByVal nBufferLength As Long, _
        ByVal lpBuffer As String) As Long

'Windows API declaration to create, and return the name of,
'a temporary filename
Private Declare Function GetTempFileName Lib "kernel32" _
    Alias "GetTempFileNameA" ( _
        ByVal lpszPath As String, _
        ByVal lpPrefixString As String, _
        ByVal wUnique As Long, _
        ByVal lpTempFileName As String) As Long

'Windows API declaration to get the text for an API error code
Private Declare Function FormatMessage Lib "kernel32" _
    Alias "FormatMessageA" ( _
        ByVal dwFlags As Long, _
        ByVal lpSource As Long, _
        ByVal dwMessageId As Long, _
        ByVal dwLanguageId As Long, _
        ByVal lpBuffer As String, _
        ByVal nSize As Long, _
        Arguments As Long) As Long

'Constant for use in the FormatMessage API function
Const FORMAT_MESSAGE_FROM_SYSTEM As Long = &H1000

'Variables to store the path, file and error message
Dim msTempPath As String
Dim msTempFile As String
Dim msErrMsg As String
Dim mbTidyUp As Boolean
```

One advantage of using a class module is that you can perform some operations when the class is initialized. In this case, you will identify the default Windows TEMP directory. The temporary file will be created in this directory, unless the calling code tells you otherwise:

```
'Get the Windows temporary path when the class is initialized
Private Sub Class_Initialize()
    'Define some variables to use in the API calls
    Dim sBuff As String * 255, lAPIResult As Long

    'Call the Windows API function to get the TEMP path
    lAPIResult = GetTempPath(255, sBuff)

    If lAPIResult = 0 Then
        'An error occurred, so get the text of the error
        msErrMsg = LastDLLerrText(Err.LastDllError)
    Else
        'Store the TEMP path
        msTempPath = Left(sBuff, lAPIResult)
    End If
End Sub
```

This is the routine to create the temporary file, returning its name (including the path). In its simplest use, the calling routine can just call this one method to create a temporary file:

```
'Create a temporary file, returning its name (including the path)
Public Function CreateFile() As String
    'Define some variables to use in the API calls
    Dim sBuff As String * 255, lAPIResult As Long

    'Try to get a temporary file name (also creates the file)
    lAPIResult = GetTempFileName(msTempPath, "", 0, sBuff)

    If lAPIResult = 0 Then
        'An error occurred, so get the text of the error
        msErrMsg = LastDLLErrText(Err.LastDllError)
    Else
        'Created a temp file OK, so store the file and "OK" error message
        msTempFile = Left(sBuff, InStr(1, sBuff, Chr(0)) - 1)
        msErrMsg = "OK"
        mbTidyUp = True

        CreateFile = msTempFile
    End If
End Function
```

In a class module, you can expose a number of properties that allow the calling routine to retrieve and modify the temporary file creation. For example, you may want to enable the calling program to set which directory to use for the temporary file. You could extend this to make the property read-only after the file has been created, raising an error in that case. The use of `Property` procedures in class modules is described in more detail in Chapter 16:

```
'Show the TEMP path as a property of the class
Public Property Get Path() As String
    'Return the path, without the final '\'
    Path = Left(msTempPath, Len(msTempPath) - 1)
End Property

'Allow the user to change the TEMP path
Public Property Let Path(sNewPath As String)
    msTempPath = sNewPath

    'Ensure path ends with a \
    If Right(msTempPath, 1) <> "\" Then
        msTempPath = msTempPath & "\"
    End If
End Property
```

You can also give the calling routine read-only access to the temporary file's name and full name (that is, including the path):

```
'Show the temporary file name as a property
Public Property Get Name() As String
    Name = Mid(msTempFile, Len(msTempPath) + 1)
End Property
```

Chapter 27: Programming with the Windows API

```
'Show the full name (directory and file) as a property
Public Property Get FullName() As String
    FullName = msTempFile
End Property
```

Give the calling program read-only access to the error messages:

```
'Show the error message as a property of the class
Public Property Get ErrorText() As String
    ErrorText = msErrMsg
End Property
```

You'll also allow the calling program to delete the temporary file after use:

```
'Delete the temporary file
Public Sub Delete()
    On Error Resume Next 'In case it has already been deleted
    Kill msTempFile
    mbTidyUp = False
End Sub
```

By default, you will delete the temporary file that you created when the class is destroyed. The calling application may not want you to, so provide some properties to control this:

```
'Whether or not to delete the temp file when the
'class is deleted
Public Property Get TidyUpFiles() As Boolean
    TidyUpFiles = mbTidyUp
End Property

'Allow the user to prevent the deletion of his/her own files
Public Property Let TidyUpFiles(bNew As Boolean)
    mbTidyUp = bNew
End Property
```

In the class's `Terminate` code, you'll delete the temporary file, unless told not to. This code is run when the instance of the class is destroyed. If declared within a procedure, this will be when the class variable goes out of scope at the end of the procedure. If declared at a module level, it will occur when the workbook is closed:

```
Private Sub Class_Terminate()
    If mbTidyUp Then Delete
End Sub
```

The same function you saw in the previous section is used to retrieve the text associated with a Windows API error code:

```
'Get the text associated with a Windows API error code
Private Function LastDLLerrText(ByVal lErrorCode As Long) As String
    Dim sBuff As String * 255, lAPIResult As Long

    'Get the text of the error and return it
```



```
lAPIResult = FormatMessage(FORMAT_MESSAGE_FROM_SYSTEM, _
    0&, lErrorCode, 0, sBuff, 255, 0)

LastDLLErrText = Left(sBuff, lAPIResult)
End Function
```

Once this class module is included in a project, the calling routine does not need to know anything about any of the API functions you're using:

```
Sub TestCTempFile()
    Dim oTempFile As New CTempFile

    If oTempFile.CreateFile = "" Then
        MsgBox "An error occurred while creating the temporary file:" & _
            vbCrLf & oTempFile.ErrorText
    Else
        MsgBox "Temporary file " & oTempFile.FullName & " created"
    End If
End Sub
```

This results in a message like this:

```
Temporary file C:\WINDOWS\TEMP\5024.TMP created
```

Note that the temporary file is created during the call to `CreateFile`. When the procedure ends, the variable `oTempFile` goes out of scope and hence is destroyed by VBA. The `Terminate` event in the class module ensures the temporary file is deleted—the calling procedure does not need to know about any clean-up routines. If `CreateFile` is called twice, only the last temporary file is deleted. A new instance of the class should be created for each temporary file required.

You can force an error by amending `TestCTempFile` to specify a nonexistent directory for the temporary file:

```
Sub TestCTempFile()
    Dim oTempFile As New CTempFile

    'Tell the class to use a non-existent path
    oTempFile.Path = "C:\NoSuchPath"

    If oTempFile.CreateFile = "" Then
        MsgBox "An error occurred while creating the temporary file:" & _
            Chr(10) & oTempFile.ErrorText
    Else
        MsgBox "Temporary file " & oTempFile.FullName & " created"
    End If
End Sub
```

This time, you get a meaningful error message (see Figure 27-1).

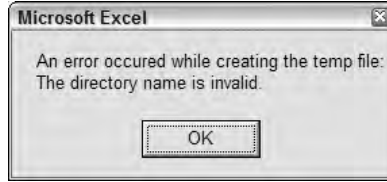


Figure 27-1

Some Example Classes

This section provides a number of common API calls to include in your projects. Note that in each case, the function and constant definitions must be put in the `Declarations` section at the top of a module.

A High-Resolution Timer Class

When testing your code, it is a good idea to time the various routines to identify and eliminate any bottlenecks. VBA includes two functions that can be used as timers:

- ❑ The `Now` function returns the current time and has a resolution of about one second.
- ❑ The `Timer` function returns the number of milliseconds since midnight, with a resolution of approximately 10 milliseconds.

Neither of these are accurate enough to time VBA routines, unless the routine is repeated many times.

Modern PCs include a high-resolution timer, which updates many thousands of times per second, accessible through API calls. You can wrap these calls in a class module to provide easy access to a high-resolution timer.

Class Module `CHighResTimer`

Note that the API Viewer shows these definitions using the `LARGE_INTEGER` data type, but they are defined as `Currency`:

```
Option Explicit

'How many times per second is the counter updated?
Private Declare Function QueryFrequency Lib "kernel32" _
    Alias "QueryPerformanceFrequency" ( _
        lpFrequency As Currency) As Long

'What is the counter's value
Private Declare Function QueryCounter Lib "kernel32" _
    Alias "QueryPerformanceCounter" ( _
        lpPerformanceCount As Currency) As Long
```

The `LARGE_INTEGER` is a 64-bit data type, usually made up of two `Long` types. The VBA `Currency` data type also uses 64 bits to store the number, so you can use it in place of a `LARGE_INTEGER`. The only differences are that the `Currency` data type is scaled down by a factor of 10,000, and that VBA can perform standard math operations with `Currency` variables:

```
'Variables to store the counter information
Dim mcyFrequency As Currency
Dim mcyOverhead As Currency
Dim mcyStarted As Currency
Dim mcyStopped As Currency
```

The API call itself takes a small amount of time to complete. For accurate timings, you should take this delay into account. You find this delay and the counter's frequency in the class's `Initialize` routine:

```
'When first initialized, determine the overhead incurred when retrieving the
'high-performance counter value
Private Sub Class_Initialize()
    Dim cyCount1 As Currency, cyCount2 As Currency

    'Get the counter frequency
    QueryFrequency mcyFrequency

    'Call the hi-res counter twice, to check how long it takes
    QueryCounter cyCount1
    QueryCounter cyCount2

    'Store the call overhead
    mcyOverhead = cyCount2 - cyCount1
End Sub

Public Sub StartTimer()
    'Get the time that you started
    QueryCounter mcyStarted
End Sub

Public Sub StopTimer()
    'Get the time that you stopped
    QueryCounter mcyStopped
End Sub

Public Property Get Elapsed() As Double
    Dim cyTimer As Currency

    'Have you stopped or not?
    If mcyStopped = 0 Then
        QueryCounter cyTimer
    Else
        cyTimer = mcyStopped
    End If

    'If you have a frequency, return the duration, in seconds
    If mcyFrequency > 0 Then
        Elapsed = (cyTimer - mcyStarted - mcyOverhead) / mcyFrequency
    End If
End Property
```

Chapter 27: Programming with the Windows API

When you calculate the elapsed time, both the timer and the frequency contain values that are a factor of 10,000 too small. Because the numbers are divided, the factors cancel out to give a result in seconds.

The High-Resolution Timer class can be used in a calling routine like this:

```
Sub TestCHighResTimer()  
    Dim i As Long  
    Dim oTimer As New CHighResTimer  
  
    oTimer.StartTimer  
  
    For i = 1 To 100000  
    Next i  
    oTimer.StopTimer  
  
    Debug.Print "100,000 iterations took " & oTimer.Elapsed & " seconds"  
End Sub
```

Freeze a UserForm

When working with UserForms, the display may be updated whenever a change is made to the form, such as adding an item to a ListBox, or enabling or disabling controls. `Application.ScreenUpdating` has no effect on UserForms; this `CFreezeForm` class provides a useful equivalent:

```
Option Explicit  
  
'Find a window  
Private Declare Function FindWindow Lib "user32" _  
    Alias "FindWindowA" ( _  
    ByVal lpClassName As String, _  
    ByVal lpWindowName As String) As Long  
  
'Freeze the window to prevent continuous redraws  
Private Declare Function LockWindowUpdate Lib "user32" ( _  
    ByVal hwndLock As Long) As Long  
  
Public Sub Freeze(oForm As UserForm)  
    Dim hWnd As Long  
  
    'Get a handle to the UserForm window,  
    'using the class name appropriate for the XL version  
    If Val(Application.Version) >= 9 Then  
        hWnd = FindWindow("ThunderDFrame", oForm.Caption)  
    Else  
        hWnd = FindWindow("ThunderXFrame", oForm.Caption)  
    End If  
  
    'If you got a handle, freeze the window  
    If hWnd > 0 Then LockWindowUpdate hWnd  
End Sub  
  
'Allow the calling routine to unfreeze the UserForm
```

```
Public Sub UnFreeze()  
    LockWindowUpdate 0  
End Sub  
  
'If they forget to unfreeze the form, do it at the end  
'of the calling routine (when you go out of scope)  
Private Sub Class_Terminate()  
    UnFreeze  
End Sub
```

To demonstrate this in action, create a new UserForm and add a ListBox and a command button. Add the following code for the command button's Click event:

```
Private Sub CommandButton1_Click()  
    Dim i As Integer  
  
    For i = 1 To 1000  
        ListBox1.AddItem "Item " & i  
        DoEvents  
    Next i  
End Sub
```

The `DoEvents` line forces the UserForm to redraw, to demonstrate the problem. In more complicated routines, the UserForm may redraw itself without using `DoEvents`. To prevent the redrawing, you can modify the routine to use the `CFreezeForm` class as follows:

```
Private Sub CommandButton1_Click()  
    Dim obFF As New CFreezeForm, i As Integer  
    'Freeze the UserForm  
    obFF.Freeze Me  
  
    For i = 1 To 1000  
        ListBox1.AddItem "Item " & i  
        DoEvents  
    Next i  
End Sub
```

This is much easier than including several API calls in every function. The class's `Terminate` event ensures that the UserForm is unfrozen when the `obFF` object variable goes out of scope. Freezing a UserForm in this way can result in a dramatic performance improvement. For example, the non-frozen version takes approximately 3.5 seconds to fill the ListBox, while the frozen version of the routine takes approximately 1.2 seconds. This should be weighted against user interaction; they may think the computer has frozen if they see no activity for some time. Consider using `Application.StatusBar` to keep them informed of progress in that case.

A System Info Class

The classic use of a class module and API functions is to provide all the information about the Windows environment that you cannot get at using VBA. The following properties are typical components of such a `CSysInfo` class.

The declarations for the constants and API functions used in these procedures must all be placed together at the top of the class module. For clarity, they are shown here with the corresponding routines.

Obtaining the screen resolution (in pixels):

```
Option Explicit

Private Const SM_CYSCREEN As Long = 1    'Screen height
Private Const SM_CXSCREEN As Long = 0    'Screen width

'API Call to retrieve system information
Private Declare Function GetSystemMetrics Lib "user32" ( _
    ByVal nIndex As Long) As Long
'Retrieve the screen height, in pixels
Public Property Get ScreenHeight() As Long
    ScreenHeight = GetSystemMetrics(SM_CYSCREEN)
End Property

'Retrieve the screen width, in pixels
Public Property Get ScreenWidth() As Long
    ScreenWidth = GetSystemMetrics(SM_CXSCREEN)
End Property
```

Obtaining the color depth (in bits):

```
Private Declare Function GetDC Lib "user32" ( _
    ByVal hwnd As Long) As Long

Private Declare Function GetDeviceCaps Lib "Gdi32" ( _
    ByVal hDC As Long, _
    ByVal nIndex As Long) As Long

Private Declare Function ReleaseDC Lib "user32" ( _
    ByVal hwnd As Long, _
    ByVal hDC As Long) As Long

Private Const BITSPIXEL = 12

Public Property Get ColorDepth() As Integer
    Dim hDC As Long

    'A device context is the canvas on which a window is drawn
    hDC = GetDC(0)
    ColorDepth = GetDeviceCaps(hDC, BITSPIXEL)
    ReleaseDC 0, hDC
End Property
```

Obtaining the width of a pixel in UserForm coordinates (where the API declarations are the same as those for the previous `ColorDepth` and repeated here for clarity):

```
Private Declare Function GetDC Lib "user32" ( _
    ByVal hwnd As Long) As Long

Private Declare Function GetDeviceCaps Lib "Gdi32" ( _
    ByVal hDC As Long, _
    ByVal nIndex As Long) As Long

Private Declare Function ReleaseDC Lib "user32" ( _
    ByVal hwnd As Long, _
    ByVal hDC As Long) As Long
```

```
Private Const LOGPIXELSX = 88

' The width of a pixel in Excel's UserForm coordinates
Public Property Get PointsPerPixel() As Double
    Dim hDC As Long

    hDC = GetDC(0)

    ' A point is defined as 1/72 of an inch and LOGPIXELSX returns
    ' the number of pixels per logical inch, so divide them to give
    ' the width of a pixel in Excel's UserForm coordinates
    PointsPerPixel = 72 / GetDeviceCaps(hDC, LOGPIXELSX)

    ReleaseDC 0, hDC
End Property
```

Reading the user's login ID:

```
Private Declare Function GetUserName Lib "advapi32.dll" _
    Alias "GetUserNameA" ( _
    ByVal lpBuffer As String, _
    ByRef nSize As Long) As Long

Public Property Get UserName() As String
    Dim sBuff As String * 255, lAPIResult As Long
    Dim lBuffLen As Long

    lBuffLen = 255

    ' The second parameter, lBuffLen is both In and Out.
    ' On the way in, it tells the function how big the string buffer is
    ' On the way out, it tells us how long the user name is (including
    ' a terminating Chr(0))
    lAPIResult = GetUserName(sBuff, lBuffLen)

    ' If you got something, return the text of the user name
    If lBuffLen > 0 Then UserName = Left(sBuff, lBuffLen - 1)
End Property
```

Reading the computer's name:

```
Private Declare Function GetComputerName Lib "kernel32" _
    Alias "GetComputerNameA" ( _
    ByVal lbuffer As String, _
    nsize As Long) As Long

Public Property Get ComputerName() As String
    Dim sBuff As String * 255, lAPIResult As Long
    Dim lBuffLen As Long

    lBuffLen = 255
    lAPIResult = GetComputerName(sBuff, lBuffLen)
    If lBuffLen > 0 Then ComputerName = Left(sBuff, lBuffLen)
End Property
```

These can be tested by using the following routine (in a standard module):

```
Sub TestCSysInfo()
    Dim oSysInfo As New CSysInfo
    Debug.Print "Screen Height = " & oSysInfo.ScreenHeight
    Debug.Print "Screen Width = " & oSysInfo.ScreenWidth
    Debug.Print "Color Depth = " & oSysInfo.ColorDepth
    Debug.Print "One pixel = " & oSysInfo.PointsPerPixel & " points"
    Debug.Print "User name = " & oSysInfo.UserName
    Debug.Print "Computer name = " & oSysInfo.ComputerName
End Sub
```

Modifying UserForm Styles

UserForms in Excel do not provide any built-in mechanism for modifying their appearance. Your only choice is a simple popup dialog with a caption and an X button to close the form, though you can choose to show it modally or non-modally.

Using API calls, you can modify the UserForm's window to do any combination of the following:

- Switching between modal and non-modal while the form is showing
- Making the form resizable
- Showing or hiding the form's caption and title bar
- Showing a small title bar, like those on a floating toolbar
- Showing a custom icon on the form
- Showing an icon in the task bar for the form
- Removing the X button to close the form
- Adding standard maximize and minimize buttons

You can find example workbooks demonstrating all these choices at www.wrox.com, with the key parts of the code explained in the following sections.

Window Styles

The appearance and behavior of a window is primarily controlled by its *style* and *extended style* properties. These styles are both `Long` values, in which each bit of the value controls a specific aspect of the window's appearance—either on or off. You can change the window's appearance using the following process:

1. Use `FindWindow` to get the UserForm's window handle.
2. Read its style using the `GetWindowLong` function.
3. Toggle one or more of the style bits.
4. Set the window to use this modified style using the `SetWindowLong` function.
5. For some changes, tell the window to redraw itself using the `ShowWindow` function.

Some of the main constants for each bit of the basic window style are (no need to type these in):

```
'Style to add a titlebar
Private Const WS_CAPTION As Long = &HC0000

'Style to add a system menu
Private Const WS_SYSMENU As Long = &H8000

'Style to add a sizable frame
Private Const WS_THICKFRAME As Long = &H4000

'Style to add a Minimize box on the title bar
Private Const WS_MINIMIZEBOX As Long = &H2000

'Style to add a Maximize box to the title bar
Private Const WS_MAXIMIZEBOX As Long = &H1000

'Cleared to show a task bar icon
Private Const WS_POPUP As Long = &H80000000

'Cleared to show a task bar icon
Private Const WS_VISIBLE As Long = &H10000000
```

And some of those for the extended window style are:

```
'Controls if the window has an icon
Private Const WS_EX_DLGMODALFRAME As Long = &H1

'Application Window: shown on taskbar
Private Const WS_EX_APPWINDOW As Long = &H40000

'Tool Window: small titlebar
Private Const WS_EX_TOOLWINDOW As Long = &H80
```

Note that this is only a subset of all the possible window style bits. See the MSDN documentation for [Window Styles for the full list](http://msdn.microsoft.com/library/en-us/winui/winui/WindowsUserInterface/Windowing/Windows/WindowReference/WindowStyles.asp) (<http://msdn.microsoft.com/library/en-us/winui/winui/WindowsUserInterface/Windowing/Windows/WindowReference/WindowStyles.asp>) and the API Viewer for their values.

Chapter 27: Programming with the Windows API

The following example uses the preceding process to remove a UserForm's close button, and it can be found in the `NoCloseButton.xlsm` example in the code download at www.wrox.com:

```
'Find the UserForm's Window
Private Declare Function FindWindow Lib "user32" _
    Alias "FindWindowA" ( _
        ByVal lpClassName As String, _
        ByVal lpWindowName As String) As Long

'Get the current window style
Private Declare Function GetWindowLong Lib "user32" _
    Alias "GetWindowLongA" ( _
        ByVal hWnd As Long, _
        ByVal nIndex As Long) As Long

'Set the new window style
Private Declare Function SetWindowLong Lib "user32" _
    Alias "SetWindowLongA" ( _
        ByVal hWnd As Long, _
        ByVal nIndex As Long, _
        ByVal dwNewLong As Long) As Long

Const GWL_STYLE = -16           'The standard style
Const WS_SYSMENU = &H80000     'The system menu style bit

Private Sub UserForm_Initialize()
    Dim hWnd As Long, lStyle As Long

    '1. Find the UserForm's window handle
    If Val(Application.Version) >= 9 Then
        hWnd = FindWindow("ThunderDFrame", Me.Caption)
    Else
        hWnd = FindWindow("ThunderXFrame", Me.Caption)
    End If

    '2. Get the current window style
    lStyle = GetWindowLong(hWnd, GWL_STYLE)

    '3. Toggle the SysMenu bit, to turn off the system menu
    lStyle = (lStyle And Not WS_SYSMENU)

    '4. Set the window to use the new style
    SetWindowLong hWnd, GWL_STYLE, lStyle
End Sub
```

The CFormChanger Class

As mentioned previously in this chapter, API calls are much easier to use when they are encapsulated within a class module. The `CFormChanger` class included in the `FormFun.xlsm` file at www.wrox.com repeats the previous code snippet for all the windows style bits mentioned in the previous section, presenting them as the following properties of the class:

- Modal
- Sizeable
- ShowCaption
- SmallCaption
- ShowIcon
- IconPath (to show a custom icon)
- ShowCloseBtn
- ShowMaximizeBtn
- ShowMinimizeBtn
- ShowSysMenu
- ShowTaskBarIcon

To use the class on your own forms, copy the entire class module into your project and call it from your form's `Activate` event, as in the following example. You can find this example in the `ToolBarForm.xlsm` workbook at www.wrox.com:

```
Private Sub UserForm_Activate()  
  
    Dim oChanger As CFormChanger  
  
    'Create a new instance of the CFormChanger class  
    Set oChanger = New CFormChanger  
  
    'Set the form changer's properties  
    oChanger.SmallCaption = True  
    oChanger.Sizeable = True  
  
    'Tell the changer which form to apply the style changes to.  
    'Also acts as the trigger for applying them  
    Set oChanger.Form = Me  
  
End Sub
```

Resizable UserForms

In Office XP, Microsoft made the `File ⇨ Open` and `File ⇨ Save As` dialogs resizable. They remember their position and size between sessions, greatly improving their usability. Using the same API calls shown in the previous section and a class module to do all the hard work, you can give your users the same experience when interacting with your UserForms.

One of the curiosities of the `UserForm` object is that it has a `Resize` event, but it doesn't have a property to specify whether or not it is resizable — the `Resize` event only fires when the form is first displayed. As shown in the previous example, you can provide your own `Sizeable` property by toggling the

Chapter 27: Programming with the Windows API

WS_THICKFRAME window style bit; when you do this, the `UserForm_Resize` event comes to life, triggered every time the users change the size of the form (though not when they move it around the screen). You can respond to this event by changing the size or position of all the controls on the form, such that they make the best use of the `UserForm`'s new size.

There are two approaches that can be used to change the size or position of all the controls on the form: absolute and relative.

Absolute Changes

Using an absolute approach, code has to be written to set the size and position of all the controls on the form, relative to the form's new dimensions and to each other. Consider a very simple form showing just a `ListBox` and an `OK` button.

The code to resize and reposition the two controls using absolute methods is as follows:

```
Private Sub UserForm_Resize()  
  
    'Handle the form's resizing by specifying the new size and position  
    'of all the controls  
  
    'Use a standard gap of 6 points between controls  
    Const dGap = 6  
  
    'Ignore errors caused by controls getting too small  
    On Error Resume Next  
  
    'The OK button is in the middle ...  
    btnOK.Left = (Me.InsideWidth - btnOK.Width) / 2  
  
    '... and at the bottom of the form, with a standard gap below it  
    btnOK.Top = Me.InsideHeight - dGap - btnOK.Height  
  
    'The list's width is the form's width,  
    'minus two gaps for the left and right edges  
    lstItems.Width = Me.InsideWidth - dGap * 2  
  
    'The list should fill the space between the top of the form  
    'and the top of the OK button, minus a gap top and bottom  
    lstItems.Height = btnOK.Top - dGap * 2  
  
End Sub
```

It works, but has a few major problems:

- ❑ Specific code has to be written for every control that changes size or position, which can be a daunting task for more complex forms. See the resize code in `FormFun.xlsx` for an example of this.
- ❑ The size and position of controls are often dependent on the size and position of other controls (such as the bottom of the `ListBox` being four pixels above the top of the `OK` button).

- ❑ If you modify the appearance of the form by adding or moving controls, you have to make corresponding changes to the resize code. For example, to add a Cancel button alongside the OK button, you have to add code to handle the Cancel button's repositioning and also change the code for the OK button.
- ❑ There is no opportunity for code reuse.

Relative Changes

Using a relative approach, information is added to each control to specify by how much that control's size and position should change as the UserForm's size changes. In the same dialog, the two controls have the following relative changes:

- ❑ The OK button should move down by the full change in the form's height (to keep it at the bottom).
- ❑ The OK button should move across by half the change in the form's width (to keep it in the middle).
- ❑ The ListBox's height and width should change by the full change in the form's height and width.

These statements can be encoded into a single string to state the percentage change for each control's `Top`, `Left`, `Height`, and `Width` properties, and can be stored against the control. A convenient place to store it is the control's `Tag` property, which allows the resizing behavior of the control to be set at design time. Using the letters `T`, `L`, `H`, and `W` for the four properties, and a decimal for the percentage change if not 100%, gives the `Tag` properties of `HW` for the list box and `TL0.5` for the OK button, as shown in Figure 27-2.

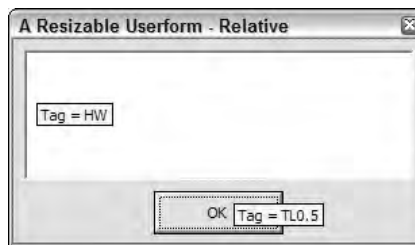


Figure 27-2

When the `UserForm_Resize` event fires, the code can calculate the change in the form's height and width and iterate through all the controls, adjusting their `Top`, `Left`, `Height`, and `Width` as specified by their `Tag` properties. The `CFormResizer` class to do this is shown in the next section.

There are a number of benefits to this approach:

- ❑ The resize behavior of each control is set at design time, while the form is being viewed, just like all the other properties of the control.
- ❑ The change in size and position of each control is independent of any other control.

- ❑ Controls can be added, moved, or deleted without having to modify the `Resize` code or change other controls' resize behavior.
- ❑ The resize code can treat every control in exactly the same way; hence, every `UserForm` uses exactly the same `Resize` code, which can be encapsulated in a separate class module.

The `CFormResizer` Class

By encapsulating all the resize code in a separate class module, any `UserForm` can be made resizable by adding just six lines of code to instantiate and call into the class, and setting the resize behavior for each control in its `Tag` property.

The `CFormResizer` class provides the following functionality:

- ❑ Sets the form to be resizable.
- ❑ Sets the initial size and position of the form, if it has been shown before.
- ❑ Resizes and repositions all the controls on the form, according to their `Tag` resizing string.
- ❑ Stores the form's size and position in the registry, for use when the same form is shown again.
- ❑ Allows the calling code to specify a key name for storing the form dimensions in the registry.
- ❑ Prevents a form being resized in either direction if none of the controls are set to respond to changes in height or width.
- ❑ Stops resizing when any control is moved to the left or top edge of the form, or when any control is reduced to zero height or width.

The code for the `CFormResizer` class is as follows, with comments in the code to explain each section. It is available for download in the `FormResizer.xlsx` workbook at www.wrox.com:

```
Option Explicit

'Find the UserForm's window handle
Private Declare Function FindWindow Lib "user32" _
    Alias "FindWindowA" ( _
        ByVal lpClassName As String, _
        ByVal lpWindowName As String) As Long

'Get the UserForm's window style
Private Declare Function GetWindowLong Lib "user32" _
    Alias "GetWindowLongA" ( _
        ByVal hWnd As Long, _
        ByVal nIndex As Long) As Long

'Set the UserForm's window style
Private Declare Function SetWindowLong Lib "user32" _
    Alias "SetWindowLongA" ( _
        ByVal hWnd As Long, _
        ByVal nIndex As Long, _
        ByVal dwNewLong As Long) As Long

'The offset of a window's style
```

```

Private Const GWL_STYLE As Long = (-16)

'Style to add a sizable frame
Private Const WS_THICKFRAME As Long = &H40000

Dim moForm As Object
Dim mhWndForm As Long
Dim mdWidth As Double
Dim mdHeight As Double
Dim msRegKey As String

'Default for the registry key to store the dimensions
Private Sub Class_Initialize()
    msRegKey = "Excel 2007 Prog Ref"
End Sub

'Properties to identify where in the registry to store the UserForm
'position information
Public Property Let RegistryKey(sNew As String)
    msRegKey = sNew
End Property

Public Property Get RegistryKey() As String
    RegistryKey = msRegKey
End Property

'We're told which form to handle the resizing for,
'set in the UserForm_Initialize event.
'Make the form resizable and set its size and position
Public Property Set Form(oNew As Object)

    Dim sSizes As String, vaSizes As Variant
    Dim iStyle As Long

    'Remember the form for later
    Set moForm = oNew

    'Get the UserForm's window handle
    If Val(Application.Version) < 9 Then
        'XL97
        mhWndForm = FindWindow("ThunderXFrame", moForm.Caption)
    Else
        'XL2000 and 2002
        mhWndForm = FindWindow("ThunderDFrame", moForm.Caption)
    End If

    'Make the form resizable
    iStyle = GetWindowLong(mhWndForm, GWL_STYLE)
    iStyle = iStyle Or WS_THICKFRAME
    SetWindowLong mhWndForm, GWL_STYLE, iStyle

    'Read its dimensions from the registry (if there)
    'The string has the form of "<Top>;<Left>;<Height>;<Width>"
    sSizes = GetSetting(msRegKey, "Forms", moForm.Name, "")

```

```
'Remember the current size for use in the Resize routine
mdWidth = moForm.Width
mdHeight = moForm.Height

If sSizes <> "" Then
    'If we got a dimension string, split it into its parts
    vaSizes = Split(sSizes, ";")

    'Make sure we got 4 elements!
    ReDim Preserve vaSizes(0 To 3)

    'Set the form's size and position
    moForm.Top = Val(vaSizes(0))
    moForm.Left = Val(vaSizes(1))
    moForm.Height = Val(vaSizes(2))
    moForm.Width = Val(vaSizes(3))

    'Set to manual startup position
    moForm.StartupPosition = 0
End If

End Property

'Called from the User_Form resize event, also triggered when we change
'the size ourself.

'This is the routine that performs the resizing, by checking each control's
'Tag property, and moving/sizing it accordingly.
Public Sub FormResize()

    Dim dWidthAdj As Double, dHeightAdj As Double
    Dim bSomeWidthChange As Boolean
    Dim bSomeHeightChange As Boolean
    Dim sTag As String, sSize As String
    Dim oCtl As MSForms.Control

    Static bResizing As Boolean

    'Resizing can be triggered from within this routine,
    'so use a flag to prevent recursion
    If bResizing Then Exit Sub
    bResizing = True

    'Calculate the change in height and width
    dHeightAdj = moForm.Height - mdHeight
    dWidthAdj = moForm.Width - mdWidth

    'Check if we can perform the adjustment
    '(i.e. widths and heights can't be negative)
    For Each oCtl In moForm.Controls

        'Read the control's Tag property, which contains the resizing info
        sTag = UCase(oCtl.Tag)

        'If we're changing the Top, check that it won't move off the top
```



```

'of the form
If InStr(1, sTag, "T", vbBinaryCompare) Then
    If oCtl.Top + dHeightAdj * ResizeFactor(sTag, "T") <= 0 Then
        moForm.Height = mdHeight
    End If

    bSomeHeightChange = True
End If

'If we're changing the Left, check that it won't move off the
'left of the form
If InStr(1, sTag, "L", vbBinaryCompare) Then
    If oCtl.Left + dWidthAdj * ResizeFactor(sTag, "L") <= 0 Then
        moForm.Width = mdWidth
    End If

    bSomeWidthChange = True
End If

'If we're changing the Height, check that it won't go negative
If InStr(1, sTag, "H", vbBinaryCompare) Then
    If oCtl.Height + dHeightAdj * ResizeFactor(sTag, "H") <= 0 Then
        moForm.Height = mdHeight
    End If

    bSomeHeightChange = True
End If

'If we're changing the Width, check that it won't go negative
If InStr(1, sTag, "W", vbBinaryCompare) Then
    If oCtl.Width + dWidthAdj * ResizeFactor(sTag, "W") <= 0 Then
        moForm.Width = mdWidth
    End If

    bSomeWidthChange = True
End If
Next          'Control

'If none of the controls move or size,
'don't allow the form to resize in that direction
If Not bSomeHeightChange Then moForm.Height = mdHeight
If Not bSomeWidthChange Then moForm.Width = mdWidth

'Recalculate the height and width changes,
'in case the previous checks reset them
dHeightAdj = moForm.Height - mdHeight
dWidthAdj = moForm.Width - mdWidth

'Loop through all the controls on the form,
'adjusting their position and size
For Each oCtl In moForm.Controls
    With oCtl
        sTag = UCase(.Tag)

        'Changing the Top

```

```
    If InStr(1, sTag, "T", vbBinaryCompare) Then
        .Top = .Top + dHeightAdj * ResizeFactor(sTag, "T")
    End If

    'Changing the Left
    If InStr(1, sTag, "L", vbBinaryCompare) Then
        .Left = .Left + dWidthAdj * ResizeFactor(sTag, "L")
    End If

    'Changing the Height
    If InStr(1, sTag, "H", vbBinaryCompare) Then
        .Height = .Height + dHeightAdj * ResizeFactor(sTag, "H")
    End If

    'Changing the Width
    If InStr(1, sTag, "W", vbBinaryCompare) Then
        .Width = .Width + dWidthAdj * ResizeFactor(sTag, "W")
    End If
End With
Next 'Control

'Remember the new dimensions of the form for next time
mdWidth = moForm.Width
mdHeight = moForm.Height

'Store the size and position in the registry
With moForm
    SaveSetting msRegKey, "Forms", .Name, Str(.Top) & ";" & _
        Str(.Left) & ";" & Str(.Height) & ";" & Str(.Width)
End With

'Reset the recursion flag, now that we're done
bResizing = False

End Sub

'Function to locate a property letter (T, L, H or W) in the Tag string
'and return the resizing factor for it
Private Function ResizeFactor(sTag As String, sChange As String)

    Dim i As Integer, d As Double

    'Locate the property letter in the tag string
    i = InStr(1, sTag, sChange, vbBinaryCompare)

    'If we found it...
    If i > 0 Then

        '... read the number following it
        d = Val(Mid$(sTag, i + 1))

        'If there was no number, use a factor of 100%
        If d = 0 Then d = 1
    End If
```

```
'Return the factor
ResizeFactor = d

End Function
```

The code to use the `CFormResizer` class in a `UserForm`'s code module is as follows:

```
'Declare an object for our CFormResizer class to handle
'resizing for this form
Dim moResizer As CFormResizer

'The Resizer class is set up in the UserForm_Initialize event
Private Sub UserForm_Initialize()

    'Create the instance of the class
    Set moResizer = New CFormResizer

    'Tell it where to store the form dimensions
    moResizer.RegistryKey = "Excel 2007 Prog Ref"

    'Tell it which form it's handling
    Set moResizer.Form = Me

End Sub

'When the form is resized, the UserForm_Resize event is raised,
'which we just pass on to the Resizer class
Private Sub UserForm_Resize()
    moResizer.FormResize
End Sub

'The OK button unloads the form
Private Sub btnOK_Click()
    Unload Me
End Sub

'The QueryClose event is called whenever the form is closed.
'We call the FormResize method one last time, to store the form's
'final size and position in the registry
Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)
    moResizer.FormResize
End Sub
```

There are a few points to remember when using this approach in your own `UserForms`:

- ❑ The resizer works by changing the control's `Top`, `Left`, `Height`, and `Width` properties in response to changes in the `UserForm` size, according to the control's resizing information.
- ❑ The control's resizing information is set in its `Tag` property, using the letters `T`, `L`, `H`, or `W` followed by a number specifying the resizing factor (if not 100%).
- ❑ The resizing factors must be in U.S. format, using a period as the decimal separator.
- ❑ If there are no controls that have `T` or `H` in their `Tag` strings, the form will not be allowed to resize vertically.

- ❑ If there are no controls that have `L` or `W` in their `Tag` strings, the form will not be allowed to resize horizontally.
- ❑ The smallest size for the form is set by the first control to be moved to the top or left edge, or to have a zero width or height.
- ❑ This can be used to set a minimum size for the form by using a hidden label with a `Tag` of `HW`, where the size of the label equals the amount that the form can be reduced in size. If the label is set to zero height and width to start with, the `UserForm` can only be enlarged from its design-time size.
- ❑ List boxes must have their `IntegralHeight` property set to `False` for this to work, but due to an old bug, they may not fully display the very last item in the list. As a workaround, add a blank entry as the last item in the list, and code to ignore it if it gets selected.

Summary

The functions defined in the Windows API provide a valuable and powerful extension to the VBA developer's tool set. The API Viewer provides the VBA definitions for most of the core functions. The definitions for the remaining functions can be converted from the C-style versions shown in the online MSDN library.

Class modules enable the user to encapsulate both the API definitions and their use into simple chunks of functionality that are easy to use and reuse in VBA applications. A number of example classes and routines have been provided in this chapter to get you started using the Windows API functions within your applications, including:

- ❑ Creating a `TEMP` file
- ❑ A high-resolution timer
- ❑ Freezing a `UserForm`
- ❑ Getting system information
- ❑ Modifying a `UserForm`'s appearance
- ❑ Making `UserForms` resizable, with a minimum of code in the form



Excel 2007 Object Model

Most of the objects in the Excel object model have objects with associated collections. The collection object is usually the plural form of the associated object. For example, the `Worksheets` collection holds a collection of `Worksheet` objects. For simplicity, each object and associated collection will be grouped together under the same heading.

Common Properties with Collections and Associated Objects

In most cases, the purpose of the collection object is only to hold a collection of the same objects. The common properties and methods of the collection objects are listed in the following section. Only unique properties, methods, or events are mentioned in each object section.

Common Collection Properties

Name	Returns	Description
<code>Application</code>	<code>Application</code>	Read-only. Returns a reference to the owning application of the current object — Excel, in this case
<code>Count</code>	<code>Long</code>	Read-only. Returns the number of objects in the collection
<code>Creator</code>	<code>Long</code>	Read-only. Returns a <code>Long</code> number that describes whether or not the object was created in Excel
<code>Parent</code>	<code>Object</code>	The <code>Parent</code> object is the owning object of the collection object. For example, <code>Workbooks.Parent</code> returns a reference to the <code>Application</code> object

Common Collection Methods

Name	Returns	Parameters	Description
Item	Single Object	Index as Variant	Returns the object from the collection with the Index value specified by the Index parameter. The Index value may also specify a unique string key describing one of the objects in the collection

Common Object Properties

Objects also have some common properties. To avoid redundancy, the common properties and methods of all objects are listed next. They will be mentioned in each object description as existing, but are only defined here.

Name	Returns	Description
Application	Application	Read-only. Returns a reference to the owning application of the current object — Excel, in this case
Creator	Long	Read-only. Returns a Long number that describes whether or not the object was created in Excel
Parent	Object	Read-only. The owning object of the current object. For example, Characters. Parent may return a reference to a Range object, since a Range object is one of the possible owners of a Characters object

Excel Objects and Their Properties, Methods, and Events

The objects are listed in alphabetical order. Each object has a general description of the object and possible parent objects. This is followed by a table format of each of the object's properties, methods, and events.

AboveAverage Object

The AboveAverage object controls the attributes and specifications of a conditional formatting rule that evaluates the values in a given scope or range against the average of that scope or range.

AboveAverage Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

AboveAverage Properties

Name	Returns	Description
<code>AboveBelow</code>	<code>xlAboveBelow</code>	Set/Get the conditional formatting rule looking for values above or below the average. Use the <code>xlAboveBelow</code> constants
<code>AppliesTo</code>	<code>Range</code>	Set/Get the range that is affected by the formatting rule
<code>Borders</code>	<code>Borders</code>	Read-only. Returns a collection that specifies the cell borders for the formatting condition
<code>CalcFor</code>	<code>xlCalcFor</code>	Set/Get the scope of data to be evaluated in a PivotTable report. Use the <code>xlCalcFor</code> constant
<code>Font</code>	<code>Font</code>	Read-only. Specifies the font formatting attributes for the conditional formatting rule
<code>FormatRow</code>	<code>Boolean</code>	Set/Get the Boolean value specifying if the entire Excel table row should be formatted. The default value is <code>False</code>
<code>Interior</code>	<code>Interior</code>	Read-only. Specifies the interior formatting attributes for the conditional formatting rule
<code>NumberFormat</code>	<code>Variant</code>	Set/Get the number format applied to a cell if the conditional formatting rule evaluates to <code>true</code>
<code>Priority</code>	<code>Long</code>	Set/Get the priority value of a conditional formatting rule, determining the order of evaluation when other rules are in effect
<code>PTCondition</code>	<code>Boolean</code>	Read-only. Indicates whether the formatting rule is applied to a PivotTable chart
<code>ScopeType</code> <code>Condition</code> <code>Scope</code>	<code>xlPivot</code>	Set/Get the scope of the formatting rule when applied to a PivotTable chart. Use the <code>xlPivotConditionScope</code> constants
<code>StopifTrue</code>	<code>Boolean</code>	Set/Get a Boolean value that determines if additional formatting rules should be applied if the current rule evaluates to <code>True</code> . The default value is <code>True</code>
<code>Type</code> <code>ConditionType</code>	<code>xlFormat</code>	Read-only. Returns an <code>xlFormatConditionType</code> constant that specifies the type of conditional formatting being applied. This object will always return a value of 12 since it corresponds to the <code>XlAboveAverageCondition</code>

AboveAverage Methods

AboveAverage Methods

Name	Returns	Parameters	Description
Delete			Deletes the object
ModifyAppliesToRange		Range As Range	Sets the range for which the formatting rule will be applied
SetFirstPriority			Sets the priority value for the formatting rule so that it is evaluated before all other rules on the worksheet
SetLastPriority			Sets the priority value for the formatting rule so that it is evaluated after all other rules on the worksheet

AboveAverage Object Example

```
Sub CreateBelowAverageCondition()  
Dim oFormatCondition As AboveAverage  
  
'Add a new Formatting rule  
Set oFormatCondition = Range("F6:F16").FormatConditions.AddAboveAverage  
  
'Highlight all values that are below the average for the range.  
oFormatCondition.AboveBelow = xlBelowAverage  
oFormatCondition.Interior.Color = 7039480  
  
End Sub
```

Action Object and the Actions Collection

The `Action` object represents an action to be executed in a PivotTable or sheet data, while the `Actions` collection contains all `Action` objects for a given series. The `Actions` collection has properties outside the common properties of `Application`, `Count`, `Item`, and `Parent`.

Action Properties

Name	Returns	Description
Caption	String	Read-only. Returns the caption assigned to a given <code>Action</code> object
Content	String	Read-only. Returns the contents associated with a given <code>Action</code> object
Coordinate	String	Read-only. Returns the coordinate property of a given <code>Action</code> object

Name	Returns	Description
Name	String	Read-only. Returns the Name of a given Action object
Type	XLActionType	Read-only. Returns the action type for a given Action object, defined by an XLActionType constant

Add-In Object and the Addins Collection

The `Addins` collection holds all of the `Addin` objects available to Excel. The Add-In must be installed (`AddIn.Installed = True`) to be able to use it in the current session. Examples of available `Addin` objects in Excel include the Analysis Toolpack, the MS Query Add-In, and the Conditional Sum Wizard.

The `Add` method of the `Addins` collection can be used to add a new `Addin` to the collection. The `Add` method requires a `FileName` to be specified (usually with an `XLL` or `XLA` file extension). The `Count` property of the `Addins` collection returns the number of Add-Ins that are available for use by the current Excel session.

Add-In Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Add-In Properties

Name	Returns	Description
CLSID	String	Read-only. Returns a unique identifier for the Add-In
FullName	String	Read-only. Returns the full path and filename of the associated Add-In
Installed	Boolean	Set/Get whether the Add-In can be used in the current session
Name	String	Read-only. Returns the filename of the Add-In
Path	String	Read-only. Returns the full file path of the associated Add-In
progID	String	Read-only. Returns the programmatic identifier for object
Title	String	Read-only. This hidden property returns the string shown in the Add-In Manager dialog box

AddIn Object and the AddIns Collection Example

This example ensures that the Analysis Toolpack is installed:

```
Sub UsingAnalysisToolpack():
```

```
    Dim oAddin As AddIn
    'Make sure the Analysis Toolpack is installed
    For Each oAddin In AddIns
        If oAddin.Name = "analys32.xll" Then
            If oAddin.Installed = True Then
                MsgBox "True"
            Else
                MsgBox "False"
            End If
        End If
    Next
End Sub
```

Note that instead of looping through the `AddIns` collection, you could use the Add-In's title:

```
Sub UsingAnalysisToolpack()
    If AddIns("Analysis Toolpak").Installed = True Then
        MsgBox "True"
    Else
        MsgBox "False"
    End If
End Sub
```

Unfortunately, this approach may not work with a non-English User-Interface language, if the Add-In's title has been localized.

Adjustments Object

The `Adjustments` object holds a collection of numbers used to move the adjustment *handles* of the parent `Shape` object. Each `Shape` object can have up to eight different adjustments. Each specific adjustment handle can have one or two adjustments associated with it, depending on if it can be moved both horizontally and vertically (two) or in just one dimension. Adjustment values are between 0 and 1 and hence are percentage adjustments — the absolute magnitude of a 100% change is defined by the shape being adjusted.

Adjustments Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Adjustments Properties

Name	Returns	Description
Count	Long	Read-only. Returns the number of adjustment values associated with the parent <code>Shape</code> object
Item	Single	Parameters: <code>Index As Long</code> . Set/Get the adjustment value or values indicated by the <code>Index</code> parameter

Adjustments Object Example

This example draws a block arrow on the sheet, and then modifies the dimensions of the arrow head:

```
Sub AddArrow()
    Dim oShp As Shape

    'Add an arrow head to the sheet
    Set oShp = ActiveSheet.Shapes.AddShape( _
        msoShapeRightArrow, 10, 10, 100, 50)

    'Set the 'head' of the arrow to start 30% of the way across
    'and the 'shaft' to start 40% of the way down.
    oShp.Adjustments(1) = 0.3    'Left/right
    oShp.Adjustments(2) = 0.4    'Up/down
End Sub
```

AllowEditRange Object and the AllowEditRanges Collection

The `AllowEditRange` object represents a range of cells on a worksheet that can still be edited when protected. Each `AllowEditRange` object can have permissions set for any number of users on your network, and can have a separate password.

Be aware of the `Locked` property of the `Range` object when using this feature. When you unlock cells, then protect the worksheet, you are allowing any user access to those cells, regardless of the `AllowEditRange` objects. When each `AllowEditRange` object's cells are locked, any user can still edit them, unless you assign a password or add users and deny them permission without using a password.

The `AllowEditRanges` collection represents all `AllowEditRange` objects that can be edited on a protected worksheet. See the `AllowEditRange` object for more details.

AllowEditRanges Collection Properties

Name	Returns	Description
Count	Long	Read-only. Returns the number of <code>AllowEditRange</code> objects that are contained in the area
Item	<code>AllowEditRange</code>	Parameter: <code>Index As Variant</code> . Returns a single <code>AllowEditRange</code> object in the <code>AllowEditRanges</code> collection

AllowEditRanges Collection Methods

Name	Returns	Parameters	Description
Add	<code>AllowEditRange</code>	<code>Title As String</code> , <code>Range As Range</code> , <code>[Password] As Variant</code>	Adds an <code>AllowEditRange</code> object to the <code>AllowEditRanges</code> collection

AllowEditRange Properties

AllowEditRange Properties

Name	Returns	Description
Range	Range	Returns a subset of the ranges that can be edited on a protected worksheet
Title	String	Returns or sets the title of the web page when the document is saved as a web page
Users	UserAccess List	Returns the list of users who are allowed access to the protected range on a worksheet

AllowEditRange Methods

Name	Parameters	Description
ChangePassword	Password As String	Sets the password for a range that can be edited on a protected worksheet
Delete		Deletes the object
Unprotect	[Password]	Removes any protection from a sheet or workbook

AllowEditRange Object Example

The following routine creates an editable range on a protected worksheet, allowing a user to edit range J2:M16 with the successful entry of the password. In this case, the password is unlock:

```
Sub AddAllowEditRange()  
    ActiveSheet.Protection.AllowEditRanges.Add _  
        Title:="EditableRange", _  
        Range:=Range("J2:M16"), _  
        Password:="unlock"  
End Sub
```

Application Object

The `Application` object is the root object of the Excel object model. All the other objects in the Excel object model can only be accessed through the `Application` object. Many objects, however, are globally available. For example, the `ActiveSheet` property of the `Application` object is also available globally. That means that the active worksheet can be accessed in at least two ways: `Application.ActiveSheet` and `ActiveSheet`.

The `Application` object holds most of the application-level attributes that can be set through the Options menu in Excel. For example, the `DefaultFilePath` is equivalent to the Default File Location text box in the Save section of the Excel Options dialog box.

Many of the `Application` object's properties and methods are equivalent to things that can be set with the Options dialog box.

The `Application` object is also used when automating Excel from another application, such as Word. The `CreateObject` function, the `GetObject` function, or the `New` keyword can be used to create a new instance of an Excel `Application` object from another application. Please refer to Chapter 18 for examples of automation from another application.

The `Application` object can also expose events. However, `Application` events are not automatically available for use. The following three steps must be completed before `Application` events can be used:

1. Create a new class module (perhaps called `cAppObject`) and declare a `Public` object variable in a class (perhaps called `AppExcel`) to respond to events. For example:

```
Public WithEvents AppExcel As Excel.Application
```

- Now the `Application` object events will be available in the class for the `AppExcel` object variable.

2. Write the appropriate event handling code in the class. For example, if you wanted a message to appear whenever a worksheet was activated, you could write the following:

```
Private Sub AppExcel_SheetActivate(ByVal Sh As Object)
    'display worksheet name
    MsgBox "The " & Sh.Name & " sheet has just been activated."
End Sub
```

3. Finally, in a procedure in a standard module, instantiate the class created in the previous step with a current `Application` object:

```
Private App As New cAppObject 'class with the above code snippets
Sub AttachEvents()
    Set App.AppExcel = Application
End Sub
```

The `EnableEvents` property of the `Application` object must also be set to `True` for events to be triggered at the appropriate time.

Application Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Application Properties

Name	Returns	Description
<code>ActiveCell</code>	<code>Range</code>	Read-only. Returns the cell in the active sheet where the cursor is located
<code>ActiveChart</code>	<code>Chart</code>	Read-only. Returns the currently selected chart in the active workbook. If no chart is currently selected, nothing is returned
<code>ActivePrinter</code>	<code>String</code>	Set/Get the name of the printer currently being used

Table continued on following page

Application Properties

Name	Returns	Description
ActiveSheet	Object	Read-only. Returns the currently active sheet in the active workbook
ActiveWindow	Window	Read-only. Returns the currently selected Excel window, if any
ActiveWorkbook	Workbook	Read-only. Returns the workbook that is currently active, if any
AddIns	AddIns	Read-only. Returns the collection of Add-Ins currently available for use in Excel
AlertBeforeOverwriting	Boolean	Set/Get whether a message pops up any time an attempt to overwrite non-blank cells by a drag-and-drop operation is made
AltStartupPath	String	Set/Get the alternative startup file location folder for Excel
AlwaysUseClearType	Boolean	Set/Get the Boolean value determining whether ClearType is used to display fonts in menu, ribbon, and dialog box text
AnswerWizardWizard	Answer	Read-only. Returns an object allowing manipulation of the Answer Wizard
ArbitraryXMLSupportAvailable	Boolean	Read-only. Returns a Boolean value indicating if the XML feature is available in Excel
AskToUpdateLinks	Boolean	Set/Get whether the user is prompted to update links whenever a workbook with links is opened
Assistance	Assistant	Read-only. Returns an object representing the Microsoft Office Help Viewer
Assistant	Assistant	Read-only. Returns an object allowing manipulation of the Office Assistant
AutoCorrect	AutoCorrect	Read-only. Returns an object allowing modification of Excel's AutoCorrect features
AutoFormatAsYouTypeReplaceHyperlinks	Boolean	Set/Get whether Excel automatically formats/creates hyperlinks as you type
AutomationSecurityAutomationSecurity	Mso	Set/Get the level of macro security used when Excel opens a file programmatically
AutoPercentEntry	Boolean	Set/Get whether Excel automatically adds a % sign when typing a number into a cell that has a Percentage format applied

Name	Returns	Description
AutoRecover	AutoRecover	Set/Get AutoRecover options such as Path and Time interval
Build	Long	Read-only. Returns the exact build number of Excel
Calculate BeforeSave	Boolean	Set/Get whether workbooks are calculated before they are saved to disk. This assumes that formula calculation is not set to automatic (Calculation property)
Calculation	XlCalculation	Set/Get when calculations are made automatically, manually, or semi-automatically
Calculation InterruptKey Interrupt Key	XlCalculation	Set/Get the key that can interrupt Excel when performing calculations
Calculation State State	XlCalculation	Read-only. Indicates whether Excel calculations are in progress, pending, or done
Calculation Version	Long	Read-only. Returns the Excel version and calculation engine version used when the file was last saved
Caller	Variant	Read-only. Parameters: [Index]. Returns information describing what invoked the current Visual Basic code (for example, cell function, document event)
CanPlaySounds	Boolean	Read-only. Returns whether audio notes are heard in Excel. Property unused from Excel 2000 onward
CanRecord Sounds	Boolean	Read-only. Returns whether sound notes can be recorded in Excel. Property unused from Excel 2000 onward
Caption	String	Set/Get the caption that appears in the main Excel window
CellDragAnd Drop	Boolean	Set/Get whether dragging and dropping cells is possible
Cells	Range	Read-only. Returns all the cells in the active sheet
Charts	Sheets	Read-only. Returns all the charts in the active workbook

Table continued on following page

Application Properties

Name	Returns	Description
Clipboard Formats	Variant	Read-only. Parameters: [Index]. Returns an array of format values (xlClipboardFormat) that are currently in the clipboard
Columns	Range	Read-only. Returns all the columns in the currently active sheet
COMAddIns	COMAddIns	Read-only. Returns the collection of installed COM Add-Ins
CommandBars	CommandBars	Read-only. Returns the collection of command bars available to Excel
CommandUnderlines Underlines	xlCommand	Set/Get how commands are underlined in Excel. Used only on Macintosh systems
Constrain Numeric	Boolean	Set/Get whether only numbers and punctuation marks are recognized by handwriting recognition. Used only by Windows for Pen Computing
Control Characters	Boolean	Set/Get whether control characters are displayed for right-to-left languages. (Language support must be installed)
CopyObjects WithCells	Boolean	Set/Get whether objects (such as embedded objects) can be cut, copied, and sorted along with cell data
Cursor Pointer	xlMouse	Set/Get which mouse pointer is seen in Microsoft Excel
Cursor Movement	Long	Set/Get what type of cursor is used: visual or logical
CustomList Count	Long	Read-only. Returns the number of custom and built-in lists used in Excel (for example, Monday, Tuesday, Wednesday())
CutCopyMode	xlCutCopyMode	Set/Get whether a cut or copy operation is currently happening
DataEntryMode	Long	Set/Get whether locked cells can be edited (xlOff for editing allowed, xlOn for editing of unlocked cells only, xlStrict for editing of unlocked cells only that cannot be canceled by pressing Esc)
DDEAppReturnCode	Long	Read-only. Returns the result (confirmation/error) of the last DDE message sent by Excel
DecimalSeparator	String	Set/Get the character used for the decimal separator. This is a global setting and will affect all workbooks when opened. Use Application UseSystemSeparators = True to globally reset custom separators

Name	Returns	Description
DefaultFilePath	String	Set/Get the default folder used when opening files
DefaultSaveFormat	XlFileFormat	Set/Get the default file format used when saving files
DefaultSheetDirection	Long	Set/Get which direction new sheets will appear in Excel
DefaultWebOptionsOptions	DefaultWeb	Read-only. Returns an object allowing manipulation of the items associated with the Web Options dialog box
Dialogs	Dialogs	Read-only. Returns a collection of all the built-in dialog boxes
DisplayAlerts	Boolean	Set/Get whether the user is prompted by typical Excel messages (for example, "Save Changes to Workbook?"), or no prompts appear and the default answer is always chosen
DisplayClipboardWindow	Boolean	Set/Get whether the Clipboard window is displayed. Used in Microsoft Office Macintosh Edition
DisplayCommentIndicatorDisplayMode	XlComment	Set/Get how Excel displays cell comments and indicators
DisplayDocumentActionTaskPane	Boolean	Set to True to display the Document Actions task pane
DisplayDocumentInformationPanel	Boolean	Set to True to display the Document Properties panel
DisplayExcel4Menus	Boolean	Set/Get whether Excel displays Excel 4.0 menus
DisplayFormulaAutoComplete	Boolean	Set to False to disable auto complete when entering formulas
DisplayFormulaBar	Boolean	Set/Get whether the formula bar is displayed
DisplayFullScreen	Boolean	Set/Get whether the Excel is in full-screen mode
DisplayFunctionToolTips	Boolean	Set/Get whether ToolTips for arguments appear in the cell when typing a function

Table continued on following page

Application Properties

Name	Returns	Description
DisplayInsertOptions	Boolean	Set/Get whether the Insert Options drop-down button appears next to a range after inserting cells, rows, or columns
DisplayNoteIndicator	Boolean	Set/Get whether comments inserted into cells have a little note indicator at the top-right corner of the cell
DisplayPasteOptions	Boolean	Set/Get whether the Paste Options drop-down button appears next to a range after a paste operation. This is an Office XP setting and therefore affects all other Office applications that use this feature
DisplayRecentFiles	Boolean	Set/Get whether the most recently opened files are displayed under the Office Icon in the upper left-hand corner of the application
DisplayScrollBars	Boolean	Set/Get whether scrollbars are displayed for all open workbooks in the current session
DisplayStatusBar	Boolean	Set/Get whether the status bar is displayed
EditDirectlyInCell	Boolean	Set/Get whether existing cell text can be modified directly in the cell. Note that cell text can still be overwritten directly
EnableAnimations	Boolean	Set/Get whether adding and deleting cells, rows, and columns are animated
EnableAutoComplete	Boolean	Set/Get whether the AutoComplete feature is enabled
EnableKeyCancelKeyCancelKey	XlEnable	Set/Get how an Excel macro reacts when the user tries to interrupt the macro (for example, Ctrl+Break). This can be used to disable any user interruption, send any interruption to the error handler, or just stop the code (default). <i>Use with care</i>
EnableEvents	Boolean	Set/Get whether events are triggered for any object in the Excel object model that supports events
EnableLargeOperationAlerts	Boolean	Set to True to alert the user when an operation will affect the number of cells that exceeds the number specified in the Office center UI
EnableLivePreview	Boolean	Set/Get whether to show gallery previews
EnableSound	Boolean	Set/Get whether sounds are enabled for Excel

Name	Returns	Description
ErrorCheckingOptions	ErrorCheckingOptions	Set/Get error-checking properties such as BackgroundChecking, IndicatorColorIndex, and InconsistentFormula. These options mirror rules found in the Formulas section of the Excel Options dialog box
Excel4IntlMacroSheets	Sheets	Read-only. Returns the collection of sheets containing Excel 4 International macros
Excel4MacroSheets	Sheets	Read-only. Returns the collection of sheets containing Excel 4 macros
ExtendList	Boolean	Set/Get whether formatting and formulas are automatically added when adding new rows or columns to the existing lists of rows or columns
FeatureInstallInstall	MsoFeature	Set/Get how Excel reacts when an Excel feature is accessed that is not installed (through the interface or programmatically)
FileConverters	Variant	Read-only. Parameters: [Index1], [Index2]. Returns an array of all the file converters available in Excel
FileDialog	FileDialog	Parameters: [fileDialogType]. Returns an object that represents an instance of one of several types of file dialog boxes
FileFind	IFind	Returns an object that can be used to search for files. Used in Microsoft Office Macintosh Edition
FindFormat	CellFormat	Set/Get search criteria for the types of cell formats to look for when using the Find and Replace methods
FixedDecimal	Boolean	Set/Get whether any numbers entered in the future will have the decimal points specified by FixedDecimalPlaces
FixedDecimalPlaces	Long	Set/Get the decimal places used for any future numbers
FormulaBarHeight	Long	Set/Get the height of the formula bar. The formula bar cannot exceed the viewable window height
GenerateGetPivotData	Boolean	Set/Get whether Excel can get PivotTable report data
GenerateTableRefs	Boolean	Determines whether the traditional notation method or the new structured referencing notation method is used for referencing tables in formulas

Table continued on following page

Application Properties

Name	Returns	Description
Height	Double	Set/Get the height of Excel's main application window. The value cannot be set if the main window is maximized or minimized
Hinstance	Long	Read-only. Returns the instance handle of the instance that is calling Excel. Used mainly by other custom applications like those written in Visual Basic
Hwnd	Long	Read-only. Returns the top-level window handle of the Excel window. Used mainly by other custom applications like those written in Visual Basic
IgnoreRemoteRequests	Boolean	Set/Get whether remote requests through DDE are ignored
Interactive	Boolean	Set/Get whether Excel accepts keyboard and mouse input
International	Variant	Read-only. Parameters: [Index]. Returns international settings for Excel. Use the <code>XlApplicationInternational</code> constants as one of the values of <code>Index</code>
Iteration	Boolean	Set/Get whether Excel will iterate through and calculate all the cells in a circular reference trying to resolve the circular reference. Use with <code>MaxIterations</code> and <code>MaxChange</code>
LanguageSettingsSettings	Language	Read-only. Returns an object describing the language settings in Excel
LargeOperationCellThousandCount	Long	Set/Get the maximum number of cells that can be affected by a given operation before triggering an alert
Left	Double	Set/Get the left edge of Excel's main application window. The value cannot be set if the main window is maximized or minimized
LibraryPath	String	Read-only. Returns the directory where Add-Ins are stored
MailSession	Variant	Read-only. Returns the hexadecimal mail session number or <code>Null</code> if mail session is active
MailSystem	XlMailSystem	Read-only. Returns what type of mail system is being used by the computer (for example, <code>xlMapi</code> , <code>xlPowerTalk</code>)

Name	Returns	Description
MapPaperSize	Boolean	Set/Get whether documents formatted for another country's/region's standard paper size (for example, A4) are automatically adjusted so that they're printed correctly on your country's/region's standard paper size (for example, Letter)
MathCoprocesor Available	Boolean	Read-only. Returns whether a math coprocessor is available
MaxChange	Double	Set/Get the minimum change between iterations of a circular reference before iterations stop
MaxIterations	Long	Set/Get the maximum number of iterations allowed for circular references before iterations stop
MeasurementUnit	xlMeasurement Unit	Set/Get the measurement unit used in the application with the xlMeasurementUnit constants
MouseAvailable	Boolean	Read-only. Returns whether the mouse is available
MoveAfter Return	Boolean	Set/Get whether the current cell changes when the user hits Enter
MoveAfter Return Direction	XlDirection	Set/Get which direction the cursor will move when the user hits Enter, changing the current cell
MultiThreaded Calculation Calculation	MultThreaded	Returns a MultThreadedCalculation object that controls the multi-threaded recalculation settings
Name	String	Read-only. Returns "Microsoft Excel"
Names	Names	Read-only. Returns the collection of defined names in an active workbook
NetworkTemplates Path	String	Read-only. Returns the location on the network where the Excel templates are kept, if any
NewWorkbook	NewFile	Returns a NewFile object
ODBCErrors	ODBCErrors	Read-only. Returns the collection of errors returned by the most recent query or Pivot-Table report that had an ODBC connection
ODBCTimeout	Long	Set/Get how long, in seconds, an ODBC connection will be kept before timing out
OLEDBErrors	OLEDBErrors	Read-only. Returns the collection of errors returned by the most recent query or Pivot-Table report that had an OLEDB connection

Table continued on following page

Application Properties

Name	Returns	Description
OnWindow	String	Set/Get the procedure that is executed every time a window is activated by the end user
Operating System	String	Read-only. Returns the name and version of the operating system
OrganizationName	String	Read-only. Returns the organization name as seen in the About Microsoft Excel dialog box
Path	String	Read-only. Returns the path where Excel is installed
PathSeparator	String	Read-only. Returns a backslash (“\”) on a PC or a colon “:” on a Macintosh
PivotTable Selection	Boolean	Set/Get whether PivotTables use structured selection. For example, when selecting a Row field title, the associated data is selected with it
Previous Selections	Variant	Read-only. Parameters: [Index]. Returns an array of the last four ranges or named areas selected by using the Name dialog box or Goto feature
ProductCode	String	Read-only. Returns the GUID for Excel
PromptFor SummaryInfo	Boolean	Set/Get whether the user is prompted to enter summary information when trying to save a file
Range	Range	Read-only. Parameters: Cell1, [Cell2]. Returns a Range object containing all the cells specified by the parameters
Ready	Boolean	Read-only. Determines whether the Excel application is ready
RecentFiles	RecentFiles	Read-only. Returns the collection of recently opened files
RecordRelative	Boolean	Read-only. Returns whether recorded macros use relative cell references (True) or absolute cell references (False)
ReferenceStyle Style	XlReference	Set/Get how cells are referenced: Letter-Number (for example, A1, A3) or RowNumber-ColumnNumber (for example, R1C1, R3C1)
Registered Functions	Variant	Read-only. Parameters: [Index1], [Index2]. Returns the array of functions and function details relating to external DLLs or code resources. Using Add-Ins will add external DLLs to your workbook

Name	Returns	Description
ReplaceFormat	CellFormat	Set/Get replacement criteria for the types of cell formats to replace when using the <code>Replace</code> method
RollZoom	Boolean	Set/Get whether scrolling with a scroll mouse will zoom instead of scroll
Rows	Range	Read-only. Returns all the rows in the active sheet
RTD	RTD	Read-only. Returns a reference to a real-time date (RTD) object connected to an RTD Server
ScreenUpdating	Boolean	Set/Get whether Excel updates its display while a procedure is running. This property can be used to speed up procedure code by turning off screen updates (setting the property to <code>False</code>) during processing. Use with the <code>ScreenRefresh</code> method to manually refresh the screen
Selection	Object	Read-only. Returns whatever object is currently selected (for example, sheet, chart)
Sheets	Sheets	Read-only. Returns the collection of sheets in the active workbook
SheetsInNewWorkbook	Long	Set/Get how many blank sheets are put in a newly created workbook
ShowChartTipNames	Boolean	Set/Get whether charts show the tip names over data points
ShowChartTipValues	Boolean	Set/Get whether charts show the tip values over data points
ShowDevTools	Boolean	Set to <code>True</code> to display the Developer tab
ShowMenuFloaties	Boolean	Set to <code>False</code> to disable mini-toolbars when right-clicking in the workbook window
ShowSelectionFloaties	Boolean	Set to <code>False</code> to disable mini-toolbars when selecting text
ShowStartupDialog	Boolean	Set/Get whether the New Workbook task pane appears when loading the Excel application
ShowToolTips	Boolean	Set/Get whether ToolTips are shown in Excel
ShowWindowsInTaskbar	Boolean	Set/Get whether each workbook is visible on the taskbar (<code>True</code>) or only one Excel item is visible in the taskbar (<code>False</code>)
SmartTagRecognizers	SmartTagRecognizers	Read-only. Returns a collection of SmartTag recognition engines (recognizers) currently being used in the application

Table continued on following page

Application Properties

Name	Returns	Description
Speech	Speech	Read-only. Allows access to the properties and methods used to programmatically control the Office speech tools
Spelling Options Options	Spelling	Read-only. Allows access to the spelling options of the application
StandardFont	String	Set/Get what font is used as the standard Excel font
Standard FontSize	Double	Set/Get what font size is used as the standard Excel font size (in points)
StartupPath	String	Read-only. Returns the folder used as the Excel startup folder
StatusBar	Variant	Set/Get the status bar text. Returns <code>False</code> if Excel has control of the status bar. Set to <code>False</code> to give control of the status bar to Excel
TemplatesPath	String	Read-only. Returns the path to the Excel templates
ThisCell	Range	Set/Get the cell in which a user-defined function is being called
ThisWorkbook	Workbook	Read-only. Returns the workbook that contains the currently running VBA code
Thousands Separator	String	Set/Get the character used for the thousands separator. This is a global setting and will affect all workbooks when opened. Use <code>Application.UseSystemSeparators = True</code> to globally reset custom separators
Top	Double	Set/Get the top of Excel's main application window. The value cannot be set if the main window is maximized or minimized
Transition MenuKey	String	Set/Get what key is used to bring up Excel's menu. The forward slash key ("/") is the default
Transition MenuKeyAction	Long	Set/Get what happens when the Transition Menu key is pressed. Either Excel menus appear (<code>xlExcelMenu</code>) or the Lotus Help dialog box (<code>xlLotusHelp</code>) appears
Transition NavigKeys	Boolean	Set/Get whether the Transition Navigation keys are active. These provide different key combinations for moving and selecting within a worksheet
UsableHeight	Double	Read-only. Returns the vertical space available in Excel's main window, in points, that is available to a sheet's Window. The value will be 1 if there is no space available

Name	Returns	Description
UsableWidth	Double	Read-only. Returns the horizontal space available in Excel's main window, in points, that is available to a sheet's Window. This property's value will be invalid if no space is available. Check the value of the UsableHeight property to check to see if there is any space available (>1)
UsedObjects	UsedObjects	Read-only. Represents objects allocated in a workbook
UseLegacyKeyboardShortcuts	Boolean	Set to True to enable the use of legacy keyboard shortcuts
UserControl	Boolean	Read-only. True if the current Excel session was started by a user, and False if the Excel session was started programmatically
UserLibraryPath	String	Read-only. Returns the location of Excel's COM Add-Ins
UserName	String	Set/Get the user name in Excel. Note that this is the name shown in the General tab of the Options dialog box, and <i>not</i> the current user's network ID or the name shown in the Excel splash screen
UseSystemSeparators	Boolean	Set/Get whether the system operators in Excel are enabled. When set to False, you can use Application.DecimalSeparator and Application.ThousandsSeparator to override the system separators, which are located in the Regional Settings/Options applet in the Windows Control Panel
Value	String	Read-only. Returns "Microsoft Excel"
VBE	VBE	Read-only. Returns an object allowing manipulation of the Visual Basic Editor
Version	String	Read-only. Returns the version of Excel
Visible	Boolean	Set/Get whether Excel is visible to the user
WarnOnFunctionNameConflict	Boolean	Set to True, this property raises an alert when a newly created function is given a name that duplicates an existing function name
Watches	Watches	Read-only. Returns a Watches object that represents all of the ranges that are tracked when a worksheet is calculated

Table continued on following page

Application Methods

Name	Returns	Description
Width	Double	Set/Get the width of Excel's main application window. The value cannot be set if the main window is maximized or minimized
Windows	Windows	Read-only. Returns all the Windows open in the current Excel session
WindowsForPens	Boolean	Read-only. Returns whether Excel is running in a Windows for Pen Computing environment
WindowState XlWindow	xlWindowState	Set/Get whether the window is maximized, minimized, or in a normal state
Workbooks	Workbooks	Read-only. Returns all the open workbooks (not including Add-Ins) in the current Excel session
Worksheet Function Function	Worksheet	Read-only. Returns an object holding all the Excel's worksheet functions that can be used in VBA
Worksheets	Sheets	Read-only. Returns all the worksheets in the active workbook

Application Methods

Name	Returns	Parameters	Description
Activate MicrosoftApp		Index As XlMS Application	Activates an application specified by XlMSApplication. Opens the application if it is not open. Acts in a similar manner as the GetObject function in VBA
AddCustomList		ListArray, [ByRow]	Adds the array of strings specified by ListArray to Excel's custom lists. The ListArray may also be a cell range
Calculate			Calculates all the formulas in all open workbooks that have changed since the last calculation. Only applicable if using manual calculation
CalculateFull			Calculates all the formulas in all open workbooks. Forces recalculation of every formula in every workbook, regardless of whether or not it has changed since the last calculation

Name	Returns	Parameters	Description
CalculateFullRebuild			Forces a full calculation of the data and rebuilds the dependencies for all open workbooks. Note that dependencies are the formulas that depend on other cells
CalculateUnit AsyncQueries Done			Runs all pending queries to OLEDB and OLAP data sources
CentimetersTo Points	Double	Centimeters As Double	Converts the Centimeters parameter to points, where 1 cm = 28.35 points
CheckAbort	–	[KeepAbort]	Stops any recalculations in an Excel application
CheckSpelling [Custom Dictionary], [Ignore Uppercase]	Boolean	Word As String,	Checks the spelling of the Word parameter and returns True if the spelling is correct, or False if there are errors
Convert Formula	Variant	Formula, FromReference Style As XlReference Style, [ToReference Style], [ToAbsolute], [RelativeTo]	Converts the Formula parameter between R1C1 references and A1 references and returns the converted formula. Also, can change the Formula parameter between relative references and absolute references using the ToReferenceStyle parameter and the XlReferenceStyle constants
DDEExecute		Channel As Long, String As String	Sends a Command to an application using DDE through the given Channel number. The properties starting with DDE are associated with the older technology, Dynamic Data Exchange, which was used to share data between applications
DDEInitiate	Long	App As String, Topic As String	Returns a channel number to use for DDE given an application name and the DDE topic
DDEPoke		Channel As Long, Item, Data	Sends Data to an item in an application using DDE through the given Channel number
DDERequest	Variant	Channel As Long, Item As String	Returns information, given a specific DDE channel and a requested item

Table continued on following page

Application Methods

Name	Returns	Parameters	Description
DDETerminate		Channel As Long	Closes the specified DDE channel
DeleteCustomList		ListNum As Long	Deletes the custom list specified by the list number. The first four lists are built into Excel and cannot be removed
DisplayXMLSourcePane			Activates the XML Source task pane
DoubleClick			Triggered by a double-click to the active cell in the active sheet
Evaluate	Variant	Name	Evaluates the Name string expression as if it were entered into a worksheet cell
ExecuteExcel4Macro	Variant	String As String	Executes the Excel 4 macro specified by the String parameter and returns the results
FindFile	Boolean		Shows the Open dialog box, allowing the user to choose a file to open. True is returned if the file opens successfully
GetCustomListContents	Variant	ListNum As Long	Returns the custom list specified by the ListNum parameter as an array of strings
GetCustomListNum	Long	ListArray	Returns the list number for the custom list that matches the given array of strings. 0 is returned if nothing matches
GetOpenFilename	Variant	FileFilter], [FilterIndex], [Title], [ButtonText], [MultiSelect]	The Open dialog box is displayed with the optional file filters, titles, and button texts specified by the parameters. The filename and path are returned from this method call. Optionally, can return an array of filenames if the MultiSelect parameter is True. Does not actually open the file
GetPhonetic	String	[Text]	Returns the phonetic text of the Japanese characters in the Text parameter. If no Text parameter is specified, then an alternate phonetic text of the previous Text parameter is returned
GetSaveAsFilename	Variant	[Initial Filename], [FileFilter], [FilterIndex], [Title], [ButtonText]	The Save As dialog box is displayed with the optional default filename, file filters, titles, and button texts specified by the parameters. The filename and path are returned from this method call. Does not actually save the file

Name	Returns	Parameters	Description
Goto		[Reference], [Scroll]	Selects the object specified by the Reference parameter and activates the sheet containing that object. The Reference parameter can be a cell, a range, or the name of a VBA procedure. The Scroll parameter, if set to True, will scroll the selected object to the top-left corner of the Excel window
Help		[HelpFile], [HelpContextID]	Displays the Help topic specified by the HelpContextID parameter in the Help file HelpFile
InchesToPoints	Double	Inches As Double	Converts the Inches parameter to points and returns the new value (1 inch = 72 points)
InputBox	Variant	Prompt As String, [Title], [Default], [Left], [Top], [HelpFile], [HelpContextID], [Type]	Displays a simple input box, very similar to a standard VBA one. However, the [Type] parameter can be used to set the return type to a formula (0), a number (1), text (2), a Boolean (4), a cell reference (8), an error value (16), or an array of values (64)
Intersect	Range	Arg1 As Range, Arg2 As Range, [Arg3], [Arg30]	Returns the intersection or overlap of the ranges specified by the parameters as a Range object
MacroOptions		[Macro], [Description], [HasMenu], [MenuText], [HasShortcut Key], [ShortcutKey], [Category], [StatusBar], [HelpContext ID], [HelpFile]	Allows modification of macro attributes such as the name, description, shortcut key, category, and associated Help file. Equivalent to the Macro Options dialog box
MailLogoff			Logs off the current MAPI mail session (for example, Exchange, Outlook)
MailLogon		[Name], [Password], [DownloadNewMail]	Logs on to the default MAPI mail client (for example, Exchange, Outlook). Credentials such as name and password can be specified

Table continued on following page

Application Methods

Name	Returns	Parameters	Description
NextLetter		Workbook	Used in Macintosh systems with PowerTalk mail extensions to open the oldest unread workbook from the In Tray. Generates an error in Windows
OnKey		Key As String, [Procedure]	Executes the procedure specified by the Procedure parameter whenever the keystroke or key combination described in the Key parameter is pressed
OnRepeat		Text As String, Procedure As String	Specifies the procedure that will run when the user chooses the Repeat command
OnTime		EarliestTime, Procedure As String, [LatestTime], [Schedule]	Chooses a procedure to run at the time specified by the EarliestTime parameter. Uses the LatestTime parameter to specify a time range
OnUndo		Text As String, Procedure As String	Specifies the procedure to run when the user chooses the Undo command
Quit			Shuts down Microsoft Excel
RecordMacro		[Basic Code], [XlmCode]	If the user is currently recording a macro, running this statement will put the code specified in the BasicCode parameter into the currently recording macro
RegisterXLL	Boolean	Filename As String	Loads the code resource specified by the Filename parameter, and registers all the functions and procedures in that code resource
Repeat			Repeats the last user action made. Must be the first line of a procedure
Run	Variant	[Macro], [Arg1], [Arg2], ([Arg30]	Runs the macro or procedure specified by the Macro parameter. Can also run Excel 4.0 macros with this method
SaveWorkspace		[Filename]	Saves the current workspace to the Filename parameter
SendKeys		Keys, [Wait]	Sends the keystrokes in the Keys parameter to Microsoft Excel user interface
Undo			Undoes the last action done with the user interface

Name	Returns	Parameters	Description
Union	Range	Arg1 As Range, Arg2 As Range, [Arg3], ([Arg30]	Returns the union of the ranges specified by the parameters
Volatile		[Volatile]	Sets the function that currently contains this statement to be either volatile (Volatile parameter to True) or not. A volatile function will be recalculated whenever the sheet containing it is calculated, even if its input values have not changed
Wait	Boolean	Time	Pauses the macro and Excel until the time in the Time parameter is reached

Application Events

Name	Parameters	Description
AfterCalculate		Triggered when all refresh and calculation activities have been completed
CalculationDone	Wb As Workbook	Triggered after a calculation has been executed
NewWorkbook	Wb As Workbook	Triggered when a new workbook is created. The new workbook is passed into the event
SheetActivate	Sh As Object	Triggered when a sheet is activated (brought up to front of the other sheets). The activated sheet is passed into the event
SheetBeforeDoubleClick	Sh As Object, Target As Range, Cancel As Boolean	Triggered when a sheet is about to be double-clicked. The sheet and the potential double-click spot are passed into the event. The double-click action can be canceled by setting the Cancel parameter to True
SheetBeforeRightClick	Sh As Object, Target As Range, Cancel As Boolean	Triggered when a sheet is about to be right-clicked. The sheet and the potential right-click spot are passed into the event. The right-click action can be canceled by setting the Cancel parameter to True
SheetCalculate	Sh As Object	Triggered when a sheet is recalculated, passing in the recalculated sheet
SheetChange	Sh As Object, Target As Range	Triggered when a range on a sheet is changed, for example, by clearing the range, entering data, deleting rows or columns, pasting data, and so on. <i>Not</i> triggered when inserting rows/columns

Table continued on following page

Application Events

Name	Parameters	Description
Sheet Deactivate	Sh As Object	Triggered when a sheet loses focus. Passes in the sheet
SheetFollow Hyperlink	Sh As Object, Target As Hyperlink	Triggered when the user clicks a hyperlink on a sheet. Passes in the sheet and the clicked hyperlink
SheetPivotTableUpdate	ByVal Sh As Object, Target As PivotTable	Triggered by an update of the PivotTable report. Passes in the sheet and the PivotTable report
SheetSelection Change	Sh As Object, Target As Range	Triggered when the user selects a new cell in a worksheet. Passes in the new range and the sheet where the change occurred
Window Activate	Wb As Workbook, Wn As Window	Triggered when a workbook window is activated (brought up to the front of other workbook windows). The workbook and the window are passed in
Window Deactivate	Wb As Workbook, Wn As Window	Triggered when a workbook window loses focus. The related workbook and the window are passed in
WindowResize	Wb As Workbook, Wn As Window	Triggered when a workbook window is resized. The resized workbook and window are passed into the event. Not triggered when Excel is resized
Workbook Activate	Wb As Workbook	Triggered when a workbook is activated (brought up to the front of other workbook windows). The workbook is passed in
WorkbookAddin Install	Wb As Workbook	Triggered when an Add-In is added to Excel that is also a workbook. The Add-In workbook is passed into the event
WorkbookAddin Uninstall	Wb As Workbook	Triggered when an Add-In is removed to Excel that is also a workbook. The Add-In workbook is passed into the event
WorkbookAfter XMLExport	Wb As Workbook Map As XMLMap URL As String Result As xlxmlexportresult	Triggered after XML data that is mapped to a worksheet is exported
WorkbookAfter XMLImport	Wb As Workbook Map As XMLMap IsRefresh As Boolean Result As xlxmlimporttresult	Triggered after an existing XML data mapping is refreshed or new XML data is imported into an existing XML map

Name	Parameters	Description
Workbook BeforeClose	Wb As Workbook, Cancel As Boolean	Triggered just before a workbook is closed. The workbook is passed into the event. The closure can be canceled by setting the <code>Cancel</code> parameter to <code>True</code>
Workbook BeforePrint	Wb As Workbook, Cancel As Boolean	Triggered just before a workbook is printed. The workbook is passed into the event. The printing can be canceled by setting the <code>Cancel</code> parameter to <code>True</code>
Workbook BeforeSave	Wb As Workbook, SaveAsUI As Boolean, Cancel As Boolean	Triggered just before a workbook is saved. The workbook is passed into the event. The saving can be canceled by setting the <code>Cancel</code> parameter to <code>True</code> . If the <code>SaveAsUI</code> is set to <code>True</code> , then the <code>Save As</code> dialog box appears
WorkbookBefore XMLExport	Wb As Workbook, Map As XMLMap, URL As String Cancel As Boolean	Triggered before XML data that is mapped to a worksheet is exported
WorkbookBefore XMLImport	Wb As Workbook, Map As XMLMap URL As String	Triggered before an existing XML data mapping is refreshed or new XML data is imported into an existing XML map
Workbook Deactivate	Wb As Workbook	Triggered when a workbook loses focus. The related workbook and the window are passed in
Workbook NewSheet	Wb As Workbook, Sh As Object	Triggered when a new sheet is added to a workbook. The workbook and new sheet are passed into the event
WorkbookOpen	Wb As Workbook	Triggered when a workbook is opened. The newly opened workbook is passed into the event
WorkbookPivot TableClose Connection	ByVal Wb As Workbook, Target As PivotTable	Triggered when a <code>PivotTable</code> report connection is closed. The selected workbook and <code>PivotTable</code> report are passed into this event
WorkbookPivot TableOpen Connection	ByVal Wb As Workbook, Target As PivotTable	Triggered when a <code>PivotTable</code> report connection is opened. The selected workbook and <code>PivotTable</code> report are passed into this event
WorkbookRowset Complete	Wb As Workbook Description As String Sheet As String Success As Boolean	Triggered when the user either drills through the recordset or invokes the rowset action on an OLAP <code>PivotTable</code>
WorkbookPivot TableOpen Connection	Wb As Workbook SyncEventType As MsoSyncEventType	Triggered when the local copy of a workbook that is part of a Document Workspace is synchronized with the copy on the server

Application Object Example

This example demonstrates how to use `Application.GetOpenFilename` to get the name of a file to open. The key to using this function is to assign its return value to a `Variant` data type:

```
Sub UsingGetOpenFilename()  
    Dim sFilter As String  
    Dim vaFile As Variant  
    'Build a filter list.  
    sFilter = "Excel 2007 Files,* .xlsx," & _  
        "Excel 2000-2003 Files,* .xls"  
    'Display the File Open dialog, putting the result in a Variant  
    vaFile = Application.GetOpenFilename(FileFilter:=sFilter, FilterIndex:=1, _  
        Title:="Open a New or Old File", MultiSelect:=False)  
    'If user cancelled, then exit, else open the selected file.  
    If vaFile <> False Then  
        Workbooks.Open Filename:=vaFile  
    Else  
        MsgBox "Action Cancelled"  
    End If  
End Sub
```

Areas Collection

The `Areas` collection holds a collection of `Range` objects. Each `Range` object represents a block of cells (for example, A1:A10) or a single cell. The `Areas` collection can hold many ranges from different parts of a workbook. The parent of the `Areas` collection is the `Range` object.

Areas Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Areas Properties

Name	Returns	Description
Count	Long	Read-only. Returns the number of <code>Range</code> objects that are contained in the area
Item	Range	Parameter: <code>Index As Long</code> . Returns a single <code>Range</code> object in the <code>Areas</code> collection. The <code>Index</code> parameter corresponds to the order of the ranges selected

Areas Collection Example

When using a `Range` containing a number of different areas, you cannot use code like `rgRange.Cells(20).Value` if the 20th cell is not inside the first area in the range. This is because Excel only looks at the first area, implicitly doing `rgRange.Areas(1).Cells(20).Value`, as this example shows — with a function to provide a workaround:

```

Sub TestMultiAreaCells()
Dim oRNg As Range
'Define a multi-area range
Set oRNg = Range("D2:F5,H2:I5")
'The 12th cell should be F5.
    MsgBox "Rng.Cells(12) is " & oRNg.Cells(12).Address & _
        vbCrLf & "Rng.Areas(1).Cells(12) is " & oRNg.Areas(1).Cells(12).Address & _
        vbCrLf & "MultiAreaCells(oRng, 12) is " & MultiAreaCells(oRNg, 12).Address
'The 13th cell of the multi-area range should be H2,
'that is the first cell in the second area.
    MsgBox "Rng.Cells(13) is " & oRNg.Cells(13).Address & _
        vbCrLf & "Rng.Areas(1).Cells(13) is " & oRNg.Areas(1).Cells(13).Address & _
        vbCrLf & "MultiAreaCells(Rng, 13) is " & MultiAreaCells(oRNg, 13).Address
End Sub

Function MultiAreaCells(oRange As Range, iCellNum As Long) As Range
Dim iTotCells As Long, oArea As Range
'Loop through all the areas in the range,
'starting again from the first if we run out
    Do
        For Each oArea In oRange.Areas
            'Is the cell we want in this area?
            'Return it and exit if Yes
                If iTotCells + oArea.Cells.Count >= iCellNum Then
                    Set MultiAreaCells = oArea.Cells(iCellNum - iTotCells)
                    Exit Function
                Else
                    iTotCells = iTotCells + oArea.Cells.Count
                End If
        Next
    Loop
End Function

```

AutoCorrect Object

The AutoCorrect object represents all of the functionality of the Excel's AutoCorrect features.

AutoCorrect Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

AutoCorrect Properties

Name	Returns	Description
AutoExpand	Boolean	Set to True to enable automatic expansion in lists ListRange
AutoFillFormulasInLists	Boolean	Set to True to enable the creation of calculated columns created by automatic fill-down lists

Table continued on following page

AutoCorrect Methods

Name	Returns	Description
CapitalizeNamesOfDays	Boolean	Set/Get whether the first letters of days of the week are capitalized
CorrectCapsLock	Boolean	Set/Get whether typing mistakes made by leaving the Caps Lock on are automatically corrected
CorrectSentenceCap	Boolean	Set/Get whether the first letter of a sentence is capitalized if accidentally left in lowercase
DisplayAutoCorrectOptions	Boolean	Displays/Hides the AutoCorrect Options button. The default value is True. This is an Office-wide setting. Changing it in Excel will also affect all the other Office applications
ReplacementList	Boolean	Returns a multidimensional array of strings. The first column of the array holds the word that will be changed, and the second column holds the replaced text. The Index parameter can be used to return an array containing a single word and its replacement
ReplaceText	Boolean	Set/Get whether Excel will automatically replace certain words with words from the AutoCorrect list
TwoInitialCapitals	Boolean	Set/Get whether Excel will automatically change the second letter of a word to lowercase if the first letter is uppercase

AutoCorrect Methods

Name	Returns	Parameters	Description
AddReplacement	Variant	What As String, Replacement As String	Adds a word (the What parameter) that will be automatically replaced with another word (the Replacement parameter) to the ReplacementList list array
DeleteReplacement	Variant	What As String	Deletes a word from the ReplacementList list so that it does not get replaced with another word automatically

AutoCorrect Object Example

This example uses the `AutoCorrect` object to find the replacement to use for a given word:

```
Sub TestAutoCorrect()
    MsgBox "'(c)' is replaced by " & UseAutoCorrect("(c)")
End Sub

Function UseAutoCorrect(ByVal sWord As String) As String
    Dim i As Integer
    Dim vaRepList As Variant
    Dim sReturn As String
    'Default to returning the word we were given
    sReturn = sWord
    'Get the replacement list into an array
    vaRepList = Application.AutoCorrect.ReplacementList
    'Go through the replacement list
    For i = LBound(vaRepList) To UBound(vaRepList)
        'If there is a match, return the replacement text, else exit loop.
        If vaRepList(i, 1) = sWord Then
            sReturn = vaRepList(i, 2)
            Exit For
        End If
    Next
    'Return the word, or its replacement if it has one
    UseAutoCorrect = sReturn
End Function
```

AutoFilter Object

The `AutoFilter` object provides the functionality equivalent to the `AutoFilter` feature in Excel. This object can programmatically filter a range of text for specific types of rows, hiding the rows that do not meet the filter criteria. Examples of filters include top 10 rows in the column, rows matching specific values, and non-blank cells in the row. The parent of the `AutoFilter` object is the `Worksheet` object (implying that a worksheet can have only one `AutoFilter`).

The `AutoFilter` object is used with the `AutoFilter` method of the `Range` object and the `AutoFilterType` property of the `Worksheet` object.

AutoFilter Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

AutoFilter Properties

Name	Returns	Description
Filter Mode	Boolean	Set/Get whether a given worksheet is in the AutoFilter filter mode
Filters	Filters	Read-only. Returns the collection of filters associated with the range that was auto-filtered (for example, non-blank rows)
Range	Range	Read-only. Returns the group of cells that have an AutoFilter applied to them
Sort	Sort	Controls the attributes and specifications of a sort with the AutoFilter object

AutoFilter Methods

Name	Returns	Parameters	Description
ApplyFilter			Applies the specified filters
ShowAllData			Removes any specified filters

AutoFilter Object Example

This example demonstrates how to use the AutoFilter, Filters, and Filter objects by displaying the complete set of auto-filters currently in use:

```
Sub ShowAutoFilterCriteria()  
Dim oAF As AutoFilter, oFlt As Filter  
Dim sField As String  
Dim sCrit1 As String, sCrit2 As String  
Dim sMsg As String, i As Integer  
'Check if the sheet is not filtered, then exit  
If ActiveSheet.AutoFilterMode = False Then  
    MsgBox "The sheet does not have an AutoFilter"  
    Exit Sub  
End If  
'Get the sheet's AutoFilter object  
Set oAF = ActiveSheet.AutoFilter  
'Loop through the Filters of the AutoFilter  
For i = 1 To oAF.Filters.Count  
'Get the field name from the first row of the AutoFilter range  
sField = oAF.Range.Cells(1, i).Value  
'Get the Filter object  
Set oFlt = oAF.Filters(i)  
'If it is on, then get the standard filter criteria  
If oFlt.On Then  
sMsg = sMsg & vbCrLf & sField & oFlt.Criteria1  
'If it's a special filter, show it
```

```

        Select Case oFlt.Operator
            Case xlAnd
                sMsg = sMsg & " And " & sField & oFlt.Criteria2
            Case xlOr
                sMsg = sMsg & " Or " & sField & oFlt.Criteria2
            Case xlBottom10Items
                sMsg = sMsg & " (bottom 10 items)"
            Case xlBottom10Percent
                sMsg = sMsg & " (bottom 10%)"
            Case xlTop10Items
                sMsg = sMsg & " (top 10 items)"
            Case xlTop10Percent
                sMsg = sMsg & " (top 10%)"
        End Select
    End If
Next
'Construct a no filters message if no filters applied
If sMsg = "" Then
    sMsg = "The range " & oAF.Range.Address & " is not filtered."
Else
'Construct a message showing any filters that are applied
    sMsg = "The range " & oAF.Range.Address & " is filtered by:" & sMsg
End If
'Display the message
MsgBox sMsg
End Sub

```

AutoRecover Object

This object allows access to the `AutoRecover` settings for the Excel application. These settings can be found on the Save section of the Excel Options dialog. Note that each workbook can choose whether or not to have `AutoRecover` applied to it—also located on the Save tab.

AutoRecover Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

AutoRecover Properties

Name	Returns	Description
Enabled	Boolean	Set/Get whether the <code>AutoRecover</code> object is enabled
Path	String	Set/Get the complete path to where the <code>AutoRecover</code> temporary files are saved
Time	Long	Set/Get the time interval for the <code>AutoRecover</code> object. Permissible values are integers from 1 to 120 minutes (default 10)

AutoRecover Object Example

AutoRecover Object Example

The following subroutine and function sets `AutoRecover` properties, then ensures that the workbook the code is in uses them:

```
Sub SetAutoRecoverOptions()  
    'Set the AutoRecover options for the application  
    ChangeAutoRecoverSettings True, "C:\Backup Files\AutoRecover\Excel", 2  
    'Make sure this workbook uses them  
    ThisWorkbook.EnableAutoRecover = True  
End Sub  
  
Function ChangeAutoRecoverSettings(Optional ByVal vEnable As Variant, _  
Optional ByVal vPath As Variant, Optional ByVal vTime As Variant)  
    'Only set the property if a value was passed  
    With Application.AutoRecover  
        If Not IsMissing(vEnable) Then  
            'Enable AutoRecover  
            .Enabled = vEnable  
        End If  
    'Only set the property if a value was passed  
    If Not IsMissing(vPath) Then  
        'Change the path to a central backup files area  
        .Path = vPath  
    End If  
    'Only set the property if a value was passed  
    If Not IsMissing(vTime) Then  
        'Save every AutoRecover file every 2 minutes  
        .Time = vTime  
    End If  
    End With  
End Function
```

Axis Object and the Axes Collection

The `Axes` collection represents all of the `Axes` in an Excel chart. Each `Axis` object is equivalent to an axis in an Excel chart (for example, X axis, Y axis, and so on). The parent of the `Axes` collection is the `Chart` object.

Unlike most other collections, the `Item` method of the `Axes` collection has two parameters: `Type` and `AxisGroup`. Use one of the `xlAxisType` constants for the `Type` parameter (`xlValue`, `xlCategory`, or `xlSeriesAxis`). The optional second parameter, `AxisGroup`, can take one of the `xlAxisGroup` constants (`xlPrimary` or `xlSecondary`).

Axis Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Axis Properties

Name	Returns	Description
AxisBetweenCategories	Boolean	Set/Get whether the value axis crosses the category axis between categories (as in Column charts) or is aligned with the category label (as in Line charts)
AxisGroup	XlAxisGroup	Read-only. Returns whether the current axis is of the primary group (xlPrimary) or the secondary group (xlSecondary)
AxisTitle	AxisTitle	Read-only. Returns an object manipulating the axis title properties
BaseUnit	XlTimeUnit	Set/Get what type of base units to have for a category axis. Use with BaseUnitIsAuto property. Fails on a value axis
BaseUnitIsAuto	Boolean	Set/Get whether the Excel automatically chooses the base units for a category axis. Fails on a value axis
Border	Border	Read-only. Returns the border's properties around the selected axis
CategoryNames	Variant	Set/Get the category names for the axis as a string array
CategoryType	XlCategoryType	Set/Get what type of axis to make the category axis. Fails on a value axis
Crosses	XlAxisCrosses	Set/Get where one axis crosses with the other axis: at the minimum value, maximum value, Excel automatic, or some custom value
CrossesAt	Double	Set/Get at which value the other axis crosses the current one. Use when the Crosses property is xlAxisCrossesCustom
DisplayUnit	XlDisplayUnit	Set/Get what sort of unit to display for the axis (for example xlThousands)
DisplayUnitCustom	Double	Set/Get the value to display units if the DisplayUnit property is set to xlCustom
DisplayUnitLabel	DisplayUnitLabel	Read-only. Returns an object that manipulates a unit label for an axis
Format	ChartFormat	Read-only. Returns the ChartFormat object, which controls the line, fill, and effect formatting for the chart axis.

Table continued on following page

Axis Properties

Name	Returns	Description
HasDisplayUnitLabel	Boolean	Set/Get whether a display unit label created using the <code>DisplayUnit</code> or <code>DisplayUnitCustom</code> property is visible on the axis
HasMajorGridlines	Boolean	Set/Get whether major gridlines are displayed on the axis
HasMinorGridlines	Boolean	Set/Get whether minor gridlines are displayed on the axis
HasTitle	Boolean	Set/Get whether the axis has a title
Height	Double	Read-only. Returns the height of the axis
Left	Double	Read-only. Returns the position of the axis from the left edge of the chart
LogBase	Double	Set/Get the base of the logarithm when using log scales
MajorGridlines	Gridlines	Read-only. Returns an object to manipulate the major gridlines formatting
MajorTickMark	XlTickMark	Set/Get what the major ticks should look like (for example, inside the axis, outside the axis)
MajorUnit	Double	Set/Get what the value is between major blocks of a unit
MajorUnitIsAuto	Boolean	Set/Get whether the value of <code>MajorUnit</code> is set automatically
MajorUnitScale	XlTimeUnit	Set/Get what type to set for the major units
MaximumScale	Double	Set/Get what the maximum value is for the axis
MaximumScaleIsAuto	Boolean	Set/Get whether the maximum value for the axis is determined automatically
MinimumScale	Double	Set/Get what the minimum value is for the axis
MinimumScaleIsAuto	Boolean	Set/Get whether the minimum value for the axis is determined automatically
MinorGridlines	Gridlines	Read-only. Returns an object to manipulate major gridline formatting
MinorTickMark	XlTickMark	Set/Get what the minor ticks should look like (for example, inside the axis, outside the axis)
MinorUnit	Double	Set/Get what the value is between minor blocks of a unit
MinorUnitIsAuto	Boolean	Set/Get whether the value of <code>MinorUnit</code> is set automatically
MinorUnitScale	XlTimeUnit	Set/Get what scale to set for the minor units

Name	Returns	Description
ReversePlot Order	Boolean	Set/Get whether the unit values on the axis should be reversed
ScaleType	XlScale Type	Set/Get the type of scale to use for the units: Linear or Logarithmic
TickLabel Position	XlTickLabel Position	Set/Get the position that the tick marks will appear in relation to the axis (for example, low, high)
TickLabels	TickLabels	Read-only. Returns an object to manipulate properties of the tick labels of an axis
TickLabel Spacing	Long	Set/Get how often to display the tick labels
TickLabel SpacingIsAuto	Boolean	Set/Get whether the value of TickLabelSpacing is set automatically
TickMark Spacing	Long	Set/Get how often to display tick marks on an axis. Fails on a value axis
Top	Double	Read-only. Returns the top of the axis in relation to the top edge of the chart
Type	XlAxisType	Set/Get the type of axis (xlCategory, xlSeriesAxis, or xlValue)
Width	Double	Read-only. Returns the width of the axis

Axis Methods

Name	Returns	Parameters	Description
Delete	Variant		Deletes the axis from the axes collection
Select	Variant		Selects the axis on the chart

Axis Object and the Axes Collection Example

This example sets the labels for the X-axis (independently of the data that's plotted) and applies some formatting:

```
Sub FormatXAxis()
    Dim oCht As Chart, oAxis As Axis
    'Get the first embedded chart on the sheet
    Set oCht = ActiveSheet.ChartObjects(1).Chart
    'Get it's X axis
    Set oAxis = oCht.Axes(xlCategory)
    'Format the X axis
    With oAxis
        .CategoryNames = Array("Item 1", "Item 2", "Item 3")
        .TickLabels.Orientation = 45
    End With
End Sub
```

AxisTitle Object

```
.AxisBetweenCategories = True
.ReversePlotOrder = False
.MinorTickMark = xlTickMarkNone
.MajorTickMark = xlTickMarkCross
End With
End Sub
```

AxisTitle Object

The `AxisTitle` object contains the formatting and words associated with a chart axis title. The parent of the `AxisTitle` object is the `Axis` object. The `AxisTitle` object is used in coordination with the `HasTitle` property of the parent `Axis` object. The `HasTitle` property must be `True` for a child `AxisTitle` object to exist.

AxisTitle Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

AxisTitle Properties

Name	Returns	Description
<code>AutoScaleFont</code>	Variant	Set/Get whether the font size will change automatically if the parent chart changes sizes
<code>Border</code>	Border	Read-only. Returns the border's properties around the selected axis title
<code>Caption</code>	String	Set/Get the axis title's text
<code>Characters</code>	Characters	Read-only. Parameters: [<code>Start</code>], [<code>Length</code>]. Returns an object containing all the characters in the axis title. Allows manipulation on a character-by-character basis
<code>Fill Format</code>	ChartFill	Read-only. Returns an object containing fill formatting options for the chart axis title
<code>Font</code>	Font	Read-only. Returns an object containing Font options for the chart axis title
<code>Format</code>	ChartFormat	Read-only. Returns the <code>ChartFormat</code> object, which controls the line, fill, and effect formatting for the axis title
<code>Horizontal Alignment</code>	<code>xlAlign</code>	Set/Get how you want the axis title to be horizontally aligned. Use the <code>xlAlign</code> constants
<code>IncludeLayout</code>	Boolean	Set/Get whether the axis title will occupy the chart layout space when a chart layout is being determined
<code>Interior</code>	Interior	Read-only. Returns an object containing options to format the area in the chart title text area (for example, interior color)

Name	Returns	Description
Left	Double	Set/Get the distance from the left edge of the axis title text area to the chart's left edge
Name	String	Read-only. Returns the name of the axis title object
Position	xlChartElementPosition	Set/Get the position of the axis title on the chart
Orientation	xlOrientation	Set/Get the angle of the text for the axis title. The value can be in degrees (from -90 to 90) or one of the xlOrientation constants
ReadingOrder	Long	Set/Get how the text is read (from left to right or right to left). Only applicable in appropriate languages
Shadow	Boolean	Set/Get whether the axis title has a shadow effect
Text	String	Set/Get the axis title's text
Top	Double	Set/Get the distance from the top edge of the axis title text area to the chart's top edge
VerticalAlignment	xlVAlign	Set/Get how you want the axis title to be horizontally aligned. Use the xlVAlign constants

AxisTitle Methods

Name	Returns	Parameters	Description
Delete	Variant		Deletes the axis title from the axis
Select	Variant		Selects the axis title on the chart

AxisTitle Object Example

This example ensures the X-axis has a title and sets the X-axis title's caption and formatting:

```
Sub FormatXAxisTitle()
    Dim oCht As Chart, oAT As AxisTitle
    'Get the first embedded chart on the sheet
    Set oCht = ActiveSheet.ChartObjects(1).Chart
    'Give the X axis a title
    oCht.Axes(xlCategory).HasTitle = True
    'Get the title
    Set oAT = oCht.Axes(xlCategory).AxisTitle
    'Format the title
    With oAT
        .AutoScaleFont = False
        .Caption = "X Axis Title"
        .Font.Bold = True
    End With
End Sub
```

Border Object and the Borders Collection

The `Borders` collection contains the properties associated with four borders around the parent object. Parent objects of the `Borders` collection are the `Range` and the `Style` object. A `Borders` collection always has four borders. Use the `xlBordersIndex` constants with the `Item` property of the `Borders` collection to access one of the `Border` objects in the collection.

Each `Border` object corresponds to a side or some sides of a border around a parent object. Some objects only allow access to all four sides of a border as a whole (for example, the left side of border cannot be accessed independently). The following objects are parents of the `Border` object (not the `Borders` collection): `Axis`, `AxisTitle`, `ChartArea`, `ChartObject`, `ChartTitle`, `DataLabel`, `DataTable`, `DisplayUnitLabel`, `Downbars`, `DropLines`, `ErrorBars`, `Floor`, `GridLines`, `HiLoLines`, `LeaderLines`, `Legend`, `LegendKey`, `OleObject`, `PlotArea`, `Point`, `Series`, `SeriesLines`, `TrendLine`, `UpBars`, and `Walls`. The following collections are also possible parents of the `Border` object: `DataLabels`, `ChartObjects`, and `OleObjects`.

Borders Collection Properties

The `Borders` collection has a few properties besides the typical collection attributes. They are listed in the following table.

Name	Returns	Description
<code>Color</code>	<code>Variant</code>	Set/Get the color for all four of the borders in the collection. Use the <code>RGB</code> function to set the <code>color</code>
<code>ColorIndex</code>	<code>Variant</code>	Set/Get the color for all four of the borders in the collection. Use the index number of a color in the current color palette to set the <code>Color</code> value
<code>LineStyle</code>	<code>xlLineStyle</code>	Set/Get the style of line to use for the borders (for example, <code>xlDash</code>). Use the <code>xlLineStyle</code> constants to set the value
<code>ThemeColor</code>	<code>xlThemeColor</code>	Set/Get the theme color in the applied color scheme associated with an object. Should the object have no association with a theme, then trying to access the <code>ThemeColor</code> property will result in an error
<code>TintAndShade</code>	<code>Single</code>	Set/Get a <code>Single</code> value from -1 (darkest) to 1 (lightest), which darkens or lightens a color. Zero (0) is neutral
<code>Value</code>	<code>xlLineStyle</code>	Set/Get the style of line to use for the borders (for example, <code>xlDash</code>). Use the <code>xlLineStyle</code> constants to set the value. Same as <code>LineStyle</code>
<code>Weight</code> <code>Weight</code>	<code>xlBorder</code>	Set/Get how thick to make the borders in the collection (for example, <code>xlThin</code> , <code>xlThick</code>). Use the <code>xlBorderWeight</code> constants

Border Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Border Properties

Name	Returns	Description
Color	Variant	Set/Get the color for a border. Use the RGB function to set the color
ColorIndex	Variant	Set/Get the color for a border. Use the index number of a color in the current color palette to set the color value
LineStyle	xlLineStyle	Set/Get the style of line to use for a border (for example, xlDash). Use the xlLineStyle constants to set the value
ThemeColor	xlThemeColor	Set/Get the theme color in the applied color scheme associated with an object. Should the object have no association with a theme, then trying to access the ThemeColor property will result in an error
TintAndShade	Single	Set/Get a Single value from -1 (darkest) to 1 (lightest), which darkens or lightens a color. Zero (0) is neutral
Weight	xlBorderWeight	Set/Get how thick to make the border (for example, xlThin, xlThick). Use the xlBorderWeight constants

Border Object and the Borders Collection Example

Applies a 3D effect to a range:

```

Sub TestFormat3D()
'Format the selected range as 3D sunken
  Format3D Selection
End Sub
Sub Format3D(oRange As Range, Optional bSunken As Boolean = True)
'Using the range(
  With oRange
'Surround it with a white border
    .BorderAround Weight:=xlMedium, Color:=RGB(255, 255, 255)

    If bSunken Then
'Sunken, so make the left and top dark-grey
      .Borders(xlEdgeLeft).Color = RGB(96, 96, 96)
      .Borders(xlEdgeTop).Color = RGB(96, 96, 96)
    Else
'Raised, so make the right and bottom dark-grey
      .Borders(xlEdgeRight).Color = RGB(96, 96, 96)
      .Borders(xlEdgeBottom).Color = RGB(96, 96, 96)
    End If
  End With
End Sub

```

CalculatedFields Collection

See the “PivotField Object, PivotFields Collection, and the CalculatedFields Collection” section.

CalculatedItems Collection

See the “PivotItem Object, PivotItems Collection, and the CalculatedItems Collection” section.

CalculatedMember Object and the CalculatedMembers Collection

The `CalculatedMembers` collection is a collection of all the `CalculatedMember` objects on the specified `PivotTable`. Each `CalculatedMember` object represents a calculated field, or calculated item.

CalculatedMembers Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

CalculatedMembers Collection Properties

Name	Returns	Description
Count	Long	Returns the number of objects in the collection
Item	CalculatedMember	Parameter: <code>Index As Variant</code> . Returns a single <code>CalculatedMember</code> object in the <code>CalculatedMembers</code> collection

CalculatedMembers Collection Methods

Name	Returns	Parameters	Description
Add	Calculated Member	<code>Name As String</code> , <code>Formula As String</code> , <code>[SolveOrder]</code> , <code>[Type]</code>	Adds a <code>CalculatedField</code> or <code>CalculatedItem</code> to a <code>PivotTable</code>

CalculatedMember Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

CalculatedMember Properties

Name	Returns	Description
Formula	String	Read-only. Returns the <code>CalculatedMember</code> 's formula in multi-dimensional expressions (MDX) syntax
IsValid	Boolean	Read-only. Indicates whether the specified <code>CalculatedMember</code> object has been successfully instantiated with the OLAP provider during the current session. Will return <code>True</code> even if the <code>PivotTable</code> is not connected to its data source
Name	String	Returns the name of the object

Name	Returns	Description
SolveOrder	Long	Read-only. Gets the value of the <code>CalculatedMember</code> 's MDX (multidimensional expression) argument (default is zero)
SourceName	String	Read-only. Gets the object's name as it appears in the original source data for the specified <code>PivotTable</code> report
Type	XlCalculated	Read-only. Gets the <code>CalculatedMember</code> object's type <code>MemberType</code>

CalculatedMember Methods

Name	Returns	Parameters	Description
Delete			Deletes the selected object

CalculatedMembers Collection and CalculatedMember Object Example

The following routine returns information about each `CalculatedMember` from the data source used by the `PivotTable` on the active worksheet. It returns messages if either the data source is not an OLAP type or there are no `CalculatedMembers`:

```

Sub ReturnCalculatedMembers()
    Dim lIcon As Long, lCount As Long
    Dim ptTable As PivotTable
    Dim oCalcMember As CalculatedMember
    Dim oCalcMembers As CalculatedMembers
    Dim sInfo As String
    'Set the reference to the PivotTable
    Set ptTable = ActiveSheet.PivotTables("PivotTable1")
    On Error Resume Next
    Set oCalcMembers = ptTable.CalculatedMembers
    On Error GoTo 0
    'Did we return a reference to Calculated Members?
    If Not oCalcMembers Is Nothing Then

    'If there's at least one Calculated Member initialize the Count and message
    variables
        If oCalcMembers.Count > 0 Then
            lCount = 1
            lIcon = vbInformation
        'Loop through each Calculated Member and store its name and formula
        For Each oCalcMember In oCalcMembers
            With oCalcMember
                sInfo = sInfo & lCount & ") " & .Name & ": " & .Formula
                lCount = lCount + 1
            End With
        Next oCalcMember

    Else

```

CalloutFormat Object

```
'It's a valid OLAP data source, but no Calculated Members are there
    lIcon = vbExclamation
    sInfo = "No Calculated Members found."
End If
Else

'oCalcMembers returned nothing. Not an OLAP data source
    lIcon = vbCritical
    sInfo = "No Calculated Members found. Data Source may not be OLAP type."
End If
MsgBox sInfo, lIcon, "Calculated Members"
End Sub
```

CalloutFormat Object

The `CalloutFormat` object corresponds to the line callouts on shapes. The parent of the `CalloutFormat` object is the `Shape` object.

CalloutFormat Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

CalloutFormat Properties

Name	Returns	Description
Accent	MsoTriState	Set/Get whether a vertical accent bar is used to separate the callout box from the line
Angle	MsoCallout AngleType	Set/Get the angle of the callout line in relation to the callout box
AutoAttach	MsoTriState	Set/Get whether a callout line automatically changes where it is attached to the callout box depending on where the line is pointing (left or right of the callout box)
AutoLength	MsoTriState	Read-only. Returns whether the callout line changes size automatically if the multisegment callout box is moved
Border	MsoTriState	Set/Get whether the callout box has a border around it
Drop	Single	Read-only. Returns the distance from the callout box to the spot where the callout line is pointing
DropType	MsoCallout DropType	Read-only. Returns the spot on the callout box that attaches to the callout line
Gap	Single	Set/Get the distance between the callout line's end and the callout box
Length	Single	Read-only. Returns the length of the first part of a callout line. <code>AutoLength</code> must be <code>False</code>
Type	MsoCallout Type	Set/Get the type of callout line used

CalloutFormat Methods

Name	Returns	Parameters	Description
AutomaticLength			Sets the AutoLength property to True
CustomDrop		Drop As Single	Uses the Drop parameter to set the distance from the callout box to the spot where the callout line is pointing
CustomLength		Length As Single	Sets the length of the first part of a callout line to the Length parameter and sets AutoLength to False
PresetDrop		DropType As MsoCallout DropType	Sets the spot on the callout box that attaches to the callout line, using the DropType parameter

CalloutFormat Object Example

This example applies the same formatting to all the callouts in a worksheet:

```

Sub FormatAllCallouts()
    Dim oShp As Shape
    Dim oCF As CalloutFormat
    'Loop through all the shapes in the sheet
    For Each oShp In ActiveSheet.Shapes

        On Error GoTo MyExit
        'Is this a callout?
        If oShp.Type = msoCallout Then

            'Yes - set its text box to autosize
            oShp.TextFrame.AutoSize = True

            'Get the CalloutFormat object
            Set oCF = oShp.Callout

            'Format the callout
            With oCF
                .Gap = 0
                .Border = msoFalse
                .Accent = msoTrue
                .Angle = msoCalloutAngle30
                .PresetDrop msoCalloutDropCenter
            End With
        End If

    Next
Exit Sub

```

CellFormat Object

```
MyExit:
    MsgBox "One or more of your Callouts do not have text"
End Sub
```

CellFormat Object

Represents both the `FindFormat` and `ReplaceFormat` property settings of the `Application` object, which are then used by the `Find` and `Replace` methods (respectively) of the `Range` object.

Set the `FindFormat` property settings before using the `Find` method to search for cell formats within a range. Set the `ReplaceFormat` property settings if you want the `Replace` method to replace formatting in cells. Any values specified in the `What` or `Replacement` arguments of either the `Find` or `Replace` methods will involve an `And` condition. For example, if you are searching for the word *Wrox* and have set the `FindFormat` property to search for **Bold**, only those cells containing both will be found.

When searching for formats, make sure the `SearchFormat` argument of the `Find` method is set to `True`. When replacing formats, make sure the `ReplaceFormat` argument of the `Replace` method is set to `True`.

When you want to search for formats only, make sure the `What` argument of the `Find` method contains nothing. When you only want to replace formats, make sure the `Replace` argument of the `Replace` method contains nothing.

When replacing one format with another, make sure you explicitly specify formats you no longer want. For example, if you are searching for cells containing both bold and red and want to replace both formats with just blue, you'll need to make sure you set the `bold` property of the `ReplaceFormat` property to `False`. If you don't, you'll end up with blue and bold text.

When you need to search or replace using different format settings (or none at all), be sure to either use the `Clear` method of the `CellFormat` object—if you've declared a variable as such—or directly access the `Clear` methods of the `FindFormat` and `ReplaceFormat` properties. Setting the `SearchFormat` and `ReplaceFormat` arguments to `False` for the `Find` and `Replace` methods will *not* prevent the `FindFormat` and/or `ReplaceFormat` settings from being used.

CellFormat Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

CellFormat Properties

Name	Returns	Description
<code>AddIndent</code>	<code>Variant</code>	Set/Get whether the text in a cell is automatically indented when the text alignment is set to equal distribution, either horizontally or vertically
<code>Borders</code>	<code>Borders</code>	Set/Get the search criteria based on the cell's border format

Name	Returns	Description
Font	Font	Set/Get the search criteria based on the cell's font format
Formula Hidden	Variant	Set/Get whether the formula will be hidden when the worksheet is protected. Returns <code>Null</code> if the specified range contains some cells with hidden formulas and some cells without
Horizontal Alignment	Variant	Set/Get the horizontal alignment for the specified object
IndentLevel	Variant	Set/Get the indent level for the cell or range
Interior	Interior	Set/Get the search criteria based on the cell's interior format
Locked	Variant	Set/Get whether cells in the range can be modified if the sheet is protected. Returns <code>Null</code> if only some of the cells in the range are locked
MergeCells	Variant	Returns <code>True</code> if the range or style contains merged cells
NumberFormat	Variant	Set/Get the number format associated with the cells in the range. <code>Null</code> if all the cells don't have the same format
NumberFormat Local	Variant	Set/Get the number format associated with the cells in the range, in the language of the end user. <code>Null</code> if all the cells don't have the same format
Orientation	Variant	Set/Get the text orientation for the cell text. A value from -90 to 90 degrees can be specified, or use an <code>XLOrientation</code> constant
ShrinkToFit	Variant	Set/Get whether the cell text will automatically shrink to fit the column width. Returns <code>Null</code> if the rows in the range have different <code>ShrinkToFit</code> properties
Vertical Alignment	Variant	Set/Get how the cells in the range are vertically aligned. Use the <code>XLVAlign</code> constants
WrapText	Variant	Set/Get whether cell text wraps in the cell. Returns <code>Null</code> if the cells in the range contain different text wrap properties

CellFormat Methods

Name	Description
Clear	Removes the criteria set in the <code>FindFormat</code> and <code>ReplaceFormat</code> properties

CellFormat Object Example

The following routine searches through the used range in the active worksheet, replacing any cells containing both a Tahoma font and a light blue background with Arial light green background:

```
Sub ReplaceFormats()  
    Dim oCellFindFormat As CellFormat  
    Dim oCellReplaceFormat As CellFormat  
    Dim rngReplace As Boolean, sMessage As String  
    'Define variables for Find and Replace formats  
    Set oCellFindFormat = Application.FindFormat  
    Set oCellReplaceFormat = Application.ReplaceFormat  
    'Set the Search criteria for the Find Formats  
    With oCellFindFormat  
        .Clear  
        .Font.Name = "Tahoma"  
        .Interior.ColorIndex = 34  
    End With  
    'Set the Replace criteria for the Replace Formats  
    With oCellReplaceFormat  
        .Clear  
        .Font.Name = "Arial"  
        .Interior.ColorIndex = 35  
    End With  
    'Perform the replace  
    ActiveSheet.UsedRange.Replace What:="", _  
        Replacement:="", SearchFormat:=True, ReplaceFormat:=True  
    'Reset the Find and Replace formats  
    oCellFindFormat.Clear  
    oCellReplaceFormat.Clear  
End Sub
```

Characters Object

The `Characters` object allows access to individual characters in a string of text. Characters can have some of the visual properties modified with this object. Possible parents of the `Characters` object are the `AxisTitle`, `ChartTitle`, `DataLabel`, and `Range` objects. Each of the parent objects can use the `Characters([Start], [Length])` property to access a part of their respective texts. The `Start` parameter can specify which character to start at, and the `Length` parameter can specify how many to take from the `Start` position.

Characters Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Characters Properties

Name	Returns	Description
<code>Caption</code>	String	Set/Get the full string contained in the <code>Characters</code> object
<code>Count</code>	Long	Read-only. Returns the number of characters in the object

Name	Returns	Description
Font	Font	Read-only. Returns an object allowing manipulation of the character's font
Phonetic Characters	String Characters	Set/Get the phonetic characters contained in the object
Text	String	Set/Get the full string contained in the Characters object

Characters Methods

Name	Returns	Parameters	Description
Delete	Variant		Deletes the characters in the collection
Insert	Variant	String As String	Replaces the characters in the collection with the specified string

Characters Object Example

This example formats all the capital letters in the active cell in red with 16-point bold text:

```

Sub FormatCellCapitals()
    Dim sText As String
    Dim oChars As Characters
    Dim i As Integer
    'Get the text of the active cell
    sText = ActiveCell.Text
    'Loop through the text
    For i = 1 To Len(sText)
        'Is this character a capital letter?
        If Asc(Mid(sText, i, 1)) > 64 And Asc(Mid(sText, i, 1)) < 91 Then

            'Yes, so get the Characters object
            Set oChars = ActiveCell.Characters(i, 1)

            'Format the Characters object in Red, 16pt Bold.
            With oChars
                .Font.Color = RGB(255, 0, 0)
                .Font.Size = 16
                .Font.Bold = True
            End With
        End If
    Next
End Sub

```

Chart Object and the Charts Collection

The `Charts` collection holds the collection of chart sheets in a workbook. The `Workbook` object is always the parent of the `Charts` collection. The `Charts` collection only holds the chart sheets. Individual charts

Charts Collection Properties and Methods

can also be embedded in worksheets and dialog sheets. The `Chart` objects in the `Charts` collection can be accessed using the `Item` property. The name of the chart can be specified either as a parameter to the `Item` property's parameter or an index number describing the position of the chart in the workbook (from left to right).

The `Chart` object allows access to all of the attributes of a specific chart in Excel. This includes chart formatting, chart types, and other charting properties. The `Chart` object also exposes events that can be used programmatically.

Charts Collection Properties and Methods

The `Charts` collection has a few properties and methods besides the typical collection attributes. These are listed in the following table.

Name	Returns	Description
<code>HPageBreaks</code>	<code>HPageBreaks</code>	Read-only. Returns a collection holding all the horizontal page breaks associated with the <code>Charts</code> collection
<code>Visible</code>	<code>Variant</code>	Set/Get whether the charts in the collection are visible. Also, you can set this to <code>xlVeryHidden</code> to prevent a user from making the charts in the collection visible
<code>VPageBreaks</code>	<code>VPageBreaks</code>	Read-only. Returns a collection holding all the vertical page breaks associated with the <code>Charts</code> collection
<code>Add</code>	<code>Chart</code>	Method. Parameters: <code>[Before]</code> , <code>[After]</code> , <code>[Count]</code> . Adds a chart to the collection. You can specify where the chart goes by choosing which sheet object will be before the new chart object (<code>Before</code> parameter) or after the new chart (<code>After</code> parameter). The <code>Count</code> parameter decides how many charts are created
<code>Copy</code>		Method. Parameters: <code>[Before]</code> , <code>[After]</code> . Adds a new copy of the currently active chart to the position specified at the <code>Before</code> or <code>After</code> parameters
<code>Delete</code>		Method. Deletes all the charts in the collection
<code>Move</code>		Method. Parameters: <code>[Before]</code> , <code>[After]</code> . Moves the current chart to the position specified by the parameters
<code>PrintOut</code>		Method. Parameters: <code>[From]</code> , <code>[To]</code> , <code>[Copies]</code> , <code>[Preview]</code> , <code>[ActivePrinter]</code> , <code>[PrintToFile]</code> , <code>[Collate]</code> , <code>[PrToFileName]</code> . Prints out the charts in the collection. The printer, number of copies, collation, and whether a print preview is desired can be specified with the parameters. Also, the sheets can be printed to a file using the <code>PrintToFile</code> and <code>PrToFileName</code> parameters. The <code>From</code> and <code>To</code> parameters can be used to specify the range of printed pages

Name	Returns	Description
PrintPreview		Method. Parameters: [EnableChanges]. Displays the current chart in the collection in a print preview mode. Set the EnableChanges parameter to False to disable the Margins and Setup buttons, hence not allowing the viewer to modify the chart's page setup
Select		Method. Parameters: [Replace]. Selects the current chart in the collection

Chart Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

Chart Properties

Name	Returns	Description
AutoScaling	Boolean	Set/Get whether Excel will stretch a 3D chart to match its 2D chart equivalent. RightAngleAxes must be true
BackWall	Walls	Read-only. Returns a Walls object allowing users to format the back wall of a 3D chart
BarShape	XlBarShape	Set/Get the basic shape used in 3D bar or column charts (for example, box, cylinder, pyramid, and so on)
ChartArea	ChartArea	Read-only. Returns the part of a chart containing axes, titles, legends, and formatting properties
ChartStyle	Variant	Set/Get a number from 1–48, indicating the chart style for the chart
ChartTitle	ChartTitle	Read-only. Returns an object manipulating the chart title's properties. Use with the HasTitle property
ChartType	XlChartType	Set/Get what the type of chart is. This property determines which other chart properties are valid. For example, if the ChartType is set to xl3DBarClustered, then the Bar3DGroup property can be used to access the chart group properties
CodeName	String	Read-only. Returns the programmatic name of the chart set at design-time in the VBA editor
DataTable	DataTable	Read-only. Returns an object to manipulate a chart's data table
DepthPercent	Long	Set/Get the percentage of a 3D chart's depth (y-axis) in relation to its width (x-axis)

Table continued on following page

Chart Properties

Name	Returns	Description
Display BlanksAs	XlDisplay BlanksAs	Set/Get how blank cells are treated when plotting data in a chart (for example, xlNotPlotted, xlZero, or xlInterpolated)
Elevation	Long	Set/Get the angle of elevation, in degrees, at which the viewer sees a 3D chart. Valid degrees vary depending on the type of 3D chart
Floor	Floor	Read-only. Returns an object with the formatting properties of the floor (base) of a 3D chart
GapDepth	Long	Set/Get the percentage depth of a data series in relation to the marker width
HasAxis	Variant	Parameters: [Index1], [Index2]. Set/Get whether axes exist for the chart. The parameters can be used to specify the axis type (using the xlAxisType constants with the first parameter) and the axis group (using the xlAxisGroup constants with the second parameter)
HasDataTable	Boolean	Set/Get whether a data table is associated (and therefore displayed). Use with the DataTable property
HasLegend	Boolean	Set/Get whether the chart has a legend. Use with the Legend property
HasTitle	Boolean	Set/Get whether the chart has a title. Use with the ChartTitle property
Height Percent	Long	Set/Get the percentage of a 3D chart's height (z-axis) in relation to its width (x-axis)
Hyperlinks	Hyperlinks	Read-only. Returns the collection of hyperlinks associated with the chart
Index	Long	Read-only. Returns the spot in the parent collection where the current chart is located
Legend	Legend	Read-only. Returns the formatting properties for a Legend. Use with the HasLegend property
MailEnvelope	MsoEnvelope	Set/Get the e-mail header for a document
Name	String	Set/Get the name of the chart
Next	Object	Read-only. Returns the next sheet in the workbook (from left to right) as an object
PageSetup	PageSetup	Read-only. Returns an object to manipulate the page setup properties for the chart
Perspective	Long	Sets the perspective, in degrees, at which a 3D chart will be viewed if the RightAngleAxes property is set to False

Name	Returns	Description
PivotLayout	PivotLayout	Read-only. Returns an object to manipulate the location of fields for a PivotChart report
PlotArea	PlotArea	Read-only. Returns an object to manipulate formatting, gridlines, data markers, and other visual items for the area where the chart is actually plotted. Inside the chart area
PlotBy	XlRowCol	Set/Get whether columns in the original data are used as individual data series (xlColumns), or if the rows in the original data are used as data series (xlRows)
PlotVisible Only	Boolean	Set/Get whether only visible cells are plotted or if invisible cells are plotted too (False)
Previous		Read-only. Returns the previous sheet in the workbook (from right to left) as an object
Protect Contents	Boolean	Read-only. Returns whether the chart and everything in it is protected from changes
ProtectData	Boolean	Set/Get whether the source data can be redirected for a chart
Protect Drawing Objects	Boolean	Read-only. Returns whether the shapes in the chart can be modified (ProtectDrawingObjects = False)
Protect GoalSeek	Boolean	Set/Get whether the user can modify the points on a chart with a mouse action
Protect Formatting	Boolean	Set/Get whether formatting can be changed for a chart
Protection Mode	Boolean	Read-only. Returns whether protection has been applied to the user interface. Even if a chart has user interface protection on, any VBA code associated with the chart can still be accessed
Protect Selection	Boolean	Set/Get whether parts of a chart can be selected and if shapes can be put into a chart
RightAngle Axes	Variant	Set/Get whether axes are fixed at right angles for 3D charts, even if the perspective of the chart changes
Rotation	Variant	Set/Get what angle of rotation around the z-axis, in degrees, the viewer sees on a 3D chart. Valid degrees vary depending on the type of 3D chart
Shapes	Shapes	Read-only. Returns all the shapes contained by the chart
ShowDataLabels OverMaximum	Boolean	Set to True, this property ensures that data labels are shown even if the data point exceeds the size of the axis; this property applies to 2D charts only

Table continued on following page

Chart Methods

Name	Returns	Description
SideWall	Walls	Read-only. Returns a Walls object allowing users to format the side wall of a 3D chart
Tab	Tab	Read-only. Returns a Tab object for a chart or a worksheet
Visible Visibility	XlSheet	Set/Get whether or not the chart is visible. The Visible property can also be set to xlVeryHidden to make the chart inaccessible to the end user
Walls	Walls	Read-only. Returns an object to manipulate the formatting of the walls on a 3D chart

Chart Methods

Name	Returns	Parameters	Description
Activate			Activates the chart, making it the ActiveChart
ApplyChart Template		FileName As String	Activates and applies a template file for the chart
ApplyData Labels		[Type As Xl DataLabels Type], [Legend Key], [Auto Text], [Has LeaderLines], [ShowSeries Name], [Show CategoryName], [ShowValue], [Show Percentage], [ShowBubble Size], [Separator]	Sets the point labels for a chart. The Type parameter specifies whether no label, a value, a percentage of the whole, or a category label is shown. The legend key can appear by the point by setting the LegendKey parameter to True
ApplyLayout		Layout As Long	Allows a user to apply any one of the predefined layouts shown in the Ribbon
Axes	Object	Type, AxisGroup As XlAxisGroup	Returns the Axis object or the Axes collection for the associated chart. The type of axis and the axis group can be specified with the parameters

Name	Returns	Parameters	Description
ChartGroups	Object	[Index]	Returns either a single chart group (ChartGroup) or a collection of chart groups (ChartGroups) for a chart
ChartObjects	Object	[Index]	Returns either a single embedded chart (ChartObject) or a collection of embedded charts (ChartObjects) in a chart
ChartWizard		[Source],	A single method to modify the key properties associated with a chart. Specify the properties that you want to change. The Source specifies the data source. Gallery specifies the chart type. Format can specify one of the 10 built-in chart auto-formats. The rest of the parameters set up how the source will be read, the source of category labels, the source of the series labels, whether a legend appears, and the titles of the chart and the axis. If Source is not specified, this method can only be used if the sheet containing the chart is active
CheckSpelling		[Custom	Checks the spelling of the text in the chart. A custom dictionary can be specified (CustomDictionary), all uppercase words can be ignored (IgnoreUppercase), and Excel can be set to display a list of suggestions (AlwaysSuggest)
ClearToMatchStyle			Resets the formatting for all chart elements to automatic
Copy		[Before], [After]	Adds a new copy of the chart to the position specified at the Before or After parameters
CopyPicture		[Appearance As XlPicture Appearance], [Format As Xl- CopyPicture Format], [Size As XlPicture Appearance]	Copies the chart into the clipboard as a picture. The Appearance parameter can be used to specify whether the picture is copied as it looks on the screen or when printed. The Format parameter can specify the type of picture that will be put into the clipboard. The Size parameter is used when dealing with chart sheets to describe the size of the picture
Delete			Deletes the chart

Table continued on following page

Chart Methods

Name	Returns	Parameters	Description
Deselect			Unselects the selected object within a chart. This is equivalent to pressing the Esc key while working in a chart
Evaluate	Variant	Name	Evaluates the Name string expression as if it were entered into a worksheet cell
Export String, [FilterName], [Interactive]	Boolean	Filename As	Saves the chart as a picture (jpg or gif format) with the name specified by Filename
ExportAsFixed Format	Boolean	Type As xlFixed FormatType, FileName As Variant, Quality As Variant, IncludeDoc Properties As Variant, IgnorePrint Areas As Variant From As Variant, To As Variant, OpenAfter Publish	Exports a file to a format specified by using the xlFixedFormatType constants
GetChart Elemer Long, ElementID As Long, Arg1 As Long, Arg2 As Long		x As Long, y As	Returns what is located at the coordinates x and y of the chart. Only the first two parameters are sent. Variables must be put in the last three parameters. After the method is run, the last three parameters can be checked for return values. The ElementID parameter will return one of the xlChartItem parameters. The Arg1 and Arg2 parameters may or may not hold data, depending on the type of element
Location xlChart Location, [Name]	Chart	Where As	Moves the chart to the location specified by the Where and Name parameters. The Where can specify if the chart is moving to become a chart sheet or an embedded object
Move [After]		[Before],	Moves the chart to the position specified by the parameters
OLEObjects	Object	[Index]	Returns either a single OLE object (OLEObject) or a collection of OLE objects (OLEObjects) for a chart

Name	Returns	Parameters	Description
Paste		[Type]	Pastes the data or pictures from the clipboard into the chart. The <code>Type</code> parameter can be used to specify if only formats, formulas, or everything is pasted
PrintOut		[From], [To], [Copies], [Preview], [ActivePrinter], [PrintToFile], [Collate], [PrToFile Name]	Prints the chart. The printer, number of copies, collation, and whether a print preview is desired can be specified with the parameters. Also, the sheets can be printed to a file by using the <code>PrintToFile</code> and <code>PrToFileName</code> parameters. The <code>From</code> and <code>To</code> parameters can be used to specify the range of printed pages
PrintPreview		[EnableChanges]	Displays the current chart in the collection in a print preview mode. Set the <code>EnableChanges</code> parameter to <code>False</code> to disable the <code>Margins</code> and <code>Setup</code> buttons, hence not allowing the viewer to modify the page setup
Protect		[Password], [DrawingObject], [Contents], [Scenarios], [User Interface Only]	Protects the chart from changes. A case-sensitive <code>Password</code> can be specified. Also, determines whether shapes are protected (<code>DrawingObjects</code>), the entire contents are protected (<code>Contents</code>), or only the user interface is protected (<code>UserInterfaceOnly</code>)
Refresh			Refreshes the chart with the data source
SaveAs		Filename As String, [FileFormat], [Password], [WriteRes Password], [ReadOnly Recommended], [CreateBackup] [AddToMru], [TextCodepage] [TextVisual Layout], [Local]	Saves the current chart into a new workbook with the filename specified by the <code>Filename</code> parameter. A file format, password, write-only password, creation of backup files, and other properties of the saved file can be specified with the parameters

Table continued on following page

Chart Events

Name	Returns	Parameters	Description
SaveChart Template		Filename As String	Saves the specified chart as a custom chart template. The chart template is saved to the template directory unless a location is explicitly provided, telling Excel to use that location instead
Select		[Replace]	Selects the chart
Series Collection	Object	[Index]	Returns either a single series (<i>Series</i>) or a collection of series (<i>SeriesCollection</i>) for a chart
SetBackground Picture String		FileName As String	Sets the chart's background to the picture specified by the <i>FileName</i> parameter
SetDefault Chart String		FileName As String	Specifies the name of the chart template that is used when new charts are created
SetElement		Element As MsoChart- ElementType	Sets/Gets the elements on a chart. Use the <i>MsoChartElementType</i> constants to identify the elements you want to set/get
SetSourceData Range, [PlotBy]		Source As String	Sets the source of the chart's data to the range specified by the <i>Source</i> parameter. The <i>PlotBy</i> parameter uses the <i>XlRowCol</i> constants to choose whether rows or columns of data will be plotted
Unprotect		[Password]	Deletes the protection set up for a chart. If the chart was protected with a password, the password must be specified now

Chart Events

Name	Parameters	Description
Activate		Triggered when a chart is made to have focus
BeforeDouble Click	ElementID As String	Triggered just before a user double-clicks a chart. The element that was double-clicked in the chart is passed into the event procedure as <i>ElementID</i> . The <i>Arg1</i> and <i>Arg2</i> parameters may or may not hold values depending on the <i>ElementID</i> . The double-click action can be canceled by setting the <i>Cancel</i> parameter to <i>True</i>
XlChartItem, Arg1 As Long, Arg2 As Long, Cancel As Boolean		

Name	Parameters	Description
BeforeRightClick	Cancel As Boolean	Triggered just before a user right-clicks a chart. The right-click action can be canceled by setting the <code>Cancel</code> parameter to <code>True</code>
Calculate		Triggered after new or changed data is plotted on the chart
Deactivate		Triggered when the chart loses focus
DragOver		Triggered when a cell range is dragged on top of a chart. Typically used to change the mouse pointer or give a status message
DragPlot		Triggered when a cell range is dropped onto a chart. Typically used to modify chart attributes
MouseDown XlMouse Button, Shift As Long, x As Long, y As Long	Button As	Triggered when the mouse button is pressed down on a chart. Which mouse button is pressed is passed in with the <code>Button</code> parameter. The <code>Shift</code> parameter holds information regarding the state of the Shift, Ctrl, and Alt keys. The <code>x</code> and <code>y</code> parameters hold the <code>x</code> and <code>y</code> coordinates of the mouse pointer
MouseMove XlMouse Button, Shift As Long, x As Long, y As Long	Button As	Triggered when the mouse is moved on a chart. Which mouse button is pressed is passed in with the <code>Button</code> parameter. The <code>Shift</code> parameter holds information regarding the state of the Shift, Ctrl, and Alt keys. The <code>x</code> and <code>y</code> parameters hold the <code>x</code> and <code>y</code> coordinates of the mouse pointer
MouseUp XlMouse Button, Shift As Long, x As Long, y As Long	Button As	Triggered when the mouse button is released on a chart. Which mouse button is pressed is passed in with the <code>Button</code> parameter. The <code>Shift</code> parameter holds information regarding the state of the Shift, Ctrl, and Alt keys. The <code>x</code> and <code>y</code> parameters hold the <code>x</code> and <code>y</code> coordinates of the mouse pointer
Resize		Triggered when the chart is resized
Select XlChartItem, Arg1 As Long, Arg2 As Long	ElementID As	Triggered when one of the elements in a chart is selected. The element that was selected in the chart is passed into the event procedure as <code>ElementID</code> . The <code>Arg1</code> and <code>Arg2</code> parameters may or may not hold values depending on the <code>ElementID</code>
SeriesChange As Long, PointIndex As Long	SeriesIndex	Triggered when the value of a point on a chart is changed. <code>SeriesIndex</code> returns the location of the series in the chart series collection. <code>PointIndex</code> returns the point location in the series

Chart Object and the Charts Collection Example

Chart Object and the Charts Collection Example

This example creates a 3D chart from a given range, formats it, and saves a picture of it as a .jpg image:

```
Sub CreateAndExportChart()  
    Dim oCht As Chart  
    'Create a new (blank) chart  
    Set oCht = Charts.Add  
    'Format the chart  
    With oCht  
        .ChartType = xl3DColumnStacked  
    'Set the data source and plot by columns  
        .SetSourceData Source:=Range("Sheet1!$B$2:$D$8"), PlotBy:=xlColumns  
    'Create a new sheet for the chart  
        .Location Where:=xlLocationAsNewSheet  
    'Size and shape matches the window it's in  
        .SizeWithWindow = True  
    'Turn of stretching of chart  
        .AutoScaling = False  
    'Set up a title  
        .HasTitle = True  
        .ChartTitle.Caption = "Main Chart"  
    'No titles for the axes  
        .Axes(xlCategory).HasTitle = False  
        .Axes(xlValue).HasTitle = False  
    'Set the 3D view of the chart  
        .RightAngleAxes = False  
        .Elevation = 50 'degrees  
        .Perspective = 30 'degrees  
        .Rotation = 20 'degrees  
        .HeightPercent = 100  
    'No data labels should appear  
        .ApplyDataLabels Type:=xlDataLabelsShowNone  
    'Save a picture of the chart as a jpg image  
        .Export "c:\\" & .Name & ".jpg", "jpg", False  
    End With  
End Sub
```

ChartArea Object

The `ChartArea` object contains the formatting options associated with a chart area. For 2D charts, `ChartArea` includes the axes, axes titles, and chart titles. For 3D charts, `ChartArea` includes the chart title and its legend. The part of the chart where data is plotted (plot area) is not part of the `ChartArea` object. Please see the `PlotArea` object for formatting related to the plot area. The parent of the `ChartArea` is always the `Chart` object.

ChartArea Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ChartArea Properties

Name	Returns	Description
AutoScaleFont	Variant	Set/Get whether the font size changes in the ChartArea whenever the Chart changes sizes
Format	ChartFormat	Read-only. Returns the ChartFormat object, which controls the line, fill, and effect formatting for the chart area
Height	Double	Set/Get the height of the chart area in points
Left	Double	Set/Get the left edge of the chart area in relation to the chart in points
Name	String	Read-only. Returns the name of the chart area
Shadow	Boolean	Set/Get whether a shadow effect appears around the chart area
Top	Double	Set/Get the top edge of the chart area in relation to the chart in points
Width	Double	Set/Get the width of the chart area in points

ChartArea Methods

Name	Returns	Parameters	Description
Clear	Variant		Clears the chart area
ClearContents	Variant		Clears the data from the chart area without affecting formatting
ClearFormats	Variant		Clears the formatting from the chart area without affecting the data
Copy	Variant		Copies the chart area into the clipboard
Select	Variant		Activates and selects the chart area

ChartArea Object Example

This example sizes the chart area for a chart and gives the chart a shadow:

```
Sub SetChartColorFormat()
    With Worksheets("Sheet4").ChartObjects("Chart 2").Chart
        .ChartArea.Height = 300
        .ChartArea.Width = 500
        .ChartArea.Shadow = True
    End With
End Sub
```

ChartColorFormat Object

ChartColorFormat Object

The `ChartColorFormat` object describes a color of the parent `ChartFillFormat`. For example, the `ChartFillFormat` object contains a `BackColor` property that returns a `ChartColorFormat` object to set the color.

ChartColorFormat Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ChartColorFormat Properties

Name	Returns	Description
<code>RGB</code>	<code>Long</code>	Read-only. Returns the RGB value associated with the color
<code>SchemeColor</code>	<code>Long</code>	Set/Get the color of <code>ChartColorFormat</code> using an index value corresponding to the current color scheme
<code>Type</code>	<code>Long</code>	Read-only. Returns whether the color is an RGB, mixed, or scheme type

ChartFillFormat Object

The `ChartFillFormat` object represents the fill formatting associated with its parent object. This object allows manipulation of foreground colors, background colors, and patterns associated with the parent object.

ChartFillFormat Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ChartFillFormat Properties

Name	Returns	Description
<code>BackColor</code>	<code>ChartColorFormat</code>	Read-only. Returns the background color through the <code>ChartColorFormat</code> object
<code>ForeColor</code>	<code>ChartColorFormat</code>	Read-only. Returns the foreground color through the <code>ChartColorFormat</code> object
<code>GradientColor Type</code>	<code>MsoGradient ColorType</code>	Read-only. Returns what type of gradient fill color concept is used
<code>GradientDegree</code>	<code>Single</code>	Read-only. Returns how dark or light the gradient fill is
<code>GradientStyle</code>	<code>MsoGradientStyle</code>	Read-only. Returns the orientation of the gradient that is used

Name	Returns	Description
GradientVariant	Long	Read-only. Returns the variant used for the gradient from the center
Pattern	MsoPatternType	Set/Get the pattern used for the fill, if any is used
PresetGradientType	MsoPresetGradientType	Read-only. Returns the type of gradient that is used
PresetTexture	MsoPresetTexture	Read-only. Returns the non-custom texture of the fill
TextureName	String	Read-only. Returns the custom texture name of the fill
TextureType	MsoTextureType	Read-only. Returns whether the texture is custom, preset, or mixed
Type	MsoFillType	Returns an MsoFillType constant that determines how transparent the fill is. From 0 (opaque) to 1 (clear)
Visible	MsoTriState	Set/Get whether the fill is a texture, gradient, solid, background, picture, or mixed

ChartFillFormat Methods

Name	Returns	Parameters	Description
OneColorGradient		Style As MsoGradientStyle, Variant As Long, Degree As Single	Set the style, variant, and degree for a one-color gradient fill
Patterned		Pattern As MsoPatternType	Set the pattern for a fill
PresetGradient		Style As MsoGradientStyle, Variant As Long, PresetGradientType As MsoPresetGradientType	Choose the style, variant, and preset gradient type for a gradient fill
PresetTextured		PresetTexture As MsoPresetTexture	Set the preset texture for a fill
Solid			Set the fill to a solid color
TwoColorGradient		Style As MsoGradientStyle, Variant As Long	Set the style for a two-color gradient fill

Table continued on following page

ChartFillFormat Object Example

Name	Returns	Parameters	Description
UserPicture		[PictureFile], [Picture Format], [PictureStackUnit], [PicturePlacement]	Set the fill to the picture in the PictureFile format
UserTextured		TextureFile As String	Set the custom texture for a fill with the TextureFile format

ChartFillFormat Object Example

```
Sub FormatPlotArea()  
    Dim oCFF As ChartFillFormat  
    'Get the ChartFillFormat for the plot area  
    Set oCFF = ActiveSheet.ChartObjects(1).Chart.PlotArea.Fill  
    'Format the fill area  
    With oCFF  
        .TwoColorGradient Style:=msoGradientDiagonalUp, Variant:=1  
        .Visible = True  
        .ForeColor.SchemeColor = 6  
        .BackColor.SchemeColor = 7  
    End With  
End Sub
```

ChartFormat Object

The `ChartFormat` object allows access to the new Office Art formatting options that are available in Excel 2007. Using this object, you can apply many of the new graphics to chart elements via VBA formatting.

ChartFormat Common Properties

The `Parent`, `Creator`, and `Application` properties are defined at the beginning of this appendix.

ChartFormat Properties

Name	Returns	Description
Fill	FillFormat	Read-only. Returns the fill formatting properties through the <code>FillFormat</code> object
Glow	GlowFormat	Read-only. Returns the glow formatting properties through the <code>GlowFormat</code> object
Line	LineFormat	Read-only. Returns the line formatting properties through the <code>LineFormat</code> object
PictureFormat	PictureFormat	Read-only. Returns the <code>PictureFormat</code> object for a chart that contains pictures
Shadow	ShadowFormat	Read-only. Returns the shadow formatting properties through the <code>ShadowFormat</code> object

Name	Returns	Description
SoftEdge	SoftEdgeFormat	Read-only. Returns the <code>SoftEdge</code> formatting properties through the <code>SoftEdgeFormat</code> object
TextFrame2	TextFrame2	Read-only. Returns the text formatting properties through the <code>TextFrame2Format</code> object
ThreeD	ThreeDFormat	Read-only. Returns the 3D effects formatting properties through the <code>ThreeDFormat</code> object

ChartGroup Object and the ChartGroups Collection

The `ChartGroups` collection holds all the plotting information associated with the parent chart. A chart can have more than one `ChartGroup` associated with it. For example, a single chart can contain both a line and a bar chart associated with it. The `ChartGroups` property of the `Chart` object can be used to access the `ChartGroups` collection.

The parent of the `ChartGroups` collection or the `ChartGroup` object is the `Chart` object.

The `ChartGroup` object includes all of the plotted points associated with a particular chart type. A `ChartGroup` can hold many series of points (each column or row of the original data). Each series can contain many points (each cell of the original data). A `Chart` can contain more than one `ChartGroup` associated with it.

ChartGroup Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ChartGroup Properties

Name	Returns	Description
AxisGroup	XlAxisGroup	Set/Get whether the chart group is primary or secondary
BubbleScale	Long	Set/Get the percentage increase in the size of bubbles from the default size. Valid values from 0 to 300 percent. Valid only for bubble chart group
DoughnutHole Size	Long	Set/Get how large the hole in a doughnut chart group is. The value is a percentage of the size of the chart. Valid values from 10 to 90 percent. Valid only on doughnut chart groups
DownBars	DownBars	Read-only. Returns an object to manipulate the formatting options of down bars on a line chart group. Valid only on line chart groups. Use with the <code>HasUpDownBars</code> property

Table continued on following page

ChartGroup Properties

Name	Returns	Description
DropLines	DropLines	Read-only. Returns an object to manipulate the formatting options of drop lines on a line or area chart group. Valid only on line or area chart groups. Use with the HasDropLines property
FirstSlice Angle	Long	Set/Get what angle to use for the first slice of a pie or doughnut chart groups (the first data point plotted on the chart)
GapWidth	Long	Set/Get how big to make the gap between the columns of different data series. Also, when dealing with Bar of Pie charts or Pie of Pie charts, the GapWidth describes the distance from the main chart to the secondary chart (when the ChartType is xlPieOfPie or xlBarOfPie for the parent chart)
Has3Dshading	Boolean	Set/Get whether 3D shading is applied to the chart group visuals
HasDropLines	Boolean	Set/Get whether the chart group has drop lines. Use with the DownLines property
HasHiLoLines	Boolean	Set/Get whether the chart group has high-low lines. Use with the HiLoLines property
HasRadarAxis Labels	Boolean	Set/Get whether the chart as axis labels. Applies only to radar charts
HasSeries Lines	Boolean	Set/Get whether the chart group has series lines. Use with the SeriesLines property
HasUpDownBars	Boolean	Set/Get whether the chart group has up and down bars. Use with the DownBars and UpBars property
HiLoLines	HiLoLines	Read-only. Returns an object to manipulate the formatting of high-low lines in a line chart. Valid only for line charts
Index	Long	Read-only. Returns the spot in the parent collection where the current ChartGroup object is located
Overlap	Long	Set/Get whether bars and columns in a series will overlap each other or have a gap between them. A value from -100 to 100 can be specified, where -100 will put a gap between each bar or column equal to the bar or column width, and 100 will stack the bars or columns on top of each other. Valid only for 2D bar and column chart groups

Name	Returns	Description
RadarAxis Labels	TickLabels	Read-only. Returns an object to manipulate the formatting and labels associated with radar axis labels. Valid only for radar chart groups
SecondPlot Size	Long	Set/Get the percentage of size of the secondary part of a Pie of Pie or Bar of Pie chart group as a percentage of the main Pie
SeriesLines	SeriesLines	Read-only. Returns an object to manipulate the formatting associated with the series lines in a chart group. A series line connects same series of data appearing in a stacked column chart groups, stacked bar chart groups, Pie of Pie chart groups, or Bar of Pie chart groups. Use with the HasSeriesLines property
ShowNegative Bubbles	Boolean	Set/Get whether bubbles with negative data values are shown. Valid only on bubble chart groups
Size Represents	XlSizeRepresents	Set/Get whether the value of the data points is represented by the size or the area of bubbles on a bubble chart group. Valid only on bubble chart groups
SplitType	XlChartSplitType	Set/Get how the two charts in Pie of Pie chart group and Bar of Pie chart group are split up. For example, the chart can be split by percentage of value (xlSplitByPercentValue) or by value (xlSplitByValue)
SplitValue	Variant	Set/Get the value that will be combined in the main pie chart but split up in the secondary chart in a Pie of Pie or Bar of Pie chart group
UpBars	UpBars	Read-only. Returns an object to manipulate the formatting options of up bars on a line chart group. Valid only on line chart groups. Use with the HasUpDownBars property
VaryBy Categories	Boolean	Set/Get whether different colors are assigned to different categories in a single series of a chart group. The chart can only contain a single data series for this to work

ChartGroup Methods

Name	Returns	Parameters	Description
Series Collection	Object	[Index]	Returns either a single series (Series) or a collection of series (SeriesCollection) for a chart

ChartGroup Object and the ChartGroups Collection Example

ChartGroup Object and the ChartGroups Collection Example

This sets the gap width of all column groups in the chart to 10% and sets each column to have a different color:

```
Sub FormatColumns()  
    Dim oCht As Chart  
    Dim oCG As ChartGroup  
    For Each oCG In Charts(1).ColumnGroups  
        oCG.GapWidth = 10  
        oCG.VaryByCategories = True  
    Next  
End Sub
```

ChartObject Object and the ChartObjects Collection

The `ChartObjects` collection holds all of the embedded `Chart` objects in a worksheet, chart sheet, or dialog sheet. This collection does not include the actual chart sheets themselves. Chart sheets can be accessed through the `Charts` collection. Each `Chart` in the `ChartObjects` collection is accessed through the `ChartObject` object. The `ChartObject` acts as a wrapper for the embedded chart itself. The `Chart` property of the `ChartObject` is used to access the actual chart. The `ChartObject` object also contains properties to modify the formatting of the embedded chart (for example, `Height` and `Width`).

The `ChartObjects` collection contains many properties besides the typical collection attributes. These properties are listed next.

ChartObjects Collection Properties and Methods

Name	Returns	Description
<code>Enabled</code>	<code>Boolean</code>	Set/Get whether any macros associated with each <code>ChartObject</code> object in the collection can be triggered by the user
<code>Height</code>	<code>Double</code>	Set/Get the height of the <code>ChartObject</code> in the collection if there is only one object in the collection
<code>Left</code>	<code>Double</code>	Set/Get the distance from the left edge of the <code>ChartObject</code> to the left edge of the parent sheet. This property only works if there is only one <code>ChartObject</code> in the collection
<code>Locked</code>	<code>Boolean</code>	Set/Get whether the <code>ChartObject</code> is locked when the parent sheet is protected. This property only works if there is only one <code>ChartObject</code> in the collection
<code>Placement</code>	<code>XlPlacement</code>	Set/Get how the <code>ChartObject</code> object is anchored to the sheet (for example, free floating, move with cells). Use the <code>XlPlacement</code> constants to set this property. This property only works if there is only one <code>ChartObject</code> in the collection

ChartObjects Collection Properties and Methods

Name	Returns	Description
PrintObject	Boolean	Set/Get whether the embedded chart on the sheet will be printed when the sheet is printed. This property only works if there is only one ChartObject in the collection
ProtectChartObject	Boolean	Set to True, this property will ensure that the embedded chart cannot be moved, resized, or deleted through the user interface
RoundedCorners	Boolean	Set/Get whether the corners of the embedded chart are rounded (True) or right angles (False). This property only works if there is only one ChartObject in the collection
Shadow	Boolean	Set/Get whether a shadow appears around the embedded chart. This property only works if there is only one ChartObject in the collection
ShapeRange	ShapeRange	Read-only. Returns the ChartObjects in the collection as Shape objects
Top	Double	Set/Get the distance from top edge of the ChartObject to the top of the parent sheet. This property only works if there is only one ChartObject object in the collection
Visible	Boolean	Set/Get whether all the ChartObject objects in the collection are visible
Width	Double	Set/Get the width of the ChartObject in the collection if there is only one ChartObject object in the collection
Add	ChartObject	Method. Parameters: Left As Double, Top As Double, Width As Double, Height As Double. Adds a ChartObject to the collection of ChartObjects. The position of the new ChartObject can be specified by using the Left, Top, Width, and Height parameters
BringToFront	VARIANT	Method. Brings all the ChartObject objects in the collection to the front of all the other objects
Copy	VARIANT	Method. Copies all the ChartObject objects in the collection into the clipboard

Table continued on following page

ChartObject Common Properties

Name	Returns	Description
CopyPicture	Variant	Method. Parameters: Appearance As XlPictureAppearance, Format As XlCopyPictureFormat. Copies the Chart objects in the collection into the clipboard as a picture. The Appearance parameter can be used to specify whether the picture is copied as it looks on the screen or when printed. The Format parameter can specify the type of picture that will be put into the clipboard
Cut	Variant	Cuts an embedded chart to the clipboard
Delete	Variant	Method. Deletes all the ChartObject objects in the collection into the clipboard
Duplicate		Method. Duplicates all the ChartObject objects in the collection into the parent sheet (for example, if you had two ChartObject objects in the parent sheet and used this method, then you would have four ChartObject objects)
Select	Variant	Method. Parameters: [Replace]. Selects all the ChartObject objects in the collection
SendToBack	Variant	Method. Brings the ChartObject objects in the collection to the back of other objects

ChartObject Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

ChartObject Properties

Name	Returns	Description
BottomRightCell	Range	Read-only. Returns the single cell range located under the lower-right corner of the ChartObject
Chart	Chart	Read-only. Returns the actual chart associated with the ChartObject
Enabled	Boolean	Set/Get whether a macro associated with the ChartObject is capable of being triggered
Height	Double	Set/Get the height of embedded chart
Index	Long	Read-only. Returns the position of the ChartObject among the parent collection
Left	Double	Set/Get the distance from the left edge of the ChartObject to the left edge of the parent sheet
Locked	Boolean	Set/Get whether the ChartObject is locked when the parent sheet is protected

Name	Returns	Description
Name	String	Set/Get the name of the ChartObject
Placement	XlPlacement	Set/Get how the ChartObject object is anchored to the sheet (for example, free floating, move with cells). Use the XlPlacement constants to set this property
PrintObject	Boolean	Set/Get whether the embedded chart on the sheet will be printed when the sheet is printed
ProtectChartObject	Boolean	Set/Get whether the embedded chart can change sizes, can be moved, or can be deleted from the parent sheet
RoundedCorners	Boolean	Set/Get whether the corners of the embedded chart are rounded (True) or right angles (False)
Shadow	Boolean	Set/Get whether a shadow appears around the embedded chart
ShapeRange	ShapeRange	Read-only. Returns the ChartObject as a Shape object
Top	Double	Set/Get the distance from top edge of the ChartObject to the top of the parent sheet
TopLeftCell	Range	Read-only. Returns the single cell range located above the top-left corner of the ChartObject
Visible	Boolean	Set/Get whether the ChartObject object is visible
Width	Double	Set/Get the width of embedded chart
ZOrder	Long	Read-only. Returns the position of the embedded chart among all the other objects on the sheet. The ZOrder also matches the location of the ChartObject in the parent collection

ChartObject Methods

Name	Returns	Parameters	Description
Activate	Variant		Makes the embedded chart the active chart
BringToFront	Variant		Brings the embedded chart to the front of all the other objects on the sheet. Changes the ZOrder
Copy	Variant		Copies the embedded chart into the clipboard

Table continued on following page

ChartObject Object and the ChartObjects Collection Example

Name	Returns	Parameters	Description
CopyPicture	Variant	Appearance As XlPicture Appearance, Format As XlCopyPicture Format	Copies the Chart object into the clipboard as a picture. The Appearance parameter can be used to specify whether the picture is copied as it looks on the screen or when printed. The Format parameter can specify the type of picture that will be put into the clipboard. The Size parameter is used when dealing with chart sheets to describe the size of the picture
Cut	Variant		Cuts an embedded chart into the clipboard
Delete	Variant		Deletes the embedded chart from the sheet
Duplicate			Duplicates the embedded chart and places the duplicate in the same parent sheet
Select	Variant	[Replace]	Sets focus to the embedded chart
SendToBack	Variant		Sends the embedded object to the back of the other objects on the sheet

ChartObject Object and the ChartObjects Collection Example

This example creates .jpg images from all the embedded charts in the active worksheet:

```
Sub ExportChartObjects()  
    Dim oCO As ChartObject  
    For Each oCO In ActiveSheet.ChartObjects  
        'Export the chart as a jpg image, giving it the  
        'name of the embedded object  
        oCO.Chart.Export "c:\" & oCO.Name & ".jpg", "jpg"  
    Next  
End Sub
```

ChartTitle Object

The ChartTitle object contains all of the text and formatting associated with a chart's title. The parent of the ChartTitle object is the Chart object. This object is usually used along with the HasTitle property of the parent Chart object.

ChartTitle Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

ChartTitle Properties

Name	Returns	Description
AutoScaleFont	Variant	Set/Get whether the font size will change automatically if the parent chart changes sizes
Caption	String	Set/Get the chart title's text
Characters	Characters	Read-only. Parameters: [Start], [Length]. Returns an object containing all the characters in the chart title. Allows manipulation on a character-by-character basis
Format	ChartFormat	Returns the ChartFormat object, which controls the line, fill, and effect formatting for the chart area
Horizontal Alignment	xlAlign	Set/Get how the chart title is horizontally aligned. Use the xlAlign constants
IncludeInLayout	Boolean	Set to True, this property ensures that the chart title will occupy the chart layout space when a chart layout is being determined
Left	Double	Set/Get the distance from the left edge of the chart title text area to the chart's left edge
Name	String	Read-only. Returns the name of the chart title object
Orientation	XlOrientation	Set/Get the angle of the text for the chart title. The value can be either in degrees (from -90 to 90) or one of the XlOrientation constants
Position	xlChartElement Position	Set/Get the position of the chart title by using the xlChartElementPosition constants
ReadingOrder	Long	Set/Get how the text is read (from left to right or right to left). Only applicable in appropriate languages
Shadow	Boolean	Set/Get whether the chart title has a shadow effect
Text	String	Set/Get the chart title's text
Top	Double	Set/Get the distance from the top edge of the chart title text area to the chart's top edge
Vertical Alignment	xlVAlign	Set/Get how you want the chart title to be horizontally aligned. Use the xlVAlign constants

ChartTitle Methods

ChartTitle Methods

Name	Returns	Parameters	Description
Delete	VARIANT		Deletes the chart title from the chart
Select	VARIANT		Selects the chart title on the chart

ChartTitle Object Example

This example adds a chart title to a chart and formats it:

```
Sub AddAndFormatChartTitle()  
    Dim oCT As ChartTitle  
    'Make sure the chart has a title  
    Charts(1).HasTitle = True  
    'Get the ChartTitle object  
    Set oCT = Charts(1).ChartTitle  
    'Format the chart title  
    With oCT  
        .Caption = "Hello World"  
        .Font.Name = "Times New Roman"  
        .Font.Size = 16  
        .Characters(1, 1).Font.Color = RGB(255, 0, 0)  
        .Characters(7, 1).Font.Color = RGB(255, 0, 0)  
        .Border.LineStyle = xlContinuous  
        .Border.Weight = xlThin  
        .Shadow = True  
    End With  
End Sub
```

ChartView Object

The `ChartView` object is returned by the `Sheets` collection and allows you to focus in on a Chart sheet.

ChartView Common Properties

The `Application` and `Parent` properties are defined at the beginning of this appendix.

ChartView Properties

Name	Returns	Description
Sheet	Sheet	Read-only. Returns the sheet name for the specified <code>ChartView</code> object

ColorFormat Object

The `ColorFormat` object describes a single color used by the parent object. Possible parents of the `ColorFormat` object are the `FillFormat`, `LineFormat`, `ShadowFormat`, and `ThreeDFormat` objects.

ColorFormat Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ColorFormat Properties

Name	Returns	Description
<code>ObjectThemeColor</code>	<code>MsoThemeColorIndex</code>	Set/Get the color of the color mapped to the theme color scheme by using the <code>MsoThemeColorIndex</code> constants
<code>RGB</code>	<code>Long</code>	Set/Get the red-green-blue value associated with the color
<code>SchemeColor</code>	<code>Integer</code>	Set/Get the color of the <code>ColorFormat</code> using an index value corresponding to the current color scheme
<code>TintAndShade</code>	<code>Single</code>	Set/Get a value that lightens or darkens the color of a specified shape. The values can be from -1 (darkest) to 1 (lightest). Zero is neutral
<code>Type</code>	<code>MsoColorType</code>	Read-only. Returns whether the color is an RGB, mixed, or scheme type

ColorFormat Object Example

Set the `ForeColor` of a shape's fill effect:

```
Sub FormatShapeColor()
    Dim oShp As Shape
    Dim oCF As ColorFormat
    Set oShp = ActiveSheet.Shapes(1)
    Set oCF = oShp.Fill.ForeColor
    oCF.SchemeColor = 53
End Sub
```

ColorScale Object

The `ColorScale` object allows you to create a color scale formatting rule by using either the `Add` or `AddColorScale` method of the `FormatConditions` collection. You can apply a two-color or three-color scale to a range of data by setting the properties of the `ColorScaleCriteria` to minimum, maximum, and midpoint thresholds. The `ColorScaleCriteria` object is a child of the `ColorScale` object.

ColorScale Properties

The `Application`, `Parent`, and `Creator` properties are defined at the beginning of this appendix.

ColorScale Properties

ColorScale Properties

Name	Returns	Description
AppliesTo	Range	Set/Get the range that is affected by the formatting rule
ColorScaleCriteria	ColorScaleCriteria	Returns a ColorScaleCriteria object, which is used to specify the type of scale to be used (two or three), the values that are to be evaluated, and the color of threshold criteria used. Read-only
FormatRow	Boolean	Set/Get the Boolean value specifying if the entire Excel table row should be formatted. The default value is False
Formula	String	Set/Get a string representing a formula that determines the values that are to be evaluated in the conditional formatting rule
Priority	Long	Set/Get the priority value of a conditional formatting rule, determining the order of evaluation when other rules are in effect
PTCondition	Boolean	Read-only. Indicates whether the formatting rule is applied to a PivotTable chart
ScopeType	xlPivotConditionScope	Set/Get the scope of the formatting rule when applied to a PivotTable chart. Use the xlPivotConditionScope constants
StopifTrue	Boolean	Set/Get a Boolean value that determines if additional formatting rules should be applied if the current rule evaluates to True. The default value is True
Type	xlFormatConditionType	Read-only. Returns an xlFormatConditionType constant that specifies the type of conditional formatting being applied. This object will always return a value of 12 since it corresponds to the xlAboveAverageCondition

ColorScale Methods

Name	Parameters	Description
Delete		Deletes the object
ModifyAppliesToRange	Range As Range	Sets the range for which the formatting rule will be applied
SetFirstPriority		Sets the priority value for the formatting rule so that it is evaluated before all other rules on the worksheet
SetLastPriority		Sets the priority value for the formatting rule so that it is evaluated after all other rules on the worksheet

ColorScale Object Example

This example adds a three-color scale formatting rule to a specified range:

```
Sub CreateColorScale()
Dim oColorScale As ColorScale
'Add a three-color scale
Set oColorScale = Range("F6:F16").FormatConditions.AddColorScale(ColorScaleType:=3)
'Set the minimum threshold to the lowest value in the range
'Set the color for the minimum threshold
    oColorScale.ColorScaleCriteria(1).Type = xlConditionValueLowestValue
    oColorScale.ColorScaleCriteria(1).FormatColor.Color = 7039480
'Set the midpoint threshold to the value in a specific cell (cell F10 in this case)
'Set the for the midpoint threshold
    oColorScale.ColorScaleCriteria(2).Type = xlConditionValueNumber
    oColorScale.ColorScaleCriteria(2).Value = "=$F$10"
    oColorScale.ColorScaleCriteria(2).FormatColor.Color = 49407
'Set the maximum threshold to the lowest value in the range
'Set the color for the maximum threshold
    oColorScale.ColorScaleCriteria(3).Type = xlConditionValueHighestValue
    oColorScale.ColorScaleCriteria(3).FormatColor.Color = 8109667
```

End SubColorScaleCriterion and the ColorScaleCriteria Collection

The `ColorScaleCriteria` collection holds each `ColorScaleCriterion` in a color scale conditional format. Each criterion specifies the minimum, midpoint, or maximum threshold for the color scale.

ColorScaleCriteria Common Properties

The `Count` and `Item` properties are defined at the beginning of this appendix.

ColorScaleCriterion Properties

Name	Returns	Description
<code>FormatColor</code> <code>Color</code>	<code>Format</code>	Returns a <code>FormatColor</code> object, which defines the color assigned to the specified color scale threshold. Read-only
<code>Index</code>	<code>Long</code>	Returns a value that represents the threshold for the criteria. For two-color scales, the index values will be 1 for the minimum threshold and 2 for the maximum threshold. For three-color scales, the values will be 1 for the minimum threshold, 2 for the midpoint, and 3 for the maximum. Read-only
<code>Type</code> <code>ValueTypes</code>	<code>xlCondition</code>	Specifies how the threshold values for a color scale conditional format are determined (number, percent, formula, or percentile). This property will return an <code>xlConditionValueTypes</code> constant
<code>Value</code>	<code>VARIANT</code>	Set/Get the value for the minimum, midpoint, and maximum thresholds in a color scale conditional formatting rule

ColorStop Object and ColorStops Collection

ColorStop Object and ColorStops Collection

The `ColorStop` object exposes the properties and methods that control cell fill. Each `ColorStop` object represents a color stop for gradient fill in a range or selection. The `ColorStops` collection contains the `ColorStop` objects in a range or selection.

ColorStops Common Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ColorStops Methods

Name	Parameters	Description
<code>Add</code>	<code>Position</code>	Adds a <code>ColorStop</code> for the active selection
<code>Clear</code>		Clears the current <code>ColorStops</code> collection
<code>Item</code>	<code>Index</code>	Returns a single object from the represented collection

ColorStop Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Color</code>	<code>Variant</code>	Set/Get the color for the active selection
<code>Position</code>	<code>Double</code>	Set/Get the position of the <code>ColorStop</code> object
<code>ThemeColor</code>	<code>Long</code>	Set/Get the theme color for the active selection
<code>TintandShade</code>	<code>Variant</code>	Set/Get the tint and shade for the active selection

ColorStops Methods

Name	Parameters	Description
<code>Delete</code>		Deletes a specified <code>ColorStop</code> object

Comment Object and the Comments Collection

The `Comments` collection holds all of the cell comments in the parent `Range` object. Each `Comment` object represents a single cell comment.

Comment Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Comment Properties

Name	Returns	Description
Author	String	Read-only. Returns the name of the person who created the comment
Shape	Shape	Read-only. Returns the comment box as a Shape object, allowing manipulation of the comment box
Visible	Boolean	Set/Get whether the comment is visible all the time (True) or only when the user hovers over the cell containing the comment

Comment Methods

Name	Returns	Parameters	Description
Delete			Deletes the comment from the cell
Next	Comment		Returns the next cell comment in the parent collection
Previous	Comment		Returns the previous cell comment in the parent collection
Text	String	[Text], [Start], [Overwrite]	Sets the text associated with the comment. The Text parameter is used to set the comment text. Use the Start parameter to specify the starting point for Text in the existing comment. Set the Overwrite parameter to True to overwrite existing text

Comment Object and the Comments Collection Example

This example removes the user name added by Excel at the start of the comment and formats the comment to make it more readable:

```
Sub FormatComments()
    Dim oComment As Comment, i As Integer
    'Loop through all the comments in the sheet
    For Each oComment In ActiveSheet.Comments
        'Using the text of the comment(
        With oComment.Shape.TextFrame.Characters

            'Find and remove the user name inserted by Excel
            i = InStr(1, .Text, ":" & vbCrLf)
            If i > 0 Then
                .Text = Mid(.Text, i + 2)
            End If

            'Increase the font size
            With .Font
```

ConditionValue Object

```
.Name = "Arial"  
.Size = 10  
.Bold = False  
End With  
End With  
  
'Make the text frame auto-fit  
oComment.Shape.TextFrame.AutoSize = True  
Next  
End Sub
```

ConditionValue Object

Returned by the `Databar` object, the `ConditionValue` object defines the type of evaluation for a data bar conditional formatting rule.

ConditionValue Properties

The `Application`, `Parent`, and `Creator` properties are defined at the beginning of this appendix.

ConditionValue Properties

Name	Returns	Description
Type	<code>xlConditionValueTypes</code>	Specifies how the threshold values for a data bar conditional format are determined (number, percent, formula, or percentile). This property will return an <code>xlConditionValueTypes</code> constant. Read-only
Value	Variant	Set/Get the shortest bar or longest bar threshold value for a data bar conditional format. You can only set this value if the condition value type is set to number, percent, percentile, or formula

ConditionValue Methods

Name	Returns	Parameters	Description
Modify		<code>newtype As xlConditionValueTypes, newvalue</code>	Allows a user to modify the method in which the longest and shortest value in a data bar conditional format rule is evaluated

Connections Object

The `Connections` object returns or sets a string that enables Excel to connect any one of the valid external data sources shown in the Data tab of the Ribbon.

Connections Common Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Connections Methods

Name	Returns	Parameters	Description
Add	Workbook Connection	Name As String, Description As String, ConnectionString As Variant, CommandText As Variant, lCmdtype	Returns a WorkbookConnection object, via a connection string defining a connection to an external data source
AddFromFile	Workbook Connection	Filename As String,	Returns a WorkbookConnection object, via an Office Data Connection (.odc) file, a Offline Cube file (.cub), or any other file that defines an external data source

Connections Example

This example adds a connection to a local cube file and then creates a PivotTable.

```
Sub Create_LocalCube_PivotTable()
    'Add a connection to the local cube file
    ActiveWorkbook.Connections.AddFromFile "C:\MyCustomCube.cub"
    'Create a Pivot Cache and Pivot Table
    ActiveWorkbook.PivotCaches.Create(SourceType:=xlExternal, _
        SourceData:=ActiveWorkbook.Connections("MyCustomCube").CreatePivotTable _
        TableDestination:=Range("A1"), TableName:="MyPivot")
End Sub
```

ConnectorFormat Object

The `ConnectorFormat` object represents the connector line used between shapes. This connector line connects two shapes together. If either of the shapes is moved, the connector automatically readjusts so the shapes still look visually connected. The parent of a `ConnectorFormat` object is the `Shape` object.

ConnectorFormat Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ConnectorFormat Properties

ConnectorFormat Properties

Name	Returns	Description
<code>BeginConnected</code>	<code>MsoTriState</code>	Read-only. Returns whether the beginning of the connector has a shape attached. Use with <code>BeginConnectedShape</code>
<code>BeginConnectedShape</code>	<code>Shape</code>	Read-only. Returns the shape that is connected to the beginning of the connector. Use with <code>BeginConnected</code>
<code>BeginConnectionSite</code>	<code>Long</code>	Read-only. Returns which connection site (connection spot) on the shape that the beginning of the connector is connected to. Use with <code>BeginConnected</code>
<code>EndConnected</code>	<code>MsoTriState</code>	Read-only. Returns whether the end of the connector has a shape attached. Use with <code>BeginConnectedShape</code>
<code>EndConnectedShape</code>	<code>Shape</code>	Read-only. Returns the shape that is connected to the end of the connector. Use with <code>EndConnected</code>
<code>EndConnectionSite</code>	<code>Long</code>	Read-only. Returns which connection site (connection spot) on the shape that the end of the connector is connected to. Use with <code>EndConnected</code>
<code>Type</code>	<code>MsoConnectorType</code>	Set/Get what type of connector is being used (for example, <code>msoConnectorStraight</code> and <code>msoConnectorCurve</code>)

ConnectorFormat Methods

Name	Parameters	Description
<code>BeginConnect</code>	<code>ConnectedShape</code> <code>AsShape, ConnectionSite</code> As <code>Long</code>	Sets the beginning of the connector to the shape specified by the <code>ConnectedShape</code> parameter at the connection site specified by the <code>ConnectionSite</code> parameter
<code>BeginDisconnect</code>		Disconnects the shape that was at the beginning of the connection. This method does not move the connection line
<code>EndConnect</code>	<code>ConnectedShape</code> <code>AsShape, ConnectionSite</code> As <code>Long</code>	Sets the end of the connector to the shape specified by the <code>ConnectedShape</code> parameter at the connection site specified by the <code>ConnectionSite</code> parameter
<code>EndDisconnect</code>		Disconnects the shape that was at the end of the connection. This method does not move the connection line

ConnectorFormat Object Example

This example formats all fully connected connectors as curved lines:

```
Sub FormatConnectors()
    Dim oShp As Shape
    Dim oCF As ConnectorFormat
    'Loop through all the Shapes in the sheet
    For Each oShp In ActiveSheet.Shapes
        'Is it a Connector?
        If oShp.Connector Then

            'Yes, so get the ConnectorFormat object
            Set oCF = oShp.ConnectorFormat

            'If the connector is connected at both ends,
            'make it a curved line.
            With oCF
                If .BeginConnected And .EndConnected Then
                    .Type = msoConnectorCurve
                End If
            End With
        End If
    Next
End Sub
```

ControlFormat Object

The `ControlFormat` object contains properties and methods used to manipulate Excel controls such as text boxes and list boxes. This object's parent is always the `Shape` object.

ControlFormat Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ControlFormat Properties

Name	Returns	Description
<code>DropDownLines</code>	Long	Set/Get how many lines are displayed in the drop-down part of a combo box. Valid only if the control is a combo box
<code>Enabled</code>	Boolean	Set/Get whether the control is enabled
<code>LargeChange</code>	Long	Set/Get the value that is added or subtracted every time the user clicks inside the scrollbar area for a scroll box. Valid only if the control is a scroll box
<code>LinkedCell</code>	String	Set/Get the range where the results of the control are placed

Table continued on following page

ControlFormat Methods

Name	Returns	Description
ListCount	Long	Read-only. Returns the number of items in the list box or combo box. Valid only for list box and combo box controls
ListFillRange	String	Set/Get the range that contains the items for a list box or combo box. Valid only for list box and combo box controls
ListIndex	Long	Set/Get the item that is currently selected in the list box or combo box. Valid only for list box and combo box controls
LockedText	Boolean	Set/Get whether the control text can be changed if the workbook is locked
Max	Long	Set/Get the maximum value allowed for a scrollbar or spinner. Valid only on a control that is a scrollbar or spinner
Min	Long	Set/Get the minimum value allowed for a scrollbar or spinner. Valid only on a control that is a scrollbar or spinner
MultiSelect	Long	Set/Get how a list box reacts to user selection. The property can be set to <code>xlNone</code> (only one item can be selected), <code>xlSimple</code> (each item the user clicks on is added to the selection), or <code>xlExtended</code> (the user has to hold down the Ctrl key to select multiple items). Valid only on list boxes
PrintObject	Boolean	Set/Get whether the control will be printed when the sheet is printed
SmallChange	Long	Set/Get the value that is added or subtracted every time the user clicks the arrow button associated with the scrollbar. Valid only if the control is a scroll box
Value	Long	Set/Get the value of the control

ControlFormat Methods

Name	Returns	Parameters	Description
AddItem		Text As String, [Index]	Adds the value of the Text parameter into a list box or combo box. Valid only for list box and combo box controls

Name	Returns	Parameters	Description
List	Variant	[Index]	Set/Get the string list array associated with a combo box or list box. Can also Set/Get individual items in the list box or combo box if the Index parameter is specified. Valid only for list box and combo box controls
RemoveAllItems			Removes all the items from a list box or combo box. Valid only for list box and combo box controls
RemoveItem		Index As Long, [Count]	Removes the item specified by the Index parameter from a list box or combo box. Valid only for list box and combo box controls

ControlFormat Object Example

This example resets all the list boxes, drop-downs, scrollbars, spinners, and checkboxes on the sheet:

```

Sub ResetFormControls()
    Dim oShp As Shape
    Dim oCF As ControlFormat
    'Loop through all the shapes in the sheet
    For Each oShp In ActiveSheet.Shapes
        'Is this a Forms control?
        If oShp.Type = msoFormControl Then

            'Yes, so get the ControlFormat object
            Set oCF = oShp.ControlFormat

            'Reset the control as appropriate
            Select Case oShp.FormControlType
                Case xlListBox, xlDropDown
                    oCF.RemoveAllItems

                Case xlSpinner, xlScrollBar
                    oCF.Value = oCF.Min

                Case xlCheckBox
                    oCF.Value = xlOff

            End Select
        End If
    Next
End Sub

```

CubeField Object and the CubeFields Collection

The `CubeFields` collection holds all of the `PivotTable` report fields based on an OLAP cube. Each `CubeField` object represents a measure or hierarchy field from the OLAP cube. The parent of the `CubeFields` collection is the `PivotTable` object.

CubeFields Common Properties

CubeFields Common Properties

The Application, Count, Creator, Item, and Parent properties are defined at the beginning of this appendix.

CubeFields Collection Methods

Name	Returns	Parameters	Description
AddSet	CubeField	Name As String, Caption As String	Adds a new CubeField object to the CubeFields collection

CubeField Common Properties

The Application, Creator, Item, and Parent properties are defined at the beginning of this appendix.

CubeField Properties

Name	Returns	Description
AllItemsVisible	Boolean	A read-only Boolean set to True by default, this property checks whether manual filtering is applied to either a PivotField or a CubeField. This property is automatically set to False when any manual filtering is applied
Caption	String	Sets/Gets the text label to use for the cube field
CubeFieldSubType	xlCubeFieldSubType	Read-only. Specifies whether a CubeField is an Attribute, Calculated Measure, Hierarchy, KPIGoal, KPIStatus, KPITrend, KPIValue, KPIWeight, Measure, or CubeSet
CubeFieldType	XlCubeField Type	Read-only. Returns whether the cube field is a hierarchy field (xlHierarchy) or a measure field (xlMeasure)
CurrentPageName	String	Set/Get the page name for a specified CubeField
DragToColumn	Boolean	Set/Get whether the field can be dragged to a column position. False for measure fields
DragToData	Boolean	Set/Get whether the field can be dragged to the data position
DragToHide	Boolean	Set/Get whether the field can be dragged off the PivotTable report and therefore hidden

Name	Returns	Description
DragToPage	Boolean	Set/Get whether the field can be dragged to the page position. <code>False</code> for measure fields
DragToRow	Boolean	Set/Get whether the field can be dragged to a row position. <code>False</code> for measure fields
EnableMultiplePageItems	Boolean	Set/Get whether multiple items in the page field area for OLAP <code>PivotTables</code> can be selected
HasMemberProperties	Boolean	Read-only. Returns <code>True</code> when there are member properties specified to be displayed for the cube field
IncludeNewItemInFilter	Boolean	When this setting is set to <code>True</code> , excluded items are tracked when manual filtering is applied. When this setting is set to <code>False</code> , included items are tracked when manual filtering is applied
IsDate	Boolean	Read-only. Returns <code>True</code> if the <code>CubeField</code> is a date
LayoutFormType	<code>XlLayoutForm</code>	Set/Get the way the specified <code>PivotTable</code> items appear
LayoutSubtotalLocationLocationType	<code>XlSubtotal</code>	Set/Get the position of the <code>PivotTable</code> field subtotals in relation to the specified field
Name	String	Read-only. Returns the name of the field
OrientationOrientation	<code>XlPivotField</code>	Set/Get where the field is located in the <code>PivotTable</code> report
PivotFields	<code>PivotFields</code>	Read-only. Returns the <code>PivotFields</code> collection
Position	Long	Set/Get the position number of the hierarchy field among all the fields in the same orientation
ShowInFieldList	Boolean	Set/Get whether a <code>CubeField</code> object will be shown in the field list
TreeviewControlControl	Treeview	Read-only. Returns an object allowing manipulation of the cube on an OLAP <code>PivotTable</code> report
Value	String	Read-only. Returns the name of the field

CubeField Methods

Name	Parameters	Description
AddMember PropertyField	Property As String, [Property Order] As Variant, [PropertyDisplayedIn] As xlPropertyDisplayedIn	Adds a member property field to the display for the cube field. Note that the property field specified will not be viewable if the PivotTable view has no fields
ClearManualFilter		Sets the Visible property to True for all items of a PivotField. It also empties the HiddenItemsList and VisibleItemsList collections in OLAP PivotTables
CreatePivotFields		Allows you to create PivotFields and apply filters to them before adding them to the PivotTable
Delete		Deletes the object

CustomProperty Object and the CustomProperties Collection

This object allows you to store information within a worksheet or SmartTag. This information can then be used as metadata for XML, or can be accessed by any routine that needs information specific to the worksheet or SmartTag.

More important to a developer is the capability of this new object to store specifics regarding a worksheet or group of worksheets so any routine can call up the CustomProperty, analyze the information contained within, and then make decisions on how to handle that worksheet. In the past, many developers used worksheet-level range names to store information about a worksheet. Worksheet-level range names only reside in that worksheet, enabling each worksheet to have the same range name, but store different values.

For example, each worksheet in a workbook containing a dozen budget worksheets and three report worksheets could contain the same range name, called IsBudget. All of the budget sheets would store the value of True in the range name, while the report sheets would store False. Routines that need to loop through the worksheets, applying different formats or calculations to budget sheets, can call on the value of the range name to determine if it's a budget sheet before running code on it.

This new CustomProperty object makes storing such information (or any information, for that matter) simpler than creating worksheet-level range names, or storing such information in a hidden worksheet or in the Registry.

The CustomProperties collection represents CustomProperty objects for either worksheets or SmartTags. CustomProperties can store information within either a worksheet or SmartTag. They are similar to the DocumentProperties object in the Office XP model, except they are stored with a worksheet or SmartTag instead of the whole document.

CustomProperties Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

CustomProperties Collection Properties

Name	Returns	Description
Count	Long	Read-only. Returns the number of objects in the collection
Item	Custom Property	Read-only. Index As Variant. Returns a single object from a collection

CustomProperties Collection Methods

Name	Returns	Parameters	Description
Add	Custom Property	Name As String, Value As Variant	Adds custom property information

CustomProperty Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

CustomProperty Properties

Name	Returns	Description
Name	String	Set/Get the name of the object
Value	Variant	Set/Get the value to store in the property

CustomProperty Methods

Name	Returns	Parameters	Description
Delete			Deletes the object

CustomProperty Object Example

This routine loops through the worksheets in a workbook and creates a CustomProperty called IsBudget. The value of IsBudget depends on whether or not the worksheet contains the phrase *Budget Analysis*. It then lists the results:

```
Sub CreateCustomProperties()
    Dim bBudget As Boolean
    Dim lRow As Long
    Dim oCustomProp As CustomProperty
    Dim rng As Range, wks As Worksheet

    'Turn off the screen and clear the search formats
```

CustomView Object and the CustomViews Collection

```
With Application
    .FindFormat.Clear
    .ScreenUpdating = False
End With

'Clear the worksheet that will contain the Custom Property list
ActiveSheet.UsedRange.Offset(1, 0).ClearContents

'Initialize the row counter
lRow = 2    'Row 1 contains the Column Headings

'Loop through the worksheet in this workbook
For Each wks In ThisWorkbook.Worksheets

'Supress errors resulting in no cells found and no Custom Property
    On Error Resume Next
        bBudget = False

        bBudget = (Len(wks.UsedRange.Find(What:="Budget Analysis").Address) > 0)
'We cannot refer to a Custom Property by its name, only its numeric index
        Set oCustomProp = wks.CustomProperties(1)
        On Error GoTo 0

'If the Custom Property exists, delete it and add it again
        If Not oCustomProp Is Nothing Then oCustomProp.Delete

'Note the value of bBudget is encased in double quotes.
'If we don't, True will be stored as -1 and False 0 (their numeric values).
        Set oCustomProp = wks.CustomProperties.Add(Name:="IsBudget", Value:="" _
            & bBudget & "")

'List the Custom Property settings on the worksheet
    With ActiveSheet
'Parent.Name returns the name of the object holding the Custom Property
'That object is the worksheet name in this case
        .Cells(lRow, 1).Value = oCustomProp.Parent.Name
        .Cells(lRow, 2).Value = oCustomProp.Name
        .Cells(lRow, 3).Value = oCustomProp.Value
    End With

    'Move down one row
    lRow = lRow + 1

Next wks
End Sub
```

CustomView Object and the CustomViews Collection

The `CustomViews` collection holds the list of custom views associated with a workbook. Each `CustomView` object holds the attributes associated with a workbook custom view. A custom view holds settings such as window size, window position, column widths, hidden columns, and print settings of a workbook. The parent object of the `CustomViews` collection is the `Workbook` object.

The `CustomViews` collection has one other property besides the typical collection attributes. The `Add` method adds a custom view to the `CustomViews` collection. The `Add` method accepts a name for the view with the `ViewName` parameter. Optionally, the `Add` method accepts whether print settings are included (`PrintSettings`) and whether hidden rows and columns are included (`RowColSettings`).

CustomView Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

CustomView Properties

Name	Returns	Description
<code>Name</code>	<code>String</code>	Read-only. Returns the name of the custom view
<code>PrintSettings</code>	<code>Boolean</code>	Read-only. Returns whether print settings are included in the custom view
<code>RowColSettings</code>	<code>Boolean</code>	Read-only. Returns whether hidden rows and columns are included in the custom view

CustomView Methods

Name	Returns	Parameters	Description
<code>Delete</code>			Deletes the custom view
<code>Show</code>			Shows the custom view and the settings associated with it

CustomView Object and the CustomViews Collection Example

Display all the custom views in the workbook as a screen-show, pausing for two seconds between each one:

```
Sub ShowCustomView()  
    Dim oCV As CustomView  
    'Cycle through all custom views in the sheet containing row/column information  
    For Each oCV In ActiveWorkbook.CustomViews  
        If oCV.RowColSettings Then  
            oCV.Show  
        End If  
  
        'Pause for 2 seconds between each view  
        Application.Wait Now + TimeValue("00:00:02")  
    Next  
End Sub
```

Databar Object

The `Databar` object allows you to apply visual formatting that uses a colored bar to represent a cell's value in relation to other cells in a specified range.

Databar Properties

The `Application`, `Parent`, and `Creator` properties are defined at the beginning of this appendix.

Databar Properties

Name	Returns	Description
<code>AppliesTo</code>	Range	Read-only. Returns the range that is affected by the formatting rule
<code>BarColor</code>	Format Color	Read-only. Returns the <code>FormatColor</code> object, which is used to specify the color of the bar shown
<code>Formula</code>	String	Set/Get a string representing a formula that determines the values that are to be evaluated in the conditional formatting rule
<code>MaxPoint</code>	Condition Value	Read-only. Returns the method by which the longest bar in the formatting rule is evaluated. Use the <code>Type</code> and <code>Value</code> properties of the <code>ConditionValue</code>
<code>MinPoint</code>	String	Read-only. Returns a <code>ConditionValue</code> object that can be used to specify the method by which the shortest bar in the formatting rule is evaluated
<code>PercentMax</code>	Long	Set/Get the length of the longest bar by using the cell width as a baseline. The value must be a whole number between 0 and 100
<code>PercentMin</code>	Long	Set/Get the length of the shortest bar by using the cell width as a baseline. The value must be a whole number between 0 and 100
<code>Priority</code>	Long	Set/Get the priority value of a conditional formatting rule, determining the order of evaluation when other rules are in effect
<code>PTCondition</code>	Boolean	Read-only. Indicates whether the formatting rule is applied to a PivotTable chart
<code>ScopeType</code>	<code>xlPivotConditionScope</code>	Set/Get the scope of the formatting rule when applied to a PivotTable chart. Use the <code>xlPivotConditionScope</code> constants
<code>ShowValue</code>	Boolean	If the <code>ShowValue</code> property is set to <code>True</code> , the actual value in the cell is displayed along with the data bar
<code>StopifTrue</code>	Boolean	Set/Get a Boolean value that determines if additional formatting rules should be applied if the current rule evaluates to <code>True</code> . The default value is <code>True</code>
<code>Type</code>	<code>xlFormatConditionType</code>	Read-only. Returns an <code>xlFormatConditionType</code> constant, which specifies the type of conditional formatting being applied

Databar Methods

Name	Returns	Parameters	Description
Delete			Deletes the object
ModifyAppliesToRange		Range As Range	Sets the range for which the formatting rule will be applied
SetFirstPriority			Sets the priority value for the formatting rule so that it is evaluated before all other rules on the worksheet
SetLastPriority			Sets the priority value for the formatting rule so that it is evaluated after all other rules on the worksheet

Databar Object Example

This example creates a data bar conditional formatting rule that sets the highest value in the specified range as the `MaxPoint` and the value in cell F14 as the `MinPoint`:

```
Sub CreateDatabar()
Dim oDatabar As Databar
'Add a Data bar
    Set oDatabar = Range("F6:F16").FormatConditions.AddDatabar
'Set the max and min parameters for the data bar
'Note the MinPoint for the databar is a value in cell F14
    With oDatabar
        .MaxPoint.Modify xlConditionValueHighestValue
        .MinPoint.Modify xlConditionValueNumber, "=$F$14"
        .BarColor.Color = 7039480
        .ShowValue = True
    End With
End Sub
```

DataLabel Object and the DataLabels Collection

The `DataLabels` collection holds all the labels for individual points or trendlines in a data series. Each series has only one `DataLabels` collection. The parent of the `DataLabels` collection is the `Series` object. Each `DataLabel` object represents a single data label for a trendline or a point. The `DataLabels` collection is used with the `HasDataLabels` property of the parent `Series` object.

The `DataLabels` collection has a few properties and methods besides the typical collection attributes. They are listed in the following table.

DataLabels Collection Properties and Methods

Name	Returns	Description
AutoScaleFont	Variant	Set/Get whether the font size will change automatically if the parent chart changes sizes
AutoText	Boolean	Set/Get whether Excel will generate the data label text automatically
Format	ChartFormat	Read-only. Returns the ChartFormat object, which controls the line, fill, and effect formatting for the chart area
HorizontalAlignment	xlAlign	Set/Get how the data labels are horizontally aligned. Use the xlAlign constants
Name	String	Read-only. Returns the name of the collection
NumberFormat	String	Set/Get the numeric formatting to use if the data labels are numeric values or dates
NumberFormatLinked	Boolean	Set/Get whether the same numerical format used for the cells containing the chart data is used by the data labels
NumberFormatLocal	Variant	Set/Get the name of the numeric format being used by the data labels, in the language being used by the user
Orientation	XlOrientation	Set/Get the angle of the text for the data labels. The value can be in degrees (from -90 to 90) or one of the XlOrientation constants
Position	XlDataLabelPosition	Set/Get where the data labels are going to be located in relation to points or trendlines
ReadingOrder	Long	Set/Get how the text is read (from left to right or right to left). Only applicable in appropriate languages
Separator	Variant	Set/Get the separator used for the data labels on a chart
Shadow	Boolean	Set/Get whether the data labels have a shadow effect

Name	Returns	Description
ShowBubbleSize	Boolean	Set/Get whether to show the bubble size for the data labels on a chart
ShowCategoryName	Boolean	Set/Get whether to display the category name for the data labels on a chart
ShowLegendKey	Boolean	Set/Get whether the key being used in the legend, usually a specific color, will show along with the data label
ShowPercentage	Boolean	Set/Get whether to display the percentage value for the data labels on a chart
ShowSeriesName	Boolean	Set/Get whether to show the series name
ShowValue	Boolean	Set/Get whether to display the specified chart's data label values
VerticalAlignment	xlVAlign	Set/Get how you want the data labels to be horizontally aligned. Use the xlVAlign constants
Delete	Variant	Method. Deletes the data labels
Select	Variant	Method. Selects the data labels on the chart

DataLabel Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

DataLabel Properties

Name	Returns	Description
AutoScaleFont	Variant	Set/Get whether the font size will change automatically if the parent chart changes sizes
AutoText	Boolean	Set/Get whether Excel will generate the data label text automatically
Caption	String	Set/Get the data label text
Characters	Characters	Read-only. Parameters: [Start], [Length]. Returns an object that represents a range of characters within the text

Table continued on following page

DataLabel Properties

Name	Returns	Description
Format	ChartFormat	Returns the ChartFormat object, which controls the line, fill, and effect formatting for the chart area
Horizontal Alignment	xlAlign	Set/Get how the data labels are horizontally aligned. Use the xlAlign constants
Left	Double	Set/Get the distance from the left edge of the data label to the parent chart's left edge
Name	String	Read-only. Returns the name of the data label
NumberFormat	String	Set/Get the numeric formatting to use if the data label is a numeric value or a date
NumberFormat Linked	Boolean	Set/Get whether the same numerical format used for the cells containing the chart data is used by the data label
NumberFormat Local	Variant	Set/Get the name of the numeric format being used by the data label, in the language being used by the user
Orientation	XlOrientation	Set/Get the angle of the text for the data label. The value can be in degrees (from -90 to 90) or one of the XlOrientation constants
Position	XlDataLabelPosition	Set/Get where the data label is going to be located in relation to points or trendlines
ReadingOrder	Long	Set/Get how the text is read (from left to right or right to left). Only applicable in appropriate languages
Separator	Variant	Set/Get the separator used for the data labels on a chart
Shadow	Boolean	Set/Get whether the data label has a shadow effect
ShowBubbleSize	Boolean	Set/Get whether to show the bubble size for the data labels on a chart
ShowCategory Name	Boolean	Set/Get whether to display the category name for the data labels on a chart

Name	Returns	Description
ShowLegendKey	Boolean	Set/Get whether the key being used in the legend, usually a specific color, will show along with the data label
ShowPercentage	Boolean	Set/Get whether to display the percentage value for the data labels on a chart
ShowSeriesName	Boolean	Set/Get whether to show the series name
ShowValue	Boolean	Set/Get whether to display the specified chart's data label values
Text	String	Set/Get the data label text
Top	Double	Set/Get the distance from the top edge of the data label to the parent chart's top edge
Type	Variant	Set/Get what sort of data label to show (for example, labels, percent, values)
Vertical Alignment	xlVAlign	Set/Get how you want the data label to be horizontally aligned. Use the xlVAlign constants

DataLabel Methods

Name	Returns	Parameters	Description
Delete	Variant		Deletes the data label
Select	Variant		Selects the data label on the chart

DataLabel Object and the DataLabels Collection Example

This example adds data labels to all the points on the chart, using the column to the left of the X values range:

```
Sub AddDataLabels()
    Dim oSer As Series
    Dim vaSplits As Variant
    Dim oXRng As Range
    Dim oLblRng As Range
    Dim oLbl As DataLabel
    Dim i As Integer

    'Loop through all the series in the chart
```

DataTable Object

```
For Each oSer In ActiveSheet.ChartObjects("Chart 1").Chart.SeriesCollection
    'Get the series formula and split it into its
    'constituent parts (Name, X range, Y range, order)
    vaSplits = Split(oSer.Formula, ",")

    'Get the X range
    Set oXRng = Range(vaSplits(LBound(vaSplits) + 1))

    'Get the column to the left of the X range
    Set oLblRng = oXRng.Offset(0, -1)

    'Show data labels for the series
    oSer.ApplyDataLabels

    'Loop through the points
    For i = 1 To oSer.Points.Count

        'Get the DataLabel object
        Set oLbl = oSer.Points(i).DataLabel
        oLbl.Caption = oLblRng.Cells(i)
    Next
Next
End Sub
```

DataTable Object

A `DataTable` object contains the formatting options associated with a chart's data table. The parent of the `DataTable` object is the `Chart` object.

DataTable Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

DataTable Properties

Name	Returns	Description
<code>AutoScaleFont</code>	Variant	Set/Get whether the font size will change automatically if the parent chart changes sizes
<code>Border</code>	Border	Read-only. Returns the border's properties around the data table
<code>Font</code>	Font	Read-only. Returns an object containing <code>Font</code> options for the data table
<code>Format</code>	<code>ChartFormat</code>	Read-only. Returns the <code>ChartFormat</code> object, which controls the line, fill, and effect formatting for the chart area
<code>HasBorderHorizontal</code>	Boolean	Set/Get whether the data table has horizontal cell borders

Name	Returns	Description
HasBorder Outline	Boolean	Set/Get whether the data table has a border around the outside
HasBorder Vertical	Boolean	Set/Get whether the data table has vertical cell borders
ShowLegendKey	Boolean	Set/Get whether the legend key is shown along with the data table contents

DataTable Methods

Name	Returns	Parameters	Description
Delete			Deletes the data table
Select			Selects the data table on the chart

DataTable Object Example

This example adds a data table to a chart and formats it to only have vertical lines between the values:

```
Sub FormatDataTable()
    Dim oChart As Chart
    'Set the target chart
    Set oChart = ActiveSheet.ChartObjects("Chart 1").Chart
    'Add data table and change font color
    oChart.HasDataTable = True
    oChart.DataTable.Font.Color = 1533
End Sub
```

DefaultWebOptions Object

The `DefaultWebOptions` object allows programmatic changes to items associated with the default settings of the Web Options dialog box. These options include what Excel does when opening an HTML page and when saving a sheet as an HTML page.

DefaultWebOptions Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

DefaultWebOptions Properties

Name	Returns	Description
AllowPNG	Boolean	Set/Get whether Portable Network Graphics Format PNG is allowed as an output format. PNG is a file format for the lossless, portable, well-compressed storage of images
AlwaysSaveInDefaultEncoding	Boolean	Set/Get whether web pages are always saved in the default encoding
CheckIfOfficeIsHTMLEditor	Boolean	Set/Get whether Office is the default web editor for Office-created pages
DownloadComponents	Boolean	Set/Get whether Office components are downloaded to the end user's machine when viewing Excel files in a web browser
Encoding	MsoEncoding	Set/Get the type of encoding to save a document as
FolderSuffix	String	Read-only. Returns what the suffix name is for the support directory created when saving an Excel document as a web page. Language-dependent
Fonts	WebPage Fonts	Read-only. Returns a collection of possible Web type fonts
LoadPictures	Boolean	Set/Get whether images are loaded when opening up an Excel file
LocationOfComponents	String	Set/Get the URL or path that contains the Office Web components needed to view documents in a web browser
OrganizeInFolder	Boolean	Set/Get whether supporting files are organized in a folder
PixelsPerInch	Long	Set/Get how dense graphics and table cells should be when viewed on a web page
RelyOnCSS	Boolean	Set/Get whether Cascading Style Sheets (CSS) is used for font formatting
RelyOnVML	Boolean	Set/Get whether image files are not created when saving a document with drawn objects. Vector Markup Language is used to create the images on the fly. VML is an XML-based format for high-quality vector graphics on the web

Name	Returns	Description
SaveHidden Data	Boolean	Set/Get whether all hidden data is saved in the web page along with the regular data
SaveNewWeb PagesAs Web Archives	Boolean	Set/Get whether a new web page can be saved as a web archive
ScreenSize	MsoScreen Size	Set/Get the target monitor's screen size
TargetBrowser	MsoTargetBrowser	Set/Get the browser version
UpdateLinksOnSave	Boolean	Set/Get whether links are updated every time the document is saved
UseLongFileNames	Boolean	Set/Get whether long filenames are used whenever possible

DefaultWebOptions Object Example

This example shows how to open a web page without loading the pictures:

```

Sub OpenHTMLWithoutPictures()
    Dim bLoadImages As Boolean
    Dim oDWO As DefaultWebOptions
    'Get the Default Web options
    Set oDWO = Application.DefaultWebOptions
    'Remember whether to load pictures
    bLoadImages = oDWO.LoadPictures
    'Tell Excel not to load pictures, for faster opening
    oDWO.LoadPictures = False
    'Open a web page, without pictures
    Workbooks.Open "http://www.wrox.com"
    'Restore the setting
    oDWO.LoadPictures = bLoadImages
End Sub

```

Dialog Object and the Dialogs Collection

The `Dialogs` collection represents the list of dialog boxes that are built into Excel. The `XlBuiltinDialog` constants are used to access an individual `Dialog` object in the `Dialogs` collection. A `Dialog` object represents a single built-in Excel dialog box. Each `Dialog` object will have additional custom properties, depending on what type of `Dialog` object it is.

Dialogs Common Properties

The `Application`, `Count`, `Creator`, `Item`, and `Parent` properties are defined at the beginning of this appendix.

Dialog Common Properties

Dialog Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Dialog Methods

Name	Returns	Parameters	Description
Show	Boolean	[Arg1], [Arg2], ... [Arg30]	Displays and executes the dialog box settings. <code>True</code> is returned if the user chose <code>OK</code> , and <code>False</code> is returned if the user chose <code>Cancel</code> . The arguments to pass depend on the dialog box

Dialog Object and the Dialogs Collection Example

```
Sub ShowPrinterSelection()  
    'Show printer selection dialog  
    Application.Dialogs(xlDialogPrinterSetup).Show  
End Sub
```

DisplayUnitLabel Object

The `DisplayUnitLabel` object contains all of the text and formatting associated with the label used for units on axes. For example, if the values on an axis are in the millions, it would be messy to display such large values on the axis. Using a unit label such as `Millions` would allow much smaller numbers to be used. The parent of the `DisplayUnitLabel` object is the `Axis` object. This object is usually used along with the `HasDisplayUnit` property of the parent `Axis` object.

DisplayUnitLabel Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

DisplayUnitLabel Properties

Name	Returns	Description
<code>AutoScaleFont</code>	Variant	Set/Get whether the font size will change automatically if the parent chart changes sizes
<code>Caption</code>	String	Set/Get the unit label's text
<code>Characters</code>	Characters	Read-only. Parameters: <code>[Start]</code> , <code>[Length]</code> . Returns an object containing all the characters in the unit label. Allows manipulation on a character-by-character basis
<code>Format</code>	ChartFormat	Read-only. Returns the <code>ChartFormat</code> object, which controls the line, fill, and effect formatting for the chart area
<code>HorizontalAlignment</code>	<code>xlAlign</code>	Set/Get how you want the unit label to be horizontally aligned. Use the <code>xlAlign</code> constants

Name	Returns	Description
Left	Double	Set/Get the distance from the left edge of the unit label text area to the chart's left edge
Name	String	Read-only. Returns the name of the DisplayUnitLabel object
Orientation	XlOrientation	Set/Get the angle of the text for the unit label. The value can be in degrees (from -90 to 90) or one of the XlOrientation constants
Position	XlChartElementPosition	Set/Get where the unit label is going to be located in relation to points or trendlines
Reading Order	Long	Set/Get how the text is read (from left to right or right to left). Only applicable in appropriate languages
Shadow	Boolean	Set/Get whether the unit label has a shadow effect
Text	String	Set/Get the unit label's text
Top	Double	Set/Get the distance from the top edge of the unit label text area to the chart's top edge
Vertical Alignment	xlVAlign	Set/Get how you want the unit label to be horizontally aligned. Use the xlVAlign constants

DisplayUnitLabel Methods

Name	Returns	Parameters	Description
Delete	Variant		Deletes the unit label from the axis
Select	Variant		Selects the unit label on the chart

DisplayUnitLabel Object Example

```

Sub AddUnitLabel()
    Dim oDUL As DisplayUnitLabel
    'Format the Y axis to have a unit label
    With ActiveSheet.ChartObjects("Chart 1").Chart.Axes(xlValue)
        .DisplayUnit = xlThousands
        .HasDisplayUnitLabel = True
    'Get the unit label
        Set oDUL = .DisplayUnitLabel
    End With
End Sub

```

DownBars Object

DownBars Object

The `DownBars` object contains formatting options for down bars on a chart. The parent of the `DownBars` object is the `ChartGroup` object. To see if this object exists, use the `HasUpDownBars` property of the `ChartGroup` object.

DownBars Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

DownBars Properties

Name	Returns	Description
<code>Format</code>	<code>ChartFormat</code>	Read-only. Returns the <code>ChartFormat</code> object, which controls the line, fill, and effect formatting for the chart area
<code>Name</code>	<code>String</code>	Read-only. Returns the name of the down bars

DownBars Methods

Name	Returns	Parameters	Description
<code>Delete</code>	<code>Variant</code>		Deletes the down bars
<code>Select</code>	<code>Variant</code>		Selects the down bars in the chart

DropLines Object

The `DropLines` object contains formatting options for drop lines in a chart. The parent of the `DropLines` object is the `ChartGroup` object. To see if this object exists, use the `HasDropLines` property of the `ChartGroup` object.

DropLines Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

DropLines Properties

Name	Returns	Description
<code>Border</code>	<code>Border</code>	Read-only. Returns the border's properties around the drop lines
<code>Format</code>	<code>ChartFormat</code>	Returns the <code>ChartFormat</code> object, which controls the line, fill, and effect formatting for the chart area
<code>Name</code>	<code>String</code>	Read-only. Returns the name of the drop lines

DropLines Methods

Name	Returns	Parameters	Description
Delete	Variant		Deletes the drop lines
Select	Variant		Selects the drop lines in the chart

Error Object and the Errors Collection

The `Error` object contains one error in the `Errors` collection representing one error in a cell containing possible errors.

The `Errors` collection represents all the errors contained within a cell. Each cell can contain multiple errors.

Note that neither the `Error` nor `Errors` objects contains a count or Boolean property that would allow you to test whether an error even exists in a cell. For this reason, additional code would be needed to loop through each error type for every desired cell checking for the `Error` object's `Value` property, which returns `True` if that type of error occurs in the cell.

Use the `Item` property of the `Errors` Collection object to loop through the error types to determine which errors might have occurred. The `Errors` Collection object can be used to check for specific error conditions available through `xlErrorChecks` constants.

Errors Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Errors Collection Properties

Name	Returns	Description
<code>Item</code>	<code>Error</code>	Returns an <code>Error</code> object that is contained in the <code>Errors</code> collection

Error Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Error Properties

Name	Returns	Description
<code>Ignore</code>	<code>Boolean</code>	Get/Set whether error checking is enabled for a range
<code>Value</code>	<code>Boolean</code>	Read-only. Returns whether all the validation criteria are met

Errors Collection Example

Errors Collection Example

```
Sub CheckForEmptyCellReference()  
' Place a formula in cell A1.  
  Range("A1").Formula = "=B1+C1"  
'If Cell B1 is empty  
  If Range("A1").Errors.Item(xlEmptyCellReferences).Value = True Then  
    MsgBox "One or more of the referenced cells are empty."  
  End If  
End Sub
```

ErrorBars Object

The `ErrorBars` object contains formatting options for error bars in a chart. The parent of the `Errors` object is the `SeriesCollection` object.

ErrorBars Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ErrorBars Properties

Name	Returns	Description
<code>Border</code>	<code>Border</code>	Read-only. Returns the border's properties around the error bars
<code>EndStyle</code>	<code>XlEndStyleCap</code>	Set/Get the style used for the ending of the error bars
<code>Format</code>	<code>ChartFormat</code>	Returns the <code>ChartFormat</code> object, which controls the line, fill, and effect formatting for the chart area.
<code>Name</code>	<code>String</code>	Read-only. Returns the name of the error bars

ErrorBars Methods

Name	Returns	Parameters	Description
<code>ClearFormats</code>	<code>Variant</code>		Clears the formatting set on the error bar
<code>Delete</code>	<code>Variant</code>		Deletes the error bars
<code>Select</code>	<code>Variant</code>		Selects the error bars in the chart

ErrorBars Object Example

```
Sub AddAndFormatErrorBars()  
  Dim oSer As Series  
  Dim oErrBars As ErrorBars  
'Add error bars to the first series (at +/- 10% of the value)  
  Set oSer = ActiveSheet.ChartObjects("Chart 1").Chart.SeriesCollection(1)  
  oSer.ErrorBar xlY, xlErrorBarIncludeBoth, xlErrorBarTypePercent, 10  
'Get the ErrorBars object
```



```

Set oErrBars = oSer.ErrorBars
'Format the error bars
With oErrBars
    .Border.Weight = xlThick
    .Border.LineStyle = xlContinuous
    .Border.ColorIndex = 7
    .EndStyle = xlCap
End With
End Sub

```

ErrorCheckingOptions Collection Object

Represents all of the Error Checking possibilities found on the Formulas section of the Excel Options dialog. Using the `BackgroundChecking` property of this object hides all of the error indicators (small triangle in the upper-right corner of cells).

Use the other properties in this object to specify which type of error checking you want Excel to perform.

The `ErrorCheckingOptions` object can be referenced through the `Application` object and therefore affects all open workbooks.

ErrorCheckingOptions Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ErrorCheckingOptions Collection Properties

Name	Returns	Description
<code>BackgroundChecking</code>	Boolean	Set/Get whether background error checking is set; that is, whether the autocorrect button will appear in cells that contain errors
<code>EmptyCellReferences</code>	Boolean	Set/Get whether error checking is on for cells containing formulas that refer to empty cells
<code>EvaluateToError</code>	Boolean	Set/Get whether error checking is on for cells that evaluate to an error value
<code>InconsistentFormula</code>	Boolean	Set/Get whether error checking is on for cells containing an inconsistent formula in a region
<code>InconsistentTableFormula</code>	Boolean	Set/Get whether error checking is on for cells containing an inconsistent formula in a Table
<code>IndicatorColorIndex</code>	<code>XlColorIndex</code>	Set/Get the color of the indicator for error checking options
<code>ListDataValidation</code>	Boolean	The property will return <code>True</code> if data validation is enabled for a list
<code>NumberAsText</code>	Boolean	Set/Get whether error checking is on for numbers written as text

Table continued on following page

FillFormat Object

Name	Returns	Description
OmittedCells	Boolean	Set/Get whether error checking is on for cells that contain formulas referring to a range that omits adjacent cells that could be included
TextDate	Boolean	Set/Get whether error checking is on for cells that contain a text date with a two-digit year
UnlockedFormulaCells	Boolean	Set/Get whether error checking is on for cells that are unlocked and contain a formula

FillFormat Object

The `FillFormat` object represents the fill effects available for shapes. For example, a `FillFormat` object defines solid, textured, and patterned fill of the parent shape. A `FillFormat` object can only be accessed through the parent `Shape` object.

FillFormat Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

FillFormat Properties

Name	Returns	Description
BackColor	ColorFormat	Read-only. Returns the background color through the <code>ColorFormat</code> object
ForeColor	ColorFormat	Read-only. Returns the foreground color through the <code>ColorFormat</code> object
GradientColorType	MsoGradientColorType	Read-only. Returns what type of gradient fill color concept is used
GradientDegree	Single	Read-only. Returns how dark or light the gradient fill is
GradientStops	GradientStops	Set/Get the end point for the gradient fill
GradientStyle	MsoGradientStyle	Read-only. Returns the orientation of the gradient that is used
GradientVariant	Integer Long	Read-only. Returns the variant used for the gradient from the center
Pattern	MsoPatternType	Read-only. Returns the pattern used for the fill, if any
PresetGradientType	MsoPresetGradientType	Read-only. Returns the type of gradient that is used
PresetTexture	MsoPresetTexture	Read-only. Returns the non-custom texture of the fill

Name	Returns	Description
RotateWithObject	Boolean	Set/Get whether the fill style should rotate with the object
TextureAlignment	MsoTextureAlignment	Set/Get the text alignment for the specified FillFormat object
TextureHorizontalScale	Single	Set/Get the value for horizontally scaling the text for the FillFormat object
TextureName	String	Read-only. Returns the custom texture name of the fill
TextureOffsetX	Single	Set/Get the offset X value for the specified fill
TextureOffsetY	Single	Set/Get the offset Y value for the specified fill
TextureTile	Boolean	Set/Get the texture tile style for the specified fill
TextureType	MsoTextureType	Read-only. Returns whether the texture is custom, preset, or mixed
TextureVerticalScale	Single	Set/Get the texture vertical scale for the specified fill
Transparency	Single	Set/Get how transparent the fill is. From 0 (opaque) to 1 (clear)
Type	MsoFillType	Read-only. Returns if the fill is a texture, gradient, solid, background, picture, or mixed
Visible	MsoTriState	Set/Get whether the fill options are visible in the parent shape

FillFormat Methods

Name	Returns	Parameters	Description
OneColorGradient		Style As MsoGradientStyle, Variant As Integer, Degree as Single	Set the style variant and degree for a one-color gradient fill
Patterned		Pattern As MsoPatternType	Set the pattern for a fill
PresetGradient		Style As MsoGradientStyle, Variant As Integer, PresetGradientType As MsoPresetGradientType	Choose the style, variant, and preset gradient type for a gradient fill

Table continued on following page

FillFormat Object Example

Name	Returns	Parameters	Description
PresetTextured		PresetTexture As MsoPresetTexture	Set the preset texture for a fill
Solid			Set the fill to a solid color
TwoColorGradient		Style As MsoGradientStyle, Variant As Integer	Set the style for a two-color gradient fill
UserPicture		PictureFile As String	Set the fill to the picture in the PictureFile format
UserTextured		TextureFile As String	Set the custom texture for a fill with the Texture-File format

FillFormat Object Example

```
Sub FormatShape()  
    Dim oFF As FillFormat  
    'Get the Fill format of the first shape  
    Set oFF = ActiveSheet.Shapes(1).Fill  
    'Format the shape  
    With oFF  
        .TwoColorGradient msoGradientFromCorner, 1  
        .ForeColor.SchemeColor = 3  
        .BackColor.SchemeColor = 5  
    End With  
End Sub
```

Filter Object and the Filters Collection

The `Filters` collection holds all of the filters associated with the specific parent `AutoFilter`. Each `Filter` object defines a single filter for a single column in an autofiltered range. The parent of the `Filters` collection is the `AutoFilter` object.

Filters Common Properties

The `Application`, `Count`, `Creator`, `Item`, and `Parent` properties are defined at the beginning of this appendix.

Filter Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Filter Properties

Name	Returns	Description
Count	Long	Read-only. Returns the number of objects in the collection
Criteria1	Variant	Read-only. Returns the first criterion defined for the filter (for example, ">=5")
Criteria2	Variant	Read-only. Returns the second criterion for the filter, if defined
On	Boolean	Read-only. Returns whether the filter is in use
Operator	XlAuto Filter Operator	Read-only. Returns what sort of operator has been defined for the filter (for example, xlTop10Items)

Floor Object

The `Floor` object contains formatting options for the floor area of a 3D chart. The parent of the `Floor` object is the `Chart` object.

Floor Common Properties

The `Application`, `Creator`, `Name`, and `Parent` properties are defined at the beginning of this appendix.

Floor Properties

Name	Returns	Description
Format	ChartFormat	Read-only. Returns the <code>ChartFormat</code> object, which controls the line, fill, and effect formatting for the chart area
Name	String	Returns a string representing the name of the object
PictureType	Variant	Set/Get how an associated picture is displayed on the floor of the 3D chart (for example, stretched or tiled). Use the <code>xlChartPictureType</code> constants
Thickness	Long	Set/Get the thickness of the floor. Default is 0

Floor Methods

Name	Returns	Parameters	Description
ClearFormats	Variant		Clears the formatting made on the <code>Floor</code> object
Paste			Pastes the picture in the clipboard into the <code>Floor</code> object
Select	Variant		Selects the floor on the parent chart

Floor Object Example

```
Sub FormatFloor()  
    Dim oChart As Chart  
'Set the target chart  
    Set oChart = ActiveSheet.ChartObjects("Chart 1").Chart  
'Format the chart floor  
    With oChart.Floor  
        .Format.Fill.PresetTextured (msoTextureWhiteMarble)  
        .Format.Fill.Visible = True  
    End With  
End Sub
```

Font Object

The `Font` object contains all of the formatting attributes related to fonts of the parent, including font type, size, and color. Possible parents of the `Font` object are the `AboveAverage`, `AxisTitle`, `CellFormat`, `Characters`, `ChartArea`, `CompareColumns`, `ChartTitle`, `DataLabel`, `Datalabels`, `DataTable`, `FormatCondition`, `Legend`, `LegendEntry`, `Phonetics`, `Range`, `Style`, `TableStyleElement`, `TickLabels`, and `Top10` objects.

Font Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Font Properties

Name	Returns	Description
Background	xlBackground	Set/Get the type of background used behind the font text (xlBackgroundAutomatic, xlBackgroundOpaque, and xlBackgroundTransparent). Use the XlBackground constants. Valid only for text on charts
Bold	Variant	Set/Get whether the font is bold
Color	Variant	Set/Get the color of the font. Use the RGB function to create the color value
ColorIndex	Variant	Set/Get the color of the font. Use the XlColorIndex constants or an index value in the current color palette
FontStyle	Variant	Set/Get what style to apply to the font (for example, "Bold")
Italic	Variant	Set/Get whether the font is italic
Name	Variant	Set/Get the name of the font
Size	Variant	Set/Get the font size of the font

Name	Returns	Description
Strikethrough	Variant	Set/Get whether the font has a strikethrough effect
Subscript	Variant	Set/Get whether the font characters look like a subscript
Superscript	Variant	Set/Get whether the font characters look like a superscript
ThemeColor	Variant	Set/Get the theme color in the applied color scheme associated with an object. Should the object have no association with a theme, then trying to access the ThemeColor property will result in an error
ThemeFont	XlThemeFont	Set/Get the font in the applied font scheme associated with an object
TintAndShade	Variant	Set/Get a Single value from -1 (darkest) to 1 (lightest), which darkens or lightens a color. Zero (0) is neutral
Underline	Variant	Set/Get whether the font is underlined

Font Object Example

```

Sub FormatCellFont()
    Dim oFont As Font
    'Get the font of the currently selected range
    Set oFont = Selection.Font
    'Format the font
    With oFont
        .Name = "Times New Roman"
        .Size = 16           'Points
        .ColorIndex = 5     'Blue
        .Bold = True
        .Underline = xlSingle
    End With
End Sub

```

FormatColor Object

The `FormatColor` object specifies the color for a given condition in a conditional formatting rule. The `FormatColor` object is applied to the thresholds of a color scale conditional format or the color of a data bar conditional format. You can pass a color to the condition using an RGB value or a color index. See the `ColorScale` object in this appendix to see how the `FormatColor` is used to assign colors.

FormatColor Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

FormatColor Properties

Name	Returns	Description
Color	Variant	Set/Get the color of the font. Use the RGB function to create the color value
ColorIndex	XlColorIndex	Set/Get the color of the font. Use the XlColorIndex constants or an index value in the current color palette
ThemeColor	xlThemeColor	Set/Get the theme color in the applied color scheme associated with an object. Should the object have no association with a theme, then trying to access the ThemeColor property will result in an error
TintAndShade	Single	Set/Get a Single value from -1 (darkest) to 1 (lightest), which darkens or lightens a color. Zero (0) is neutral

FormatCondition Object and the FormatConditions Collection

The `FormatConditions` collection contains the conditional formatting associated with the particular range of cells. The Parent of the `FormatConditions` collection is the `Range` object. Each `FormatCondition` object represents some formatting that will be applied if the condition is met.

FormatConditions Common Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

FormatConditions Methods

Name	Returns	Parameters	Description
Add	Object	Several	Adds a new formatting condition
AddAboveAverage	AboveAverage		Adds a new AboveAverage formatting condition
AddColorScale	ColorScale	ColorScaleType As Long	Adds a new ColorScale formatting condition
AddCompareColumns	CompareColumns		Adds a new CompareColumns formatting condition
AddDatabar	Databar		Adds a new Databar formatting condition

Name	Returns	Parameters	Description
AddIconSetCondition	IconSetCondition		Adds a new IconSetCondition formatting condition
AddTop10	Top10		Adds a new Top10 formatting condition
AddUniqueValues	UniqueValues		Adds a new UniqueValues formatting condition
Delete			Deletes a formatting condition
Item			Returns a single object from the collection

FormatCondition Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

FormatCondition Properties

Name	Returns	Description
AppliesTo	Range	Read-only. Returns the range that is affected by the formatting rule
Borders	Borders	Read-only. Returns a collection holding all the individual border attributes for the formatting condition
Font	Font	Read-only. Returns an object containing Font options for the formatting condition
FormatRow	Boolean	Set/Get the Boolean value specifying if the entire Excel table row should be formatted. The default value is False
Formula1	String	Read-only. Returns the value that the cells must contain or an expression or formula evaluating to True/False. If the formula or expression evaluates to True, then the formatting is applied
Formula2	String	Read-only. Returns the value that the cells must contain or an expression evaluating to True/False. Valid only if the Operator property is xlBetween or xlNotBetween
Interior	Interior	Read-only. Returns an object containing options to format the inside area for the formatting condition (for example, interior color)
NumberFormat	Variant	Set/Get the number format applied to a cell if the conditional formatting rule evaluates to true

Table continued on following page

FormatCondition Methods

Name	Returns	Description
Operator	xlFormatConditionOperator	Read-only. Returns the operator to apply to the Formula1 and Formula2 property. Use the xlFormatConditionOperator constants
Period	xlTimePeriods	Set/Get the time period used to apply the conditional formatting rule. Use one of the xlTimePeriods constants
Priority	Long	Set/Get the priority value of a conditional formatting rule, determining the order of evaluation when other rules are in effect
PTCondition	Boolean	Read-only. Indicates whether the formatting rule is applied to a PivotTable chart
ScopeType	xlPivotConditionScope	Set/Get the scope of the formatting rule when applied to a PivotTable chart. Use the xlPivotConditionScope constants
StopifTrue	Boolean	Set/Get a Boolean value that determines if additional formatting rules should be applied if the current rule evaluates to True. The default value is True
Text	String	Set/Get text string used in the formatting rule
TextOperator	XlContainsOperator	Set/Get an XlContainsOperator constant, specifying the text search performed by the formatting rule
Type	xlFormatConditionType	Read-only. Returns an xlFormatConditionType constant, which specifies the type of conditional formatting being applied

FormatCondition Methods

Name	Returns	Parameters	Description
Delete			Deletes the formatting condition
Modify		Type As xlFormatConditionType, [Operator], [Formula1], [Formula2] [String], [Operator2]	Modifies the formatting condition. Since all the properties are read-only, this is the only way to modify the format condition
ModifyAppliesToRange		Range As Range	Sets the range for which the formatting rule will be applied

FormatCondition Object and the FormatConditions Collection Example

Name	Returns	Parameters	Description
SetFirst Priority			Sets the priority value for the formatting rule so that it is evaluated before all other rules on the worksheet
SetLast Priority			Sets the priority value for the formatting rule so that it is evaluated after all other rules on the worksheet

FormatCondition Object and the FormatConditions Collection Example

Refer to the `AboveAverage` object in this appendix to see how the `FormatCondition` object is used.

FreeformBuilder Object

The `FreeformBuilder` object is used by the parent `Shape` object to create new freehand shapes. The `BuildFreeform` method of the `Shape` object is used to return a `FreeformBuilder` object.

FreeformBuilder Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

FreeformBuilder Methods

Name	Returns	Parameters	Description
AddNodes		<code>SegmentType</code> As <code>MsoSegmentType</code> , <code>EditingType</code> As <code>MsoEditingType</code> , <code>X1</code> As Single, <code>Y1</code> As Single, <code>[X2]</code> , <code>[Y2]</code> , <code>[X3]</code> , <code>[Y3]</code>	This method adds a point in the current shape being drawn. A line is drawn from the current node being added to the last node added. <code>SegmentType</code> describes the type of line to add between the nodes. <code>X1</code> , <code>Y1</code> , <code>X2</code> , <code>Y2</code> , <code>X3</code> , <code>Y3</code> is used to define the position of the current node being added. The coordinates are taken from the upper-left corner of the document
ConvertToShape	Shape		Converts the nodes added above into a <code>Shape</code> object

FreeformBuilder Object Example

```
Sub MakeArch()  
    Dim oFFB As FreeformBuilder  
    'Create a new freeform builder  
    Set oFFB = ActiveSheet.Shapes.BuildFreeform(msoEditingCorner, 100, 300)  
    'Add the lines to the builder  
    With oFFB  
        .AddNodes msoSegmentLine, msoEditingAuto, 100, 200  
        .AddNodes msoSegmentCurve, msoEditingCorner, 150, 150, 0, 0, 200, 200  
        .AddNodes msoSegmentLine, msoEditingAuto, 200, 300  
        .AddNodes msoSegmentLine, msoEditingAuto, 100, 300  
    'Convert it to a shape  
        .ConvertToShape  
    End With  
End Sub
```

Graphic Object

Represents a picture that can be placed in any one of the six locations of the Header and Footer in the Page Setup of a sheet. It's analogous to using both the Insert Picture and Format Picture buttons in the Header or Footer dialogs inside the Page Setup command.

It's important to note that none of the Property settings of this object will result in anything appearing in the Header or Footer, unless you insert "&G" (via VBA code) in any of the six different areas of the Header or Footer.

Graphic Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

Graphic Properties

Name	Returns	Description
Brightness	Single	Set/Get the brightness of the specified picture. This property's value must be from 0.0 (dimkest) to 1.0 (brightest)
ColorType	MsoPicture ColorType	Set/Get the color transformation applied to the specified picture or OLE object
Contrast	Single	Set/Get the contrast of the specified picture. This property's value must be from 0.0 (least) to 1.0 (greatest)
CropBottom	Single	Set/Get the number of points that are cropped off the bottom of the specified picture or OLE object
CropLeft	Single	Set/Get the number of points that are cropped off the left-hand side of the specified picture or OLE object
CropRight	Single	Set/Get the number of points that are cropped off the right-hand side of the specified picture or OLE object
CropTop	Single	Set/Get the number of points that are cropped off the top of the specified picture or OLE object

Name	Returns	Description
Filename	String	Set/Get the URL or path to where the specified object was saved
Height	Single	Set/Get the height of the object
LockAspect Ratio	MsoTriState	Set/Get whether the specified shape retains its original proportions when you resize it
Width	Single	Set/Get the width of the object

Graphic Object Example

The following routine prompts the user for a graphic file. If chosen, it places the graphic in the header of the active sheet as a watermark and sizes it to fit the page:

```

Sub AddWatermark()
    Dim oSheet As Object
    Dim sFile As String

    On Error Resume Next
        Set oSheet = ActiveSheet
    On Error GoTo 0

    'Make sure there is an active sheet
    If Not oSheet Is Nothing Then

        'Set the properties of the File Open dialog
        With Application.FileDialog(msoFileDialogFilePicker)

            'Change the default dialog title
            .Title = "Insert Graphic In Center Header"

            'Allow only one file
            .AllowMultiSelect = False

            'Clear the filters and create your own
            'Switch to the custom filter before showing the dialog

            .Filters.Add "All Pictures", _
            "*.gif; *.jpg; *.jpeg; *.bmp; *.wmf; *.gif;*.emf;*.dib;*.jfif;*.jpe", 1

            'Show thumbnails to display small representation
            ' of the image
            .InitialView = msoFileDialogViewThumbnail

            'Show the dialog
            '-1 means they didn't cancel
            If .Show = -1 Then
                'Store the chosen file
                sFile = .SelectedItems(1)

                'Set up the graphic in the Header

```

Gridlines Object

```
With oSheet.PageSetup
    With .CenterHeaderPicture
        .Filename = sFile
        .ColorType = msoPictureWatermark
        .LockAspectRatio = True

        'Make it fill the page
        'c Assumes a letter size portrait)
        .Width = Application.InchesToPoints(17)
    End With

    'Make the graphic appear
    'Without this, nothing happens
    .CenterHeader = "&G"
End With

End If

'Remove the filter when done
.Filters.Clear

End With

End If

End Sub
```

Gridlines Object

The `Gridlines` object contains formatting properties associated with the major and minor gridlines on a chart's axes. The gridlines are an extension of the tick marks seen in the background of a chart, allowing the end user to more easily see what a chart object's value is. The parent of the `Gridlines` object is the `Axis` object. To make sure the object is valid and to create the `Gridlines` object, use the `HasMajorGridlines` and `HasMinorGridlines` properties of the `Axis` object first.

Gridlines Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Gridlines Properties

Name	Returns	Description
<code>Border</code>	<code>Border</code>	Read-only. Returns the border's properties around the gridlines
<code>Format</code>	<code>ChartFormat</code>	Returns the <code>ChartFormat</code> object, which controls the line, fill, and effect formatting for the chart area
<code>Name</code>	<code>String</code>	Read-only. Returns the name of the <code>Gridlines</code> object

Gridlines Methods

Name	Returns	Parameters	Description
Delete	Variant		Deletes the Gridline object
Select	Variant		Selects the gridlines on the chart

Gridlines Object Example

```

Sub AddAndFormatGridLines()
    Dim oChart As Chart
    Dim oGridline As Gridlines

    'Set the target chart and set up the gridline object
    Set oChart = ActiveSheet.ChartObjects("Chart 1").Chart
    Set oGridline = oChart.Axes(xlValue).MajorGridlines

    'Add MajorGridlines
    oChart.Axes(xlValue).HasMajorGridlines = True

    'Format gridlines
    With oGridline
        .Border.Weight = xlMedium
        .Border.LineStyle = xlContinuous
        .Border.ColorIndex = 3
    End With
End Sub

```

GroupShapes Collection

The `GroupShapes` collection holds all of the shapes that make up a grouped shape. The `GroupShapes` collection holds a collection of `Shape` objects. The parent of the `GroupShapes` object is the `Shape` object.

The `GroupShapes` collection only has one property besides the typical collection attributes. The `Range` property returns a subset of the shapes in the `Shapes` collection.

HeaderFooter Object

The `HeaderFooter` object represents a single header or footer. The `HeaderFooter` object is a member of the `HeadersFooters` collection. The `HeadersFooters` collection includes all headers and footers in the specified workbook section. You will notice that the `HeadersFooters` collection is not exposed through the object model. As a result, you cannot explicitly add `HeaderFooter` objects to the `HeadersFooters` collection.

HeaderFooter Properties

HeaderFooter Properties

Name	Returns	Description
Picture	Graphic	Read-only. Returns a <code>Picture</code> object that represents a picture field included in the specified header or footer
Text	String	Set/Get a <code>Text</code> object that represents text included in the specified header or footer

HiLoLines Object

The `HiLoLines` object contains formatting attributes for a chart's high-low lines. The parent of the `HiLoLines` object is the `ChartGroup` object. High-low lines connect the largest and smallest points on a 2D line chart group.

HiLoLines Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

HiLoLines Properties

Name	Returns	Description
Border	Border	Read-only. Returns the border's properties around the high-low lines
Format	ChartFormat	Returns the <code>ChartFormat</code> object, which controls the line, fill, and effect formatting for the <code>HiLoLines</code>
Name	String	Read-only. Returns the name of the <code>HiLoLines</code> object

HiLoLines Methods

Name	Returns	Parameters	Description
Delete	Variant		Deletes the high-low lines
Select			Selects the object

HPageBreak Object and the HPageBreaks Collection

The `HPageBreaks` collection contains all of the horizontal page breaks in the printable area of the parent object. Each `HPageBreak` object represents a single horizontal page break for the printable area of the parent object. Possible parents of the `HPageBreaks` collection are the `WorkSheet`, `Worksheets`, and `Chart` objects.

The `HPageBreaks` collection contains one method besides the typical collection attributes. The `Add` method is used to add an `HPageBreak` object to the collection (and horizontal page break to the sheet). The `Add` method has a `Before` parameter to specify the range above where the horizontal page break will be added.

HPageBreak Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

HPageBreak Properties

Name	Returns	Description
<code>Extent</code>	<code>XlPageBreak Extent</code>	Read-only. Returns whether the horizontal page break is full-screen or only for the print area
<code>Location</code>	<code>Range</code>	Set/Get the cell where the horizontal page break is located. The top edge of the cell is the location of the page break
<code>Type</code>	<code>XlPageBreak</code>	Set/Get whether the page break is automatic or manually set

HPageBreak Methods

Name	Returns	Parameters	Description
<code>Delete</code>			Deletes the page break
<code>DragOff</code>		<code>Direction As XlDirection</code> , <code>RegionIndex As Long</code>	Drags the page break out of the printable area. The <code>Direction</code> parameter specifies the direction the page break is dragged. The <code>RegionIndex</code> parameter specifies which print region the page break is being dragged out of

HPageBreak Object and the HPageBreaks Collection Example

```
Sub AddHPageBreaks()
    Dim oCell As Range
    'Loop through all the cells in the first column of the sheet
    For Each oCell In ActiveSheet.UsedRange.Columns(1).Cells
        'If the font size is 16, add a page break above the cell
        If oCell.Font.Size = 16 Then
            ActiveSheet.HPageBreaks.Add oCell
        End If
    Next
End Sub
```

Hyperlink Object and the Hyperlinks Collection

The `Hyperlinks` collection represents the list of hyperlinks in a worksheet or range. Each `Hyperlink` object represents a single hyperlink in a worksheet or range. The `Hyperlinks` collection has an `Add` and `Delete` method besides the typical collection of properties and methods. The `Add` method takes the text

Hyperlink Common Properties

or graphic that is to be converted into a hyperlink (`Anchor`) and the URL address or filename (`Address`), and creates a `Hyperlink` object. The `Delete` method deletes the `Hyperlinks` in the collection.

Hyperlink Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Hyperlink Properties

Name	Returns	Description
<code>Address</code>	<code>String</code>	Set/Get the filename or URL address of the hyperlink
<code>EmailSubject</code>	<code>String</code>	Set/Get the e-mail subject line if the address is an e-mail address
<code>Name</code>	<code>String</code>	Read-only. Returns the name of the hyperlink
<code>Range</code>	<code>Range</code>	Read-only. Returns the range in the document where the hyperlink is
<code>ScreenTip</code>	<code>String</code>	Set/Get the text that appears when the mouse hovers over the hyperlink
<code>Shape</code>	<code>Shape</code>	Read-only. Returns the shape associated with the hyperlink, if any
<code>SubAddress</code>	<code>String</code>	Set/Get the spot in the target location that the hyperlink points to
<code>TextToDisplay</code>	<code>String</code>	Set/Get the text to be displayed for the specified hyperlink. The default value is the address of the hyperlink
<code>Type</code>	<code>Long</code>	Set/Get the target location of the HTML frame of the address

Hyperlink Methods

Name	Parameters	Description
<code>AddTo Favorites</code>		Adds the <code>Address</code> property to the <code>Favorites</code> folder
<code>CreateNew Document</code>	<code>Filename</code> As <code>String</code> , <code>EditNow</code> As <code>Boolean</code> , <code>Overwrite</code> As <code>Boolean</code>	Creates a new document with the <code>FileName</code> name from the results of the hyperlink's address. Set the <code>EditNow</code> property to <code>True</code> to open up the document in the appropriate editor. Set <code>Overwrite</code> to <code>True</code> to overwrite any existing document with the same name
<code>Delete</code>		Deletes the <code>Hyperlink</code> object

Name	Parameters	Description
Follow	[NewWindow], [AddHistory], [ExtraInfo], [Method], [HeaderInfo]	Opens up the target document specified by the Address property. Setting NewWindow to True opens up a new window with the target document. Set AddHistory to True to display the item in the history folder. Use the Method parameter to choose if the ExtraInfo property is sent as a Get or a Post

Hyperlink Object and the Hyperlinks Collection Example

This example creates a hyperlink-based Table of Contents worksheet:

```
Sub CreateHyperlinkTOC()  
    Dim oBk As Workbook  
    Dim oShtTOC As Worksheet, oSht As Worksheet  
    Dim iRow As Integer  
    Set oBk = ActiveWorkbook  
    'Add a new sheet to the workbook  
    Set oShtTOC = oBk.Worksheets.Add  
    With oShtTOC  
        'Add the title to the sheet  
        .Range("A1").Value = "Table of Contents"  
        'Add Mail and web hyperlinks  
        .Hyperlinks.Add .Range("A3"), "mailto:Me@MyISP.com", _  
            TextToDisplay:="Email your comments"  
        .Hyperlinks.Add .Range("A4"), "http://www.wrox.com", _  
            TextToDisplay:="Visit Wrox Press"  
    End With  
    'Loop through the sheets in the workbook  
    'adding location hyperlinks  
    iRow = 6  
    For Each oSht In oBk.Worksheets  
    If oSht.Name <> oShtTOC.Name Then  
        oShtTOC.Hyperlinks.Add oShtTOC.Cells(iRow, 1), "", _  
            SubAddress:="" & oSht.Name & "!A1", _  
            TextToDisplay:=oSht.Name  
        iRow = iRow + 1  
    End If  
    Next  
End Sub
```

Icon Object

The `Icon` object represents a single icon used in a conditional formatting rule.

Icon Common Properties

Icon Common Properties

The `Application` and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Index</code>	<code>Long</code>	Read-only. Returns a value that represents the index number of a given icon within the <code>IconSet</code> object

IconCriteria and the IconCriteria Collection

The `IconCriteria` collection holds each `IconCriteria` in an icon set conditional format. Each criterion defines the range of values and the threshold type associated with each icon in an icon set conditional formatting rule.

IconCriteria Common Properties

The `Count` and `Item` properties are defined at the beginning of this appendix.

IconCriteria Properties

Name	Returns	Description
<code>Index</code>	<code>Long</code>	Returns a value that represents the threshold for the criteria. 1 would represent the first threshold, 2 would represent the second, and so on. Read-only
<code>Operator</code>	<code>XlFormatConditionOperator</code>	Set/Get whether the threshold is greater than or greater than or equal to the threshold value. This property uses either the <code>xlGreater</code> or <code>xlGreaterEqual</code>
<code>Type</code>	<code>xlConditionValueTypes</code>	Specifies how the threshold values for an icon set conditional format are determined (number, percent, formula, or percentile). This property will return an <code>xlConditionValueTypes</code> constant. Read-only
<code>Value</code>	<code>Variant</code>	Set/Get the value for each threshold in an icon set conditional formatting rule

IconSet and the IconSets Collection

The `IconSets` collection holds each `IconSet` that is available for use in building a formatting condition. Each `IconSet` represents a single set of icons and is assigned by using the `xlIconSet` enumeration as an index of the `IconSet` property of the `Workbook` object.

IconSets Common Properties

The `Application`, `Count`, `Creator`, `Item`, and `Parent` properties are defined at the beginning of this appendix.

IconSet Properties

The `Application`, `Count`, `Creator`, `Item`, and `Parent` properties are defined at the beginning of this appendix.

IconSet Properties

Name	Returns	Description
ID	<code>xlIconSet</code>	Read-only. Returns an <code>xlIconSet</code> constant identifying the icons that are used in the formatting condition

IconSetCondition Object

The `IconSetCondition` object allows you to create a formatting rule by using either the `Add` or `AddIconSetCondition` method of the `FormatConditions` collection. You can apply any one of the built-in icon sets to a range of values, assigning each value an icon based on thresholds set via the `IconCriteria` collection.

IconSetCondition Properties

The `Application`, `Parent`, and `Creator` properties are defined at the beginning of this appendix.

IconSetCondition Properties

Name	Returns	Description
<code>AppliesTo</code>	<code>Range</code>	Read-only. Returns the range that is affected by the formatting rule
<code>FormatRow</code>	<code>Boolean</code>	Set/Get the Boolean value specifying if the entire Excel table row should be formatted. The default value is <code>False</code>
<code>Formula</code>	<code>String</code>	Set/Get a string representing a formula that determines the values that are to be evaluated in the conditional formatting rule
<code>IconCriteria</code>	<code>IconCriteria</code>	Read-only. Returns an <code>IconCriteria</code> collection, which is used to specify the thresholds of the criteria used in the formatting condition
<code>IconSet</code>	<code>IconSets</code>	Set/Get the specific built-in icon set used in the formatting condition

Table continued on following page

IconSetCondition Methods

Name	Returns	Description
Percentile Values	Boolean	Set/Get whether the thresholds for an icon set conditional format are determined by using percentiles. Setting this property to <code>True</code> will ensure that all values are evaluated based on percentiles
Priority	Long	Set/Get the priority value of a conditional formatting rule, determining the order of evaluation when other rules are in effect
PTCondition	Boolean	Read-only. Indicates whether the formatting rule is applied to a PivotTable chart
ReverseOrder	Boolean	Set/Get whether the formatting rule should be applied in reverse order. For example, where low values are tagged as red, setting this property would tag low values as green
ScopeType	<code>xlPivotConditionScope</code>	Set/Get the scope of the formatting rule when applied to a PivotTable chart. Use the <code>xlPivotConditionScope</code> constants
ShowIconOnly	Boolean	Setting this property to <code>True</code> ensures that only the icons will be displayed in the cell, as opposed to both the data and the icon
StopifTrue	Boolean	Set/Get a Boolean value that determines if additional formatting rules should be applied if the current rule evaluates to <code>True</code> . The default value is <code>True</code>
Type	<code>xlFormatConditionType</code>	Read-only. Returns an <code>xlFormatConditionType</code> constant, which specifies the type of conditional formatting being applied. This object will always return a value of 12 since it corresponds to the <code>xlAboveAverageCondition</code>

IconSetCondition Methods

Name	Parameters	Description
Delete		Deletes the object
ModifyAppliesToRange	Range As Range	Sets the range for which the formatting rule will be applied
SetFirstPriority		Sets the priority value for the formatting rule so that it is evaluated before all other rules on the worksheet
SetLastPriority		Sets the priority value for the formatting rule so that it is evaluated after all other rules on the worksheet

IconSetCondition Object Example

This example adds an icon condition that tags a range of values with one of three colored flags:

```
Sub CreateIconConditions()
Dim oIconCondition As IconSetCondition

'Add an icon set condition
Set oIconCondition = Range("F6:F16").FormatConditions.AddIconSetCondition

'Choose the 3Flags icon set
oIconCondition.IconSet = ActiveWorkbook.IconSets(xl3Flags)

'Set the threshold for Green flag; above 80 percent is green.
With oIconCondition.IconCriteria(3)
.Type = xlConditionValuePercent
.Operator = xlGreater
.Value = 80
End With

'Set the threshold for Yellow flag; 70 percent and above is yellow.
'Values that qualify for green will not be tagged as yellow.
'Any values that do not qualify for yellow or green will be red.
With oIconCondition.IconCriteria(2)
.Type = xlConditionValuePercent
.Operator = xlGreater
.Value = 70
End With

'Show only the flag icons; not the data values
oIconCondition.ShowIconOnly = True

End Sub
```

Interior Object

The **Interior** object contains the formatting options associated with the inside area of the parent object. Possible parents of the **Interior** object are the **AboveAverage**, **AxisTitle**, **ChartArea**, **CellFormat**, **ChartObject**, **ChartTitle**, **DataLabel**, **DownBars**, **Floor**, **FormatCondition**, **Legend**, **LegendKey**, **OLEObject**, **PlotArea**, **Point**, **Range**, **Series**, **Style**, **TableStyleElement**, **Top10**, **Style**, **Upbars**, and **Walls** objects. The **ChartObjects**, **DataLabels**, **OLEObjects**, and **UniqueValues** collections also are possible parents of the **Interior** object.

Interior Common Properties

The **Application**, **Creator**, and **Parent** properties are defined at the beginning of this appendix.

Interior Properties

Name	Returns	Description
Color	Variant	Set/Get the color of the interior. Use the RGB function to create the color value
ColorIndex	Variant	Set/Get the color of the interior. Use the XlColorIndex constants or an index value in the current color palette
GradientColorType	MsoGradientColorType	Read-only. Returns what type of gradient fill color concept is used
GradientStyle	MsoGradientStyle	Read-only. Returns the orientation of the gradient that is used
GradientType	Long	Read-only. Returns the gradient type used for the specified Interior object
GradientVariant	Integer	Read-only. Returns the variant used for the gradient from the center
InvertIfNegative	Variant	Set/Get whether the color in the interior of the parent object is inverted if the values are negative
InteriorGradientStops	Long	Read-only. Returns an InteriorGradientStops value of an Interior object
Pattern	XlPattern	Set/Get the pattern to use for the interior of the parent object. Use one of the XlPattern constants
PatternColor	Variant	Set/Get the color of the interior pattern. Use the RGB function to create the color value
PatternColorIndex	XlColorIndex	Set/Get the color of the interior pattern. Use the XlColorIndex constants or an index value in the current color palette
PatternThemeColor	Variant	Set/Get a theme color pattern for an Interior object
PatternTintAndShade	Variant	Set/Get a tint and shade pattern for an Interior object
ThemeColor	xlThemeColor	Set/Get the theme color in the applied color scheme associated with an object. Should the object have no association with a theme, then trying to access the ThemeColor property will result in an error
TintAndShade	Single	Set/Get a Single value from -1 (darkest) to 1 (lightest), which darkens or lightens a color. 0 is neutral

Interior Object Example

```

Sub FormatRange()
    Dim oInt As Interior
    'Get the interior of the current selection
    Set oInt = Selection.Interior
    'Format the interior in solid yellow
    '(color depends on the workbook palette)
    With oInt
        .Pattern = xlSolid
        .ColorIndex = 6
    End With
End Sub

```

IRtdServer Object

This object allows the ability to connect to a Real-Time Data Server (RTD). This type of server allows Excel to receive timed interval data updates without the need for extra coding. In prior versions of Excel, when regular updates were needed, you could use the `OnTime` method to set up regular data update intervals. RTDs send updates automatically based on an interval set within the server, or by using the `HeartbeatInterval` method of the `IRTDUpdateEvent` object.

This object is similar in nature to using the RTD worksheet function, which displays data at regular intervals in a worksheet cell.

Note that to use this object, you must instantiate it using the `Implements` keyword.

IRtdServer Methods

Name	Returns	Parameters	Description
<code>ConnectData</code>		<code>TopicID</code> As Long, <code>Strings</code> As Variant, <code>GetNewValues</code> As Boolean	Called when a file is opened that contains real-time data (RTD) functions, or when a new formula that contains a RTD function is entered
<code>DisconnectData</code>		<code>TopicID</code> As Long	Used to notify the RTD server that a topic is no longer in use
<code>Heartbeat</code>	Long		Checks to see if the RTD server is still active. Negative numbers or zero are a failure, while positive numbers indicate success

Table continued on following page

IRTDUpdateEvent Object

Name	Returns	Parameters	Description
RefreshData	Variant	ByRef TopicCount As Long	This method is called to get new data, but only after being notified by the RTD server that there is new data
ServerStart	Long	CallbackObject As IRTDUpdateEvent	Called immediately after an RTD server is instantiated. Negative numbers or zero are a failure, while positive numbers indicate success
ServerTerminate			Used to terminate the connection to the server

IRTDUpdateEvent Object

Represents real-time update events. This object is used to set the interval between updates for an `IrtDServer` object using the `HeartbeatInterval` property. This object is returned when you use the `ServerStart` method of the `IrtDServer` object to connect to a real-time data server.

IRTDUpdateEvent Properties

Name	Returns	Description
Heartbeat Interval	Long	Set/Get the interval between updates for RTD

IRTDUpdateEvent Methods

Name	Returns	Parameters	Description
Disconnect			Instructs the RTD server to disconnect from the specified object
UpdateNotify			Excel is informed by the RTD server that new data has been received

LeaderLines Object

The `LeaderLines` object contains the formatting attributes associated with leader lines on charts connecting data labels to the actual points. The parent of the `LeaderLines` object is the `Series` object. Use the `HasLeaderLines` property of the `Series` object to create a `LeaderLines` object and to make sure one exists.

LeaderLines Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

LeaderLines Properties

Name	Returns	Description
<code>Border</code>	<code>Border</code>	Read-only. Returns the border's properties around the leader lines
<code>Format</code>	<code>ChartFormat</code>	Returns the <code>ChartFormat</code> object, which controls the line, fill, and effect formatting for the chart area

LeaderLines Methods

Name	Returns	Parameters	Description
<code>Delete</code>			Deletes the <code>LeaderLines</code> object
<code>Select</code>			Selects the leader lines on the chart

LeaderLines Object Example

```
Sub AddAndFormatLeaderLines()
    Dim oChart As Chart
    Dim oLeaderLines As LeaderLines

    'Set the target chart
    Set oChart = ActiveSheet.ChartObjects("Chart 1").Chart

    'Apply labels and add leaderlines
    With oChart.SeriesCollection(1)
        .ApplyDataLabels
        .HasLeaderLines = True
    End With

    'Target the leaderlines object
    Set oLeaderLines = oChart.SeriesCollection(1).LeaderLines

    'Format the leaderlines
    With oLeaderLines
        .Border.LineStyle = xlContinuous
        .Border.ColorIndex = 5
    End With
End Sub
```

Legend Object

Legend Object

The `Legend` object contains the formatting options and legend entries for a particular chart. The parent of the `Legend` object is the `Chart` object. Use the `HasLegend` property of the `Chart` object to create a `Legend` object and to make sure one exists.

Legend Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Legend Properties

Name	Returns	Description
<code>AutoScaleFont</code>	<code>Variant</code>	Set/Get whether the font size will change automatically if the parent chart changes sizes
<code>Format</code>	<code>ChartFormat</code>	Read-only. Returns the <code>ChartFormat</code> object that controls the line, fill, and effect formatting for the chart area
<code>Height</code>	<code>Double</code>	Set/Get the height of the legend box
<code>IncludeInLayout</code>	<code>Boolean</code>	When set to <code>True</code> , the legend will occupy the chart layout space while a chart layout is being determined. Default is <code>True</code>
<code>Left</code>	<code>Double</code>	Set/Get the distance from the left edge of the legend box to the left edge of the chart containing the legend
<code>Name</code>	<code>String</code>	Read-only. Returns the name of the <code>Legend</code> object
<code>Position</code>	<code>XlLegendPosition</code>	Set/Get the position of the legend on the chart (for example, <code>xlLegendPositionCorner</code> , <code>xlLegendPositionLeft</code>)
<code>Shadow</code>	<code>Boolean</code>	Set/Get whether the legend has a shadow effect
<code>Top</code>	<code>Double</code>	Set/Get the distance from the top edge of the legend box to the top edge of the chart containing the legend
<code>Width</code>	<code>Double</code>	Set/Get the width of the legend box

Legend Methods

Name	Returns	Parameters	Description
<code>Clear</code>	<code>Variant</code>		Clears the legend
<code>Delete</code>	<code>Variant</code>		Deletes the legend

Name	Returns	Parameters	Description
LegendEntries	Object	[Index]	Returns either one <code>LegendEntry</code> object or a <code>LegendEntries</code> collection, depending if an <code>Index</code> parameter is specified. Contains all the legend text and markers
Select	Variant		Selects the legend on the chart

Legend Object Example

```

Sub AddAndFormatLegend()
    Dim oChart As Chart
    Dim oLegend As Legend

    'Set the target chart
    Set oChart = ActiveSheet.ChartObjects("Chart 1").Chart

    'Apply Legend
    oChart.HasLegend = True

    'Target the leaderlines object
    Set oLegend = oChart.Legend

    'Format the leaderlines
    With oLegend
        .Position = xlLegendPositionBottom
        .Border.LineStyle = xlNone
        .AutoScaleFont = False
    End With
End Sub

```

LegendEntry Object and the LegendEntries Collection

The `LegendEntries` collection contains the collection of entries in a legend. Each `LegendEntry` object represents a single entry in a legend. This consists of the legend entry text and the legend entry marker. The legend entry text is always the associated series name or trendline name. The parent of the `LegendEntries` collection is the `Legend` object. The `LegendEntries` collection has no properties or methods, outside the typical collection attributes listed at the beginning of this appendix.

LegendEntry Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

LegendEntry Properties

LegendEntry Properties

Name	Returns	Description
AutoScaleFont	Variant	Set/Get whether the font size will change automatically if the parent chart changes sizes
Font	Font	Read-only. Returns an object containing Font options for the legend entry text
Format	ChartFormat	Read-only. Returns the ChartFormat object, which controls the line, fill, and effect formatting for the chart area
Height	Double	Read-only. Returns the height of the legend entry
Index	Long	Read-only. Returns the position of the LegendEntry in the LegendEntries collection
Left	Double	Read-only. Returns the distance from the left edge of the legend entry box to the left edge of the chart
LegendKey	LegendKey	Read-only. Returns an object containing formatting associated with the legend entry marker
Top	Double	Read-only. Returns the distance from the top edge of the legend entry box to the top edge of the chart
Width	Double	Read-only. Returns the width of the legend entry

LegendEntry Methods

Name	Returns	Parameters	Description
Delete	Variant		Deletes the LegendEntry object
Select	Variant		Selects the legend entry on the chart

LegendEntry Object and the LegendEntries Collection Example

```
Sub FormatEachLegendEntry()  
    Dim oChart As Chart  
    Dim oEntry As LegendEntry  
    Dim oEntries As LegendEntries  
    'Set the target chart  
    Set oChart = ActiveSheet.ChartObjects("Chart 4").Chart  
  
    'Apply Legend  
    oChart.HasLegend = True  
  
    'Target the leaderlines object  
    Set oEntries = oChart.Legend.LegendEntries  
  
    For Each oEntry In oEntries  
        oEntry.Font.Size = 16  
    End For  
End Sub
```

```

oEntry.Font.ColorIndex = 15
Next oEntry
End Sub

```

LegendKey Object

The `LegendKey` object contains properties and methods to manipulate the formatting associated with a legend key entry marker. A legend key is a visual representation, such as a color, that identifies a specific series or trendline.

LegendKey Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

LegendKey Properties

Name	Returns	Description
<code>Format</code>	<code>ChartFormat</code>	Read-only. Returns the <code>ChartFormat</code> object that controls the line, fill, and effect formatting for the chart area
<code>Height</code>	<code>Double</code>	Read-only. Returns the height of the legend entry key
<code>InvertIf Negative</code>	<code>Boolean</code>	Set/Get whether the color in the legend key is inverted if the values are negative
<code>Left</code>	<code>Double</code>	Read-only. Returns the distance from the left edge of the legend key entry box to the left edge of the chart
<code>Marker Background Color</code>	<code>Long</code>	Set/Get the color of the legend key background. Use the <code>RGB</code> function to create the color value
<code>Marker Background ColorIndex</code>	<code>XlColorIndex</code>	Set/Get the color of the legend key background. Use the <code>XlColorIndex</code> constants or an index value in the current color palette
<code>Marker Foreground Color</code>	<code>Long</code>	Set/Get the color of the legend key foreground. Use the <code>RGB</code> function to create the color value
<code>Marker Foreground ColorIndex</code>	<code>XlColor Index</code>	Set/Get the color of the legend key foreground. Use the <code>XlColorIndex</code> constants or an index value in the current color palette
<code>MarkerSize</code>	<code>Long</code>	Set/Get the size of the legend key marker
<code>MarkerStyle</code>	<code>XlMarker Style</code>	Set/Get the type of marker to use as the legend key (for example, square, diamond, triangle, picture, and so on)
<code>PictureType</code>	<code>Long</code>	Set/Get how an associated picture is displayed on the legend (for example, stretched, tiled). Use the <code>XlPicture- Type</code> constants

Table continued on following page

LegendKey Methods

Name	Returns	Description
PictureUnit	Long	Set/Get how many units a picture represents if the <code>PictureType</code> property is set to <code>x1Scale</code>
PictureUnit2	Double	Set/Get how many units a picture represents if the <code>PictureType</code> property is set to <code>x1Scale</code>
Shadow	Boolean	Set/Get whether a shadow effect appears around the legend entry key
Smooth	Boolean	Set/Get whether the legend key has smooth curving enabled
Top	Double	Read-only. Returns the distance from the top edge of the legend entry key box to the top edge of the chart
Width	Double	Read-only. Returns the width of the legend entry key box

LegendKey Methods

Name	Returns	Parameters	Description
ClearFormats	Variant		Clears the formatting made on the <code>LegendKey</code> object
Delete	Variant		Deletes the <code>LegendKey</code> object

LinearGradient Object

The `LinearGradient` object transitions through a series of colors in a linear manner along a specific angle. Attempting to access a `Gradient` property of an `Interior` object that does not have an existing gradient fill will result in a run-time error. Be aware of the `Interior.Pattern` property before accessing the `Gradient` property.

LinearGradient Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

LinearGradient Properties

Name	Returns	Description
ColorStops	ColorStops	Read-only. Returns the <code>ColorStops</code> for the <code>LinearGradient</code> object
Degree	Double	Set/Get the angle of the linear gradient fill within a selection

LineFormat Object

The `LineFormat` object represents the formatting associated with the line of the parent `Shape` object. The `Line` property of the `Shape` object is used to access the `LineFormat` object. The `LineFormat` object is commonly used to change line properties such as arrowhead styles and directions.

LineFormat Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

LineFormat Properties

Name	Returns	Description
<code>BackColor</code>	<code>ColorFormat</code>	Read-only. Returns an object allowing manipulation of the background color of the line
<code>BeginArrowheadLength</code>	<code>MsoArrowheadLength</code>	Set/Get the arrowhead length on the start of the line
<code>BeginArrowheadStyle</code>	<code>MsoArrowheadStyle</code>	Set/Get how the arrowhead looks on the start of the line
<code>BeginArrowheadWidth</code>	<code>MsoArrowheadWidth</code>	Set/Get the arrowhead width on the start of the line
<code>DashStyle</code>	<code>MsoLineDashStyle</code>	Set/Get the style of the line
<code>EndArrowheadLength</code>	<code>MsoArrowheadLength</code>	Set/Get the arrowhead length on the end of the line
<code>EndArrowheadStyle</code>	<code>MsoArrowheadStyle</code>	Set/Get how the arrowhead looks on the end of the line
<code>EndArrowheadWidth</code>	<code>MsoArrowheadWidth</code>	Set/Get the arrowhead width on the end of the line
<code>ForeColor</code>	<code>ColorFormat</code>	Read-only. Returns an object allowing manipulation of the background color of the line
<code>Pattern</code>	<code>MsoPatternType</code>	Set/Get the pattern used on the line
<code>Style</code>	<code>MsoLineStyle</code>	Set/Get the line style
<code>Transparency</code>	<code>Single</code>	Set/Get how transparent (1) or opaque (0) the line is
<code>Visible</code>	<code>MsoTriState</code>	Set/Get whether the line is visible
<code>Weight</code>	<code>Single</code>	Set/Get how thick the line is

LineFormat Object Example

```
Sub AddAndFormatLine()  
    Dim oShp As Shape  
    Dim oLF As LineFormat  
    'Add a line shape  
    Set oShp = ActiveSheet.Shapes.AddLine(100, 100, 200, 250)  
    'Get the line format object  
    Set oLF = oShp.Line  
    'Set the line format  
    With oLF  
        .BeginArrowheadStyle = msoArrowheadOval  
        .EndArrowheadStyle = msoArrowheadTriangle  
        .EndArrowheadLength = msoArrowheadLong  
        .EndArrowheadWidth = msoArrowheadWide  
        .Style = msoLineSingle  
    End With  
End Sub
```

LinkFormat Object

The `LinkFormat` object represents the linking attributes associated with an OLE object or picture. The `LinkFormat` object is associated with a `Shape` object. Only `Shape` objects that are valid OLE objects can access the `LinkFormat` object.

LinkFormat Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

LinkFormat Properties

Name	Returns	Description
<code>AutoUpdate</code>	Boolean	Set/Get whether the parent <code>Shape</code> object is updated whenever the source file changes, or when the parent object is opened
<code>Locked</code>	Boolean	Set/Get whether the parent <code>Shape</code> object does not update itself against the source file

LinkFormat Methods

Name	Returns	Parameters	Description
<code>Update</code>			Updates the parent <code>Shape</code> object with the source file data

LinkFormat Object Example

```

Sub UpdateShapeLinks ()
    Dim oShp As Shape
    Dim oLnkForm As LinkFormat
    'Loop through all the shapes in the sheet
    For Each oShp In ActiveSheet.Shapes
        'Is it a linked shape?
        If oShp.Type = msoLinkedOLEObject Or oShp.Type = msoLinkedPicture Then
            'Yes, so get the link format
            Set oLnkForm = oShp.LinkFormat
            'and update the link
            oLnkForm.Update
        End If
    Next
End Sub

```

ListColumn and ListColumns Collection

The `ListColumn` object represents a column in a `List`. The `ListColumns` collection contains all columns within a list, represented by one or more `ListColumn` objects.

ListColumns Common Properties

The `Application`, `Count`, `Creator`, `Item`, and `Parent` properties are defined at the beginning of this appendix. The only method for the `ListColumns` object is the `Add` method, which adds a new column to a given list.

ListColumn Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ListColumn Properties

Name	Returns	Description
<code>DataBodyRange</code>	<code>Range</code>	Read-only. Returns the range that holds the values for the list object. Excludes header row
<code>Index</code>	<code>Long</code>	Index number of <code>ListColumn</code> object in the <code>List</code>
<code>Name</code>	<code>String</code>	Name of this object
<code>Range</code>	<code>Range</code>	Returns a range representing the range for which the current <code>ListColumn</code> applies to the current list. This includes the header row
<code>Total</code>		Read-only. Returns the Total row for a <code>ListColumn</code> object
<code>TotalsCalulation</code>	<code>XlTotalsCalculation</code>	Determines the type of calculation in the totals row
<code>XPath</code>	<code>XPath</code>	Returns an <code>XPath</code> object for the element mapped to the specific range

ListColumn Methods

ListColumn Methods

Name	Returns	Parameters	Description
Delete			Deletes a column of data in the list

ListColumns Object Example

```
Sub CreateEmptyListObject()  
Dim oList As ListObject  
Dim oColumn As ListColumn  
Dim i As Integer  
  
'Create a new list in A1.  
Range("A1").Select  
Set oList = ActiveSheet.ListObjects.Add  
  
'Add 9 more columns to the list object  
For i = 1 To 9  
    oList.ListColumns.Add  
Next i  
  
'Enumerate through each column and name it  
i = 1  
For Each oColumn In oList.ListColumns  
oColumn.Name = "Field-" & i  
i = i + 1  
Next  
End Sub
```

ListDataFormat Object

The `ListDataFormat` object holds all of the data type properties for a `ListColumn` object. All properties of the `ListDataFormat` object are read-only.

ListDataFormat Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ListDataFormat Properties

Name	Returns	Description
AllowFillIn	Boolean	Determines if the user can provide values or if they are restricted to using information from a list
Choices	String Array	Contains a string array of choices
DecimalPlaces	Long	Determines the number of decimal places to show
DefaultValue	Variant	Default value to use for this item

Name	Returns	Description
IsPercent	Boolean	Determines if this item should be displayed as a percentage
LCID	Long	Determines the currency symbol that is used
MaxCharacters	Long	The maximum number of characters that can be entered
MaxNumber	VARIANT	The largest number used
MinNumber	VARIANT	The minimum number that can be used
ReadOnly	Boolean	Determines if the item should be read-only and therefore disallow changes
Required	Boolean	Determines if the item is required
Type	XlListDataType	Used when a item links to a SharePoint site

ListObject Object and the ListObjects Collection

The `ListObject` object represents a list table within a workbook. The `ListObjects` collection represents all of the list objects on a worksheet.

ListObjects Common Properties

The `Application`, `Count`, `Creator`, `Item`, and `Parent` properties are defined at the beginning of this appendix. The only method for the `ListObjects` object is the `Add` method, which adds a new `ListObject` to a given worksheet.

ListObject Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ListObject Properties

Name	Returns	Description
Active	Boolean	Read-only. Indicates whether the active cell is inside the range of the <code>ListObject</code> object
AutoFilter	AutoFilter	Read-only. Filters a list using the <code>AutoFilter</code>
Comment	String	Set/Get any comments associated with the list object
DataBodyRange	Range	Read-only. Returns the range that holds the values for the list object. Excludes header row
DisplayName		Set/Get the display name for the specified <code>ListObject</code> object

Table continued on following page

ListObject Properties

Name	Returns	Description
DisplayRight ToLeft	Boolean	Read-only. Determines whether a given list object is displayed left to right or right to left. If the property is set to <code>True</code> , then the values are displayed right to left
HeaderRowRange	Range	Read-only. Returns the range that holds the header row for the list object
InsertRowRange	Range	Read-only. Returns the range that represents the insert row
ListColumns	ListColumns	Read-Only. Returns a collection of all columns within the given list object
ListRows	ListRows	Read-only. Returns a collection of all rows within the given list object
Name	String	Set/Get the name of the list object
QueryTable	QueryTable	Read-only. Returns a <code>QueryTable</code> object that provides a link for the <code>ListObject</code> object to a list server
Range	Range	Set/Get the range to which the list object is applied
SharePointURL	String	Read-only. Returns the URL of the SharePoint list to which the given list object is linked
ShowAutoFilter	Boolean	Set/Get whether <code>AutoFilter</code> is applied. Default is <code>True</code>
ShowHeaders	Boolean	Set/Get whether the header row is displayed. Default is <code>True</code>
ShowTableStyle ColumnStripes	Boolean	Set/Get whether the alternate color banding is applied to the columns of the <code>ListObject</code>
ShowTableStyle FirstColumn	Boolean	Set/Get whether the first column is displayed for a given <code>ListObject</code> object
ShowTableStyle LastColumn	Boolean	Set/Get whether the last column is displayed for a given <code>ListObject</code> object
ShowTableStyle RowStripes	Boolean	Set/Get whether the alternate color banding is applied to the rows of the <code>ListObject</code>
ShowTotals	Boolean	Set/Get whether the Total row is displayed. Default is <code>True</code>
Sort	Sort	Read-only. Returns the sort criteria for the <code>ListObject</code>
SourceType	xListObject SourceType	Set/Get an <code>xListObjectSourceType</code> constant that determines the source of the given list object
TableStyle	Variant	Set/Get the style applied to the <code>ListObject</code>
TotalsRowRange	Range	Read-only. Returns the range that represents the Totals row
XmlMap	XmlMap	Read-only. Returns the schema map used as the source for the given list object

ListObject Methods

Name	Returns	Parameters	Description
Delete			Deletes an item from the collection
ExportToVisio			Exports a ListObject object to Visio
Publish	String	[Target - String Array] [LinkSource - Boolean]	Publishes the object to a Windows Sharepoint Server. The Target string array must contain the following elements: 0 - URL of SharePoint Server 1 - ListName 2 - Description of list
Refresh			Refreshes the current data from a Windows SharePoint Server
Resize		Range	Allows a ListObject to be resized of a certain range
Unlink			Removes the current link to a Windows SharePoint Server
Unlist			Removes the list functionality from a ListObject, thus turning all data into a regular range of data

List Object Example

```

Sub DeletSpecificListColumn()
Dim olist As ListObject
Dim i As Integer

' Set olist to the first list object on the active sheet
Set olist = ActiveSheet.ListObjects(1)

' Find the column named "Field-3" and delete it
For i = 1 To olist.ListColumns.Count - 1
    If olist.ListColumns(i).Name = "Field-3" Then
        olist.ListColumns(i).Delete
    End If
Next
End Sub

```

ListRow Object and the ListRows Collection

The `ListRow` object represents a single row in a given list object. The `ListRows` collection represents all of the rows in a list object.

ListRows Common Properties

ListRows Common Properties

The `Application`, `Count`, `Creator`, `Item`, and `Parent` properties are defined at the beginning of this appendix. The only method for the `ListRows` object is the `Add` method, which adds new rows to a given list object.

ListRow Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ListRow Properties

Name	Returns	Description
<code>Index</code>	<code>Long</code>	Read-only. Returns an index number for an item within a collection
<code>Range</code>	<code>Range</code>	Read-only. Returns a range to which the current object applies

ListRow Methods

Name	Returns	Parameters	Description
<code>Delete</code>			Deletes an item from the collection

Mailer Object

The `Mailer` object is used on the Macintosh to mail Excel files using the PowerTalk Mailer.

Mailer Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Mailer Properties

Name	Returns	Description
<code>BCCRecipients</code>	<code>Variant</code>	Set/Get the list of blind copies
<code>CCRecipients</code>	<code>Variant</code>	Set/Get the list of carbon copies
<code>Enclosures</code>	<code>Variant</code>	Set/Get the list of enclosures
<code>Received</code>	<code>Boolean</code>	Read-only. Returns whether the mail message was received
<code>SendDateTime</code>	<code>Date</code>	Read-only. Returns the date and time the message was sent
<code>Sender</code>	<code>String</code>	Read-only. Returns the name of the mail message sender
<code>Subject</code>	<code>String</code>	Set/Get the subject line of the mail message
<code>ToRecipients</code>	<code>Variant</code>	Set/Get the array of recipient names
<code>WhichAddress</code>	<code>Variant</code>	Set/Get the address that the mail message originates from

MultiThreadedCalculation Object

The `MultiThreadedCalculation` object exposes the properties that allow for programmatic control of the multithreaded calculations (calculations that can be performed across multiple threads).

MultiThreadedCalculation Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

MultiThreadedCalculation Properties

Name	Returns	Description
<code>Enabled</code>	Boolean	Set/Get whether <code>MultiThreadedCalculation</code> objects are enabled at run time
<code>ThreadCount</code>	Long	Returns the total count of the process threads that are a part of the specified <code>MultiThreadedCalculation</code> object
<code>ThreadMode</code>	<code>XlThreadMode</code>	Set/Get the thread mode for the specified <code>MultiThreadedCalculation</code> object using an <code>XlThreadMode</code> constant

Name Object and the Names Collection

The `Names` collection holds the list of named ranges in a workbook. Each `Name` object describes a range of cells in a workbook that can be accessed by the name. Some `Name` objects are built-in (for example, `Print_Area`) and others are user-defined. The parent of the `Names` collection can be the `Workbook`, `Application`, or `Worksheet` object. The `Name` object can also be accessed through the `Range` object.

The `Names` collection has an `Add` method besides the typical collection attributes. The `Add` method adds a `Name` object to the collection. The parameters of the `Add` method correspond to the properties of the `Name` object.

Name Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name Properties

Name	Returns	Description
<code>Category</code>	String	Set/Get the category of the <code>Name</code> in the language used to create the macro. Valid only if the <code>Name</code> is a custom function or command
<code>CategoryLocal</code>	String	Set/Get the category of the <code>Name</code> in the language of the end user. Valid only if the <code>Name</code> is a custom function or command
<code>Comment</code>	String	Set/Get any comments associated with a name

Table continued on following page

Name Methods

Name	Returns	Description
Index	Long	Read-only. Returns the spot where Name is located in the Names collection
IsWorkbookParameter	Boolean	Set/Get the specified Name object as a workbook parameter
MacroType	XlXLMacroType	Set/Get if the Name refers to a command, a function, or just a range
Name	String	Set/Get the name of the Name object in the language of the macro
NameLocal	String	Set/Get the name of the Name object in the language of the end user
RefersTo	Variant	Set/Get the range text that the Name refers to in the language of the macro and in A1 notation style
RefersToLocal	Variant	Set/Get the range text that the Name refers to in the language of the user and in A1 notation style
RefersToR1C1	Variant	Set/Get the range text that the Name refers to in the language of the macro and in R1C1 notation style
RefersToR1C1Local	Variant	Set/Get the range text that the Name refers to in the language of the user and in R1C1 notation style
RefersToRange	Range	Read-only. Returns the range that the Name refers to
ShortcutKey	String	Set/Get the shortcut key to trigger a Microsoft Excel 4.0 macro associated with a Name
ValidWorkbookParameter	Boolean	Set/Get the specified Name object if a valid workbook parameter
Value	String	Set/Get the range text that the Name refers to in the language of the macro and in A1 notation style
Visible	Boolean	Set/Get whether the name of the Name object appears in the Names dialog box in Excel

Name Methods

Name	Returns	Parameters	Description
Delete			Deletes the Name object from the collection

Name Object and the Names Collection Example

```
Sub DeleteInvalidNames()  
    Dim oName As Name  
    'Loop through all the names in the active workbook  
    For Each oName In ActiveWorkbook.Names  
        'Is it an invalid name?  
        If InStr(1, oName.RefersTo, "#REF") > 0 Then  
  
            'Yes, so log it  
            Debug.Print "Deleted name " & oName.Name & " - " & oName.RefersToLocal  
  
            'and delete it from the collection  
            oName.Delete  
        End If  
    Next  
End Sub
```

ODBCConnection Object

When an ODBC connection is stored in an Excel workbook, Excel opens an in-memory ODBC connection each time the workbook opens. This in-memory connection in the `ODBCConnection` object holds the connection string that allows an Excel workbook to connect to an external data source. The `ODBCConnection` object typically contains the name of the server to connect to, the name of the objects to be opened, and authentication parameters.

ODBCConnection Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ODBCConnection Properties

Name	Returns	Description
<code>AlwaysUseConnectionFile</code>	Boolean	Set/Get whether an external connection file will be used to establish a connection to the data source. When this property is set to <code>True</code> , any embedded connection information will be ignored, and the external connection file will be used
<code>BackgroundQuery</code>	Boolean	Set/Get if the processing of queries is done asynchronously
<code>CommandText</code>	VARIANT	Set/Get a command string that passes commands to the data source. This property essentially replaces the <code>SQL</code> property, which still exists only for compatibility purposes

Table continued on following page

ODBCConnection Properties

Name	Returns	Description
CommandType	xlCmdType	Set/Get one of the xlCmdType constants that define the type of command that is passed. The default is the xlCmdSQL constant
Connection	String	Set/Get the connection string for to an ODBC data source
EnableRefresh	Boolean	Set/Get whether an external connection can be refreshed via the user interface. Default is True
RefreshDate	Date	Read-only. Returns date of last refresh
Refreshing	Boolean	Set/Get whether a background ODBC query is currently being run
RefreshOnFileOpen	Boolean	Set/Get whether a connection is automatically refreshed when the workbook is opened. Default is False
RefreshPeriod	Long	Set/Get the number of minutes between refreshes. This property can be set to any value between 0 and 32767. Setting this property to 0 will effectively disable automatic periodic refreshing
RobustConnect	xlRobustConnect	Set/Get the method by which the ODBCConnection object connects to your data source. The RobustConnect property allows an IT Department to maintain and update connections in a central place, permitting users to automatically fetch those updates in the event that the connection cached within their workbook fails
SavePassword	Boolean	Set/Get whether password information is saved in the connection string
ServerCredentialsMethod	xlCredentialsMethod	Set/Get the type of credentials to be used for server authentication. Your choices through the xlCredentialsMethod are integrated, none, prompted, and stored
ServerSSOApplicationID	String	Set/Get a single sign-on application (SSO) identifier that is used to do a lookup in the SSO database for credentials
SourceConnectionFile	String	Set/Get the name of the file that was used to create the connection
SourceData	Variant	Set/Get the data source for the connection
SourceDataFile	String	Set/Get the name of the source data file for the connection

ODBCConnection Methods

Name	Returns	Parameters	Description
Cancel Refresh			Cancels any refresh or query operations specified by the ODBCConnection object
Refresh			Refreshes the ODBC connection
SaveAsODC		ODCFileName as string, [Description] As Variant, [Keywords] As Variant	Saves the ODBC connection as an Office Data Connection file (.odc)

ODBCError Object and the ODBCErrors Collection

The ODBCErrors collection contains a list of errors that occurred on the most recent query using an ODBC connection. Each ODBCError object contains information describing an error that occurred on the most recent query using an ODBC connection. If the most recent query against an ODBC source did not generate any errors, then the collection is empty. The ODBCErrors collection has no properties and methods outside the typical collection attributes listed at the beginning of this appendix.

ODBCError Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

ODBCError Properties

Name	Returns	Description
ErrorString	String	Read-only. Returns the error string generated from the ODBC connection
SqlState	String	Read-only. Returns the SQL state error generated from the ODBC connection

ODBCError Object and the ODBCErrors Collection Example

```
Sub CheckODBCErrors()
    Dim oErr As ODBCError
    Dim sMsg As String
    'Continue after errors
    On Error Resume Next
    'Don't show logon prompts etc
    Application.DisplayAlerts = False
    'Update an ODBC query table
    ActiveSheet.QueryTables(1).Refresh
    'Any errors?
    If Application.ODBCErrors.Count = 0 Then
        'No, so all OK
        MsgBox "Updated OK"
```

OLEDBConnection Object

```
Else
    'Yes, so list them all
    sMsg = "The following error(s) occurred during the update"
    For Each oErr In Application.OBCErrors
        sMsg = sMsg & vbCrLf & oErr.ErrorString & " (" & oErr.SqlState & ")"
    Next
    MsgBox sMsg
End If
End Sub
```

OLEDBConnection Object

When an OLEDB connection is stored in an Excel workbook, Excel opens an in-memory OLEDB connection each time the workbook opens. This in-memory connection in the `OLEDBConnection` object holds the connection string that allows an Excel workbook to connect to an external data source. The `OLEDBConnection` object typically contains the name of the server to connect to, the names of the objects to be opened, and authentication parameters.

OLEDBConnection Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

OLEDBConnection Properties

Name	Returns	Description
<code>ADOConnection</code>	Object	Read-only. Returns an ADO connection object
<code>AlwaysUseConnectionFile</code>	Boolean	Set/Get whether an external connection file will be used to establish a connection to the data source. When this property is set to <code>True</code> , any embedded connection information will be ignored, and the external connection file will be used
<code>BackgroundQuery</code>	Boolean	Set/Get the processing of queries asynchronously. <code>False</code> for OLAP data sources
<code>CommandText</code>	VARIANT	Set/Get a command string that passes commands to the data source. This property essentially replaces the <code>SQL</code> property, which still exists only for compatibility purposes
<code>CommandType</code>	<code>xlCmdType</code>	Set/Get one of the <code>xlCmdType</code> constants that define the type of command that is passed. The default is the <code>xlCmdSQL</code> constant
<code>Connection</code>	String	Set/Get the connection string for to an OLEDB data source
<code>EnableRefresh</code>	Boolean	Set/Get whether an external connection can be refreshed via the user interface. Default is <code>True</code>
<code>IsConnected</code>	Boolean	Read-only. Checks whether an OLEDB connection is connected to its data source
<code>LocalConnection</code>	String	Set/Get the connection string for an offline cube file. For a non-OLAP data source, the value of the <code>LocalConnection</code> property is an empty string

Name	Returns	Description
Maintain Connection	Boolean	Set/Get whether the connection to the data source is maintained after the refresh operation and until the workbook is closed. Setting this property to <code>True</code> will keep the connection open
MaxDrillthrough Records	Long	Used with OLAP data sources, this property sets or gets the maximum number of records that can be retrieved from the data source at any given time
OLAP	Boolean	Read-only. Returns <code>True</code> if the target data source is an OLAP server
RefreshDate	Date	Read-only. Returns date of last refresh
Refreshing	Boolean	Set/Get whether a background OLEDB query is currently being run
RefreshOn FileOpen	Boolean	Set/Get whether a connection is automatically refreshed when the workbook is opened. Default is <code>False</code>
RefreshPeriod	Long	Set/Get the number of minutes between refreshes. This property can be set to any value between 0 and 32767. Setting this property to 0 will effectively disable automatic periodic refreshing
RetrieveIn OfficeUILang	Boolean	Set/Get whether the errors are to be retrieved in the Office user interface display language
RobustConnect	<code>xlRobustConnect</code>	Set/Get the method by which the <code>OLEDBConnection</code> object connects to your data source. The <code>RobustConnect</code> property allows an IT Department to maintain and update connections in a central place, permitting users to automatically fetch those updates in the event that the connection cached within their workbook fails
SavePassword	Boolean	Set/Get whether password information is saved in the connection string
Server Credentials Method	<code>xlCredentialsMethod</code>	Set/Get the type of credentials to be used for server authentication. Your choices through the <code>xlCredentialsMethod</code> are integrated, none, prompted, and stored
ServerFillColor	Boolean	Set/Get whether to retrieve the OLAP fill color formatting when connected to an OLAP server
ServerFontStyle	Boolean	Set/Get whether to retrieve the OLAP font style formatting when connected to an OLAP server
ServerNumber Format	Boolean	Set/Get whether to retrieve the OLAP number formatting when connected to an OLAP server

Table continued on following page

OLEDBConnection Methods

Name	Returns	Description
ServerSSO ApplicationID	String	Set/Get a single sign-on application (SSO) identifier that is used to do a lookup in the SSO database for credentials
ServerTextColor	Boolean	Set/Get whether to retrieve the OLAP text color formatting when connected to an OLAP server
Source Connection File	String	Set/Get the name of the file that was used to create the connection
SourceData File	String	Read-only. Returns the name of the source data file for the connection
UseLocal Connection	Boolean	Set/Get if the LocalConnection property is used to set the data source. False means the Connection property is used. Allows you to store some data sources offline

OLEDBConnection Methods

Name	Returns	Parameters	Description
Cancel Refresh			Cancels any refresh or query operations specified by the ODBCConnection object
Make Connection			Establishes a connection when a connection drops or when the user wants to reestablish the connection. This method will result in a run-time error if the MaintainConnection property has been set to False
Refresh			Refreshes the ODBC connection
SaveAsODC		ODCFileName As String, [Description]As Variant, [Keywords]As Variant	Saves the ODBC connection as an Office Data Connection file (.odc)

OLEDBError Object and the OLEDBErrors Collection

The `OLEDBErrors` collection contains a list of errors that occurred on the most recent query using an OLE DB provider. Each `OLEDBError` object contains information describing an error that occurred on the most recent query using an OLE DB provider. If the most recent query against an OLE DB provider did not generate any errors, then the collection is empty. The `OLEDBErrors` collection has no properties and methods outside the typical collection attributes listed at the beginning of this appendix.

OLEDBError Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

OLEDBError Properties

Name	Returns	Description
ErrorString	String	Read-only. Returns the error string generated from the OLE DB provider
Native	Long	Read-only. Returns a provider-specific error number describing the error
Number	Long	Read-only. Returns the error number describing the error
SqlState	String	Read-only. Returns the SQL state error generated from the OLE DB provider
Stage	Long	Read-only. Returns the stage of an error generated from the OLE DB provider

OLEDBError Object and the OLEDBErrors Collection Example

```

Sub CheckOLEDBErrors()
    Dim oErr As OLEDBError
    Dim sMsg As String
    'Continue after errors
    On Error Resume Next
    'Don't show logon prompts etc
    Application.DisplayAlerts = False
    'Update an OLE DB pivot table
    ActiveSheet.PivotTables(1).Refresh
    'Any errors?
    If Application.OLEDBErrors.Count = 0 Then
        'No, so all OK
        MsgBox "Updated OK"
    Else
        'Yes, so list them all
        sMsg = "The following error(s) occurred during the update"
        For Each oErr In Application.OLEDBErrors
            sMsg = sMsg & vbCrLf & oErr.ErrorString & " (" & oErr.SqlState & ")"
        Next
        MsgBox sMsg
    End If
End Sub

```

OLEFormat Object

The `OLEFormat` object, returned by the `OLEFormat` property of the `Shape` object, exposes the object properties for a linked or embedded object.

OLEFormat Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

OLEFormat Properties

OLEFormat Properties

Name	Returns	Description
Object	OLEObject	Read-only. Returns the OLE Automation object associated with a given OLE object
progID	String	Read-only. Returns the programmatic identifiers for the given OLE object

OLEFormat Methods

Name	Returns	Parameters	Description
Activate	Variant		Sets the focus and activates the given OLE object
Verb	Variant	Verb As X1OLEVerb	Performs an action on the parent OLE object that triggers a reaction in the OLE object (for example, x1Open)

OLEObject Object and the OLEObjects Collection

The `OLEObjects` collection holds all the ActiveX controls, linked OLE objects, and embedded OLE objects on a worksheet or chart. An OLE object represents an ActiveX control, a linked OLE object, or an embedded OLE object on a worksheet or chart.

The `OLEObjects` collection has many properties and methods besides the typical collection attributes. These are listed in the following table.

OLEObjects Collection Properties and Methods

Name	Returns	Description
AutoLoad	Boolean	Set/Get whether the OLE object is automatically loaded when the workbook is opened. Not valid for ActiveX controls. Usually set to <code>False</code> . This property only works if there is one <code>OLEObject</code> in the collection
Border	Border	Read-only. Returns the border's properties around the OLE object. This property only works if there is one <code>OLEObject</code> in the collection
Enabled	Boolean	Set/Get whether the <code>OLEObject</code> is enabled. This property only works if there is one <code>OLEObject</code> in the collection
Height	Double	Set/Get the height of <code>OLEObject</code> frame. This property only works if there is one <code>OLEObject</code> in the collection

OLEObjects Collection Properties and Methods

Name	Returns	Description
Interior	Interior	Read-only. Returns an object containing options to format the inside area of the OLE object (for example, interior color). This property only works if there is one OLEObject in the collection
Left	Double	Set/Get the distance from the left edge of the OLEObject frame to the left edge of the sheet. This property only works if there is one OLEObject in the collection
Locked	Boolean	Set/Get whether editing will be possible when the parent sheet is protected. This property only works if there is one OLEObject in the collection
Placement	XlPlacement	Set/Get how the OLEObject object is anchored to the sheet (for example, free floating, move with cells). Use the XlPlacement constants to set this property. This property only works if there is one OLEObject in the collection
PrintObject	Boolean	Set/Get whether the OLEObject on the sheet will be printed when the sheet is printed. This property only works if there is one OLEObject in the collection
Shadow	Boolean	Set/Get whether a shadow appears around the OLE object. This property only works if there is one OLEObject in the collection
ShapeRange	ShapeRange	Read-only. Returns the OLE object as a Shape object. This property only works if there is one OLEObject in the collection
SourceName	String	Set/Get the link source name of the OLE object. This property only works if there is one OLEObject in the collection
Top	Double	Set/Get the distance from top edge of the OLE object to the top of the parent sheet. This property only works if there is one OLEObject in the collection
Visible	Boolean	Set/Get whether all the OLEObjects in the collection are visible
Width	Double	Set/Get the width of the OLE object frame. This property only works if there is one OLEObject in the collection
ZOrder	Long	Read-only. Returns the position of the OLE object among all the other objects on the sheet. This property only works if there is one OLEObject in the collection
Add	OLEObject	Method. Parameters: [ClassType], [Filename], [Link], [DisplayAsIcon], [IconFileName], [IconIndex], [IconLabel], [Left], [Top], [Width], [Height]. Adds an OLE object to the collection of OLEObjects. The position of the new OLE object can be specified by using the Left, Top, Width, and Height parameters. The type of OLEObject (ClassType) or its location (Filename) can be specified as well. The other parameters have equivalent OLEObject properties

Table continued on following page

OLEObject Common Properties

Name	Returns	Description
BringToFront	Variant	Method. Brings all the OLE objects in the collection to the front of all the other objects
Copy	Variant	Method. Copies all the OLE objects in the collection into the clipboard
CopyPicture	Variant	Method. Parameters: Appearance As XlPictureAppearance, Format As XlCopyPictureFormat. Copies the OLE objects in the collection into the clipboard as a picture. The Appearance parameter can be used to specify whether the picture is copied as it looks on the screen or when printed. The Format parameter can specify the type of picture that will be put into the clipboard
Cut	Variant	Method. Cuts all the OLE objects in the collection into the clipboard
Delete	Variant	Method. Deletes all the OLEObject objects in the collection from the clipboard
Duplicate	Object	Method. Duplicates all the OLEObject objects in the collection into the parent sheet
Select	Variant	Method. Parameters: [Replace]. Selects all the OLEObject objects in the collection
SendToBack	Variant	Method. Brings the OLEObject objects in the collection to the back of other objects

OLEObject Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

OLEObject Properties

Name	Returns	Description
AutoLoad	Boolean	Set/Get whether the OLE object is automatically loaded when the workbook is opened. Not valid for ActiveX controls. Usually set to False
AutoUpdate	Boolean	Set/Get whether the OLE object is automatically updated when the source changes. Valid only for linked objects (OLE-Type=xlOLELink)
Border	Border	Read-only. Returns the border's properties around the OLE object
BottomRightCell	Range	Read-only. Returns the single cell range located under the lower-right corner of the OLE object
Enabled	Boolean	Set/Get whether the OLEObject is enabled

Name	Returns	Description
Height	Double	Set/Get the height of OLEObject frame
Index	Long	Read-only. Returns the spot in the collection where the current OLEObject is located
Interior	Interior	Read-only. Returns an object containing options to format the inside area of the OLE object (for example, interior color)
Left	Double	Set/Get the distance from the left edge of the OLEObject frame to the left edge of the sheet
LinkedCell	String	Set/Get the range that receives the value from the results of the OLE object
ListFill Range	String	Set/Get the range that holds the values used by an ActiveX list box
Locked	Boolean	Set/Get whether editing will be possible when the parent sheet is protected
Name	String	Set/Get the name of the OLE object
Object	Object	Read-only. Returns access to some of the properties and methods of the underlying object in the OLE object
OLEType	Variant	Read-only. Returns the type OLE object: xLOLELink or xLOLEEmbed. Use the XLOLEType constants
Placement	XlPlacement	Set/Get how the OLEObject object is anchored to the sheet (for example, free floating, move with cells). Use the XlPlacement constants to set this property
PrintObject	Boolean	Set/Get whether the OLEObject on the sheet will be printed when the sheet is printed
ProgId	String	Read-only. Returns the programmatic identifier associated with the OLE object (for example, "Excel.Application")
Shadow	Boolean	Set/Get whether a shadow appears around the OLE object
ShapeRange	ShapeRange	Read-only. Returns the OLE object as a Shape object
SourceName	String	Set/Get the link source name of the OLE object
Top	Double	Set/Get the distance from top edge of the OLE object to the top of the parent sheet
TopLeftCell	Range	Read-only. Returns the single cell range located above the top-left corner of the OLE object
Visible	Boolean	Set/Get whether the OLEObject is visible
Width	Double	Set/Get the width of the OLE object frame
ZOrder	Long	Read-only. The position of the OLE object among all the other objects on the sheet

OLEObject Methods

Name	Returns	Parameters	Description
Activate	Variant		Sets the focus and activates the OLE object
BringToFront	Variant		Brings the OLE object to the front of all the other objects
Copy	Variant		Copies the OLE object into the clipboard
CopyPicture	Variant	Appearance As Xl Picture Appearance, Format As XlCopy Picture Format	Copies the OLE object into the clipboard as a picture. The Appearance parameter can be used to specify whether the picture is copied as it looks on the screen or when printed. The Format parameter can specify the type of picture that will be put into the clipboard
Cut	Variant		Cuts the OLE object into the clipboard
Delete	Variant		Deletes the OLEObject object from the clipboard
Duplicate	Object		Duplicates the OLEObject object into the parent sheet
Select	Variant	[Replace]	Selects the OLEObject object
SendToBack	Variant		Brings the OLEObject object to the back of other objects
Update	Variant		Updates the OLE object link, if applicable
Verb	Variant	Verb As XlOLEVerb	Performs an action on the parent OLE object that triggers a reaction in the OLE object (for example, xlOpen)

OLEObject Events

Name	Parameters	Description
GotFocus		Triggered when the OLE object gets focus
LostFocus		Triggered when the OLE object loses focus

Outline Object

The Outline object represents the outline feature in Excel. The parent of the Outline object is the Worksheet object.

Outline Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Outline Properties

Name	Returns	Description
<code>AutomaticStyles</code>	Boolean	Set/Get whether the outline has styles automatically assigned by Excel
<code>SummaryColumn</code>	<code>XlSummaryColumn</code>	Set/Get whether the summary columns are to the left (<code>xlLeft</code>) or the right (<code>xlRight</code>) of the detail columns
<code>SummaryRow</code>	<code>XlSummaryRow</code>	Set/Get whether the summary rows are above (<code>xlAbove</code>) or below (<code>xlBelow</code>) the detail rows

Outline Methods

Name	Returns	Parameters	Description
<code>ShowLevels</code>	Variant	[<code>RowLevels</code>], [<code>ColumnLevels</code>]	Show the details of rows and columns at a higher level as specified by the <code>RowLevels</code> and <code>ColumnLevels</code> parameters, respectively. The rest of the details for the other levels are hidden

Outline Object Example

```
Sub ShowOutlines()
    Dim oOutl As Outline
    'Group some rows
    ActiveSheet.Range("4:5").Group
    'Get the Outline object
    Set oOutl = ActiveSheet.Outline
    'Format the outline display
    With oOutl
        .ShowLevels 1
        .SummaryRow = xlSummaryAbove
    End With
End Sub
```

Page Object and the Pages Collection

The `Page` object represents a single page in a given workbook. The `Pages` collection represents a collection of `Page` objects in a workbook. The `Pages` collection exposes only two properties, the standard `Count` property and the `Item` property, which returns a single `Page` object.

Page Properties

Name	Returns	Description
CenterFooter	HeaderFooter	Read-only. Specifies that a picture or text should be center-aligned in the page footer
CenterHeader	HeaderFooter	Read-only. Specifies that a picture or text should be center-aligned in the page header
LeftFooter	HeaderFooter	Read-only. Specifies that a picture or text should be left-aligned in the page footer
LeftHeader	HeaderFooter	Read-only. Specifies that a picture or text should be left-aligned in the page header
RightFooter	HeaderFooter	Read-only. Specifies that a picture or text should be right-aligned in the page footer
RightHeader	HeaderFooter	Read-only. Specifies that a picture or text should be right-aligned in the page header

PageSetup Object

The `PageSetup` object contains the functionality of the Page Setup dialog box. Possible parents of the `PageSetup` object are the `Chart` and `Worksheet` objects.

PageSetup Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

PageSetup Properties

Name	Returns	Description
AlignMargins HeaderFooter	Boolean	Set/Get whether the header and footer for a given document are aligned with margins set in the page setup options
BlackAndWhite	Boolean	Set/Get whether worksheet items will be printed in black and white only. Not valid when parents are <code>Chart</code> objects
BottomMargin	Double	Set/Get the bottom margin of the page in points
CenterFooter	String	Set/Get the text for the center part of the footer
CenterFooter Picture	Graphic	Read-only. Returns the picture for the center section of the footer
CenterHeader	String	Set/Get the text for the center part of the header
CenterHeader Picture	Graphic	Read-only. Returns the picture for the center section of the footer

Name	Returns	Description
Center Horizontally	Boolean	Set/Get whether the worksheet or chart will be horizontally centered on the page
Center Vertically	Boolean	Set/Get whether the worksheet or chart will be vertically centered on the page
Different FirstPage HeaderFooter	Boolean	Set/Get whether different header or footer is used on the first page
Draft	Boolean	Set/Get whether graphics will be printed. True means graphics will not be printed
EvenPage	Page	Set/Get the alignment of text on the even page of a workbook or section
FirstPage	Page	Set/Get the alignment of text on the first page of a workbook or section
FirstPage Number	Long	Set/Get which number will be used as the first page number. Use <code>x1Automatic</code> to have Excel choose this (default)
FitToPages Tall	Variant	Set/Get how many pages tall the sheet will be scaled to. Setting this property to <code>False</code> will mean the <code>FitToPagesWide</code> property will be used
FitToPagesWide	Variant	Set/Get how many pages wide the sheet will be scaled to. Setting this property to <code>False</code> will mean the <code>FitToPagesTall</code> property will be used
FooterMargin	Double	Set/Get the distance from the page's bottom to the footer of the page in points
HeaderMargin	Double	Set/Get the distance from the page's top to the header of the page in points
LeftFooter	String	Set/Get the text for the left part of the footer
LeftFooter Picture	Graphic	Read-only. Returns the picture for the left section of the footer
LeftHeader	String	Set/Get the text for the center part of the header
LeftHeader Picture	Graphic	Read-only. Returns the picture for the left section of the header
LeftMargin	Double	Set/Get the left margin of the page in points
OddAndEvenPages HeaderFooter	Boolean	Set/Get whether the specified <code>PageSetup</code> object has different headers and footers for odd-numbered and even-numbered pages

Table continued on following page

PageSetup Properties

Name	Returns	Description
Order	XlOrder	Set/Get the manner in which Excel numbers pages for large worksheets (for example, xlDownTheOver, xlOverThenDown). Not valid for parents that are Chart objects
Orientation	XlPage Orientation	Set/Get the page orientation: xlLandscape or xlPortrait
Pages	Pages	Set/Get the count or item number of the pages in Pages collection
PaperSize	XlPaper Size	Set/Get the paper size (for example, xlPaperLetter, xlPaperLegal, and so on)
PrintArea	String	Set/Get the range on a worksheet that will be printed. If this property is set to False, then the entire sheet is printed. Not valid for parents that are Chart objects
PrintComments	XlPrint Location	Set/Get how comments are printed, or if they are at all (for example, xlPrintInPlace, xlPrintNo-Comments)
PrintErrors	XlPrint Errors	Set/Get the type of print error displayed. This allows the suppression of error values when printing a worksheet
Print Gridlines	Boolean	Set/Get whether cell gridlines are printed for a worksheet. Not valid for parents that are Chart objects
PrintHeadings	Boolean	Set/Get whether row and column headings are printed
PrintNotes	Boolean	Set/Get whether notes attached to the cells are printed at the end as endnotes. Not valid if parents are Chart objects
PrintTitle Columns	String	Set/Get which columns to repeat on the left side of every printed page
PrintTitleRows	String	Set/Get which rows to repeat on the top of every page
RightFooter	String	Set/Get the text for the right part of the footer
RightFooter Picture	Graphic	Read-only. Returns the picture for the right section of the footer
RightHeader	String	Set/Get the text for the right part of the header
RightHeader Picture	Graphic	Read-only. Returns the picture for the right section of the header

Name	Returns	Description
RightMargin	Double	Set/Get the right margin of the page in points
ScaleWithDoc HeaderFooter	Boolean	Set/Get whether the header and footer should be scaled with the document when the size of the document changes
TopMargin	Double	Set/Get the top margin of the page in points
Zoom	Variant	Set/Get the percentage scaling that will occur for the worksheet. Not valid for parents that are Chart objects (10 to 400 percent)

PageSetup Methods

Name	Returns	Parameters	Description
PrintQuality	Variant	[Index]	Set/Get the print quality. The Index parameter can be used to specify horizontal (1) or vertical (2) print quality

PageSetup Object Example

```

Sub SetUpPage()
    Dim oPS As PageSetup
    'Get the sheet's PageSetup object
    Set oPS = ActiveSheet.PageSetup
    'Set up the page
    With oPS
        'Set the paper size to the local default
        .PaperSize = fnLocalPaperSize
        .Orientation = xlPortrait
        'etc.
    End With
End Sub

Function fnLocalPaperSize() As XlPaperSize
    'Remember the paper size when we've read it
    Static iPaperSize As XlPaperSize
    'Is it set?
    If iPaperSize = 0 Then
        'No, so create a new workbook and read off the paper size
        With Workbooks.Add
            iPaperSize = .Worksheets(1).PageSetup.PaperSize
            .Close False
        End With
    End If
    'Return the paper size
    fnLocalPaperSize = iPaperSize
End Function

```

Pane Object and the Panes Collection

The `Panes` collection allows manipulation of the different panes of a window. A `Pane` object is equivalent to the single pane of a window. The parent object of the `Panes` collection is the `Window` object. The `Panes` collection has no properties or methods outside the typical collection attributes listed at the beginning of this appendix.

Pane Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Pane Properties

Name	Returns	Description
<code>Index</code>	Long	Read-only. Returns the spot in the collection where the <code>Pane</code> object is located
<code>ScrollColumn</code>	Long	Set/Get which column number is the leftmost column in the pane window
<code>ScrollRow</code>	Long	Set/Get which row number is the top row in the pane window
<code>VisibleRange</code>	Range	Read-only. Returns the cell range that is visible in the pane

Pane Methods

Name	Returns	Parameters	Description
<code>Activate</code>	Boolean		Activates the pane
<code>LargeScroll</code>	Variant	[Down], [Up], [ToRight], [ToLeft]	Causes the document to scroll in a certain direction a screenful at a time, as specified by the parameters
<code>PointsToScreen PixelsX</code>	Long	Points	Set/Get a pixel point on the screen
<code>PointsToScreen PixelsY</code>	Long	Points	Set/Get the location of the pixel on the screen
<code>ScrollInto View</code>		Left As Long, Top As Long, Width As Long, Height As Long, [Start]	Scrolls the spot specified by the <code>Left</code> , <code>Top</code> , <code>Width</code> , and <code>Height</code> parameters to either the upper-left corner of the pane (<code>Start = True</code>) or the lower-right corner of the pane (<code>Start = False</code>). The <code>Left</code> , <code>Top</code> , <code>Width</code> , and <code>Height</code> parameters are specified in points
<code>SmallScroll</code>	Variant	[Down], [Up], [ToRight], [ToLeft]	Causes the document to scroll in a certain direction a document line at a time, as specified by the parameters

Pane Object and the Panes Collection Example

```
Sub ScrollActivePane()
    Dim oPane As Pane
    Dim oRNg As Range
    'The range to show in the pane
    Set oRNg = Range("G3:J10")
    'Get the active pane
    Set oPane = Application.ActiveWindow.ActivePane
    'Scroll the pane to show the range in the top-left corner
    oPane.ScrollColumn = oRNg.Column
    oPane.ScrollRow = oRNg.Row
End Sub
```

Parameter Object and the Parameters Collection

The `Parameters` collection holds the list of parameters associated with a query table. If no parameters exist, then the collection has no `Parameter` objects inside of it. Each `Parameter` object represents a single parameter for a query table. The parent of the `Parameters` collection is the `QueryTable` object.

The `Parameters` collection has two extra properties and methods besides the typical collection attributes. They are listed in the following table.

Parameters Collection Properties and Methods

Name	Returns	Description
Add	Parameter	Method. Parameters: Name As String, [iDataType]. Adds a parameter to the collection creating a new query parameter for the parent query table. The type of parameter can be specified by iDataType. Use the <code>XlParamaterDataType</code> constants for iDataType
Delete		Method. Deletes the parameters in the collection

Parameter Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Parameter Properties

Name	Returns	Description
DataType	XlParameter DataType	Set/Get the data type of the parameter
Name	String	Set/Get the name of the parameter
Prompt String	String	Read-only. Returns the prompt that is displayed to the user when prompted for a parameter value

Table continued on following page

Parameter Methods

Name	Returns	Description
RefreshOn Change	Boolean	Set/Get whether the query table results are refreshed when the parameter value changes
Source Range	Range	Read-only. Returns the range of text that contains the parameter value
Type	XlParameter Type	Read-only. Returns the type of parameter (for example, xlConstant, xlPrompt, or xlRange). xlConstant means that the Value parameter has the value of the parameter. xlPrompt means that the user is prompted for the value. xlRange means that the value defines the cell range that contains the value
Value	Variant	Read-only. Returns the parameter value

Parameter Methods

Name	Returns	Parameters	Description
SetParam		Type As XlParameter Type, Value	Set/Get the type of the parameter and the value of the parameter

Parameter Object and the Parameters Collection Example

```
Sub UpdateQuery()  
    Dim oParam As Parameter  
    'Using the Query Table(  
    With ActiveSheet.ListObjects(1).QueryTable           'Get the first parameter  
        Set oParam = .Parameters(1)  
  
        'Set its value  
        oParam.SetParam xlConstant, "Company"  
  
        'Refresh the query  
        .Refresh  
    End With  
End Sub
```

Phonetic Object and the Phonetics Collection

The `Phonetics` collection holds all of the phonetic text in a range. The `Phonetic` object represents a single phonetic text string. The parent of the `Phonetics` object is the `Range` object.

The `Phonetics` collection has a few properties and methods besides the typical collection attributes. They are listed in the following table.

Phonetics Collection Properties and Methods

Name	Returns	Description
Alignment	Long	Set/Get the alignment for the phonetic text. Use the <code>XlPhoneticAlignment</code> constants
Character Type	Long	Set/Get the type of phonetic text to use. Use the <code>XlPhoneticCharacterType</code> constants
Font	Font	Read-only. Returns an object containing <code>Font</code> options for the text in the <code>Phonetics</code> collection
Length	Long	Read-only. Returns the number of phonetic text characters starting from the <code>Start</code> parameter
Start	Long	Read-only. Returns the position that represents the first character of the phonetic text strings. Valid only if there is only one <code>Phonetic</code> object in the collection
Text	String	Set/Get the phonetic text. Valid only if there is only one <code>Phonetic</code> object in the collection
Visible	Boolean	Set/Get whether the phonetic text is visible to the end user. Valid only if there is only one <code>Phonetic</code> object in the collection
Add		Method. Parameters: <code>Start</code> As Long, <code>Length</code> As Long, <code>Text</code> As String. Adds a <code>Phonetic</code> object to the collection at the cell specified by the parent <code>Range</code> object
Delete		Method. Deletes all the <code>Phonetic</code> objects in the collection

Phonetic Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Phonetic Properties

Name	Returns	Description
Alignment	Long	Set/Get the alignment for the phonetic text. Use the <code>XlPhoneticAlignment</code> constants
CharacterType	Long	Set/Get the type of phonetic text to use. Use the <code>XlPhoneticCharacterType</code> constants
Font	Font	Read-only. Returns an object containing <code>Font</code> options for the phonetic text
Text	String	Set/Get the phonetic text
Visible	Boolean	Set/Get whether the phonetic text is visible to the end user

PictureFormat Object

The `PictureFormat` object allows manipulation of the picture properties of the parent `Shape` object.

PictureFormat Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

PictureFormat Properties

Name	Returns	Description
Brightness	Single	Set/Get the brightness of the parent shape (0 to 1, where 1 is the brightest)
ColorType	MsoPicture ColorType	Set/Get the type of color setting of the parent shape
Contrast	Single	Set/Get the contrast of the parent shape (0 to 1, where 1 is the greatest contrast)
CropBottom	Single	Set/Get how much is cropped off the bottom
CropLeft	Single	Set/Get how much is cropped off the left
CropRight	Single	Set/Get how much is cropped off the right
CropTop	Single	Set/Get how much is cropped off the top
Transparency Color	Long	Set/Get the color used for transparency
Transparent Background	MsoTriState	Set/Get whether transparent colors appear transparent

PictureFormat Methods

Name	Returns	Parameters	Description
Increment Brightness		Increment As Single	Increases the brightness by the Increment value
Increment Contrast		Increment As Single	Increases the contrast by the Increment value

PictureFormat Object Example

```
Sub SetPictureFormat()  
    Dim oShp As Shape  
    Dim oPF As PictureFormat  
    For Each oShp In ActiveSheet.Shapes  
        If oShp.Type = msoPicture Then  
  
            'Get the PictureFormat
```



```

    Set oPF = oShp.PictureFormat
    'Format the picture

    With oPF
        .TransparentBackground = msoTrue
        .TransparencyColor = RGB(255, 0, 0)
        .ColorType = msoPictureWatermark
    End With
End If
Next
End Sub

```

PivotAxis Object

The `PivotAxis` object allows for asymmetric drilling of a `PivotTable` via the `PivotRowAxis` and the `PivotColumnAxis`.

PivotAxis Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

PivotAxis Properties

Name	Returns	Description
<code>PivotLines</code>	<code>PivotLines</code>	Read-only. Returns the <code>PivotLines</code> attached to a given <code>PivotAxis</code> object

PivotCache Object and the PivotCaches Collection

The `PivotCaches` collection holds the collection of memory *caches* holding the data associated with a `PivotTable` report. Each `PivotCache` object represents a single memory cache for a `PivotTable` report. The parent of the `PivotCaches` collection is the `Workbook` object. Also, a possible parent of the `PivotCache` object is the `PivotTable` object.

The `PivotCaches` collection has a `Create` method besides the typical collection attributes. The `Create` method takes a `SourceType` constant (from the `XlPivotTableSourceType` constants) and `SourceData` to add a `PivotCache` to the collection.

PivotCache Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

PivotCache Properties

PivotCache Properties

Name	Returns	Description
ADoConnection	Object	Read-only. Returns an ADO connection object if the PivotCache is connected to an OLE DB data source
BackgroundQuery	Boolean	Set/Get if the processing of queries in a PivotTable report is done asynchronously. False for OLAP data sources
CommandText	VARIANT	Set/Get the SQL command used to retrieve data
CommandType	XlCmdType	Set/Get the type of CommandText (for example, xlCmdSQL, xlCmdTable)
Connection	VARIANT	Set/Get the OLE DB connection string, ODBC string, web data source, path to a text file, or path to a database
EnableRefresh	Boolean	Set/Get whether the PivotTable cache data can be refreshed. Always False for OLAP data sources
Index	Long	Read-only. Returns the spot in the collection for the specific cache
IsConnected	Boolean	Read-only. Returns whether the PivotCache is still connected to a data source
LocalConnection	String	Set/Get the connection string to an offline cube file. Blank for non-OLAP data sources. Use with UseLocalConnection
MaintainConnection	Boolean	Set/Get whether the connection to the data source does not close until the workbook is closed. Valid only against an OLE DB source
MemoryUsed	Long	Read-only. Returns the number of bytes used by the PivotTable cache
MissingItemsLimit	XlPivotTableMissingItems	Set/Get the maximum number of unique items that are retained per PivotTable field, even when they have no supporting data in the cache records
OLAP	Boolean	Read-only. Returns whether the PivotCache is connected to an OLAP server
OptimizeCache	Boolean	Set/Get whether the PivotTable cache is optimized when it is built. Always False for OLE DB data sources
QueryType	xlQueryType	Read-only. Returns the type of connection associated with the query table. (For example, xlOLEDBQuery, xlDAOQuery, xlTextImport)

Name	Returns	Description
RecordCount	Long	Read-only. Returns the number of records in the <code>PivotTable</code> cache
Recordset	Object	Set/Get the recordset used as the data source for the <code>PivotTable</code> cache
RefreshDate	Date	Read-only. Returns the date the cache was last refreshed
RefreshName	String	Read-only. Returns the name of the person who last refreshed the cache
RefreshOnFile Open	Boolean	Set/Get whether the <code>PivotTable</code> cache is refreshed when the workbook is opened
RefreshPeriod	Long	Set/Get how long (in minutes) between automatic refreshes from the data source. Set to 0 to disable
RobustConnect	<code>XlRobustConnect</code>	Set/Get the method by which the <code>PivotCache</code> connects to its data source
SavePassword	Boolean	Set/Get whether an ODBC connection password is saved with the query table
Source Connection File	String	Set/Get the name of the file that was used to create the <code>PivotTable</code>
SourceData	Variant	Set/Get the data source for the <code>PivotTable</code> report
SourceData File	String	Read-only. Returns the name of the source data file for the <code>PivotCache</code>
SourceType	<code>XlPivotTableSourceType</code>	Read-only. Returns a value that identifies the type of item being published
UpgradeOnRefresh	Boolean	Set/Get whether to upgrade the <code>PivotCache</code> and all connected <code>PivotTables</code> on the next refresh
UseLocal Connection	Boolean	Set/Get if the <code>LocalConnection</code> property is used to set the data source. <code>False</code> means the <code>Connection</code> property is used. Allows you to store some data sources offline
Version	Variant	Read-only. Returns the version in which the <code>PivotTable</code> was created
Workbook Connection	Variant	Read-only. Establishes a connection between the active workbook and the <code>PivotCache</code> object

PivotCache Methods

Name	Returns	Parameters	Description
CreatePivotTable	PivotTable	Table Destination As Variant, [TableName], [ReadData], [Default Version]	Creates a PivotTable report that is based on the current PivotCache object. The TableDestination parameter specifies where the new PivotTable report will be located. A TableName can also be specified. Set ReadData to True to fill the cache with all the records from the external database. Set ReadData to False to only retrieve some of the data. DefaultVersion is the default version of the PivotTable report
Make Connection			Makes a connection for the specified PivotCache
Refresh			Refreshes the data in the PivotTable cache with the latest copy of the external data
ResetTimer			Resets the time for the automatic refresh set by RefreshPeriod property
SaveAsODC		ODCFileName As String, [Description], [Keywords]	Saves the PivotCache source as an Office Data Connection (ODC) file. ODCFileName is the location where the file is to be saved. Description is the description that will be saved in the file. Keywords is a list of space-separated keywords that can be used to search for this file

PivotCache Object and the PivotCaches Collection Example

```
Sub RefreshPivotCache()  
Dim oPC As PivotCache  
Set oPC = ActiveWorkbook.PivotCaches(1)  
With oPC  
    'Only refresh if the data is over 1 hour old
```

```

    If .RefreshDate < Now - TimeValue("01:00:00") Then
        .Refresh
    End If
End With
End Sub

```

PivotCell Object

Represents a cell somewhere inside a `PivotTable`. You access the `PivotCell` object through the range object. Once obtained, you can use the various properties of the `PivotCell` object to retrieve data from a `PivotTable`. For example, you can use the `PivotCellType`, `ColumnItems`, and `RowItems` properties to locate a particular sales person's total sales for a specific region.

This object mirrors the functionality of the `GETPIVOTDATA` worksheet function and the `GetPivotData` method of the `PivotTable` object. The difference is that the `PivotCell` object can render information about where the cell is in the report. The `GETPIVOTDATA` worksheet function and the `GetPivotData` method do just the opposite. They yield the value associated with the row and column heading you provide.

PivotCell Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

PivotCell Properties

Name	Returns	Description
<code>ColumnItems</code>	<code>PivotItemList</code>	Read-only. Returns the items that represent the selected range on the column axis
<code>Custom Subtotal Function</code>	<code>XlConsolidationFunction</code>	Read-only. Returns the custom subtotal function field setting of the <code>PivotCell</code>
<code>DataField</code>	<code>PivotField</code>	Read-only. Returns the selected data field
<code>PivotCellType</code>	<code>XlPivotCellType</code>	Read-only. Returns the <code>PivotTable</code> entity that the selected cell corresponds to
<code>PivotColumnLine</code>	<code>PivotLine</code>	Returns the <code>PivotLine</code> on a column if the specified <code>PivotCell</code> is in the Column Area. This property will return an error for <code>PivotCells</code> in the Row area
<code>PivotField</code>	<code>PivotField</code>	Read-only. Returns the <code>PivotTable</code> field containing the upper-left corner of the specified range
<code>PivotItem</code>	<code>PivotItem</code>	Read-only. Returns the <code>PivotTable</code> item containing the upper-left corner of the specified range
<code>PivotRowLine</code>	<code>PivotLine</code>	Returns the <code>PivotLine</code> on a row if the specified <code>PivotCell</code> is in the Row Area. This property will return an error for <code>PivotCells</code> in the Column area

Table continued on following page

PivotField Object, PivotFields Collection, and the CalculatedFields Collection

Name	Returns	Description
PivotTable	PivotTable	Read-only. Returns the PivotTable report containing the upper-left corner of the specified range, or the PivotTable report associated with the PivotChart report
Range	Range	Read-only. Returns the range to which the specified PivotCell applies
RowItems	PivotItemList	Read-only. Returns the items that represent the selected range on the row axis

PivotField Object, PivotFields Collection, and the CalculatedFields Collection

The PivotFields collection holds the collection of fields associated with the parent PivotTable report. The CalculatedFields collection holds the collection of calculated fields associated with the parent PivotTable report. Each PivotField object represents single field in a PivotTable report. The parent of the PivotFields and CalculatedFields collection is the PivotTable object.

The CalculatedFields collection has an Add method that adds a new calculated field to the collection, given a Name, a Formula, and a UseStandardFormula parameter.

PivotField Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

PivotField Properties

Name	Returns	Description
AllItemsVisible	Boolean	A read-only Boolean set to True by default, this property checks whether manual filtering is applied to either a PivotField or a CubeField. This property is automatically set to False when any manual filtering is applied
AutoShowCount	Long	Read-only. Returns the number of top or bottom items that are automatically displayed in the PivotTable field
AutoShowField	String	Read-only. Returns the name of the data field used to figure out what top or bottom items to show automatically for a PivotTable field
AutoShowRange	Long	Read-only. Returns either xlTop if the top items are shown automatically or xlBottom if the bottom items are shown

Name	Returns	Description
AutoShowType	Long	Read-only. Returns either <code>xlAutomatic</code> if <code>AutoShow</code> is <code>True</code> or <code>xlManual</code> if <code>AutoShow</code> is disabled
AutoSortCustomSubtotal	Long	Read-only. Returns a value representing the type of operation used to automatically sort a given <code>PivotField</code> . The possible values are: 1 (Automatic), 2 (Sum), 3 (Count), 4 (Average), 5 (Max), 6 (Min), 7 (Product), 8 (Count Nums), 9 (StdDev), 10 (StdDevp), 11 (Var), and 12 (Varp)
AutoSortField	String	Read-only. Returns the data field name that will be used to sort the <code>PivotTable</code> field automatically
AutoSortOrder	<code>XLSortOrder</code>	Read-only. Returns one of the <code>XLSortOrder</code> constants specifying the automatic sort order type used for the field
AutoSortPivotLine	<code>PivotLine</code>	Read-only. Returns the name of the <code>PivotLine</code> used to sort the specified <code>PivotTable</code> field automatically
BaseField	Variant	Set/Get the base field used for a calculation. Data fields only
BaseItem	Variant	Set/Get the base item in the base field used for a calculation. Data fields only
Calculation	<code>XL PivotField Calculation</code>	Set/Get the type of calculation to do on the data field
Caption	String	Set/Get the text label to use for the field
ChildField	<code>PivotField</code>	Read-only. Returns the child field of the current field, if any
ChildItems	Variant	Read-only. Parameters: [Index]. Returns an object or collection containing a single <code>PivotTable</code> item (<code>PivotItem</code>) or group of <code>PivotTable</code> items (<code>PivotItems</code>) associated with the field
CubeField	<code>CubeField</code>	Read-only. Returns the cube field that the current <code>PivotTable</code> field comes from
CurrentPage	Variant	Set/Get the current page showing for the page field. Page fields only
CurrentPageList	Variant	Set/Get an array of strings corresponding to the list of items included in a multiple-item page field of a <code>PivotTable</code> report

Table continued on following page

PivotField Properties

Name	Returns	Description
CurrentPageName	String	Set/Get the displayed page of the PivotTable report
DatabaseSort	Boolean	Set/Get whether manual sorting and repositioning of data items in an OLAP PivotTable is allowed. Note that setting the DatabaseSort to either True or False causes an update
DataRange	Range	Read-only. Returns a range containing the data or items in the field
DataType	XlPivotField DataType	Read-only. Returns the data type of the PivotTable field
DisplayAsCaption	Boolean	Read-only. This property returns True when a given member property is used as a caption, and False when a member property PivotField is not used as caption
DisplayAsTooltip	Boolean	Read-only. This property returns True when a given member property is displayed in ToolTips, and False when a member property PivotField is not displayed in ToolTips
DisplayInReport	Boolean	Read-only. This property returns True when a given member property is displayed in the PivotTable, and False when a member property is not displayed in the PivotTable
DragToColumn	Boolean	Set/Get whether the field can be dragged to a column position
DragToData	Boolean	Set/Get whether the field can be dragged to the data position
DragToHide	Boolean	Set/Get whether the field can be dragged off the PivotTable report and therefore hidden
DragToPage	Boolean	Set/Get whether the field can be dragged to the page position
DragToRow	Boolean	Set/Get whether the field can be dragged to a row position
DrilledDown	Boolean	Set/Get whether the PivotTable field can be drilled down
EnableItem Selection	Boolean	Set/Get whether the ability to use the field drop-down in the user interface is enabled
Enable Multiple PageItems	Boolean	Set/Get whether checkboxes are enabled in the drop-down boxes of the fields in the PivotTable's Filter area

Name	Returns	Description
Formula	String	Set/Get the formula associated with the field, if any
Function	XlConsolidationFunction	Set/Get the type of function used to summarize the PivotTable field
GroupLevel	Variant	Read-only. Returns how the field is placed within a group of fields
Hidden	Boolean	Set/Get whether a given level in an OLAP hierarchy is visible. Set this property to True to hide levels, and False to make them visible
HiddenItems	Variant	Read-only. Parameters: [Index]. Returns an object or collection containing a single hidden PivotTable item (PivotItem) or group of hidden PivotTable items (PivotItems) associated with the field
HiddenItemsList	Variant	Set/Get an array of strings that are hidden items for the PivotField
IncludeNewItemInFilter	Boolean	Set/Get whether excluded or included items should be tracked when manual filtering is applied to a given PivotField. Set this property to True to keep track of excluded items, and set to False to keep track of included items
IsCalculated	Boolean	Read-only. Returns whether the PivotTable field is calculated
IsMemberProperty	Boolean	Read-only. Returns whether the PivotField contains member properties
LabelRange	Range	Read-only. Returns the cell range containing the field's label
LayoutBlankLine	Boolean	Set/Get whether a blank row is added just after the current row field
LayoutCompactRow	Boolean	Set/Get whether a given PivotField is compacted when rows are selected
LayoutForm	XlLayoutFormType	Set/Get how the items will appear in the field
LayoutPageBreak	Boolean	Set/Get whether a page break is inserted after each field
LayoutSubtotalLocation	XlSubtotalLocationType	Set/Get the location for the field subtotals as compared to the current field

Table continued on following page

PivotField Properties

Name	Returns	Description
MemberPropertyCaption	String	Set/Get which member property is used as caption for a given level. Note that this setting only has a visual effect when the UseMemberPropertyAsCaption property is set to True for the PivotField
MemoryUsed	Long	Read-only. Returns the number of bytes of computer memory being used for the field
Name	String	Set/Get the name of the field
NumberFormat	String	Set/Get the format used for numbers in the field
Orientation	XlPivotFieldOrientation	Set/Get where the field is located in the PivotTable report
ParentField	PivotField	Read-only. Returns the parent field of the current field, if any
ParentItems	Variant	Read-only. Parameters: [Index]. Returns an object or collection containing a single parent PivotTable item (PivotItem) or group of parent PivotTable items (PivotItems) associated with the field
PivotFilters	Variant	Sets/Gets the PivotFilters for a given PivotField
Position	Variant	Set/Get the position number of the field among all the fields in the same orientation
PropertyOrder	Long	Set/Get the display position of the member property within the cube field to which it belongs (setting will rearrange the order). Valid only for PivotField objects that are member property fields
PropertyParentField	PivotField	Read-only. Returns the field to which the properties in this field are linked
ServerBased	Boolean	Set/Get whether only items that match the page field selection are retrieved from the external data source
ShowAllItems	Boolean	Set/Get whether all items in the field are displayed
ShowingInAxis	Boolean	Read-only. Returns True if the PivotField is currently visible and returns False if the PivotField is currently not visible
SourceCaption	Variant	Read-only. Returns the original caption as it existed on the OLAP server for a given PivotField

Name	Returns	Description
SourceName	String	Read-only. Returns the name of the source data for the field
StandardFormula	String	Set/Get the formulas with standard U.S. formatting
SubtotalName	String	Set/Get the label used for the subtotal column or row for this field
Subtotals	Variant	Parameters: [Index]. Set/Get the subtotals displayed for the field
TotalLevels	Variant	Read-only. Returns the total number of fields in the current field group
UseMemberPropertyAsCaptions	Boolean	Set/Get whether member property captions are used for captions in a given PivotField. If set to True, then the MemberPropertyCaption property is used to define which member property caption to display. If none is specified, then the first member property of that PivotField (in data source order) will be displayed as the caption for items of that PivotField
Value	String	Set/Get the name of the field
VisibleItems	Variant	Read-only. Parameters: [Index]. Returns an object or collection containing a single visible PivotTable item (PivotItem) or group of visible PivotTable items (PivotItems) associated with the field
VisibleItemsList	Variant	Set/Get an array that represents the included items in the manual filter applied to a given PivotField

PivotField Methods

Name	Returns	Parameters	Description
AddPageItem		Item As String, [ClearList]	Adds an additional item to a multiple item page field. Item is the source name of a PivotItem object, corresponding to the specific OLAP member unique name. ClearList indicates whether to delete all existing items before adding the new item

Table continued on following page

PivotField Methods

Name	Returns	Parameters	Description
AutoShow		Type As Long, Range As Long, Count As Long, Field As String	Set the number of top or bottom items to display for a row, page, or column field. Type describes whether the items are shown as xlAutomatic or xlManual. Range is the location to start showing items. Count is the number of items to show for the field. Field is the base data field name
AutoSort		Order As Long, Field As String, PivotLine As Variant, Custom Subtotal As Variant	Sets the field to sort automatically based on the Order specified (using XlSortOrder constants) and the base data Field
Calculated Items	Calculated Items		Returns the group of calculated PivotTable items associated with the field
ClearAll Filters			Clears all filters applied to the PivotField, including manual filters and those applied from the PivotFilters collection of the PivotField
ClearLabel Filters			Clears all Label and Date filters applied to the PivotFilters collection of the PivotField
ClearManual Filters			Sets the Visible property to True for all items in a given PivotField. This property will also clear the HiddenItemsList and VisibleItemsList collections in OLAP PivotTables
ClearValue Filters			Clears all Value filters applied to the PivotFilters collection of the PivotField
Delete			Deletes the PivotField object

PivotField Object, PivotFields Collection, and the CalculatedFields Example

Name	Returns	Parameters	Description
DrillTo			Allows for drilling to a specified <code>PivotField</code> , so long as the hierarchy containing the requested field is an actual <code>PivotTable</code> and the field being drilled is in the same hierarchy or attribute chain
PivotItems	Variant	[Index]	Returns an object or collection containing a single <code>PivotTable</code> item (<code>PivotItem</code>) or group of <code>PivotTable</code> items (<code>PivotItems</code>) associated with the field

PivotField Object, PivotFields Collection, and the CalculatedFields Example

```
Sub AddField()  
Dim oPT As PivotTable  
Dim oPF As PivotField  
'Set target pivot table  
Set oPT = ActiveSheet.PivotTables(1)  
'Add a calculated field  
Set oPF = oPT.CalculatedFields.Add("Total", "=Price * Volume", True)  
oPF.Orientation = xlDataField  
End Sub
```

PivotFilter Object and the PivotFilters Collection

The `PivotFilters` collection holds the collection of filters actively being applied to a `PivotTable`. Each `PivotFilter` object represents a single filter in the `PivotFilters` collection. The `PivotFilters` collection has no properties and methods outside the typical collection attributes listed at the beginning of this appendix.

PivotFilter Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

PivotFilter Properties

Name	Returns	Description
Active	Boolean	Read-only. Returns <code>True</code> when the filtered <code>PivotField</code> is in the <code>PivotTable</code> , and the filter is evaluated when the <code>PivotTable</code> is updated. It returns <code>False</code> when the filtered <code>PivotField</code> is not in the <code>PivotTable</code> and has no effect on the <code>PivotTable</code> calculation
DataCubeField	CubeField	Returns the Value field actively being filtered by in the Value filter of an OLAP <code>PivotTable</code>
DataField	PivotField	Returns the Value field actively being filtered by in the Value filter of a Non-OLAP <code>PivotTable</code>
Description	String	Read-only. Returns the description for the <code>PivotFilter</code> object
FilterType	xlPivotFilter	Read-only. Returns an <code>xlPivotFilter</code> constant, defining the type of filter that is to be applied
IsMemberProperty Filter	Boolean	Read-only. Specifies whether the label filter is based on the <code>PivotItem</code> captions of a member property of the field or on the <code>PivotItem</code> captions of the <code>PivotField</code> itself. Returns <code>True</code> if the label filter is based on <code>PivotItem</code> captions of a member property of the <code>PivotField</code> . Returns <code>False</code> if the filter is based on the <code>PivotItem</code> captions of the <code>PivotField</code>
MemberProperty Field	PivotField	Returns the property of the <code>PivotField</code> on which the filter is based. Note that this property is valid only for Label filters and OLAP <code>PivotField</code> , which contain at least one member property
Name	String	Set/Get the name of the filter
Order	Integer Long	Set/Get the evaluation order of all Value and Top Nth type filters applied to a <code>PivotTable</code>
PivotField	PivotField	Read-only. Returns the name of the <code>PivotField</code> to which a filter is being applied
Value1	Variant	Set/Get any parameter used to define a filter for a <code>PivotField</code>
Value2	Variant	Set/Get any parameter used to define a filter for a <code>PivotField</code>

PivotFilter Methods

Name	Returns	Parameters	Description
Delete			Deletes the filter and removes it from the filter collections of the <code>PivotField</code> and the <code>PivotTable</code>

PivotFilters Object Example

```
Sub CreatePivotFilters()
Dim oPivotTable As PivotTable

'Set target pivot table
Set oPivotTable = ActiveSheet.PivotTables("PivotTable1")

'Add two filters, filtering out drums with over $500 in revenue
With oPivotTable
.PivotFields("Product").PivotFilters.Add Type:=xlCaptionContains, Value1:="Drums"
.PivotFields("Revenue").PivotFilters.Add Type:=xlValueIsGreaterThan, Value1:=500
End With
End Sub
```

PivotFormula Object and the PivotFormulas Collection

The `PivotFormulas` collection holds the formulas associated with the `PivotTable`. Each `PivotFormula` object represents a formula being used in a `PivotTable` report. The parent of the `PivotFormulas` collection is the `PivotTable` object.

The `PivotFormulas` collection has an `Add` method besides the typical collection attributes. The `Add` method adds a `PivotFormula` to the collection but requires the `Formula` and `UseStandardFormula` parameters to be specified.

PivotFormula Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

PivotFormula Properties

Name	Returns	Description
<code>Formula</code>	<code>String</code>	Set/Get the formula associated with the table. Use the A1-style reference notation
<code>Index</code>	<code>Long</code>	Set/Get the order that the formulas in the parent collection will be processed
<code>StandardFormula</code>	<code>String</code>	Set/Get the formulas with standard U.S. formatting
<code>Value</code>	<code>String</code>	Set/Get the formula associated with the table

PivotFormula Methods

Name	Returns	Parameters	Description
Delete			Deletes the formula from the parent collection

PivotItem Object, PivotItems Collection, and the CalculatedItems Collection

The `PivotItems` collection holds the collection of individual data entries in a field. The `CalculatedItems` collection holds the collection of individual calculated entries in a field. Each `PivotItem` object represents a single entry in a data field. The parent of the `PivotItems` and `CalculatedItems` collections is the `PivotField` object.

The `Add` method of the `PivotItems` collection adds another item to the collection (only a `Name` is required). The `Add` method of the `CalculatedItems` collection adds another item to the collection but requires the `Name`, `Formula`, and `UseStandardFormula` parameters to be specified.

PivotItem Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

PivotItem Properties

Name	Returns	Description
<code>Caption</code>	<code>String</code>	Set/Get the label text associated with the item
<code>ChildItems</code>	<code>Variant</code>	Read-only. Parameters: [<code>Index</code>]. Returns an object or collection containing a single <code>PivotTable</code> item (<code>PivotItem</code>) or group of <code>PivotTable</code> items (<code>PivotItems</code>) associated with the item
<code>DataRange</code>	<code>Range</code>	Read-only. Returns a range containing the data or items in the item
<code>DrilledDown</code>	<code>Boolean</code>	Set/Get whether the <code>PivotTable</code> item is drilled down
<code>Formula</code>	<code>String</code>	Set/Get the formula associated with item, if any
<code>IsCalculated</code>	<code>Boolean</code>	Read-only. Returns whether the item that was calculated is a data item
<code>LabelRange</code>	<code>Range</code>	Read-only. Returns the cell range containing the field's item
<code>Name</code>	<code>String</code>	Set/Get the name of the item
<code>ParentItem</code>	<code>PivotItem</code>	Read-only. Returns the parent item of the current item, if any

Name	Returns	Description
ParentShowDetail	Boolean	Read-only. Returns whether the current item is being shown because the one of the item's parents is set to show detail
Position	Long	Set/Get the position number of the item among all the items in the same orientation
RecordCount	Long	Read-only. Returns the number of records in the PivotTable cache that contain the item
ShowDetail	Boolean	Set/Get whether the detail items are being displayed
SourceName	Variant	Read-only. Returns the name of the source data for the item
SourceNameStandard	String	Read-only. Returns the PivotItem's source name in standard U.S. format settings
StandardFormula	String	Set/Get the formulas with standard U.S. formatting
Value	String	Set/Get the name of the specified item
Visible	Boolean	Set/Get whether the item is visible

PivotItem Methods

Name	Description
Delete	Deletes the item from the collection
DrillTo	Allows for drilling to a specified PivotField, so long as the hierarchy containing the requested field is on the actual PivotTable and the field being drilled is in the same hierarchy or attribute chain

PivotItem Object, PivotItems Collection, and the CalculatedItems Collection Example

```

Sub ShowPivotItemData()
    Dim oPT As PivotTable
    Dim oPI As PivotItem
    'Get the pivot table
    Set oPT = ActiveSheet.PivotTables(1)
    'Get the pivot item
    Set oPI = oPT.PivotFields("Product").PivotItems("Oranges")
    'Show all the source data rows for that pivot item
    oPI.ShowDetail = True
End Sub

```

PivotItemList Object

PivotItemList Object

Represents a list of `PivotItems` associated with a particular cell in a `PivotTable`. You access the list through the `PivotCell` object. `PivotItemLists` are accessed either through the `ColumnItems` or `RowItems` properties of the `PivotCell` object. How many row and column items there are in the `PivotItemList` depend on the structure of the `PivotTable`.

For example, cell D5 is in a `PivotTable` called `WroxSales1`. In the row area to the left of cell D5 is the row heading OR (Oregon). To the left of OR is another row label called `Region1`. Based on this information, the following will yield 2:

```
MsgBox wksPivotTable.Range("D5").PivotCell.RowItems.Count
```

The following will yield `Region1`, the farthest label to the left of cell D5:

```
MsgBox wksPivotTable.Range("D5").PivotCell.RowItems(1)
```

Finally, the following will yield OR, the second farthest label to the left of cell D5:

```
MsgBox wksPivotTable.Range("D5").PivotCell.RowItems(2)
```

A use for both the `PivotItemList` and `PivotCell` objects has yet to be found. Normally, you are looking for the opposite. You want to retrieve information based on row or column items (headings) you provide, something the `GetPivotData` method and the `GETPIVOTDATA` worksheet function can obtain.

PivotItemList Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

PivotItemList Properties

Name	Returns	Description
Count	Long	Returns the number of objects in the collection

PivotItemList Methods

Name	Returns	Parameters	Description
Item	<code>PivotItem</code>	Index As Variant	Returns a single <code>PivotItem</code> from the <code>PivotItemList</code>

PivotLayout Object

The `PivotLayout` object describes how the fields of a `PivotChart` are placed in the parent chart. Either the `Chart` object or the `ChartGroup` object is the parent of the `PivotChart` object.

PivotLayout Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

PivotLayout Properties

Name	Returns	Description
<code>PivotTable</code>	<code>PivotTable</code>	Read-only. Returns the <code>PivotTable</code> associated with the <code>PivotChart</code>

PivotLine Object, the PivotLines Collection, and the PivotLinesCells Collection

The `PivotLine` object defines the profile of a particular row or column. Each row and column in a `PivotTable` is a Regular line, a Blank line, a SubTotal line, or a GrandTotal line. The `PivotLines` collection contains all of the `PivotLine` objects in a given `PivotTable`. The `PivotCells` collection represents the each `PivotCell` within a given `PivotLine`. The `PivotLines` and `PivotLinesCells` collections have no properties and methods outside the typical collection attributes listed at the beginning of this appendix.

PivotLine Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

PivotLine Properties

Name	Returns	Description
<code>LineType</code>	<code>xlPivotLineType</code>	Read-only. Returns the line type for the <code>PivotLine</code> object
<code>PivotLineCells</code>	<code>PivotLineCells</code>	Read-only. Returns the collection of <code>PivotCell</code> objects contained in a given <code>PivotLine</code> object
<code>Position</code>	<code>Position</code>	Set/Get the position of a given <code>PivotLine</code> object

PivotTable Object and the PivotTables Collection

The `PivotTables` collection contains the collection of `PivotTables` in the parent worksheet. Each `PivotTable` object in the collection allows manipulation and creation of Excel `PivotTables`. The parent of the `PivotTables` collection is the `Worksheet` object.

The `PivotTables` collection has an `Add` method besides the typical collection attributes. The `Add` method takes a new `PivotTable` cache (containing the data) and the destination single cell range determining the upper-left corner of the `PivotTable` report to create a new `PivotTable` report. The name of the new `PivotTable` report can also be specified in the `Add` method.

PivotTable Common Properties

PivotTable Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

PivotTable Properties

Name	Returns	Description
<code>ActiveFilters</code>	<code>ActiveFilter</code>	Read-only. Indicates the current active filter in a given <code>PivotTable</code>
<code>AllowMultipleFilters</code>	<code>Boolean</code>	Set/Get whether a given <code>PivotField</code> can have multiple filters applied to it. Default is <code>False</code> for any new <code>PivotField</code>
<code>CacheIndex</code>	<code>Long</code>	Set/Get the index number pointing to the <code>PivotTable</code> cache of the current <code>PivotTable</code>
<code>CalculatedMembers</code>	<code>CalculatedMembers</code>	Read-only. Returns all the calculated fields and calculated items for the <code>PivotTable</code>
<code>ColumnFields</code>	<code>Object</code>	Read-only. Parameters: [<code>Index</code>]. Returns an object or collection containing the <code>PivotTable</code> field (<code>PivotField</code>) or <code>PivotTable</code> fields (<code>PivotFields</code>) associated with the columns of the <code>PivotTable</code>
<code>ColumnGrand</code>	<code>Boolean</code>	Set/Get whether grand totals are shown for columns in the <code>PivotTable</code>
<code>ColumnRange</code>	<code>Range</code>	Read-only. Returns the range of cells containing the column area in the <code>PivotTable</code> report
<code>CompactLayoutColumnHeader</code>	<code>String</code>	Set/Get the column header caption that is displayed when the <code>PivotTable</code> is in compact row layout form
<code>CompactLayoutRowHeader</code>	<code>String</code>	Set/Get the row header caption that is displayed when the <code>PivotTable</code> is in compact row layout form
<code>CompactRowIndent</code>	<code>Long</code>	Set/Get the indent increment for <code>PivotItems</code> when the <code>PivotTable</code> is in compact row layout form
<code>CubeFields</code>	<code>CubeFields</code>	Read-only. Returns the collection of cube fields associated with the <code>PivotTable</code> report
<code>DataBodyRange</code>	<code>Range</code>	Read-only. Returns the range of cells containing the data area of the <code>PivotTable</code> report
<code>DataFields</code>	<code>Object</code>	Read-only. Parameters: [<code>Index</code>]. Returns an object or collection containing the <code>PivotTable</code> field (<code>PivotField</code>) or <code>PivotTable</code> fields (<code>PivotFields</code>) associated with the data fields of the <code>PivotTable</code>
<code>DataLabelRange</code>	<code>Range</code>	Read-only. Returns the range of cells that contain the labels for the data fields in the <code>PivotTable</code> report
<code>DataPivotField</code>	<code>PivotField</code>	Read-only. Returns all the data fields in a <code>PivotTable</code>

Name	Returns	Description
DisplayContext Tooltips	Boolean	Set/Get whether context ToolTips are displayed within the PivotTable
DisplayEmpty Column	Boolean	Read-only. Returns whether the non-empty MDX keyword is included in the query to the OLAP provider for the value axis
DisplayEmptyRow	Boolean	Read-only. Returns whether the non-empty MDX keyword is included in the query to the OLAP provider for the category axis
DisplayError String	Boolean	Set/Get whether the string in the ErrorString property is displayed in cells that contain errors
DisplayField Captions	Boolean	Set/Get whether the field captions and filter buttons are displayed on the PivotTable itself
DisplayImmediate Items	Boolean	Set/Get whether items in the row and column areas are visible when the data area of the PivotTable is empty
Display MemberProperty Tooltips	Boolean	Set/Get whether the member properties of PivotItems are displayed in ToolTips
DisplayNull String	Boolean	Set/Get whether the string in the NullString property is displayed in cells that contain null values
EnableData ValueEditing	Boolean	Set/Get whether to show an alert when the user overwrites values in the data area of the PivotTable
Enable Drilldown	Boolean	Set/Get whether drilldown in the PivotTable report is enabled
EnableField Dialog	Boolean	Set/Get whether the PivotTable Field dialog box is displayed when the user double-clicks a PivotTable field
EnableField List	Boolean	Set/Get whether to disable the ability to display the field well for the PivotTable. If the list was already visible, it disappears
EnableWizard	Boolean	Set/Get whether the PivotTable Wizard is available
ErrorString	String	Set/Get the string that is displayed in cells that contain errors. Use with the DisplayErrorString property
FieldListSort Ascending	Boolean	Set/Get sort order of fields in a given PivotField for a non-OLAP PivotTable. True sets the sort order to ascending, while False sets the sort order to descending

Table continued on following page

PivotTable Properties

Name	Returns	Description
GrandTotal Name	String	Set/Get the string label that will be displayed on the grand total column or row heading of a PivotTable report. Default is "Grand Total"
HasAuto Format	Boolean	Set/Get whether the PivotTable report is automatically reformatted when the data is refreshed or the fields are moved around
HiddenFields	Object	Read-only. Parameters: [Index]. Returns an object or collection containing the PivotTable field (PivotField) or PivotTable fields (PivotFields) associated with the hidden fields of the PivotTable
InGridDropZones	Boolean	Set/Get whether drop zones are available directly on the spreadsheet. Setting this property to True allows for dragging pivot fields directly into the PivotTable object on the spreadsheet, just as in Excel 2000-2003
InnerDetail	Boolean	Set/Get the name of the field that will show the detail when the ShowDetail property is True
LayoutRowDefault	xlLayout RowType	Set/Get the default layout settings for PivotFields that are added to the PivotTable for the first time. Use the xlLayoutRowType constants to define whether the default layout will be compact, tabular, or outline
Location	String	Specifies the location of the given PivotTable
ManualUpdate	Boolean	Set/Get whether the PivotTable report is only recalculated manually. Default for this property is False
MDX	String	Read-only. Returns the MDX (Multidimensional Expression) that would be sent to the provider to populate the current PivotTable view
MergeLabels	Boolean	Set/Get whether the outer-row item, column item, subtotal, and grand total labels of a PivotTable report have merged cells
Name	String	Set/Get the name of the PivotTable report
NullString	String	Set/Get the string that is displayed in cells that contain null strings. Use with the DisplayNullString property
PageFieldOrder	Long	Set/Get how new page fields are added to a PivotTable report's layout. Use the XLOrder constants
PageFields	Object	Read-only. Parameters: [Index]. Returns an object or collection containing the PivotTable field (PivotField) or PivotTable fields (PivotFields) associated with the page fields of the PivotTable

Name	Returns	Description
PageFieldStyle	String	Set/Get the style used for a page field area in a PivotTable
PageFieldWrapCount	Long	Set/Get how many page fields are in each column or row of the PivotTable report
PageRange	Range	Read-only. Returns the range containing the page area in the PivotTable report
PageRangeCells	Range	Read-only. Returns the range containing the page fields and item drop-down lists in the PivotTable report
PivotColumnAxis	PivotAxis	Read-only. Returns a PivotAxis object that represents the entire column axis
PivotFormulas	PivotFormulas	Read-only. Returns the collection of formulas used in the PivotTable report
PivotRowAxis	PivotAxis	Read-only. Returns a PivotAxis object that represents the entire row axis
PivotSelection	String	Set/Get the data and label selection in the PivotTable using the standard PivotTable report selection format. For example, to select the data and label for the Country equal to "Canada", the string would be "Country[Canada]"
PivotSelectionStandard	String	Set/Get the PivotTable selection in standard PivotTable report format using U.S. settings
PreserveFormatting	Boolean	Set/Get whether formatting of the PivotTable report is preserved when the report is changed, sorted, pivoted, refreshed, or recalculated
PrintDrillIndicators	Boolean	Set/Get whether drill indicators are printed along with the PivotTable
PrintTitles	Boolean	Set/Get whether the print title set on the PivotTable report is printed whenever the parent worksheet is printed
RefreshDate	Date	Read-only. Returns the date that the PivotTable report data was refreshed last
RefreshName	String	Read-only. Returns the name of the user who last refreshed the PivotTable report data
RepeatItemsOnEachPrintedPage	Boolean	Set/Get whether row, column, and item labels appear on the first row of each page when the PivotTable report is printed

Table continued on following page

PivotTable Properties

Name	Returns	Description
RowFields	Object	Read-only. Parameters: [Index]. Returns an object or collection containing the PivotTable field (PivotField) or PivotTable fields (PivotFields) associated with the rows of the PivotTable
RowGrand	Boolean	Set/Get whether grand totals are shown for rows in the PivotTable
RowRange	Range	Read-only. Returns the range of cells containing the row area in the PivotTable report
SaveData	Boolean	Set/Get whether the PivotTable report data is saved with the workbook
SelectionMode	XlPTSelectionMode	Set/Get how the PivotTable report selection mode is set (for example, xlLabelOnly)
ShowDrillIndicators	Boolean	Set/Get whether drill indicators are displayed on the PivotTable report
ShowCellBackgroundFromOLAP	Boolean	Set/Get whether the MDX that Excel asks for includes the BackColor property for each cell in the data area that corresponds to a cell in the OLAP data set
ShowPageMultipleItemLabel	Boolean	Set/Get whether “(Multiple Items)” will appear in the PivotTable cell whenever items are hidden and an aggregate of non-hidden items is shown in the PivotTable view
ShowTableStyleColumnHeaders	Boolean	Set/Get whether column headers are displayed
ShowTableStyleColumnStripes	Boolean	Set/Get whether the alternate color banding is applied to columns
ShowTableStyleRowHeaders	Boolean	Set/Get whether row headers are displayed
ShowTableStyleRowStripes	Boolean	Set/Get whether the alternate color banding is applied to rows
SmallGrid	Boolean	Set/Get whether a two-by-two grid is used for a newly created PivotTable report (True) or a blank stencil outline (False)
SortUsingCustomLists	Boolean	Set/Get whether custom sort lists are used to sort the items in a PivotField. Note that setting this property to True may adversely impact PivotTable performance

Name	Returns	Description
SourceData	Variant	Set/Get the source of the PivotTable report data. Can be a cell reference, an array, multiple ranges, and another PivotTable report. Not valid to use with OLE DB data sources
SubtotalHiddenPageItems	Boolean	Set/Get whether hidden page fields are included in row and column subtotals, block totals, and grand totals
TableRange1	Range	Read-only. Returns the range containing the whole PivotTable report, not including page fields
TableRange2	Range	Read-only. Returns the range containing the whole PivotTable report, with page fields
TableStyle2	String	Set/Get the current PivotTable report body style
Tag	String	Set/Get a string to be saved with the PivotTable report (for example, a description of the PivotTable report)
TotalsAnnotation	Boolean	When this property is set to <code>True</code> , the asterisk indicates that hidden items are included in the total. For non-OLAP data sources, the value of this property is always <code>False</code>
VacatedStyle	String	Set/Get the style to use for vacated cells when a PivotTable report is refreshed
Value	String	Set/Get the name of the PivotTable report
Version	XlPivotTableVersionList	Read-only. Returns the version number of Excel
ViewCalculatedMembers	Boolean	Set/Get whether calculated members for OLAP PivotTables can be viewed
VisibleFields	Object	Read-only. Parameters: [Index]. Returns an object or collection containing the PivotTable field (<code>PivotField</code>) or PivotTable fields (<code>PivotFields</code>) associated with the visible fields of the PivotTable
VisualTotals	Boolean	Set/Get whether PivotTables should retotal after an item has been hidden from view

PivotTable Methods

Name	Returns	Parameters	Description
AddDataField	PivotField	Field As Object, [Caption], [Function]	Adds a data field to a PivotTable report. Field is the unique field on the server, Caption is the label used to identify this data field, and Function is the function performed in the added data field
AddFields	Variant	[RowFields], [ColumnFields], [PageFields], [AddToTable]	Adds row, column, and page fields to a PivotTable report. RowFields, ColumnFields, and PageFields can hold a single string field name or an array of string field names. Set AddToTable to True to add the fields to the report. Set AddToTable to False to replace the fields in the report
CalculatedFields	CalculatedFields		Returns the collection of calculated fields in the PivotTable report
ChangeConnection		[Connection]	Change the connection string for a given pivot cache
ChangePivotCache		[PivotCache]	Change the PivotCache for a given PivotTable
ClearAllFilters			Clears all filters applied to the PivotField, including manual filters and those applied from the PivotFilters collection of the PivotField

Name	Returns	Parameters	Description
ClearTable			Removes all fields from the PivotTable, deletes all applied filters, deletes all applied sorting, and returns the PivotTable to an empty state
ConvertToFormulas		ConvertFilters As Boolean	Converts a given OLAP PivotTable to Cube formulas
CreateCubeFile	String	File As String, [Measures], [Levels], [Members], [Properties]	Creates a cube file from a PivotTable report connected to an OLAP data source. File is the name of the cube file to be created, Measures is an array of unique names of measures that are to be part of the slice, and Levels an array of strings where each array item is a unique level name. Members is an array of string arrays where the elements correspond, in order, to the hierarchies represented in the Levels array. Properties should be set to False if you don't want member properties being included in the slice
DisplayAllMemberPropertiesInTooltip	Boolean		Determines whether member properties are listed in a ToolTip when hovering over a member
GetData	Double	Name As String	Get the value of a specific cell in the PivotTable report. The Name parameter must be in the standard PivotTable report selection format

Table continued on following page

PivotTable Methods

Name	Returns	Parameters	Description
GetPivotData	Range	[DataField], [Field1], [Item1], [Field2], [Item2], [Field3], [Item3], [Field4], [Item4], [Field5], [Item5], [Field6], [Item6], [Field7], [Item7], [Field8], [Item8], [Field9], [Item9], [Field10], [Item10], [Field11], [Item11], [Field12], [Item12], [Field13], [Item13], [Field14], [Item14]	Returns information about a data item in a PivotTable report. FieldN is the name of a column or row field in the PivotTable report, and ItemN is the name of an item in FieldN

Name	Returns	Parameters	Description
ListFormulas			Creates a separate worksheet with the list of all the calculated PivotTable items and fields
PivotCache	PivotCache		Returns a data cache associated with the current PivotTable
PivotFields	Object	[Index]	Returns an object or collection containing the PivotTable field (PivotField) or PivotTable fields (PivotFields) associated with the fields of the PivotTable
PivotSelect		Name As String, [Mode As XlPTSelection Mode], [UseStandardName]	Selects the part of the PivotTable specified by Name parameter in the standard PivotTable report selection format. Mode decides which part of the PivotTable to select (for example, xlBlanks). Set UseStandardName to True for recorded macros that will play back in other locales

Table continued on following page

PivotTable Methods

Name	Returns	Parameters	Description
PivotTable Wizard		[SourceType], [SourceData], [TableDestination], [TableName], [RowGrand], [ColumnGrand], [SaveData], [HasAutoFormat], [AutoPage], [Reserved], [BackgroundQuery], [OptimizeCache], [PageFieldOrder], [PageFieldWrapCount], [ReadData], [Connection]	Creates a PivotTable report. The <code>SourceType</code> uses the <code>XLpivotTable</code> <code>SourceType</code> constants to specify the type of <code>SourceData</code> being used for the PivotTable. The <code>TableDestination</code> holds the range in the parent worksheet where the report will be placed. <code>TableName</code> holds the name of the new report. Set <code>RowGrand</code> or <code>ColumnGrand</code> to <code>True</code> to show grand totals for rows and columns, respectively. Set <code>HasAutoFormat</code> to <code>True</code> for Excel to format the report automatically when it is refreshed or changed. Use the <code>AutoPage</code> parameter to set if a page field is created for consolidation automatically. Set <code>BackgroundQuery</code> to <code>True</code> for Excel to query the data source asynchronously. Set <code>OptimizeCache</code> to <code>True</code> for Excel to optimize the cache when it is built. Use the <code>PageFieldOrder</code> with the <code>XLOrder</code> constants to set how new page fields are added to the report. Use the <code>PageFieldWrapCount</code> to set the number of page fields in each column or row. Set <code>ReadData</code> to <code>True</code> to copy the data from the external database into a cache. Finally, use the <code>Connection</code> parameter to specify an ODBC connection string for the PivotTable's cache.

PivotTable Object and the PivotTables Collection Example

Name	Returns	Parameters	Description
RefreshTable	Boolean		Refreshes the PivotTable report from the source data and returns True if successful
RowAxisLayout		RowLayout As XlLayout RowType	Sets the layout options for all existing PivotFields. Note that if layout options cannot be set on any of the PivotFields, no change will be made on any of the fields in the PivotTable
ShowPages	VARIANT	[PageField]	Creates a new PivotTable report for each item in the page field (PageField) in a new worksheet
Subtotal Location		Location As XlSubtotal LocationType	Sets the Layout SubtotalLocation property for all existing PivotFields. Note that changing the subtotal location has a visual effect only for those fields in outline form
Update			Updates the PivotTable report

PivotTable Object and the PivotTables Collection Example

```
Sub MakePrettyPivotTable()  
    Dim oPT As PivotTable  
    'Set the target pivot tablet  
    Set oPT = ActiveSheet.PivotTables("PivotTable1")  
  
    'Change layout, add color bandings, apply a style.  
    With oPT  
        .RowAxisLayout xlTabularRow  
        .ShowTableStyleColumnStripes = True  
        .ShowTableStyleRowStripes = True  
        .TableStyle2 = "PivotStyleMedium10"  
    End With  
End Sub
```

PlotArea Object

The `PlotArea` object contains the formatting options associated with the plot area of the parent chart. For 2D charts, the `PlotArea` includes trendlines, data markers, gridlines, data labels, and the axis labels—but not titles. For 3D charts, the `PlotArea` includes the walls, floor, axes, axis titles, check marks, and all of the items mentioned for the 2D charts. The area surrounding the plot area is the chart area. Please see the `ChartArea` object for formatting related to the chart area. The parent of the `PlotArea` is always the `Chart` object.

PlotArea Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

PlotArea Properties

Name	Returns	Description
<code>Format</code>	<code>ChartFormat</code>	Read-only. Returns the <code>ChartFormat</code> object, which controls the line, fill, and effect formatting for the chart area
<code>Height</code>	<code>Double</code>	Set/Get the height of the chart's plot area
<code>InsideHeight</code>	<code>Double</code>	Set/Get the height inside the plot area that does not include the axis labels
<code>InsideLeft</code>	<code>Double</code>	Set/Get the distance from the left edge of the plot area, not including axis labels, to the chart's left edge
<code>InsideTop</code>	<code>Double</code>	Set/Get the distance from the left edge of the plot area, not including axis labels, to the chart's left edge
<code>InsideWidth</code>	<code>Double</code>	Set/Get the width inside the plot area that does not include the axis labels
<code>Left</code>	<code>Double</code>	Set/Get the distance from the left edge of the plot area to the chart's left edge
<code>Name</code>	<code>String</code>	Read-only. Returns the name of the plot area
<code>Position</code>	<code>XLChartElementPosition</code>	Set/Get the position of a given chart's plot area
<code>Top</code>	<code>Double</code>	Set/Get the distance from the top edge of the plot area to the chart's top edge
<code>Width</code>	<code>Double</code>	Set/Get the width of the chart's plot area

PlotArea Methods

Name	Returns	Parameters	Description
<code>ClearFormats</code>	<code>Variant</code>		Clears any formatting made to the plot area
<code>Select</code>	<code>Variant</code>		Selects the plot area on the chart

PlotArea Object Example

This example uses the `PlotArea` object to make all the embedded charts in the workbook (not chart sheets) have the same size and position plot area, regardless of the formatting of the axes (for example, different fonts and number scales):

```

Sub MakeChartAreasSameSizeAsFirst()
Dim oCht As Chart, oPA As PlotArea
Dim dWidth As Double, dHeight As Double
Dim dTop As Double, dLeft As Double
'Get the dimensions of the inside of the plot area of the first chart
  With ActiveSheet.ChartObjects(1).Chart.PlotArea
    dWidth = .InsideWidth
    dHeight = .InsideHeight
    dLeft = .InsideLeft
    dTop = .InsideTop
  End With

'Loop through the charts in the workbook
  For Each oCht In Charts
'Get the PlotArea
    Set oPA = oCht.PlotArea

'Size and move the plot area
    With oPA
      If .InsideWidth > dWidth Then
        'Too big, make it smaller
        .Width = .Width - (.InsideWidth - dWidth)
      Else
        'Too small, move it left and make bigger
        .Left = .Left - (dWidth - .InsideWidth)
        .Width = .Width + (dWidth - .InsideWidth)
      End If

      If .InsideHeight > dHeight Then
        'Too big, make it smaller
        .Height = .Height - (.InsideHeight - dHeight)
      Else
        'Too small, move it left and make bigger
        .Top = .Top - (dHeight - .InsideHeight)
        .Height = .Height + (dHeight - .InsideHeight)
      End If

      'Set the position of the inside of the plot area
      .Left = .Left + (dLeft - .InsideLeft)
      .Top = .Top + (dTop - .InsideTop)
    End With
  Next
End Sub

```

Point Object and the Points Collection

The `Points` collection holds all of the data points of a particular series of a chart. In fact, a chart (`Chart` object) can have many chart groups (`ChartGroups` / `ChartGroup`) that can contain many series (`SeriesCollection` / `Series`), which in turn can contain many points (`Points` / `Point`). A `Point`

Point Common Properties

object describes the particular point of a series on a chart. The parent of the `Points` collection is the `Series` object. The `Points` collection has no properties and methods outside the typical collection attributes listed at the beginning of this appendix.

Point Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Point Properties

Name	Returns	Description
<code>ApplyPictToEnd</code>	Boolean	Set/Get whether pictures are added to the end of the point
<code>ApplyPictToFront</code>	Boolean	Set/Get whether pictures are added to the front of the point
<code>ApplyPictToSides</code>	Boolean	Set/Get whether pictures are added to the sides of the point
<code>DataLabel</code>	<code>DataLabel</code>	Read-only. Returns an object allowing you to manipulate the data label attributes (for example, formatting, text). Use with <code>HasDataLabel</code>
<code>Explosion</code>	Long	Set/Get how far out a slice (point) of a pie or doughnut chart will explode out. 0 for no explosion
<code>Format</code>	<code>ChartFormat</code>	Read-only. Returns the <code>ChartFormat</code> object, which controls the line, fill, and effect formatting for the chart area.
<code>Has3DEffect</code>	Boolean	Set/Get whether the point has a three-dimensional appearance
<code>HasDataLabel</code>	Boolean	Set/Get whether the point has a data label. Use with <code>DataLabel</code>
<code>Interior</code>	<code>Interior</code>	Read-only. Returns an object containing options to format the inside area of the point (for example, interior color)
<code>InvertIfNegative</code>	Boolean	Set/Get whether the point's color will be inverted if the point value is negative
<code>MarkerBackgroundColor</code>	Long	Set/Get the color of the point marker background Use the <code>RGB</code> function to create the color value
<code>MarkerBackgroundColorIndex</code>	<code>XIColorIndex</code>	Set/Get the color of the point marker background. Use the <code>XIColorIndex</code> constants or an index value in the current color palette

Name	Returns	Description
Marker Foreground Color	Long	Set/Get the color of the point marker foreground. Use the RGB function to create the color value
Marker Foreground ColorIndex	XlColor Index	Set/Get the color of the point marker foreground. Use the XlColorIndex constants or an index value in the current color palette
MarkerSize	Long	Set/Get the size of the point key marker
MarkerStyle	XlMarker Style	Set/Get the type of marker to use as the point key (for example, square, diamond, triangle, picture, and so on)
PictureType	XlChart PictureType	Set/Get how an associated picture is displayed on the point (for example, stretched, tiled). Use the XlPictureType constants
PictureUnit2	Long	Set/Get how many units a picture represents if the PictureType property is set to xlScale
Secondary Plot	Boolean	Set/Get if the point is on the secondary part of a Pie of Pie chart of a Bar of Pie chart
Shadow	Boolean	Set/Get whether the point has a shadow effect

Point Methods

Name	Returns	Parameters	Description
ApplyData Labels	Variant	[Type As XlDataLabels Type], [LegendKey], [AutoText], [HasLeader Lines], [ShowSeries Name], [ShowCategory Name], [ShowValue], [Show Percentage], [ShowBubble Size], [Separator]	Applies the data label properties specified by the parameters to the point. The Type parameter specifies whether no label, a value, a percentage of the whole, or a category label is shown. The legend key can appear by the point by setting the LegendKey parameter to True. Set AutoText to True if the object automatically generates appropriate text based on content. HasLeaderLines should be set to True if the series has leader lines. All the other parameters are simply the properties of the data labels that they describe

Table continued on following page

Point Object and the Points Collection Example

Name	Returns	Parameters	Description
ClearFormats	Variant		Clears the formatting made to a point
Copy	Variant		Cuts the point and places it in the clipboard
Delete	Variant		Deletes the point
Paste	Variant		Pastes the picture in the clipboard into the current point so it becomes the marker
Select	Variant		Selects the point on the chart

Point Object and the Points Collection Example

```
Sub ExplodePie()  
    Dim oPt As Point  
    'Get the first data point in the pie chart  
    Set oPt = ActiveSheet.ChartObjects(1).Chart.SeriesCollection(1).Points(1)  
    'Add a label to the first point only and  
    'set it away from the pie  
    With oPt  
        .ApplyDataLabels xlDataLabelsShowLabelAndPercent  
        .Explosion = 20  
    End With  
End Sub
```

Protection Object

Represents the group of sheet protection options. When you protect a sheet, you now have the option to only allow unlocked cells selected, allow cell, column, and row formatting, allow insertion and deletion of rows and columns, allow sorting, and more.

Setting Protection options is done via the `Protect` method of the `Worksheet` object. Use the `Protection` property of the `Worksheet` object to check the current protection settings:

```
MsgBox ActiveSheet.Protection.AllowFormattingCells
```

Protection Properties

Name	Returns	Description
AllowDeletingColumns	Boolean	Read-only. Returns whether the deletion of columns is allowed on a protected worksheet
AllowDeletingRows	Boolean	Read-only. Returns whether the deletion of rows is allowed on a protected worksheet
AllowEditRanges	AllowEditRanges	Read-only. Returns an <code>AllowEditRanges</code> object

Name	Returns	Description
Allow Filtering	Boolean	Read-only. Returns whether the user is allowed to make use of an <code>AutoFilter</code> that was created before the sheet was protected
Allow Formatting Cells	Boolean	Read-only. Returns whether the formatting of cells is allowed on a protected worksheet
AllowFormatting Columns	Boolean	Read-only. Returns whether the formatting of columns is allowed on a protected worksheet
AllowFormatting Rows	Boolean	Read-only. Returns whether the formatting of rows is allowed on a protected worksheet
AllowInserting Columns	Boolean	Read-only. Returns whether the inserting of columns is allowed on a protected worksheet
AllowInserting Hyperlinks	Boolean	Read-only. Returns whether the inserting of hyperlinks is allowed on a protected worksheet
AllowInserting Rows	Boolean	Read-only. Returns whether the inserting of rows is allowed on a protected worksheet
AllowSorting	Boolean	Read-only. Returns whether the sorting option is allowed on a protected worksheet
AllowUsing PivotTables	Boolean	Read-only. Returns whether the manipulation of <code>PivotTables</code> is allowed on a protected worksheet

Protection Object Example

The following routine sets `Protection` options based on the user name associated with the application (found in the Popular section of the Excel Options dialog box) and that user's settings on a table on the worksheet. If the user isn't found, a message appears and the default settings are used:

```

Sub ProtectionSettings()
    Dim rngUsers As Range, rngUser As Range
    Dim sCurrentUser As String

    'Grab the current username
    sCurrentUser = Application.UserName

    'Define the list of users in the table
    With wksAllowEditRange
        Set rngUsers = .Range(.Range("Users"), .Range("Users").End(xlToRight))
    End With

    'Locate the current user on the table
    Application.FindFormat.Clear
    Set rngUser = rngUsers.Find(What:=sCurrentUser, SearchOrder:=xlByRows,
    MatchCase:=False, SearchFormat:=False)

    'If current user is found on the table(

```

PublishObject Object and the PublishObjects Collection

```
If Not rngUser Is Nothing Then
    'Set the Protection properties based
    ' on a table
    wksAllowEditRange.Protect Password:="wrox1", _
    DrawingObjects:=True, _
    Contents:=True, _
    AllowFormattingCells:=rngUser.Offset(1, 0).Value, _
    AllowFormattingColumns:=rngUser.Offset(2, 0).Value, _
    AllowFormattingRows:=rngUser.Offset(3, 0).Value, _
    AllowSorting:=rngUser.Offset(4, 0).Value, _
    UserInterfaceOnly:=True

    'Select Unlocked cells, Locked and Unlocked cells, or neither
    ' is NOT part of the Protection object
    If rngUser.Offset(5, 0).Value = True Then
        wksAllowEditRange.EnableSelection = xlUnlockedCells
    Else
        wksAllowEditRange.EnableSelection = xlNoRestrictions
    End If
Else
    'Current user is not on the table
    MsgBox "User not found on User Table. Default Options will be used.",
    vbExclamation, "Protection Settings"
    wksAllowEditRange.Protect , True, True, False, False, False, _
        False, False, False, False, False, _
        False, False, False, False, False

    wksAllowEditRange.EnableSelection = xlNoRestrictions

End If

End Sub
```

PublishObject Object and the PublishObjects Collection

The `PublishObjects` collection holds all of the things in a workbook that have been saved to a web page. Each `PublishObject` object contains items from a workbook that have been saved to a web page and may need some occasional refreshing of values on the web page side. The parent of the `PublishObjects` collection is the `Workbook` object. The `PublishObjects` collection has no properties outside the typical collection attributes listed at the beginning of this appendix.

PublishObjects Methods

Name	Returns	Description
Add	Publish Object	Method. Parameters: <code>SourceType As XlSourceType, Filename As String, [Sheet], [Source], [HtmlType], [DivID], [Title]</code> . Adds a <code>PublishObject</code> to the collection
Delete		Method. Deletes the <code>PublishObject</code> objects from the collection
Publish		Method. Publishes all the items associated with the <code>PublishObject</code> objects to a web page

PublishObject Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

PublishObject Properties

Name	Returns	Description
<code>AutoRepublish</code>	Boolean	Set/Get whether an item in the <code>PublishObjects</code> collection should be republished when a workbook is saved
<code>DivID</code>	String	Read-only. Returns the <code>id</code> used for the <code><DIV></code> tag on a web page
<code>Filename</code>	String	Set/Get the URL or path that the object will be saved to as a web page
<code>HtmlType</code>	<code>XlHtmlType</code>	Set/Get what type of web page to save (for example, <code>xlHtmlStatic</code> , <code>xlHtmlChart</code>). Pages saved as other than <code>xlHtmlStatic</code> need special ActiveX components
<code>Sheet</code>	String	Read-only. Returns the Excel sheet that will be saved as a web page
<code>Source</code>	String	Read-only. Returns the specific item, like range name, chart name, or report name, from the base type specified by the <code>SourceType</code> property
<code>SourceType</code>	<code>XlSourceType</code>	Read-only. Returns the type of source being published (for example, <code>xlSourceChart</code> , <code>xlSourcePrintArea</code> , and so on)
<code>Title</code>	String	Set/Get the web page title for the published web page

PublishObject Methods

Name	Returns	Parameters	Description
<code>Delete</code>			Deletes the <code>PublishObject</code> object
<code>Publish</code>		[<code>Create</code>]	Publishes the source items specified by the <code>PublishObject</code> as a web file. Set the <code>Create</code> parameter to <code>True</code> to overwrite existing files. <code>False</code> will append to the existing web page with the same name, if any

PublishObject Object and the PublishObjects Collection Example

```
Sub UpdatePublishedCharts()  
    Dim oPO As PublishObject  
    For Each oPO In ActiveWorkbook.PublishObjects  
        If oPO.SourceType = xlSourceChart Then  
            oPO.Publish  
        End If  
    Next  
End Sub
```

QueryTable Object and the QueryTables Collection

The `QueryTables` collection holds the collection of data tables created from an external data source. Each `QueryTable` object represents a single table in a worksheet filled with data from an external data source. The external data source can be an ODBC source, an OLE DB source, a text file, a Data Finder, a web-based query, or a DAO/ADO recordset. Possible parents of the `QueryTables` collection are the `Worksheet` and `ListObject` objects.

The `QueryTables` collection has a few properties and methods not typical of a collection. These atypical attributes are listed next. The `QueryTables` collection has no properties outside the typical collection attributes listed at the beginning of this appendix.

QueryTables Methods

Name	Returns	Description
Add	QueryTable	Method. Parameters: <code>Connection</code> , <code>Destination As Range</code> , [<code>Sql</code>]. Adds a <code>QueryTable</code> to the collection. The <code>Connection</code> parameter can specify the ODBC or OLE DB connection string, another <code>QueryTable</code> object, a DAO or ADO recordset object, a web-based query, a Data Finder string, or a text file name. The <code>Destination</code> parameter specifies the upper-left corner that the query table results will be placed. The <code>SQL</code> parameter can specify the SQL for the connection, if applicable

QueryTable Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

QueryTable Properties

Name	Returns	Description
AdjustColumn Width	Boolean	Set/Get whether the column widths automatically adjust to best fit the data every time the query table is refreshed
Background Query	Boolean	Set/Get if the query table processing is done asynchronously
CommandText	VARIANT	Set/Get the SQL command used to retrieve data (or table name if CommandType is xlCmdTable)
CommandType	xlCmdType	Set/Get the type of CommandText (for example, xlCmdSQL, xlCmdTable)
Connection	VARIANT	Set/Get the OLE DB connection string, the ODBC string, web data source, path to a text file, or path to a database
Destination	Range	Read-only. Returns the upper-left corner cell where the query table results will be placed
EditWebPage	VARIANT	Set/Get the URL for a web query
Enable Editing	Boolean	Set/Get whether the query table data can be edited or only refreshed (False)
Enable Refresh	Boolean	Set/Get whether the query table data can be refreshed
FetchesRow Overflow	Boolean	Read-only. Returns whether the last query table refresh retrieved more rows than available on the worksheet
FieldNames	Boolean	Set/Get whether the field names from the data source become column headings in the query table
FillAdjacent Formulas	Boolean	Set/Get whether formulas located to the right of the query table will update automatically when the query table data is refreshed
ListObject	ListObject	Returns a ListObject object
Maintain Connection	Boolean	Set/Get whether the connection to the data source does not close until the workbook is closed. Valid only against an OLE DB source
Name	String	Set/Get the name of the query table
Parameters	Parameters	Read-only. Returns the parameters associated with the query table
PostText	String	Set/Get the post message sent to the web server to return data from a web query

Table continued on following page

QueryTable Properties

Name	Returns	Description
Preserve Column Info	Boolean	Set/Get whether column location, sorting, and filtering does not disappear when the data query is refreshed
Preserve Formatting	Boolean	Set/Get whether common formatting associated with the first five rows of data is applied to new rows in the query table
QueryType	xlQuery Type	Read-only. Returns the type of connection associated with the query table. (For example, xlOLEDBQuery, xlDAOQuery, xlTextImport)
Recordset	Object	Read-only. Returns a recordset associated with the data source query
Refreshing	Boolean	Read-only. Returns whether an asynchronous query is currently in progress
RefreshOn FileOpen	Boolean	Set/Get whether the query table is refreshed when the workbook is opened
Refresh Period	Long	Set/Get how long (in minutes) between automatic refreshes from the data source. Set to 0 to disable
RefreshStyle	XlCell Insertion Mode	Set/Get how worksheet rows react when data rows are retrieved from the data source. Worksheet cells can be overwritten (xlOverwriteCells), cell rows can be partially inserted/deleted as necessary (xlInsert-DeleteCells), or only cell rows that need to be added can be added (xlInsertEntireRows)
ResultRange	Range	Read-only. Returns the cell range containing the results of the query table
Robust Connect	XlRobust Connect	Set/Get how the QueryTable connects to its data source
RowNumbers	Boolean	Set/Get whether a worksheet column is added to the left of the query table containing row numbers
SaveData	Boolean	Set/Get whether query table data is saved with the workbook
SavePassword	Boolean	Set/Get whether an ODBC connection password is saved with the query table
Sort	Sort	Read-only. Returns the sort criteria for the Query-Table
SourceConnection File	String	Set/Get the name of the file that was used to create the QueryTable
SourceData File	String	Read-only. Returns the name of the source data file for the QueryTable

Name	Returns	Description
TextFile Column DataTypes	Variant	Set/Get the array of column constants representing the data types for each column. Use the XlColumnDataType constants. Used only when QueryType is xlTextImport
TextFile Comma Delimiter	Boolean	Set/Get whether a comma is the delimiter for text file imports into a query table. Used only when QueryType is xlTextImport and for a delimited text file
TextFile Consecutive Delimiter	Boolean	Set/Get whether consecutive delimiters (for example, “,”) are treated as a single delimiter. Used only when QueryType is xlTextImport
TextFile Decimal Separator	String	Set/Get the type of delimiter to use to define a decimal point. Used only when QueryType is xlTextImport
TextFile Fixed ColumnWidths	Variant	Set/Get the array of widths that correspond to the columns. Used only when QueryType is xlTextImport and for a fixed-width text file
TextFile Other Delimiter	String	Set/Get the character that will be used to delimit columns from a text file. Used only when QueryType is xlTextImport and for a delimited text file
TextFile ParseType	XlText ParsingType	Set/Get the type of text file that is being imported: xlDelimited or xlFixedWidth. Used only when QueryType is xlTextImport
TextFilePlatform	XlPlatform	Set/Get which code pages to use when importing a text file (for example, xlMSDOS, xlWindows). Used only when QueryType is xlTextImport
TextFile PromptOn Refresh	Boolean	Set/Get whether the user is prompted for the text file to use to import into a query table every time the data is refreshed. Used only when QueryType is xlTextImport. The prompt does not appear on the initial refresh of data
TextFile Semicolon Delimiter	Boolean	Set/Get whether the semicolon is the text file delimiter for importing text files. Used only when QueryType is xlTextImport and the file is a delimited text file
TextFile Space Delimiter	Boolean	Set/Get whether the space character is the text file delimiter for importing text files. Used only when QueryType is xlTextImport and the file is a delimited text file
TextFile StartRow	Long	Set/Get which row number to start importing from a text file. Used only when QueryType is xlTextImport

Table continued on following page

QueryTable Properties

Name	Returns	Description
TextFileTab Delimiter	Boolean	Set/Get whether the tab character is the text file delimiter for importing text files. Used only when QueryType is xlTextImport and the file is a delimited text file
TextFileText Qualifier	XlText Qualifier	Set/Get which character will be used to define string data when importing data from a text file. Used only when QueryType is xlTextImport
TextFile Thousands Separator	String	Set/Get which character is used as the thousands separator in numbers when importing from a text file (for example, ",")
TextFile TrailingMinus Numbers	Boolean	Set/Get whether to treat numbers imported as text that begin with a "-" symbol as negative numbers
TextFile VisualLayout	XlText Visual LayoutType	Returns 1 or 2 depending on the visual layout of the file. A value of 1 represents left-to-right, while 2 represents right-to-left
Web Consecutive DelimitersAsOne	Boolean	Set/Get whether consecutive delimiters are treated as a single delimiter when importing data from a web page. Used only when QueryType is xlWebQuery
WebDisable Date Recognition	Boolean	Set/Get whether data that looks like dates is parsed as text when importing web page data. Used only when QueryType is xlWebQuery
WebDisable Redirections	Boolean	Set/Get whether web query redirections are disabled for the QueryTable object
WebFormatting	xlWeb Formatting	Set/Get whether to keep any of the formatting when importing a web page (for example, xlAll, xlNone). Used only when QueryType is xlWebQuery
WebPre Formatted TextToColumns	Boolean	Set/Get whether HTML data with the <PRE> tag is parsed into columns when importing web pages. Used only when QueryType is xlWebQuery
WebSelection Type	xlWeb Selection Type	Set/Get which data from a web page is imported, either all tables (xlAllTables), the entire page (xlEntirePage), or specified tables (xlSpecifiedTables). Used only when QueryType is xlWebQuery
WebSingleBlock TextImport	Boolean	Set/Get whether all the web page data with the <PRE> tags is imported all at once. Used only when QueryType is xlWebQuery

Name	Returns	Description
WebTables	String	Set/Get a comma-delimited list of all the table names that will be imported from a web page. Used only when QueryType is xlWebQuery and WebSelectionType is xlSpecifiedTables
Workbook Connection	Workbook Connection	Read-only. Returns the connection that the QueryTable uses

QueryTable Methods

Name	Returns	Parameters	Description
CancelRefresh			Cancels an asynchronously running query table refresh
Delete			Deletes the query table
Refresh	Boolean	[Background Query]	Refreshes the data in the query table with the latest copy of the external data. Sets the BackgroundQuery parameter to True to get the data to refresh asynchronously
ResetTimer			Resets the time for the automatic refresh set by RefreshPeriod property
SaveAsODC		ODCFileName As String, [Description], [Keywords]	Saves the PivotCache source as an Office Data Connection file. ODCFileName is the location of the source file. Description is the description that will be saved in the file. Keywords is a list of space-separated keywords that can be used to search for this file

QueryTable Events

Name	Parameters	Description
AfterRefresh	Success As Boolean	Triggered after a query is completed or cancelled
BeforeRefresh	Cancel As Boolean	Triggered just before a refresh of the query table

QueryTable Object and the QueryTables Collection Example

```
Sub UpdateAllWebQueries()  
    Dim oQT As QueryTable  
    For Each oQT In ActiveSheet.QueryTables  
        If oQT.QueryType = xlWebQuery Then  
            oQT.BackgroundQuery = False  
            oQT.Refresh  
        End If  
    Next  
End Sub
```

Range Object and the Ranges Collection Object

The `Range` object is one of the more versatile objects in Excel. A range can be a single cell, a column, a row, a contiguous block of cells, or a non-contiguous range of cells. The main parent of a `Range` object is the `Worksheet` object. However, most of the objects in the Excel object model use the `Range` object. The `Range` property of the `Worksheet` object can be used to choose a certain range of cells using the `Cell11` and `Cell12` parameters. New to Excel 2007, the `Ranges` object holds a collection of `Range` objects. The `Ranges` collection has no properties outside the typical collection attributes listed at the beginning of this appendix.

Range Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Range Properties

Name	Returns	Description
<code>AddIndent</code>	Variant	Set/Get whether text in a cell is automatically indented if the text alignment in a cell is set to equally distribute
<code>Address</code>	String	Read-only. Parameters: <code>RowAbsolute</code> , <code>ColumnAbsolute</code> , <code>ReferenceStyle As XlReferenceStyle</code> , <code>[External]</code> , <code>[RelativeTo]</code> . Returns the address of the current range as a string in the macro's language. The type of address (reference, absolute, A1 reference style, R1C1 reference style) is specified by the parameters
<code>AddressLocal</code>	String	Read-only. Parameters: <code>RowAbsolute</code> , <code>ColumnAbsolute</code> , <code>ReferenceStyle As XlReferenceStyle</code> , <code>[External]</code> , <code>[RelativeTo]</code> . Returns the address of the current range as a string in the user's language. The type of address (reference, absolute, A1 reference style, R1C1 reference style) is specified by the parameters
<code>AllowEdit</code>	Boolean	Read-only. Returns <code>True</code> if the range can be edited on a protected worksheet
<code>Areas</code>	Areas	Read-only. Returns an object containing the different non-contiguous ranges in the current range

Name	Returns	Description
Borders	Borders	Read-only. Returns all the individual borders around the range. Each border side can be accessed individually in the collection
Cells	Range	Read-only. Returns the cells in the current range. The Cells property will return the same range as the current range
Characters	Characters	Read-only. Parameters: [Start], [Length]. Returns all the characters in the current range, if applicable
Column	Long	Read-only. Returns the column number of the first column in the range
Columns	Range	Read-only. Returns a range of the columns in the current range
ColumnWidth	Variant	Set/Get the column width of all the columns in the range. Returns Null if the columns in the range have different widths
Comment	Comment	Read-only. Returns an object representing the range comment, if any
Count	Long	Read-only. Returns the number of cells in the current range
CountLarge	Variant	Read-only. Counts the largest value in a given range of values
CurrentArray	Range	Read-only. Returns a Range object that represents the array associated with the particular cell range, if the cell is part of an array
CurrentRegion	Range	Read-only. Returns the current region that contains the Range object. A region is defined as an area that is surrounded by blank cells
Dependents	Range	Read-only. Returns the dependents of a cell on the same sheet as the range
Direct Dependents	Range	Read-only. Returns the direct dependents of a cell on the same sheet as the range
Direct Precedents	Range	Read-only. Returns the direct precedents of a cell on the same sheet as the range
End	Range	Read-only. Parameters: Direction As XlDirection. Returns the cell at end of the region containing the Range object. Which end of the region is specified by the Direction parameter
EntireColumn	Range	Read-only. Returns the full worksheet column(s) occupied by the current range

Table continued on following page

Range Properties

Name	Returns	Description
EntireRow	Range	Read-only. Returns the full worksheet row(s) occupied by the current range
Errors	Errors	Read-only. Returns the <code>Errors</code> collection associated with the <code>Range</code> object
Font	Font	Read-only. Returns an object containing <code>Font</code> options for the text in the range
FormatConditions	FormatConditions	Read-only. Returns an object holding conditional formatting options for the current range
Formula	Variant	Set/Get the formula of the cells in the range
FormulaArray	Variant	Set/Get the array formula of the cells in the range
FormulaHidden	Variant	Set/Get whether the formula will be hidden if the workbook/worksheet is protected
FormulaLocal	Variant	Set/Get the formula of the range in the language of the user using the A1 style references
FormulaR1C1	Variant	Set/Get the formula of the range in the language of the macro using the R1C1 style references
FormulaR1C1Local	Variant	Set/Get the formula of the range in the language of the user using the R1C1 style references
HasArray	Variant	Read-only. Returns whether a single cell range is part of an array formula
HasFormula	Variant	Read-only. Returns whether all the cells in the range contain formulas (<code>True</code>). If only some of the cells contain formulas, then <code>Null</code> is returned
Height	Variant	Read-only. Returns the height of the range
Hidden	Variant	Set/Get whether the cells in the range are hidden. Only works if the range contains whole columns or rows
HorizontalAlignment	Variant	Set/Get how the cells in the range are horizontally aligned. Use the <code>XLHAlign</code> constants
Hyperlinks	Hyperlinks	Read-only. Returns the collection of hyperlinks in the range
ID	String	Set/Get the <code>ID</code> used for the range if the worksheet is saved as a web page
IndentLevel	Variant	Set/Get the indent level for the range
Interior	Interior	Read-only. Returns an object containing options to format the inside area of the range, if applicable (for example, interior color)

Name	Returns	Description
Left	Variant	Read-only. Returns the distance from the left edge of the leftmost column in the range to the left edge of ColumnA
ListHeader Rows	Long	Read-only. Returns the number of header rows in the range
ListObject	ListObject	Returns a ListObject object
LocationIn Table	XLLocation InTable	Read-only. Returns the location of the upper-left corner of the range
Locked	Variant	Set/Get whether cells in the range can be modified if the sheet is protected. Returns Null if only some of the cells in the range are locked
MDX	String	Returns the MDX (Multidimensional Expression) that can be sent to an OLAP provider
MergeArea	Range	Read-only. Returns a range containing the merged range of the current cell range
MergeCells	Variant	Set/Get whether the current range contains merged cells
Name	Variant	Set/Get the Name object that contains the name for the range
Next	Range	Read-only. Returns the next range in the sheet
NumberFormat	Variant	Set/Get the number format associated with the cells in the range. Null if all the cells don't have the same format
NumberFormat Local	Variant	Set/Get the number format associated with the cells in the range, in the language of the end user. Null if all the cells don't have the same format
Offset	Range	Read-only. Parameters: [RowOffset], [ColumnOffset]. Returns the cell as a Range object that is the offset from the current cell as specified by the parameters. A positive RowOffset offsets the row downward. A negative RowOffset offsets the row upward. A positive ColumnOffset offsets the column to the right, and a negative ColumnOffset offsets the column to the left
Orientation	Variant	Set/Get the text orientation for the cell text. A value from -90 to 90 degrees can be specified, or use an XLOrientation constant
OutlineLevel	Variant	Set/Get the outline level for the row or column range
PageBreak	Long	Set/Get how page breaks are set in the range. Use the XLPageBreak constants
Phonetic	Phonetic	Read-only. Returns the Phonetic object associated with the cell range

Table continued on following page

Range Properties

Name	Returns	Description
Phonetics	Phonetics	Read-only. Returns the Phonetic objects in the range
PivotCell	PivotCell	Read-only. Returns a PivotCell object that represents a cell in a PivotTable report
PivotField	PivotField	Read-only. Returns the PivotTable field associated with the upper-left corner of the current range
PivotItem	PivotItem	Read-only. Returns the PivotTable item associated with the upper-left corner of the current range
PivotTable	PivotTable	Read-only. Returns the PivotTable report associated with the upper-left corner of the current range
Precedents	Range	Read-only. Returns the range of precedents of the current cell range on the same sheet as the range
Prefix Character	Variant	Read-only. Returns the character used to define the type of data in the cell range. For example, "" for a text label
Previous	Range	Read-only. Returns the previous range in the sheet
QueryTable	QueryTable	Read-only. Returns the query table associated with the upper-left corner of the current range
Range	Range	Read-only. Parameters: Cell1, [Cell2]. Returns a Range object as defined by the Cell1 and optionally Cell2 parameters. The cell references used in the parameters are relative to the range. For example, Range.Range ("A1") would return the first column in the parent range, but not necessarily the first column in the worksheet
ReadingOrder	Long	Set/Get whether the text is from right-to-left (xlRTL), left-to-right (xlLTR), or context-sensitive (xlContext)
Resize	Range	Read-only. Parameters: [RowSize], [ColumnSize]. Returns a new resized range as specified by the RowSize and ColumnSize parameters
Row	Long	Read-only. Returns the row number of the first row in the range
RowHeight	Variant	Set/Get the height of the rows in the range. Returns Null if the rows in the range have different row heights
Rows	Range	Read-only. Returns a Range object containing the rows of the current range
ServerActions	Actions	Read-only. Returns the actions that can be performed on the SharePoint server for a Range object
ShowDetail	Variant	Set/Get if all the outline levels in the range are expanded. Applicable only if a summary column or row is the range

Name	Returns	Description
ShrinkToFit	Variant	Set/Get whether the cell text will automatically shrink to fit the column width. Returns <code>Null</code> if the rows in the range have different <code>ShrinkToFit</code> properties
SmartTags	SmartTags	Read-only. Returns a <code>SmartTags</code> object representing the identifier for the specified cell
SoundNote	SoundNote	Property is kept for backwards compatibility only
Style	Variant	Set/Get the <code>Style</code> object associated with the range
Summary	Variant	Read-only. Returns whether the range is an outline summary row or column
Text	Variant	Read-only. Returns the text associated with a range cell
Top	Variant	Read-only. Returns the distance from the top edge of the topmost row in the range to the top edge of <code>Row1</code>
UseStandardHeight	Variant	Set/Get whether the row height is the standard height of the sheet. Returns <code>Null</code> if the rows in the range contain different heights
UseStandardWidth	Variant	Set/Get whether the column width is the standard width of the sheet. Returns <code>Null</code> if the columns in the range contain different widths
Validation	Validation	Read-only. Returns the data validation for the current range
Value	Variant	Parameters: [<code>RangeValueDataType</code>]. Set/Get the value of a cell or an array of cells depending on the contents of the <code>Range</code> object
Value2	Variant	Set/Get the value of a cell or an array of cells depending on the contents of the <code>Range</code> object. No <code>Currency</code> or <code>Date</code> types are returned by <code>Value2</code>
VerticalAlignment	Variant	Set/Get how the cells in the range are vertically aligned. Use the <code>XLVAlign</code> constants
Width	Variant	Read-only. Returns the height of the range
Worksheet	Worksheet	Read-only. Returns the worksheet that has the <code>Range</code> object
WrapText	Variant	Set/Get whether cell text wraps in the cell. Returns <code>Null</code> if the cells in the range contain different text wrap properties
XPath	Xpath	Represents the XPath element of the object at the current range

Range Methods

Name	Returns	Parameters	Description
Activate	Variant		Selects the range cells
AddComment	Comment	[Text]	Adds the text specified by the parameter to the cell specified in the range. Must be a single cell range
Advanced Filter	Variant	[Action] As XlFilterAction, [CriteriaRange], [CopyToRange], [Unique]	Copies or filters the data in the current range. The Action parameter specifies whether a copy or filter is to take place. CriteriaRange optionally specifies the range containing the criteria. CopyToRange specifies the range that the filtered data will be copied to (if Action is xlFilterCopy)
ApplyNames	Variant	[Names], [IgnoreRelativeAbsolute], [UseRowColumnNames], [OmitColumn], [OmitRow], [Order] As XlApplyNamesOrder, [AppendLast]	Applies defined names to the formulas in a range. For example, if a cell contained =\$A\$1*100 and \$A\$1 was given the name "TopLeft", you could apply the "TopLeft" name to the range, resulting in the formula changing to =TopLeft*100. Note that there is no UnApplyNames method
ApplyOutline Styles	Variant		Applies the outline styles to the range
AutoComplete	String	String As String	Returns and tries to AutoComplete the word specified in the String parameter. Returns the complete word, if found. Returns an empty string if no word or more than one word is found
AutoFill	Variant	[Destination] As Range, [Type]	Uses the current range as the source to figure out how to AutoFill the range specified by the Destination parameter. The Type parameter can also be used to specify the type of fill to use (for example, xlFillCopy, xlFillDays)

Name	Returns	Parameters	Description
AutoFilter	Variant	[Field], [Criteria1], [Operator], [Criteria2], [Visible Drop Down]	Creates an auto-filter on the data in the range. See the <code>AutoFilter</code> object for details on the parameters
AutoFit	Variant		Changes the column widths in the range to best fit the data in the cells. The range must contain full rows or columns
AutoOutline	Variant		Creates an outline for the range
BorderAround	Variant	[LineStyle], [Weight] As XlBorder Weight, [ColorIndex], [Color]	Creates a border around the range with the associated line style (LineStyle), thickness (Weight), and color (ColorIndex)
Calculate	Variant		Calculates all the formulas in the range
CheckSpelling	Variant	[Custom Dictionary], [Ignore Uppercase], [Always Suggest], [SpellLang]	Checks the spelling of the text in the range. A custom dictionary can be specified (CustomDictionary), all uppercase words can be ignored (IgnoreUppercase), and Excel can be set to display a list of suggestions (AlwaysSuggest)
Clear	Variant		Clears the text in the cells of the range
ClearComments			Clears all the comments in the range cells
ClearContents	Variant		Clears the formulas and values in a range
ClearFormats	Variant		Clears the formatting in a range
ClearNotes	Variant		Clears comments from the cells in the range
ClearOutline	Variant		Clears the outline used in the current range
Column Differences	Range	[Comparison]	Returns the range of cells that are specific to the cell specified by the Comparison parameter

Table continued on following page

Range Methods

Name	Returns	Parameters	Description
Consolidate	Variant	[Sources], [Function], [TopRow], [Left Column], [Create Links]	Consolidates the source array of range reference strings in the Sources parameter and returns the results to the current range. The Function parameter can be used to set the consolidation function. Use the XLConsolidationFunction constants
Copy	Variant	[Destination]	Copies the current range to the range specified by the parameter, or to the clipboard if no destination is specified
CopyFromRecordset	Long	[Data] As Recordset, [MaxRows], [MaxColumns]	Copies the records from the ADO or DAO recordset specified by the Data parameter into the current range. The recordset can't contain OLE objects
CopyPicture	Variant	[Appearance], [Format]	Copies the range into the clipboard as a picture. The Appearance parameter can be used to specify whether the picture is copied as it looks on the screen or when printed. The Format parameter can specify the type of picture that will be put into the clipboard
CreateNames	Variant	[Top], [Left], [Bottom], [Right]	Creates a named range for the items in the current range. Set Top to True to make the first row hold the names for the ranges below. Set Bottom to True to use the bottom row as the names. Set Left or Right to True to make the left or right column contain the Names, respectively
Cut	Variant	[Destination]	Cuts the current range to the range specified by the parameter, or to the clipboard if no destination is specified
DataSeries	Variant	[Rowcol], [Type], [Date], [Step], [Stop], [Trend]	Creates a data series at the current range location

Name	Returns	Parameters	Description
Delete	Variant	[Shift]	Deletes the cells in the current range and optionally shifts the cells in the direction specified by the Shift parameter. Use the xlDeleteShift Direction constants for the Shift parameter
DialogBox	Variant		Displays a dialog box defined by an Excel 4.0 macro sheet
Dirty			Selects a range to be recalculated when the next recalculation occurs
EditionOptions	Variant	[Type] As XlEditionType, [Option] As XlEditionOptionsOption, [Name], [Reference], Appearance, [ChartSize], [Format]	Used on the Macintosh. EditionOptions set how the range should act when being used as the source (publisher) or target (subscriber) of the link. Editions are basically the same as Windows' DDE links
ExportAsFixedFormat		[Type] As XlFixedFormatType, [FileName], [Quality], [IncludeDocProperties], [IgnorePrintAreas] [From], [To], [OpenAfterPublish]	Exports a file to a format specified by using the xlFixedFormatType constants
FillDown	Variant		Copies the contents and formatting from the top row into the rest of the rows in the range
FillLeft	Variant		Copies the contents and formatting from the rightmost column into the rest of the columns in the range
FillRight	Variant		Copies the contents and formatting from the leftmost column into the rest of the columns in the range
FillUp	Variant		Copies the contents and formatting from the bottom row into the rest of the rows in the range

Table continued on following page

Range Methods

Name	Returns	Parameters	Description
Find	Range	[What] As Variant, [After], [LookIn], [LookAt], [SearchOrder], [Search Direction] As XlSearch Direction, [MatchCase], [MatchByte], [Search Format]	Looks through the current range for the text of data type specified by the What parameter. Use a single cell range in the After parameter to choose the starting position of the search. Use the LookIn parameter to decide where the search is going to take place
FindNext	Range	[After]	Finds the next instance of the search criteria defined with the Find method
FindPrevious	Range	[After]	Finds the previous instance of the search criteria defined with the Find method
Function Wizard	Variant		Displays the Function Wizard for the upper-left cell of the current range
GoalSeek	Boolean	[Goal], [ChangingCell] As Range	Returns True if the value specified by the Goal parameter is returned when changing the ChangingCell cell range
Group	Variant	[Start], [End], [By], [Periods]	Either demotes the outline in the range or groups the discontinuous ranges in the current Range object
Insert	Variant	[Shift], [CopyOrigin]	Inserts the equivalent rows or columns in the range into the range's worksheet
InsertIndent		[InsertAmount] As Long	Indents the range by the amount specified by the InsertAmount parameter
Justify	Variant		Evenly distributes the text in the cells from the current range
ListNames	Variant		Pastes the names of all the named ranges in the current range starting at the top-left cell in the range

Name	Returns	Parameters	Description
Merge		[Across]	Merges the cells in the range. Set the <code>Across</code> parameter to <code>True</code> to merge each row as a separate cell
NavigateArrow	Variant	[Toward Precedent], [ArrowNumber], [LinkNumber]	Moves through the tracer arrows in a workbook from the current range, returning the range of cells that make up the tracer arrow destination. Tracer arrows must be turned on. Use the <code>ShowDependents</code> and <code>ShowPrecedents</code> methods
NoteText	String	[Text], [Start], [Length]	Set/Get the cell notes associated with the cell in the current range
Parse	Variant	[ParseLine], [Destination]	Parses the string specified by the <code>ParseLine</code> parameter and returns it to the current range parsed out by column. Optionally, can specify the destination range with the <code>Destination</code> parameter. The <code>ParseLine</code> string should be in the "[ColumnA][ColumnB]" format
PasteSpecial	Variant	[Paste], [Operation], [SkipBlanks], [Transpose]	Pastes the range from the clipboard into the current range. Use the <code>Paste</code> parameter to choose what to paste (for example, formulas, values). Use the <code>Operation</code> parameter to specify what to do with the paste. Set <code>SkipBlanks</code> to <code>True</code> to keep blank cells in the clipboard's range from being pasted. Set <code>Transpose</code> to <code>True</code> to transpose columns with rows
PrintOut	Variant	[From], [To], [Copies], [Preview], [ActivePrinter], [PrintToFile], [Collate], [PrToFile Name]	Prints the charts in the collection. The printer, number of copies, collation, and whether a print preview is desired can be specified with the parameters. Also, the sheets can be printed to a file using the <code>PrintToFile</code> and <code>PrToFileName</code> parameters. The <code>From</code> and <code>To</code> parameters can be used to specify the range of printed pages

Table continued on following page

Range Methods

Name	Returns	Parameters	Description
PrintPreview	Variant	[EnableChanges]	Displays the current range in a print preview. Set the EnableChanges parameter to False to disable the Margins and Setup buttons, hence not allowing the viewer to modify the page setup
Remove Duplicates		[Columns], [Header]	Removes duplicate values from a range of values
Remove Subtotal	Variant		Removes subtotals from the list in the current range
Replace	Boolean	[What] As Variant, [Replacement] As Variant, [LookAt], [SearchOrder], [MatchCase], [MatchByte], [SearchFormat], [ReplaceFormat]	Finds the text specified by the What parameter in the range. Replaces the found text with the Replacement parameter. Use the SearchOrder parameters with the XLSearchOrder constants to choose whether the search occurs by rows or by columns
Row Differences	Range	[Comparison]	Returns the range of cells that are specific to the cell specified by the Comparison parameter
Run	Variant	[Arg1], [Arg2], ([Arg30]	Runs the Excel 4.0 macro specified by the current range. The potential arguments to the macro can be specified with the Argx parameters
Select	Variant		Selects the cells in the range
SetPhonetic			Creates a Phonetic object for each cell in the range
Show	Variant		Scrolls the Excel window to display the current range. This only works if the range is a single cell

Name	Returns	Parameters	Description
Show Dependents	Variant	[Remove]	Displays the dependents for the current single cell range using tracer arrows
ShowErrors	Variant		Displays the source of the errors for the current range using tracer arrows
ShowPrecedents	Variant	[Remove]	Displays the precedents for the current single cell range using tracer arrows
Sort	Variant	[Key1], [Order1 As XlSortOrder], [Key2], [Type], [Order2 As XlSortOrder], [Key3], [Order3 As XlSortOrder], [Header As XlYesNoGuess], [OrderCustom], [MatchCase], [Orientation As XlSortOrientation], [SortMethod As XlSortMethod], [DataOption1 As XlSortDataOption], [DataOption2 As XlSortDataOption], [DataOption3 As XlSortDataOption]	Sorts the cells in the range. If the range contains only one cell, then the active region is searched. Use the Key1, Key2, and Key3 parameters to set which columns will be the sort columns. Use the Order1, Order2, and Order3 parameters to set the sort order. Use the Header parameter to set whether the first row contains headers. Set the MatchCase parameter to True to sort data and to treat uppercase and lowercase characters differently. Use the Orientation parameter to choose whether rows are sorted or columns are sorted. Finally, the SortMethod parameter is used to set the sort method for other languages (for example, xlStroke or xlPinYin). Use the SortSpecial method for sorting in East Asian languages

Table continued on following page

Range Methods

Name	Returns	Parameters	Description
SortSpecial	Variant	[SortMethod As XlSortMethod], [Key1], [Order1 As XlSortOrder], [Type], [Key2], [Order2 As XlSortOrder], [Key3], [Order3 As XlSortOrder], [Header As XlYesNoGuess], [OrderCustom], [MatchCase], [Orientation As XlSortOrientation], [DataOption1 As XlSortDataOption], [DataOption2 As XlSortDataOption], [DataOption3 As XlSortDataOption]	Sorts the data in the range using East Asian sorting methods. The parameters are the same as the Sort method
Speak		[Speak Direction], [Speak Formulas]	Causes the cells of the range to be spoken in row order or column order
SpecialCells	Range	Type As XlCellType, [Value]	Returns the cells in the current range that contain some special attribute as defined by the Type parameter. For example, if Type is xlCellTypeBlanks, then a Range object containing all of the empty cells is returned
SubscribeTo	Variant	[Edition] As String, [Format]	Only valid on the Macintosh. Defines the source of a link that the current range will contain

Name	Returns	Parameters	Description
Subtotal	Variant	[GroupBy] As Long, [Function] As Xl Consolidation Function, [TotalList], [Replace], [PageBreaks], [SummaryBelowData]	Creates a subtotal for the range. If the range is a single cell, then a subtotal is created for the current region. The GroupBy parameter specifies the field to group (for subtotaling). The Function parameter describes how the fields will be grouped. The TotalList parameter uses an array of field offsets that describe the fields that will be subtotaled. Set the Replace parameter to True to replace existing subtotals. Set PageBreaks to True for page breaks to be added after each group. Use the Summary-BelowData parameter to choose where the summary row will be added
Table	Variant	[RowInput], [ColumnInput]	Creates a new data table at the current range
TextToColumns	Variant	[Destination], [DataType As XlTextParsingType], [TextQualifier As XlTextQualifier], [ConsecutiveDelimiter], [Tab], [Semicolon], [Comma], [Space], [Other], [OtherChar], [FieldInfo], [DecimalSeparator], [ThousandsSeparator], [TrailingMinusNumbers]	Parses text in cells into several columns. The Destination specifies the range that the parsed text will go into. The DataType parameter can be used to choose whether the text is delimited or fixed width. The TextQualifier parameter can specify which character denotes string data when parsing. Set the ConsecutiveDelimiter to True for Excel to treat consecutive delimiters as one. Set the Tab, Semicolon, Comma, or Space parameter to True to use the associated character as the delimiter. Set the Other parameter to True and specify an OtherChar to use another character as the delimiter. FieldInfo takes a two-dimensional array containing more parsing information. The DecimalSeparator and ThousandsSeparator can specify how numbers are treated when parsing

Table continued on following page

RecentFile Object and the RecentFiles Collection

Name	Returns	Parameters	Description
Ungroup	Variant		Either promotes the outline in the range or ungroups the range in a PivotTable report
UnMerge			Splits up a merged cell into single cells

RecentFile Object and the RecentFiles Collection

The `RecentFiles` collection holds the list of recently modified files, equivalent to the files listed under the Office icon in the left-hand corner of the application. Each `RecentFile` object represents one of the recently modified files.

`RecentFiles` has a few attributes besides the typical collection ones. The `Maximum` property can be used to set or return the maximum number of files that Excel will “remember” modifying. The value can range from 0 to 9. The `Add` method is used to add a file (with the `Name` parameter) to the collection.

RecentFile Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

RecentFile Properties

Name	Returns	Description
Index	Long	Read-only. Returns the spot in the collection where the current object is located
Name	String	Read-only. Returns the name of the recently modified file
Path	String	Read-only. Returns the file path of the recently modified file

RecentFile Methods

Name	Returns	Description
Delete		Deletes the object from the collection
Open	Workbook	Opens up the recent file and returns the opened workbook

RecentFile Object and the RecentFiles Collection Example

```
Sub CheckRecentFiles()  
    Dim oRF As RecentFile  
    'Remove any recent files that refer to the floppy drive  
    For Each oRF In Application.RecentFiles  
        If Left(oRF.Path, 2) = "A:" Then  
            oRF.Delete  
        End If  
    Next oRF  
End Sub
```

```

End If
Next
End Sub

```

RectangularGradient Object

The `LinearGradient` object transitions through a series of colors in a linear manner along a specific angle. Attempting to access a `Gradient` property of an `Interior` object that does not have an existing gradient fill will result in a run-time error. Be aware of the `Interior.Pattern` property before accessing the `Gradient` property.

RectangleGradient Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

RectangleGradient Properties

Name	Returns	Description
<code>ColorStops</code>	<code>ColorStops</code>	Read-only. Returns the <code>ColorStops</code> for the <code>LinearGradient</code> object
<code>RectangleBottom</code>	<code>Double</code>	Set/Get the point or vector that the gradient fill converges to
<code>RectangleLeft</code>	<code>Double</code>	Set/Get the point or vector that the gradient fill converges to
<code>RectangleRight</code>	<code>Double</code>	Set/Get the point or vector that the gradient fill converges to
<code>RectangleTop</code>	<code>Double</code>	Set/Get the point or vector that the gradient fill converges to

RoutingSlip Object

The `RoutingSlip` object represents the properties and methods of the routing slip of an Excel document. The parent object of the `RoutingSlip` object is the `Workbook` object. The `HasRoutingSlip` property of the `Workbook` object has to be set to `True` before the `RoutingSlip` object can be manipulated.

RoutingSlip Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

RoutingSlip Properties

RoutingSlip Properties

Name	Returns	Description
Delivery	XlRoutingSlip Delivery	Set/Get how the delivery process will proceed
Message	Variant	Set/Get the body text of the routing slip message
Recipients	Variant	Parameters: [Index]. Returns the list of recipient names to send the parent workbook to
ReturnWhen Done	Boolean	Set/Get whether the message is returned to the original sender
Status	XlRoutingSlipStatus	Read-only. Returns the current status of the routing slip
Subject	Variant	Set/Get the subject text for the routing slip message
TrackStatus	Boolean	Set/Get whether the message is sent to the original sender each time the message is forwarded

RoutingSlip Methods

Name	Returns	Description
Reset	Variant	Resets the routing slip

RTD Object

Represents a Real-Time Data object, like one referenced using the `IrtDServer` object. As of this writing, there was very little documentation.

RTD Properties

Name	Returns	Description
Throttle Interval	Long	Set/Get the time interval between updates

RTD Methods

Name	Description
RefreshData	Requests an update of RTD from the RTD server
Restart Servers	Reconnects to servers for RTD

Scenario Object and the Scenarios Collection

The `Scenarios` collection contains the list of all the scenarios associated with a worksheet. Each `Scenario` object represents a single scenario in a worksheet. A scenario holds the list of saved cell values that can later be substituted into the worksheet. The parent of the `Scenarios` collection is the `Worksheet` object. The `Scenarios` collection has no properties outside the typical collection attributes listed at the beginning of this appendix.

Scenarios Methods

Name	Returns	Description
Add	Scenario	Method. Parameters: Name As String, ChangingCells, [Values], [Comment], [Locked], [Hidden]. Adds a scenario to the collection. The Name parameter specifies the name of the scenario. See the Scenario object for a description of the parameters
CreateSummary	Variant	Method. Parameters: ReportType As XlSummaryReportType, [ResultCells]. Creates a worksheet containing a summary of all the scenarios of the parent worksheet. The ReportType parameter can specify the report type. The ResultCells parameter can be a range of cells containing the formulas related to the changing cells
Merge	Variant	Method. Parameters: Source. Merges the scenarios in the Source parameter into the current worksheet

Scenario Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Scenario Properties

Name	Returns	Description
ChangingCells	Range	Read-only. Returns the range of cells in the worksheet that will have values plugged in for the specific scenario
Comment	String	Set/Get the scenario comment
Hidden	Boolean	Set/Get whether the scenario is hidden
Index	Long	Read-only. Returns the spot in the collection where the current Scenario object is located
Locked	Boolean	Set/Get whether the scenario cannot be modified when the worksheet is protected
Name	String	Set/Get the name of the scenario
Values	Variant	Read-only. Parameters: [Index]. Returns an array of the values to plug into the changing cells for this particular scenario

Scenario Methods

Name	Returns	Parameters	Description
Change Scenario	Variant	Changing Cells, [Values]	Changes which set of cells in the worksheet are able to change for the scenario. Optionally, can choose new values for the scenario
Delete	Variant		Deletes the Scenario object from the collection
Show	Variant		Shows the scenario results by putting the scenario values into the worksheet

Scenario Object and the Scenarios Collection Example

```
Sub GetBestScenario()  
    Dim oScen As Scenario  
    Dim oBestScen As Scenario  
    Dim dBestSoFar As Double  
    'Loop through the scenarios in the sheet  
    For Each oScen In ActiveSheet.Scenarios  
        'Show the scenario  
        oScen.Show  
  
        'Is it better?  
        If Range("Result").Value > dBestSoFar Then  
            dBestSoFar = Range("Result").Value  
            'Yes - remember it  
            Set oBestScen = oScen  
        End If  
    Next  
    'Show the best scenario  
    oBestScen.Show  
    MsgBox "The best scenario is " & oBestScen.Name  
End Sub
```

Series Object and the SeriesCollection Collection

The `SeriesCollection` collection holds the collection of series associated with a chart group. Each `Series` object contains a collection of points associated with a chart group in a chart. For example, a simple line chart contains a series (`Series`) of points brought in from the originating data. Because some charts can have many series plotted on the same chart, the `SeriesCollection` is used to hold that information. The parent of the `SeriesCollection` is the `ChartGroup`. The `SeriesCollection` object has no properties outside the typical collection attributes listed at the beginning of this appendix.

SeriesCollection Methods

Name	Returns	Description
Add	Series	Method. Parameters: Source, Rowcol, [SeriesLabels], [CategoryLabels], [Replace]. Adds a Series to the collection. The Source parameter specifies either a range or an array of data points describing the new series (and all the points in it). The Rowcol parameter sets whether the row or the column of the Source contains a series of points. Set SeriesLabels or CategoryLabels to True to make the first row or column of the Source contain the labels for the series and category, respectively
Extend	Variant	Method. Parameters: Source, [Rowcol], [CategoryLabels]. Adds the points specified by the range or array of data points in the Source parameter to the SeriesCollection. See the Add method for details on the other parameters
Paste	Variant	Method. Parameters: Rowcol, [SeriesLabels], [CategoryLabels], [Replace], [NewSeries]. Pastes the data from the Clipboard into the SeriesCollection as a new Series. See the Add method for details on the other parameters
NewSeries	Series	Method. Creates a new series and returns the newly created series

Series Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

Series Properties

Name	Returns	Description
ApplyPictToEnd	Boolean	Set/Get whether pictures are added to the end of the points in the series
ApplyPictToFront	Boolean	Set/Get whether pictures are added to the front of the points in the series
ApplyPictToSides	Boolean	Set/Get whether pictures are added to the sides of the points in the series
AxisGroup	XlAxisGroup	Set/Get the axis type being used by the series (primary or secondary)
BarShape	XlBarShape	Set/Get the type of shape to use in a 3D bar or column chart (for example, xlBox)

Table continued on following page

Series Properties

Name	Returns	Description
BubbleSizes	Variant	Set/Get the cell references (A1 reference style) that contain data relating to how big the bubble should be for bubble charts
ChartType	XlChartType	Set/Get the type of chart to use for the series
ErrorBars	ErrorBars	Read-only. Returns the error bars in a series. Use with HasErrorBars
Explosion	Long	Set/Get how far out the slices (points) of a pie or doughnut chart will explode out. 0 for no explosion
Format	ChartFormat	Returns the ChartFormat object, which controls the line, fill, and effect formatting for the chart area
Formula	String	Set/Get the type of formula label to use for the series
FormulaLocal	String	Set/Get the formula of the series in the language of the user using the A1 style references
FormulaR1C1	String	Set/Get the formula of the series in the language of the macro using the R1C1 style references
FormulaR1C1Local	String	Set/Get the formula of the series in the language of the user using the R1C1 style references
Has3DEffect	Boolean	Set/Get if bubble charts have a 3D appearance
HasDataLabels	Boolean	Set/Get if the series contains data labels
HasErrorBars	Boolean	Set/Get if the series contains error bars. Use with the ErrorBars property
HasLeaderLines	Boolean	Set/Get if the series contains leader lines. Use with the LeaderLines property
InvertIfNegative	Boolean	Set/Get whether the color of the series' points should be the inverse if the value is negative
LeaderLines	LeaderLines	Read-only. Returns the leader lines associated with the series
MarkerBackgroundColor	Long	Set/Get the color of the series marker background. Use the RGB function to create the color value
MarkerBackgroundColorIndex	XlColorIndex	Set/Get the color of the series marker background. Use the XlColorIndex constants or an index value in the current color palette
MarkerForegroundColor	Long	Set/Get the color of the series points marker foreground. Use the RGB function to create the color value
MarkerForegroundColorIndex	XlColorIndex	Set/Get the color of the series points marker foreground. Use the XlColorIndex constants or an index value in the current color palette

Name	Returns	Description
MarkerSize	Long	Set/Get the size of the point key marker
MarkerStyle	XlMarkerStyle	Set/Get the type of marker to use as the point key (for example, square, diamond, triangle, picture, and so on)
Name	String	Set/Get the name of the series
PictureType	XlChartPictureType	Set/Get how an associated picture is displayed on the series (for example, stretched, tiled). Use the XlPictureType constants
PictureUnit2	Long	Set/Get how many units a picture represents if the PictureType property is set to xlScale
PlotOrder	Long	Set/Get the plotting order for this particular series in the SeriesCollection
Shadow	Boolean	Set/Get whether the points in the series will have a shadow effect
Smooth	Boolean	Set/Get whether scatter or line charts will have curves smoothed
Type	Long	Set/Get the type of series
Values	Variant	Set/Get the range containing the series values or an array of fixed values containing the series values
XValues	Variant	Set/Get the array of x values coming from a range or an array of fixed values

Series Methods

Name	Returns	Parameters	Description
ApplyData Labels	Variant	[Type As XlDataLabelsType], [LegendKey], [AutoText], [HasLeaderLines], [ShowSeriesName], [ShowCategoryName], [ShowValue], [ShowPercentage], [ShowBubbleSize], [Separator]	Applies the data label properties specified by the parameters to the series. The Type parameter specifies whether no label, a value, a percentage of the whole, or a category label is shown. The legend key can appear by the point by setting the LegendKey parameter to True. Set the HasLeaderLines to True to add leader lines to the series

Table continued on following page

Series Methods

Name	Returns	Parameters	Description
ClearFormats	Variant		Clears the formatting made on the series
Copy	Variant		Copies the series into the clipboard
DataLabels	Object	[Index]	Returns the collection of data labels in a series. If the Index parameter is specified, then only a single data label is returned
Delete	Variant		Deletes the series from the series collection
ErrorBar	Variant	[Direction As XlErrorBar Direction], [Include As XlErrorBar Include], [Type As XlErrorBar Type], [Amount], [MinusValues]	Adds error bars to the series. The Direction parameter chooses whether the bar appears on the x or y axis. The Include parameter specifies which error parts to include. The Type parameter decides the type of error bar to use. The Amount parameter is used to choose an error amount. The MinusValues parameter takes the negative error amount to use when the Type parameter is xlErrorBarTypeCustom
Paste	Variant		Uses the picture in the Clipboard as the marker on the points in the series
Points		[Index]	Returns either the collection of points associated with the series or a single point if the Index parameter is specified
Select	Variant		Selects the series' points on the chart
Trendlines		[Index]	Returns either the collection of trendlines associated with the series or a single trendline if the Index parameter is specified

See the `DataLabel` object in this appendix for an example of using the `Series` object.

SeriesLines Object

The `SeriesLines` object accesses the series lines connecting data values from each series. This object only applies to 2D stacked bar or column chart groups. The parent of the `SeriesLines` object is the `ChartGroup` object.

SeriesLines Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

SeriesLines Properties

Name	Returns	Description
<code>Border</code>	<code>Border</code>	Read-only. Returns the border's properties around the series lines
<code>Format</code>	<code>ChartFormat</code>	Returns the <code>ChartFormat</code> object, which controls the line, fill, and effect formatting for the chart area
<code>Name</code>	<code>String</code>	Read-only. Returns the name of the <code>SeriesLines</code> object

SeriesLines Methods

Name	Returns	Description
<code>Delete</code>	<code>VARIANT</code>	Deletes the <code>SeriesLines</code> object
<code>Select</code>	<code>VARIANT</code>	Selects the series lines in the chart

SeriesLines Object Example

```
Sub FormatSeriesLines()
    Dim oCG As ChartGroup
    Dim oSL As SeriesLines
    'Loop through the column groups on the chart
    For Each oCG In ActiveSheet.ChartObjects("Chart 1").Chart.ColumnGroups
        'Make sure we have some series lines
        oCG.HasSeriesLines = True
        'Get the series lines
        Set oSL = oCG.SeriesLines
        'Format the lines
        With oSL
            .Border.Weight = xlThin
            .Border.ColorIndex = 5
        End With
    Next
End Sub
```

ServerViewableItems Collection

The `ServerViewableItems` collection allows you to programmatically define the objects in a workbook that will be viewable through Excel Services. By default, when you choose to publish your workbook to Excel Services, the entire workbook is shown. After you set the `ServerViewableItems`, you can view the collection of objects that are marked as viewable in the Excel Services Options dialog box. Note that only one `ServerViewableItems` object can exist per workbook. The `ServerViewableItems` collection has no properties outside the typical collection attributes listed at the beginning of this appendix.

ServerViewableItems Methods

Name	Returns	Parameters	Description
Add	Object	Object as Variant	Adds a reference to the <code>ServerViewableItems</code> collection for the workbook
Delete		Index As Long	Deletes a reference to an object within the <code>ServerViewableItems</code> collection
DeleteAll			Deletes the references to all objects within the <code>ServerViewableItems</code> collection for the workbook

ServerViewableItems Collection Example

The following example ensures that the chart and the PivotTable on the Summary sheet are the only items that will be viewable in Excel Services:

```
Sub MakeObjectsViewable()  
    'Clear the ServerViewableItems collection  
    ActiveWorkbook.ServerViewableItems.DeleteAll  
  
    'Make only the pivot table and the chart on Summary sheet viewable  
    With ActiveWorkbook.ServerViewableItems  
        .Add ActiveWorkbook.Sheets("Summary").PivotTables("PivotTable1")  
        .Add ActiveWorkbook.Sheets("Summary").ChartObjects("Chart 1")  
    End With  
End Sub
```

ShadowFormat Object

The `ShadowFormat` object allows manipulation of the shadow formatting properties of a parent `Shape` object. Use the `Shadow` property of the `Shape` object to access the `ShadowFormat` object.

ShadowFormat Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ShadowFormat Properties

Name	Returns	Description
Blur	Single	Set/Get the degree of blurriness of the specified shadow
ForeColor	ColorFormat	Set/Get the shadow forecolor
Obscured	MsoTriState	Set/Get whether or not the shape obscures the shadow
OffsetX	Single	Set/Get the horizontal shadow offset
OffsetY	Single	Set/Get the vertical shadow offset
RotateWith Shape	Boolean	Set/Get whether to rotate the shadow when rotating the shape
Size	Single	Set/Get the size of the specified shadow
Style	MsoShadowStyle	Set/Get the style of the specified shadow
Transparency	Single	Set/Get the transparency of the shadow (0 to 1, where 1 is clear)
Type	MsoShadow Type	Set/Get the shadow type
Visible	MsoTriState	Set/Get whether the shadow is visible

ShadowFormat Methods

Name	Parameters	Description
Increment OffsetX	Increment As Single	Changes the horizontal shadow offset
Increment OffsetY	Increment As Single	Changes the vertical shadow offset

ShadowFormat Object Example

```

Sub AddShadow()
    Dim oSF As ShadowFormat
    Set oSF = ActiveSheet.Shapes.Range(1).Shadow
    With oSF
        .Type = msoShadow6
        .OffsetX = 5
        .OffsetY = 5
        .ForeColor.SchemeColor = 2
        .Visible = True
    End With
End Sub

```

Shape Object and the Shapes Collection

The `Shapes` collection holds the list of shapes for a sheet. The `Shape` object represents a single shape such as an `AutoShape`, a freeform shape, an OLE object (such as an image), an ActiveX control, or a picture. Possible parent objects of the `Shapes` collection are the `Worksheet` and `Chart` objects.

The `Shapes` collection has a few methods and properties besides the typical collection attributes. They are listed in the following table.

Shapes Collection Properties and Methods

Name	Returns	Description
<code>Range</code>	<code>ShapeRange</code>	Read-only. Parameters: <code>Index</code> . Returns a <code>ShapeRange</code> object containing only some of the shapes in the <code>Shapes</code> collection
<code>AddCallout</code>	<code>Shape</code>	Method. Parameters: <code>Type As MsoCalloutType</code> , <code>Left As Single</code> , <code>Top As Single</code> , <code>Width As Single</code> , <code>Height As Single</code> . Adds a callout line shape to the collection
<code>AddChart</code>	<code>Shape</code>	Method. Parameters: <code>Type As xlChartType</code> , <code>Left</code> , <code>Top</code> , <code>Width</code> , <code>Height</code> . Adds a chart at a specified location
<code>AddConnector</code>	<code>Shape</code>	Method. Parameters: <code>Type As MsoConnectorType</code> , <code>BeginX As Single</code> , <code>BeginY As Single</code> , <code>EndX As Single</code> , <code>EndY As Single</code> . Adds a connector shape to the collection
<code>AddCurve</code>	<code>Shape</code>	Method. Parameters: <code>SafeArrayOfPoints</code> . Adds a Bezier curve to the collection
<code>AddFormControl</code>	<code>Shape</code>	Method. Parameters: <code>Type As XlFormControl</code> , <code>Left As Long</code> , <code>Top As Long</code> , <code>Width As Long</code> , <code>Height As Long</code> . Adds an Excel control to the collection
<code>AddLabel</code>	<code>Shape</code>	Method. Parameters: <code>Orientation As MsoTextOrientation</code> , <code>Left As Single</code> , <code>Top As Single</code> , <code>Width As Single</code> , <code>Height As Single</code> . Adds a label to the collection
<code>AddLine</code>	<code>Shape</code>	Method. Parameters: <code>BeginX As Single</code> , <code>BeginY As Single</code> , <code>EndX As Single</code> , <code>EndY As Single</code> . Adds a line shape to the collection
<code>AddOLEObject</code>	<code>Shape</code>	Method. Parameters: <code>[ClassType]</code> , <code>[Filename]</code> , <code>[Link]</code> , <code>[DisplayAsIcon]</code> , <code>[IconFileName]</code> , <code>[IconIndex]</code> , <code>[IconLabel]</code> , <code>[Left]</code> , <code>[Top]</code> , <code>[Width]</code> , <code>[Height]</code> . Adds an OLE control to the collection
<code>AddPicture</code>	<code>Shape</code>	Method. Parameters: <code>Filename As String</code> , <code>LinkToFile As MsoTriState</code> , <code>SaveWithDocument As MsoTriState</code> , <code>Left As Single</code> , <code>Top As Single</code> , <code>Width As Single</code> , <code>Height As Single</code> . Adds a picture object to the collection

Name	Returns	Description
AddPolyline	Shape	Method. Parameters: SafeArrayOfPoints. Adds an open polyline or a closed polygon to the collection
AddShape	Shape	Method. Parameters: Type As MsoAutoShapeType, Left As Single, Top As Single, Width As Single, Height As Single. Adds a shape using the Type parameter to the collection
AddTextbox	Shape	Method. Parameters: Orientation As MsoTextOrientation, Left As Single, Top As Single, Width As Single, Height As Single. Adds a textbox to the collection
AddTextEffect	Shape	Method. Parameters: PresetTextEffect As MsoPresetTextEffect, Text As String, FontName As String, FontSize As Single, FontBold As MsoTriState, FontItalic As MsoTriState, Left As Single, Top As Single. Adds a WordArt object to the collection
BuildFreeform	Freeform Builder	Method. Parameters: EditingType As MsoEditingType, X1 As Single, Y1 As Single. Accesses an object that allows creation of a new shape based on ShapeNode objects
SelectAll		Method. Selects all the shapes in the collection

Shape Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

Shape Properties

Name	Returns	Description
Adjustments	Adjustments	Read-only. An object accessing the adjustments for a shape
AlternativeText	String	Set/Get the alternate text to appear if the image is not loaded. Used with a web page
AutoShapeType	MsoAutoShapeType	Set/Get the type of AutoShape used
BackgroundStyle	MsoBackgroundStyleIndex	Set/Get an MsoBackgroundStyleIndex constant representing the background style
BlackWhiteMode	MsoBlackWhiteMode	Property used for compatibility with other drawing packages only. Does not do anything
BottomRightCell	Range	Read-only. Returns the single cell range that describes the cell under the lower-right corner of the shape

Table continued on following page

Shape Properties

Name	Returns	Description
Callout	Callout Format	Read-only. An object accessing the callout properties of the shape
Chart	Chart	Read-only. Returns the chart object contained in a given shape
Child	MsoTriState	Read-only. Returns whether the specified shape is a child shape, or if all shapes in a shape range are child shapes of the same parent
Connection SiteCount	Long	Read-only. Returns the number of potential connection points (sites) on the shape for a connector
Connector	MsoTriState	Read-only. Returns whether the shape is a connector
Connector Format	Connector Format	Read-only. Returns an object containing formatting options for a connector shape. Shape must be a connector shape
ControlFormat	Control Format	Read-only. Returns an object containing formatting options for an Excel control. Shape must be an Excel control
Diagram	Diagram	Read-only. Returns a Diagram object
DiagramNode	DiagramNode	Read-only. Returns a node in the diagram
Fill	FillFormat	Read-only. Returns an object containing fill formatting options for the Shape object
FormControl Type	XlForm Control	Read-only. Returns the type of Excel control the current shape is (for example, xlCheck Box). Shape must be an Excel control
Glow	GlowFormat	Read-only. Returns the glow formatting properties through the GlowFormat object
GroupItems	GroupShapes	Read-only. Returns the shapes that make up the current shape
HasChart	MsoTriState	Read-only. Returns whether a shape or shape range contains a chart
Height	Single	Set/Get the height of the shape
Horizontal Flip	MsoTriState	Read-only. Returns whether the shape has been flipped
Hyperlink	Hyperlink	Read-only. Returns the hyperlink of the shape, if any
ID	Long	Read-only. Returns the type for the specified object
Left	Single	Set/Get the horizontal position of the shape
Line	LineFormat	Read-only. An object accessing the line formatting of the shape

Name	Returns	Description
LinkFormat	LinkFormat	Read-only. An object accessing the OLE linking properties
LockAspect Ratio	MsoTriState	Set/Get whether the dimensional proportions of the shape are kept when the shape is resized
Locked	Boolean	Set/Get whether the shape can be modified if the sheet is locked (True = cannot modify)
Name	String	Set/Get the name of the Shape object
Nodes	ShapeNodes	Read-only. An object accessing the nodes of the freeform shape
OLEFormat	OLEFormat	Read-only. An object accessing OLE object properties, if applicable
OnAction	String	Set/Get the macro to run when the shape is clicked
ParentGroup	Shape	Read-only. Returns the common parent shape of a child shape or a range of child shapes
PictureFormat	Picture Format	Read-only. An object accessing the picture format options
Placement	XlPlacement	Set/Get how the object will react with the cells around the shape
Reflection	Reflection Format	Read-only. Returns the reflection formatting properties through the ReflectionFormat object
Rotation	Single	Set/Get the degrees rotation of the shape
Script	Script	Read-only. Returns the VBScript associated with the shape
Shadow	Shadow Format	Read-only. An object accessing the shadow properties
ShapeStyle	MsoShapte StyleIndex	Set/Get an MsoShapteStyleIndex constant representing the shape style of the Shape object
SoftEdge	SoftEdge Format	Read-only. Returns the SoftEdge formatting properties through the SoftEdgeFormat object
TextEffect	TextEffect Format	Read-only. An object accessing the text effect properties
TextFrame	TextFrame	Read-only. An object accessing the text frame properties
TextFrame2	TextFrame2	Read-only. An object accessing the text frame properties
ThreeD	ThreeD Format	Read-only. An object accessing the 3D effect formatting properties
Top	Single	Set/Get the vertical position of the shape

Table continued on following page

Shape Methods

Name	Returns	Description
TopLeftCell	Range	Read-only. Returns the single cell range that describes the cell over the upper-left corner of the shape
Type	MsoShape Type	Read-only. Returns the type of shape
VerticalFlip	MsoTriState	Read-only. Returns whether the shape has been vertically flipped
Vertices	Variant	Read-only. Returns a series of coordinate pairs describing the Freeform's vertices
Visible	MsoTriState	Set/Get whether the shape is visible
Width	Single	Set/Get the width of the shape
ZOrder Position	Long	Read-only. Returns where the shape is in the ZOrder of the collection (for example, front, back)

Shape Methods

Name	Returns	Parameters	Description
Apply			Applies formatting that has been copied using the <code>PickUp</code> method
Copy			Copies the shape to the clipboard
CopyPicture		[Appearance As <code>XLPicture</code> Appearance], [Format As <code>XLCopyPicture</code> Format]	Copies the range into the clipboard as a picture. The <code>Appearance</code> parameter can be used to specify whether the picture is copied as it looks on the screen or when printed. The <code>Format</code> parameter can specify the type of picture that will be put into the clipboard
Cut			Cuts the shape and places it in the clipboard
Delete			Deletes the shape
Duplicate	Shape		Duplicates the shape returning the new shape
Flip		<code>FlipCmd</code> As <code>MsoFlipCmd</code>	Flips the shape using the <code>FlipCmd</code> parameter

Name	Returns	Parameters	Description
Increment Left		Increment As Single	Moves the shape horizontally
Increment Rotation		Increment As Single	Rotates the shape using the Increment parameter as degrees
IncrementTop		Increment As Single	Moves the shape vertically
PickUp			Copies the format of the current shape so another shape can then apply the formats
Reroute Connections			Optimizes the route of the current connector shape connected between two shapes. Also, this method may be used to optimize all the routes of connectors connected to the current shape
ScaleHeight		Factor As Single, RelativeTo OriginalSize As MsoTriState, [Scale]	Scales the height of the shape by the Factor parameter
ScaleWidth		Factor As Single, RelativeTo OriginalSize As MsoTriState, [Scale]	Scales the width of the shape by the Factor parameter
Select		[Replace]	Selects the shape in the document
SetShapes Default Properties			Sets the formatting of the current shape as a default shape in Word
Ungroup	ShapeRange		Breaks apart the shapes that make up the Shape object
ZOrder		ZOrderCmd As MsoZ OrderCmd	Changes the order of the shape object in the collection

ShapeNode Object and the ShapeNodes Collection

The `ShapeNodes` collection has the list of nodes and curved segments that make up a freeform shape. The `ShapeNode` object specifies a single node or curved segment that makes up a freeform shape. The `Nodes` property of the `Shape` object is used to access the `ShapeNodes` collection. The `ShapeNodes` collection has no properties outside the typical collection attributes listed at the beginning of this appendix.

ShapeNodes Collection Methods

Name	Returns	Description
<code>Count</code>	<code>Integer</code>	Read-only. Returns the number of <code>ShapeNode</code> objects in the collection
<code>Delete</code>		Method. Parameters: <code>Index As Long</code> . Deletes the node specified by the <code>Index</code>
<code>Insert</code>		Method. Parameters: <code>Index As Long</code> , <code>SegmentType As MsoSegmentType</code> , <code>EditingType As MsoEditingType</code> , <code>X1, Y1, X2, Y2, X3, Y3</code> . Inserts a node or curved segment in the <code>Nodes</code> collection
<code>SetEditingType</code>		Method. Parameters: <code>Index As Long</code> , <code>EditingType As MsoEditingType</code> . Sets the editing type for a node
<code>SetPosition</code>		Method. Parameters: <code>Index As Long</code> , <code>X1 As Single</code> , <code>Y1 As Single</code> . Moves the specified node
<code>SetSegmentType</code>		Method. Parameters: <code>Index As Long</code> , <code>SegmentType As MsoSegmentType</code> . Changes the segment type following the node

ShapeNode Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ShapeNode Properties

Name	Returns	Description
<code>EditingType</code>	<code>MsoEditingType</code>	Read-only. Returns the editing type for the node
<code>Points</code>	<code>Variant</code>	Read-only. Returns the positional coordinate pair
<code>SegmentType</code>	<code>MsoSegmentType</code>	Read-only. Returns the type of segment following the node

ShapeNode Object and the ShapeNodes Collection Example

```

Sub ToggleArch()
    Dim oShp As Shape
    Dim oSN As ShapeNodes
    Set oShp = ActiveSheet.Shapes(1)
    'Is the Shape a freeform?
    If oShp.Type = msoFreeform Then

        'Yes, so get its nodes
        Set oSN = oShp.Nodes
        'Toggle segment 3 between a line and a curve
        If oSN.Item(3).SegmentType = msoSegmentCurve Then
            oSN.SetSegmentType 3, msoSegmentLine
        Else
            oSN.SetSegmentType 3, msoSegmentCurve
        End If
    End If
End Sub

```

ShapeRange Collection

The `ShapeRange` collection holds a collection of `Shape` objects for a certain range or selection in a document. Possible parent items are the `Range` and `Selection` objects. The `ShapeRange` collection has many properties and methods besides the typical collection attributes. These items are listed next.

It's important to note that some operations will cause an error if performed on a `ShapeRange` collection with multiple shapes.

ShapeRange Properties

Name	Returns	Description
<code>Adjustments</code>	<code>Adjustments</code>	Read-only. An object accessing the adjustments for a shape
<code>AlternativeText</code>	<code>String</code>	Set/Get the alternative text to appear if the image is not loaded. Used with a web page
<code>AutoShapeType</code>	<code>MsoAutoShapeType</code>	Set/Get the type of <code>AutoShape</code> used
<code>BackgroundStyle</code>	<code>MsoBackgroundStyleIndex</code>	Set/Get an <code>MsoBackgroundStyleIndex</code> constant representing the background style
<code>BlackWhiteMode</code>	<code>MsoBlackWhiteMode</code>	Property used for compatibility with other drawing packages only. Does not do anything
<code>Callout</code>	<code>CalloutFormat</code>	Read-only. An object accessing the callout properties of the shape
<code>Chart</code>	<code>Chart</code>	Read-only. Returns the chart object contained in a given shape range

Table continued on following page

ShapeRange Properties

Name	Returns	Description
Child	MsoTriState	Read-only. Returns whether the specified shape is a child shape, or if all shapes in a shape range are child shapes of the same parent
Connection SiteCount	Long	Read-only. Returns the number of potential connection points (sites) on the shape for a connector
Connector	MsoTriState	Read-only. Returns whether the shape is a connector
Connector Format	Connector Format	Read-only. Returns an object containing formatting options for a connector shape. Shape must be a connector shape
Fill	FillFormat	Read-only. An object accessing the fill properties of the shape
Glow	GlowFormat	Read-only. Returns the glow formatting properties through the GlowFormat object
GroupItems	GroupShapes	Read-only. Returns the shapes that make up the current shape
HasChart	MsoTriState	Read-only. Returns whether a shape or shape range contains a chart
Height	Single	Set/Get the height of the shape
Horizontal Flip	MsoTriState	Read-only. Returns whether the shape has been flipped
ID	Long	Read-only. Returns the type for the specified object
Left	Single	Set/Get the horizontal position of the shape
Line	LineFormat	Read-only. An object accessing the line formatting of the shape
LockAspect Ratio	MsoTriState	Set/Get whether the dimensional proportions of the shape are kept when the shape is resized
Name	String	Set/Get the name of the shape
Nodes	ShapeNodes	Read-only. Returns the nodes associated with the shape
ParentGroup	Shape	Read-only. Returns the common parent shape of a child shape or a range of child shapes
PictureFormat	Picture Format	Read-only. An object accessing the picture format options

Name	Returns	Description
Reflection	ReflectionFormat	Read-only. Returns the reflection formatting properties through the <code>ReflectionFormat</code> object
Rotation	Single	Set/Get the degrees rotation of the shape
Shadow	ShadowFormat	Read-only. An object accessing the shadow properties
ShapeStyle	MsoShappteStyleIndex	Set/Get an <code>MsoShappteStyleIndex</code> constant representing the shape style of the <code>Shape</code> object
SoftEdge	SoftEdgeFormat	Read-only. Returns the <code>SoftEdge</code> formatting properties through the <code>SoftEdgeFormat</code> object
TextEffect	TextEffectFormat	Read-only. An object accessing the text effect properties
TextFrame	TextFrame	Read-only. An object accessing the text frame properties
TextFrame2	TextFrame2	Read-only. Returns the text formatting properties through the <code>TextFrame2</code> object
ThreeD	ThreeDFormat	Read-only. An object accessing the 3D effect formatting properties
Top	Single	Set/Get the vertical position of the shape
Type	MsoShapeType	Read-only. Returns the type of shape
VerticalFlip	MsoTriState	Read-only. Returns whether the shape has been vertically flipped
Vertices	Variant	Read-only. Returns a series of coordinate pairs describing the <code>Freeform</code> 's vertices
Visible	MsoTriState	Set/Get whether the shape is visible
Width	Single	Set/Get the width of the shape
ZOrderPosition	Long	Read-only. Changes the order of the object in the collection

ShapeRange Methods

Name	Returns	Parameters	Description
Align		AlignCmd As MsoAlignCmd, RelativeTo As MsoTriState	Aligns the shapes in the collection to the alignment properties set by the parameters

Table continued on following page

ShapeRange Methods

Name	Returns	Parameters	Description
Apply			Applies the formatting that was set by the <code>PickUp</code> method
Delete			Deletes the shape
Distribute		<code>DistributeCmd As MsoDistributeCmd, RelativeTo As MsoTriState</code>	Distributes the shapes in the collection evenly, either horizontally or vertically
Duplicate	ShapeRange		Duplicates the shape and returns a new <code>ShapeRange</code>
Flip		<code>FlipCmd As MsoFlipCmd</code>	Flips the shape using the <code>FlipCmd</code> parameter
Group	Shape		Groups the shapes in the collection
IncrementLeft		<code>Increment As Single</code>	Moves the shape horizontally
Increment Rotation		<code>Increment As Single</code>	Rotates the shape using the <code>Increment</code> parameter as degrees
IncrementTop		<code>Increment As Single</code>	Moves the shape vertically
PickUp			Copies the format of the current shape so another shape can then apply the formats
Regroup	Shape		Regroups any previously grouped shapes
Reroute Connections			Optimizes the route of the current connector shape connected between two shapes. Also, this method may be used to optimize all the routes of connectors connected to the current shape
ScaleHeight		<code>Factor As Single, RelativeTo OriginalSize As MsoTriState, [Scale]</code>	Scales the height of the shape by the <code>Factor</code> parameter

Name	Returns	Parameters	Description
ScaleWidth		Factor As Single, RelativeTo OriginalSize As MsoTriState, [Scale]	Scales the width of the shape by the Factor parameter
Select		[Replace]	Selects the shape in the document
SetShapesDefaultProperties			Sets the formatting of the current shape as a default shape in Word
Ungroup	ShapeRange		Breaks apart the shapes that make up the Shape object
ZOrder		ZOrderCmd As MsoZOrderCmd	Changes the order of the Shape object in the collection

ShapeRange Collection Example

```
Sub AlignShapeRanges()
    Dim oSR As ShapeRange
    'Get the first two shapes on the sheet
    Set oSR = ActiveSheet.Shapes.Range(Array(1, 2))
    'Align the left-hand edges of the shapes
    oSR.Align msoAlignLefts, msoFalse
End Sub
```

Sheets Collection

The `Sheets` collection contains all of the sheets in the parent workbook. Sheets in a workbook consist of chart sheets and worksheets. Therefore, the `Sheets` collection holds both the `Chart` objects and `Worksheet` objects associated with the parent workbook. The parent of the `Sheets` collection is the `Workbook` object.

Sheets Common Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Sheets Properties

Sheets Properties

Name	Returns	Description
HPageBreaks	HPageBreaks	Read-only. Returns a collection holding all the horizontal page breaks associated with the <code>Sheets</code> collection
Visible	Variant	Set/Get whether the sheets in the collection are visible. Also, can set this to <code>xlVeryHidden</code> to prevent a user from making the sheets in the collection visible
VPageBreaks	VpageBreaks	Read-only. Returns a collection holding all the vertical page breaks associated with the worksheets of the <code>Sheets</code> collection

Sheets Methods

Name	Returns	Parameters	Description
Add	Object	[Before], [After], [Count], [Type]	Adds a sheet to the collection. You can specify where the sheet goes by choosing which sheet object will be before the new sheet object (<code>Before</code> parameter) or after the new sheet (<code>After</code> parameter). The <code>Count</code> parameter decides how many sheets are created. The <code>Type</code> parameter can be used to specify the type of sheet using the <code>XLSheetType</code> constants
Copy		[Before], [After]	Adds a new copy of the currently active sheet to the position specified in the <code>Before</code> or <code>After</code> parameters
Delete			Deletes all the sheets in the collection. Remember, a workbook must contain at least one sheet
FillAcross Sheets		RangeAs Range, Type As <code>XlFillWith</code>	Copies the values in the <code>Range</code> parameter to all the other sheets at the same location. The <code>Type</code> parameter can be used to specify whether cell contents, formulas, or everything is copied

Name	Returns	Parameters	Description
Move		[Before], [After]	Moves the current sheet to the position specified by the parameters. See the Add method
PrintOut		[From], [To], [Copies], [Preview], [Active Printer], [Print ToFile], [Collate], [PrToFile Name]	Prints out the sheets in the collection. The printer, number of copies, collation, and whether a print preview is desired can be specified with the parameters. Also, the sheets can be printed to a file using the PrintToFile and PrToFileName parameters. The From and To parameters can be used to specify the range of printed pages
Print Preview		[Enable Changes]	Displays the current sheet in the collection in a print preview mode. Set the EnableChanges parameter to False to disable the Margins and Setup buttons, hence not allowing the viewer to modify the page setup
Select		[Replace]	Selects the current sheet in the collection

SheetViews Object

The `Sheetviews` collection represents all the sheet views in the specified or active workbook window. This collection has no properties or methods outside of the common `Application`, `Count`, `Item`, and `Parent` properties, which are defined at the beginning of this appendix.

SmartTag Object and the SmartTags Collection Object

The `SmartTag` object represents an identifier that is assigned to a cell. Excel comes with many SmartTags, such as the Stock Ticker and Date recognizer, built in. However, you may also write your own SmartTags in Visual Basic.

The `SmartTags` collection represents all the SmartTags assigned to cells in an application.

SmartTags Collection Common Properties

Along with the typical collection attributes, the `SmartTags` collection has an `Add` method that adds a `SmartTag` object to the collection.

SmartTag Common Properties

SmartTag Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

SmartTag Properties

Name	Returns	Description
<code>DownloadURL</code>	<code>String</code>	Read-only. Returns a URL to save along with the corresponding SmartTag
<code>Name</code>	<code>String</code>	Read-only. Returns the name of the SmartTag
<code>Properties</code>	<code>Custom Properties</code>	Read-only. Returns the properties of the SmartTag
<code>Range</code>	<code>Range</code>	Read-only. Returns the range to which the specified SmartTag applies
<code>SmartTag Actions</code>	<code>SmartTag Actions</code>	Read-only. Returns the type of action for the selected SmartTag
<code>XML</code>	<code>String</code>	Read-only. Returns a sample of the XML that would be passed to the action handler

SmartTag Methods

Name	Description
<code>Delete</code>	Deletes the object

SmartTagAction Object and the SmartTagActions Collection Object

The `SmartTagAction` object represents an action that can be performed by a SmartTag. This may involve displaying the latest price for a stock symbol, or setting up an appointment on a certain date.

The `SmartTagActions` collection represents all of the `SmartTagAction` objects in the application.

SmartTagAction Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

SmartTagAction Properties

Name	Returns	Description
<code>ActiveXControl</code>	<code>Object</code>	Reference to an ActiveX control that is currently in the Document Actions task pane
<code>CheckboxState</code>	<code>Boolean</code>	Returns <code>True</code> if the checkbox is checked; otherwise, <code>False</code> is returned

Name	Returns	Description
ExpandHelp	Boolean	Returns <code>True</code> if the smart document help control is currently expanded. If not, <code>False</code> is returned
ListSelection	Long	Returns a index number for an item within a <code>List</code> control
Name	String	Read-only. Returns the name of the <code>SmartTag</code>
PresentInPane	Boolean	Returns a <code>Boolean</code> value indicating if a smart document control is currently being shown in the Document Actions task pane
RadioGroup Selection	Long	Returns an index number to the currently selected radio button within a <code>RadioGroup</code> control
TextboxText	String	Returns the text within a <code>TextBox</code> control
Type	<code>xlSmartTagControlType</code>	Returns a constant of the <code>xlSmartTagControlType</code> enumeration, representing the type of Smart Document control displayed in the Document Actions task pane

SmartTagAction Methods

Name	Description
Execute	Activates the <code>SmartTag</code> action

SmartTagOptions Collection Object

The `SmartTagOptions` collection represents all the options of a `SmartTag`. For instance, it determines whether `SmartTags` should be embedded in the worksheet, or if they should be displayed at all.

SmartTagOptions Collection Properties

Name	Returns	Description
DisplaySmart Tags	<code>XlSmartTagDisplayMode</code>	Set/Get the display features for <code>SmartTags</code>
EmbedSmartTags	Boolean	Set/Get whether to embed <code>SmartTags</code> on the specified workbook

SmartTagRecognizer Object and the SmartTagRecognizers Collection Object

The `SmartTagRecognizer` object represents the recognizer engines that label the data in the worksheet. These can be user-defined, and as such, any kind of information can be identified by `SmartTags`.

The `SmartTagRecognizers` collection represents all of the `SmartTagRecognizer` objects in the application.

SmartTagRecognizers Collection Properties

SmartTagRecognizers Collection Properties

Name	Returns	Description
Recognize	Boolean	Set/Get whether data can be labeled with a SmartTag

SmartTagRecognizer Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

SmartTagRecognizer Properties

Name	Returns	Description
Enabled	Boolean	Set/Get whether the object is recognized
FullName	String	Read-only. Returns the name of the object, including its path on disk, as a string
ProgId	String	Read-only. Returns the programmatic identifiers for the object

Sort Object

The `Sort` object exposes the properties and methods to programmatically manipulate the sorting of a range of data.

Sort Common Properties

The `Application` and `Parent` properties are defined at the beginning of this appendix.

Sort Properties

Name	Returns	Description
Header	XlYesNoGuess	Set/Get whether the first row contains header information
MatchCase	Boolean	Set/Get whether the sort is case-sensitive
Orientation	XlSortOrientation	Set/Get the orientation for the sort (Column sort or Row sort)
Rng	Range	Read-only. Returns the range of values on which the sort is performed
SortFields	SortFields	Read-only. Allows for the storing of sort state on workbooks, lists, and autofilters
SortMethod	XlSortMethod	Set/Get the sort method for Chinese languages

Sort Methods

Name	Returns	Parameters	Description
Apply			Applies the copied sort formatting
SetRange		[Rng] AS Range	Sets the starting and ending character positions for <code>Sort</code> object

SortField Object and the SortFields Collection

The `SortFields` collection is a collection of `SortField` objects that allow developers to store a sort state on workbooks, lists, and autofilters.

SortFields Common Properties

The `Application`, `Count`, and `Parent` properties are defined at the beginning of this appendix.

SortFields Methods

Name	Returns	Parameters	Description
Add			Applies the copied sort formatting
Clear			Clears all the <code>SortFields</code> objects

SortField Common Properties

The `Application` and `Parent` properties are defined at the beginning of this appendix.

SortField Properties

Name	Returns	Description
<code>CustomOrder</code>	Variant	Set/Get a custom order to sort the fields
<code>DataOption</code>	<code>XlSortDataOption</code>	Set/Get how to sort text in the range specified in <code>SortField</code> object
<code>Key</code>	Range	Read-only. Specifies the sort field that determines the values to be sorted
<code>Order</code>	<code>XlSortOrder</code>	Set/Get the sort order for the values specified in the key
<code>Priority</code>	Long	Set/Get the priority for the sort field
<code>SortOn</code>	<code>XlSortOn</code>	Set/Get the attribute of the cell on which to sort
<code>SortOnValue</code>	Object	Read-only. Returns the value on which the sort is performed for the specified <code>SortField</code> object

SortField Methods

SortField Methods

Name	Returns	Parameters	Description
Delete			Removes the specified <code>SortField</code> object from the <code>SortFields</code> collection
ModifyKey		[Key] As Range	Modifies the key value by which values are sorted in the field
SetIcon		[Icon] as Icon	Sets an icon for a <code>SortField</code> object

SoundNote Object

The `SoundNote` object is not used in the current version of Excel. It is kept here for compatibility purposes only. The list of its methods is shown next.

SoundNote Methods

Name	Returns	Parameters
Delete	Variant	
Import	Variant	Filename As String
Play	Variant	
Record	Variant	

Speech Object

Represents the speech recognition applet that comes with Office. This new Speech feature allows text to be read back on demand, or when you enter data on a document. For Excel, you have the option of having each cell's contents read back as they are entered on the worksheet. Use the `SpeakCellOnEnter` property of this object to enable this feature.

Speech is accessible through the `Application` object.

Speech Properties

Name	Returns	Description
Direction	<code>XlSpeakDirection</code>	Set/Get the order in which the cells will be spoken
<code>SpeakCellOnEnter</code>	Boolean	Set/Get whether to turn on Excel's mode where the active cell will be spoken when the Enter key is pressed, or when the active cell is finished being edited

Speech Methods

Name	Returns	Parameters	Description
Speak		Text As String, [SpeakAsync], [SpeakXML], [Purge]	The Text is spoken by Excel. If Purge is True, the current speech will be terminated and any buffered text will be purged before Text is spoken

Speech Object Example

The following routine reads off the expense totals for all items that are greater than a limit set in another cell on the sheet:

```
Sub ReadHighExpenses ()

    Dim lTotal As Long
    Dim lLimit As Long
    Dim rng As Range

    'Grab the limitation amount
    lLimit = wksAllowEditRange.Range("Limit")

    'Loop through the expense totals
    For Each rng In wksAllowEditRange.Range("Expenses")
        'Store the current expense total
        lTotal = rng.Offset(0, 5).Value

        'If the current total is greater than
        ' the limit, read it off
        If lTotal > lLimit Then
            Application.Speech.Speak rng.Text
            Application.Speech.Speak lTotal
        End If
    Next rng

End Sub
```

SpellingOptions Collection Object

Represents the spelling options in Excel. These options can be found on the Proofing section of the Excel Options dialog box and are accessed through the Application object. Hence, this object is accessible through the Application object.

SpellingOptions Collection Properties

Name	Returns	Description
ArabicModes	XlArabic Modes	Set/Get the mode for the Arabic spelling checker
DictLang	Long	Set/Get the dictionary language used by Excel for checking spelling
GermanPost Reform	Boolean	Set/Get whether to check the spelling of words using the German post-reform rules
HebrewModes	XlHebrew Modes	Set/Get the mode for the Hebrew spelling checker
IgnoreCaps	Boolean	Set/Get whether to check for uppercase words, or lowercase words during spelling checks
IgnoreFile Names	Boolean	Set/Get whether to check for Internet and file addresses during spelling checks
IgnoreMixed Digits	Boolean	Set/Get whether to check for mixed digits during spelling checks
KoreanCombine Aux	Boolean	Set/Get whether to combine Korean auxiliary verbs and adjectives when using the spelling checker
KoreanProcess Compound	Boolean	Set/Get whether to process Korean compound nouns when using the spelling checker
KoreanUseAuto ChangeList	Boolean	Set/Get whether to use the auto-change list for Korean words when using the spelling checker
SuggestMain Only	Boolean	Set/Get whether to suggest words from only the main dictionary for using the spelling checker
UserDict	String	Set/Get whether to create a custom dictionary to which new words can be added when performing spelling checks

SpellingOptions Collection Object Example

The following routine sets some spelling options and creates a new custom dictionary where added words during a spellcheck can be found:

```
Sub SetSpellingOptions()  
  
    'This one is as simple as it gets  
    With Application.SpellingOptions  
        .IgnoreCaps = True  
        .IgnoreFileNames = True  
        .IgnoreMixedDigits = True  
        .SuggestMainOnly = False  
  
    'This property creates a custom dictionary  
    ' called Wrox.dic, which can be found and directly edited
```

```

' in C:\WINDOWS\Application Data\Microsoft\Proof.
'Added words during a spellcheck will now appear
' in this custom dictionary.
.UserDict = "Wrox.dic"
End With
End Sub

```

Style Object and the Styles Collection

The `Styles` collection holds the list of user-defined and built-in formatting styles, such as `Currency` and `Normal`, in a workbook or range. Each `Style` object represents formatting attributes associated with the parent object. There are some Excel built-in `Style` objects, such as `Currency`. Also, new styles can be created. Possible parents of the `Styles` collection are the `Range` and `Workbook` objects.

The `Styles` collection has two extra attributes besides the typical collection ones. The `Add` method uses the `Name` parameter to add a new style to the collection. The `BasedOn` parameter of the `Add` method can be used to specify a range that the new style will be based on. The `Merge` method merges the styles in the workbook specified by the `Workbook` parameter into the current parent workbook.

Style Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Style Properties

Name	Returns	Description
<code>AddIndent</code>	<code>Boolean</code>	Set/Get whether text associated with the style is automatically indented if the text alignment in a cell is set to equally distribute
<code>Borders</code>	<code>Borders</code>	Read-only. Returns the collection of borders associated with the style. Each border side can be accessed individually
<code>BuiltIn</code>	<code>Boolean</code>	Read-only. Returns whether the style is built-in
<code>Font</code>	<code>Font</code>	Read-only. Returns an object containing <code>Font</code> options for the associated style
<code>FormulaHidden</code>	<code>Boolean</code>	Set/Get whether formulas associated with the style will be hidden if the workbook/worksheet is protected
<code>HorizontalAlignment</code>	<code>XlHAlign</code>	Set/Get how the cells associated with the style are horizontally aligned. Use the <code>XlHAlign</code> constants
<code>IncludeAlignment</code>	<code>Boolean</code>	Set/Get whether the styles include properties associated with alignment (that is, <code>AddIndent</code> , <code>HorizontalAlignment</code> , <code>VerticalAlignment</code> , <code>WrapText</code> , and <code>Orientation</code>)

Table continued on following page

Style Properties

Name	Returns	Description
IncludeBorder	Boolean	Set/Get whether border attributes are included with the style (that is, Color, ColorIndex, LineStyle, and Weight)
IncludeFont	Boolean	Set/Get whether font attributes are included in the style (that is, Background, Bold, Color, ColorIndex, FontStyle, Italic, Name, OutlineFont, Shadow, Size, Strikethrough, Subscript, Superscript, and Underline)
IncludeNumber	Boolean	Set/Get whether the NumberFormat property is included in the style
IncludePatterns	Boolean	Set/Get whether interior pattern related properties are included in the style (that is, Color, ColorIndex, InvertIfNegative, Pattern, PatternColor, and PatternColorIndex)
IncludeProtection	Boolean	Set/Get whether the locking related properties are included with the style (that is, FormulaHidden and Locked)
IndentLevel	Long	Set/Get the indent level for the style
Interior	Interior	Read-only. Returns an object containing options to format the inside area of the style (for example, interior color)
Locked	Boolean	Set/Get whether the style properties can be changed if the workbook is locked
MergeCells	Variant	Set/Get whether the current style contains merged cells
Name	String	Read-only. Returns the name of the style
NameLocal	String	Read-only. Returns the name of the style in the language of the user's computer
NumberFormat	String	Set/Get the number format associated with the style
NumberFormatLocal	String	Set/Get the number format associated with the style in the language of the end user
Orientation	XlOrientation	Set/Get the text orientation for the cell text associated with the style. A value from -90 to 90 degrees can be specified or an XlOrientation constant
ReadingOrder	Long	Set/Get whether the text associated with the style is from right-to-left (xlRTL), left-to-right (xlLTR), or context sensitive (xlContext)
ShrinkToFit	Boolean	Set/Get whether the cell text associated with the style will automatically shrink to fit the column width

Name	Returns	Description
Value	String	Read-only. Returns the name of the style
Vertical Alignment	XlVAlign	Set/Get how the cells associated with the style are vertically aligned. Use the XlVAlign constants
WrapText	Boolean	Set/Get whether cell text wraps in cells associated with the style

Style Methods

Name	Returns	Parameters	Description
Delete	Variant		Deletes the style from the collection

Style Object and the Styles Collection Example

```

Sub UpdateStyles()
    Dim oStyle As Style
    Set oStyle = ActiveWorkbook.Styles("Accent4")
    'Update the Editing style to be unlocked with a default background
    With oStyle
        .IncludePatterns = True
        .IncludeProtection = True
        .Locked = False
        .Interior.Pattern = xlNone
    End With
End Sub

```

Tab Object

Represents the Sheet tab at the bottom of an Excel chart sheet or worksheet. You will note that you can customize the sheet's tab color by using either the `Color` or `ColorIndex` properties of this object.

Tab Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Tab Properties

Name	Returns	Description
Color	Variant	Set/Get the primary color of the Tab object. Use the RGB function to create a color value
ColorIndex	XlColorIndex	Set/Get the color of the interior

Table continued on following page

Tab Object Example

Name	Returns	Description
ThemeColor	xlTheme Color	Set/Get the theme color in the applied color scheme associated with an object. Should the object have no association with a theme, then trying to access the ThemeColor property will result in an error
TintAndShade	Single	Set/Get a Single value from -1 (darkest) to 1 (lightest), which darkens or lightens a color. Zero (0) is neutral

Tab Object Example

The following routine changes the tab color for all budget worksheets in a workbook based on a setting in a custom property for each worksheet:

```
Sub ColorBudgetTabs()  
    Dim bBudget As Boolean  
    Dim oCustomProp As CustomProperty  
    Dim oCustomProps As CustomProperties  
    Dim wks As Worksheet  
  
    'Loop through each worksheet in this workbook  
    For Each wks In ThisWorkbook.Worksheets  
        'Loop through all of the custom properties  
        ' for the current worksheet until the  
        ' "IsBudget" property name is found  
        For Each oCustomProp In wks.CustomProperties  
            If oCustomProp.Name = "IsBudget" Then  
                'Grab its value and exit the loop  
                bBudget = CBool(oCustomProp.Value)  
                Exit For  
            End If  
        Next oCustomProp  
  
        'Use the value in the custom property to determine  
        ' whether the tab should be colored.  
        If bBudget Then wks.Tab.ColorIndex = 20 'Light blue  
  
    Next wks  
  
End Sub
```

TableStyle Object and the TableStyles Collection Object

The `TableStyle` object defines the formatting style applied to any given table. The `TableStyles` collection contains all `TableStyles`.

TableStyles Collection Common Properties

Along with the typical collection attributes, the `TableStyles` collection has an `Add` method, which adds a `TableStyle` object to the collection.

TableStyle Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

TableStyle Properties

Name	Returns	Description
<code>BuiltIn</code>	Boolean	Read-only. Returns <code>True</code> if the specified style is built-in
<code>Name</code>	String	Read-only. Returns the name of the <code>TableStyle</code> object
<code>NameLocal</code>	String	Read-only. Returns the name of the <code>TableStyle</code> object. If the style is a built-in style, this property returns the name of the style in the language of the current locale
<code>ShowAsAvailable</code>	Boolean PivotTable Style	Set/Get whether a style is shown in the gallery for PivotTable styles. Setting this property to <code>False</code> tells Excel not to show the specified style in either the gallery or when the active cell is in the PivotTable.
<code>ShowAsAvailable TableStyle</code>	Boolean	Set/Get whether a style is shown in the gallery for Table styles. Setting this property to <code>False</code> tells Excel not to show the specified style either in the gallery or when the active cell is in the table.
<code>TableStyleElements</code>	TableStyle Elements	Read-only. Returns the <code>TableStyleElements</code> object

TableStyle Methods

Name	Returns	Parameters	Description
<code>Delete</code>			Deletes the specified object
<code>Duplicate</code>	TableStyle	<code>NewTableStyleName</code>	Adds a duplicate copy of the specified <code>TableStyle</code> object and returns a reference to the new copy

TableStyle Object and TableStyles Example

```
Sub AddCustomStyle()
    Dim oStyle As TableStyle

    'Add a new style
        Set oStyle = ActiveWorkbook.TableStyles.Add("JustMyStyle")

    'Apply desired formats
```

TableStyleElement Object and the TableStyleElements Collection Object

```
With oStyle
    .TableStyleElements(xlHeaderRow).Font.ColorIndex = 50
    .TableStyleElements(xlHeaderRow).Interior.ColorIndex = 44
'Ensure your style is shown in the styles gallery
    .ShowAsAvailableTableStyle = True
End With

'Make your style the default for the workbook
    ActiveWorkbook.DefaultTableStyle = "JustMyStyle"
End Sub
```

TableStyleElement Object and the TableStyleElements Collection Object

The `TableStyleElement` object defines the individual elements that make up the look and feel of a table style. The `TableStyleElements` collection contains all elements that, together, make up the formatting of a `TableStyle` object.

TableStyleElement Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

TableStyleElement Properties

Name	Returns	Description
<code>Borders</code>	<code>Borders</code>	Read-only. Returns a <code>Borders</code> collection that represents the borders of a table style element
<code>Font</code>	<code>Font</code>	Read-only. Returns a <code>Font</code> object that represents the font of the specified object
<code>HasFormat</code>	<code>Boolean</code>	Set/Get whether a table style element has formatting applied to the specified element
<code>Interior</code>	<code>Interior</code>	Read-only. Returns an <code>Interior</code> object that represents the interior of the specified object
<code>StripeSize</code>	<code>Long</code>	Set/Get banding size

TableStyleElement Methods

Name	Returns	Parameters	Description
<code>Clear</code>			Clears all formatting for the specified <code>TableStyleElement</code> object

TextEffectFormat Object

The `TextEffectFormat` object contains all the properties and methods associated with `WordArt` objects. The parent object of the `TextEffectFormat` is always the `Shape` object.

TextEffectFormat Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

TextEffectFormat Properties

Name	Returns	Description
<code>Alignment</code>	<code>MsoTextEffectAlignment</code>	Set/Get the alignment of the <code>WordArt</code>
<code>FontBold</code>	<code>MsoTriState</code>	Set/Get whether the <code>WordArt</code> is bold
<code>FontItalic</code>	<code>MsoTriState</code>	Set/Get whether the <code>WordArt</code> is italic
<code>FontName</code>	<code>String</code>	Set/Get the font used in the <code>WordArt</code>
<code>FontSize</code>	<code>Single</code>	Set/Get the font size in the <code>WordArt</code>
<code>KernedPairs</code>	<code>MsoTriState</code>	Set/Get whether the characters are kerned in the <code>WordArt</code>
<code>NormalizedHeight</code>	<code>MsoTriState</code>	Set/Get whether both the uppercase and lowercase characters are the same height
<code>PresetShape</code>	<code>MsoPresetTextEffectShape</code>	Set/Get the shape of the <code>WordArt</code>
<code>PresetTextEffect</code>	<code>MsoPresetTextEffect</code>	Set/Get the effect associated with the <code>WordArt</code>
<code>RotatedChars</code>	<code>MsoTriState</code>	Set/Get whether the <code>WordArt</code> has been rotated by 90 degrees
<code>Text</code>	<code>String</code>	Set/Get the text in the <code>WordArt</code>
<code>Tracking</code>	<code>Single</code>	Set/Get the spacing ratio between characters

TextEffectFormat Methods

Name	Description
<code>ToggleVerticalText</code>	Toggles the text from vertical to horizontal and back

TextEffectFormat Object Example

TextEffectFormat Object Example

```
Sub FormatTextArt()  
    Dim oTEF As TextEffectFormat  
    Dim oShp As Shape  
    Set oShp = ActiveSheet.Shapes(1)  
    Set oTEF = oShp.TextEffect  
  
    With oTEF  
        .FontName = "Times New Roman"  
        .FontBold = True  
        .PresetTextEffect = msoTextEffect14  
        .Text = "Hello World!"  
    End With  
End Sub
```

TextFrame Object

The `TextFrame` object contains the properties and methods that can manipulate text-frame shapes. Possible parent objects of the `TextFrame` object are the `Shape` and `ShapeRange` objects.

TextFrame Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

TextFrame Properties

Name	Returns	Description
<code>AutoMargins</code>	Boolean	Set/Get whether Excel will calculate the margins of the text frame automatically. Set this property to <code>False</code> to use the <code>MarginLeft</code> , <code>MarginRight</code> , <code>MarginTop</code> , and <code>MarginBottom</code> properties
<code>AutoSize</code>	Boolean	Set/Get whether the size of the text frame changes to match the text inside
<code>HorizontalAlignment</code>	<code>XLHAlign</code>	Set/Get how the text frame is horizontally aligned. Use the <code>XLHAlign</code> constants
<code>MarginBottom</code>	Single	Set/Get the bottom spacing in a text frame
<code>MarginLeft</code>	Single	Set/Get the left spacing in a text frame
<code>MarginRight</code>	Single	Set/Get the right spacing in a text frame
<code>MarginTop</code>	Single	Set/Get the top spacing in a text frame
<code>Orientation</code>	<code>MsoTextOrientation</code>	Set/Get the orientation of the text in the text frame

Name	Returns	Description
ReadingOrder	Long	Set/Get whether the text in the frame is read from right-to-left (xlRTL), is read from left-to-right (xlLTR), or is context sensitive (xlContext)
Vertical Alignment	XlVAlign	Set/Get how the text frame is vertically aligned. Use the XlVAlign constants

TextFrame Methods

Name	Returns	Parameters	Description
Characters	Characters	[Start], [Length]	Returns an object containing all the characters in the text frame. Allows manipulation on a character-by-character basis and retrieves only a subset of text in the frame

TextFrame Object Example

```
Sub SetShapeAutoSized()
    Dim oTF As TextFrame
    Dim oShp As Shape
    Set oShp = ActiveSheet.Shapes(1)
    Set oTF = oShp.TextFrame
    oTF.AutoSize = True
End Sub
```

TextFrame2 Object

The `TextFrame2` object represents the text frame in a `Shape`, `ShapeRange`, or `ChartFormat` object. This object contains the text in the text frame, as well as the properties and methods that control the alignment and anchoring of the text frame.

TextFrame2 Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

TextFrame2 Properties

Name	Returns	Description
AutoSize	MsoAutoSize	Set/Get the size of the specified object that changes automatically to fit text within its boundaries
Column	TextColumn2	Read-only. Returns the <code>TextColumn2</code> object that represents the columns within the text frame

Table continued on following page

TextFrame2 Properties

Name	Returns	Description
HasText	Boolean	Read-only. Returns whether the specified text frame has text
HorizontalAnchor	MsoHorizontalAnchor	Set/Get the horizontal anchor type for the specified text
MarginBottom	Single	Set/Get the distance (in points) between the bottom of the text frame and the bottom of the inscribed rectangle of the shape that contains the text
MarginLeft	Single	Set/Get the distance (in points) between the left edge of the text frame and the left edge of the inscribed rectangle of the shape that contains the text
MarginRight	Single	Set/Get the distance (in points) between the right edge of the text frame and the right edge of the inscribed rectangle of the shape that contains the text
MarginTop	Single	Set/Get the distance (in points) between the top of the text frame and the top of the inscribed rectangle of the shape that contains the text
Orientation	MsoTextOrientation	Set/Get a value that represents the text frame orientation
PathFormat	MsoPathFormat	Set/Get the path type for the specified text frame
Ruler	Ruler	Read-only. Returns a Ruler object that represents the ruler for the specified text
TextRange	TextRange2	Read-only. Returns the TextRange2 object that represents the text in the object
ThreeD	ThreeDFormat	Read-only. Returns a ThreeDFormat object that contains 3D-effect formatting properties for the specified text
VerticalAnchor	MsoVerticalAnchor	Set/Get the vertical anchor type for the specified text
WarpFormat	MsoWarpFormat	Set/Get the warp type for the specified text frame
WordArtFormat	MsoPresetTextEffect	Set/Get the Word Art type for the specified text frame
WordWrap	Boolean	Set/Get whether text break lines are within or past the boundaries of the shape

ThreeDFormat Methods

Name	Parameters	Description
DeleteText		Deletes the text from a text frame and all the associated text properties

ThreeDFormat Object

The `ThreeDFormat` object contains all of the three-dimensional formatting properties of the parent `Shape` object. The `ThreeD` property of the `Shape` object is used to access the `ThreeDFormat` object.

ThreeDFormat Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ThreeDFormat Properties

Name	Returns	Description
BevelBottomDepth	Single	Set/Get the bottom depth when using the bevel effect on a <code>ThreeDFormat</code> object
BevelBottomInset	Single	Set/Get a value indicating whether the bottom insert bevel should be raised for a <code>ThreeDFormat</code> object
BevelBottomType	MsoBevelType	Set/Get the bottom bevel type for a <code>ThreeDFormat</code> object
BevelTopDepth	Single	Set/Get the top depth when using the bevel effect on a <code>ThreeDFormat</code> object
BevelTopInset	Single	Set/Get a value indicating whether the top insert bevel should be raised for a <code>ThreeDFormat</code> object
BevelTopType	MsoBevelType	Set/Get the top Bevel type for a <code>ThreeDFormat</code> object
ContourColor	ColorFormat	Read-only. Returns the contour color for a <code>ThreeDFormat</code> object
ContourWidth	Single	Set/Get the contour width for a <code>ThreeDFormat</code> object
Depth	Single	Set/Get the depth of a 3D shape
ExtrusionColor	ColorFormat	Read-only. An object manipulating the color of the extrusion
ExtrusionColorType	MsoExtrusionColorType	Set/Get how the color for the extrusion is set

Table continued on following page

ThreeDFormat Methods

Name	Returns	Description
FieldOfView	Single	Set/Get the angle at which a ThreeDFormat object can be viewed
LightAngle	Single	Set/Get the angle of the extrusion lights set on a ThreeDFormat object
Perspective	MsoTriState	Set/Get whether the shape's extrusion has perspective
PresetCamera	MsoPresetCamera	Read-only. Returns the extrusion preset camera for a ThreeDFormat object
Preset Extrusion Direction	MsoPreset Extrusion Direction	Read-only. Returns the direction of the extrusion
PresetLighting	MsoLightRigType	Read-only. Returns the extrusion preset lighting for a ThreeDFormat object
Preset Lighting Direction	MsoPreset Lighting Direction	Set/Get the direction of the light source
Preset Lighting Softness	MsoPreset Lighting Softness	Set/Get the softness of the light source
Preset Material	MsoPreset Material	Set/Get the surface material of the extrusion
PresetThreeD Format	MsoPreset ThreeD Format	Read-only. Returns the preset extrusion format
RotationX	Single	Set/Get how many degrees the extrusion is rotated
RotationY	Single	Set/Get how many degrees the extrusion is rotated
RotationZ	Single	Set/Get how many degrees the extrusion is rotated
Visible	MsoTriState	Set/Get whether the 3D shape is visible
Z	Single	Set/Get the Z order of the specified ThreeDFormat object

ThreeDFormat Methods

Name	Parameters	Description
IncrementRotation Horizontal	Increment As Single	Changes the rotation of the specified shape horizontally by the specified number of degrees
IncrementRotation Vertical	Increment As Single	Changes the rotation of the specified shape vertically by the specified number of degrees

Name	Parameters	Description
Increment RotationX	Increment As Single	Changes the RotationX property
Increment RotationY	Increment As Single	Changes the RotationY property
Increment RotationZ	Increment As Single	Changes the RotationZ property
ResetRotation		Resets RotationX and RotationY to 0
SetExtrusion Direction	Preset Extrusion Direction As MsoPreset Extrusion Direction	Changes the extrusion direction
SetPresetCamera		Sets the camera for the specified ThreeDFormat object
SetThreeD Format	PresetThreeD Format As MsoPreset ThreeDFormat	Sets the preset extrusion format

ThreeDFormat Object Example

```

Sub SetShape3D()
    Dim o3DF As ThreeDFormat
    Dim oShp As Shape
    Set oShp = ActiveSheet.Shapes(1)
    Set o3DF = oShp.ThreeD
    With o3DF
        .Depth = 10
        .SetExtrusionDirection msoExtrusionBottomRight
    End With
End Sub

```

TickLabels Object

The `TickLabels` object contains the formatting options associated with the tick-mark labels for tick marks on a chart axis. The parent of the `TickLabels` object is the `Axis` object.

TickLabels Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

TickLabels Properties

TickLabels Properties

Name	Returns	Description
Alignment	Long	Set/Get the alignment of the tick labels. Use the <code>XlHAlign</code> constants
AutoScaleFont	Variant	Set/Get whether the font size will change automatically if the parent chart changes sizes
Depth	Long	Read-only. Returns how many levels of category tick labels are on the axis
Font	Font	Read-only. Returns an object containing Font options for the tick label text
Format	ChartFormat	Read-only. Returns the <code>ChartFormat</code> object, which controls the line, fill, and effect formatting for the chart area
MultiLevel	Boolean	Set/Get whether an axis is multilevel
Name	String	Read-only. Returns the name of the <code>TickLabels</code> object
NumberFormat	String	Set/Get the numeric formatting to use if the tick labels are numeric values or dates
NumberFormatLinked	Boolean	Set/Get whether the same numerical format used for the cells containing the chart data is used by the tick labels
NumberFormatLocal	Variant	Set/Get the name of the numeric format being used by the tick labels in the language being used by the user
Offset	Long	Set/Get the percentage distance between levels of labels as compared to the axis label's font size
Orientation	XlTickLabelOrientation	Set/Get the angle of the text for the tick labels. The value can be in degrees (from -90 to 90) or one of the <code>XlTickLabelOrientation</code> constants
ReadingOrder	Long	Set/Get how the text is read (from left to right or right to left). Only applicable in appropriate languages

TickLabels Methods

Name	Returns	Description
Delete	Variant	Deletes the tick labels from the axis labels
Select	Variant	Selects the tick labels on the chart

TickLabels Object Example

```

Sub FormatTickLabels ()
    Dim oTL As TickLabels
    Set oTL = ActiveSheet.ChartObjects(1).Chart.Axes(xlValue).TickLabels
    With oTL
        .NumberFormat = "#,##0"
        .Font.Size = 12
    End With
End Sub

```

Top10 Object

The `Top10` object controls the attributes and specifications of a conditional formatting rule that evaluates values in a given scope or range against each other to determine the rank of each value in that scope or range.

Top10 Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Top10 Properties

Name	Returns	Description
<code>AppliesTo</code>	Range	Read-only. Returns the range that is affected by the formatting rule
<code>Borders</code>	Borders	Read-only. Returns a collection that specifies the cell borders for the formatting condition
<code>CalcFor</code>	<code>xlCalcFor</code>	Set/Get the scope of data to be evaluated in a PivotTable report. Use the <code>xlCalcFor</code> constant
<code>Font</code>	Font	Read-only. Specifies the font formatting attributes for the conditional formatting rule
<code>FormatRow</code>	Boolean	Set/Get the Boolean value specifying if the entire Excel table row should be formatted. The default value is <code>False</code>
<code>Interior</code>	Interior	Read-only. Specifies the Interior formatting attributes for the conditional formatting rule
<code>NumberFormat</code>	Variant	Set/Get the number format applied to a cell if the conditional formatting rule evaluates to true
<code>Percent</code>	Boolean	Set/Get whether the ranking of values within the given scope or range is determined by a percentage value
<code>Priority</code>	Long	Set/Get the priority value of a conditional formatting rule, determining the order of evaluation when other rules are in effect

Table continued on following page

Top10 Methods

Name	Returns	Description
PTCondition	Boolean	Read-only. Indicates whether the formatting rule is applied to a PivotTable chart
Rank	Long	Set/Get either the number or percentage rank value for the rule
ScopeType	xlPivotConditionScope	Set/Get the scope of the formatting rule when applied to a PivotTable chart. Use the xlPivotConditionScope constants
StopifTrue	Boolean	Set/Get a Boolean value that determines if additional formatting rules should be applied if the current rule evaluates to True. The default value is True
TopBottom	xlTopBottom	Set/Get whether the ranking is evaluated from the top or the bottom. Use the xlTopBottom constants
Type	xlFormatConditionType	Read-only. Returns an xlFormatConditionType constant that specifies the type of conditional formatting being applied

Top10 Methods

Name	Returns	Parameters	Description
Delete			Deletes the object
ModifyAppliesToRange		Range As Range	Sets the range for which the formatting rule will be applied
SetFirstPriority			Sets the priority value for the formatting rule so that it is evaluated before all other rules on the worksheet
SetLastPriority			Sets the priority value for the formatting rule so that it is evaluated after all other rules on the worksheet

Top10 Object Example

```
Sub CreateTopPercentCondition()  
Dim oFormatCondition As Top10  
  
'Add a new Formatting rule
```

```

Set oFormatCondition = Range("F5:F22").FormatConditions.AddTop10

'Highlight all values that make up the Top 20% of the total range
With oFormatCondition
    .TopBottom = xlTop10Top
    .Rank = 20
    .Percent = True
    .Font.Bold = True
    .Interior.Color = 7039480
End With
End Sub

```

TreeviewControl Object

The `TreeviewControl` object allows manipulation of the hierarchical member-selection of a cube field. This object is usually used by macro recordings and not when building VBA code. The parent of the `TreeviewControl` object is the `CubeField` object.

TreeviewControl Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

TreeviewControl Properties

Name	Returns	Description
Drilled	Variant	Set/Get a string array describing the drilled status of the members of the parent cube field
Hidden	Variant	Set/Get the hidden status of the members in a cube field

Trendline Object and the Trendlines Collection

The `Trendlines` collection holds the collection of trendlines in a chart. Each `Trendline` object describes a trendline on a chart of a particular series. `Trendlines` are used to graphically show trends in the data and help predict future values. The parent of the `Trendlines` collection is the `Series` object.

The `Trendlines` collection has one method besides the typical collection attributes. The `Add` method adds a trendline to the current chart. The `Add` method has a `Type`, `Order`, `Period`, `Forward`, `Backward`, `Intercept`, `DisplayEquation`, `DisplayRSquared`, and `Name` parameter. See the “Trendline Properties” section for more information.

Trendline Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Trendline Properties

Trendline Properties

Name	Returns	Description
Backward	Long	Set/Get how many periods the trendline extends back
Backward2	Double	Set/Get how many periods the trendline extends back
Border	Border	Read-only. Returns the border's properties around the trendline
DataLabel	DataLabel	Read-only. Returns an object to manipulate the trendline's data label
Display Equation	Boolean	Set/Get whether the equation used for the trendline is displayed on the chart
Display RSquared	Boolean	Set/Get whether the R-squared value for the trendline is displayed on the chart
Format	ChartFormat	Returns the <code>ChartFormat</code> object, which controls the line and effect formatting for the trendline
Forward2	Double	Set/Get how many periods the trendline extends forward
Index	Long	Read-only. Returns the spot in the collection where the current object is
Intercept	Double	Set/Get at which point the trendline crosses the value (y) axis
InterceptIs Auto	Boolean	Set/Get whether the point at which the trendline crosses the value axis is automatically calculated with regression
Name	String	Set/Get the name of the <code>Trendline</code> object
NameIsAuto	Boolean	Set/Get whether Excel automatically chooses the trendline name
Order	Long	Set/Get the order of a polynomial trendline. The <code>Type</code> property must be <code>x1Polynomial</code>
Period	Long	Set/Get what the period is for the moving-average trendline
Type	<code>XlTrendline Type</code>	Set/Get the type of the trendline (for example, <code>x1Exponential</code> , <code>x1Linear</code> , and so on)

Trendline Methods

Name	Returns	Description
ClearFormats	Variant	Clears any formatting made on the trendlines
Delete	Variant	Deletes the trendlines
Select	Variant	Selects the trendlines on the chart

Trendline Object and the Trendlines Collection Example

```

Sub AddTrendLine()
    Dim oSer As Series
    Dim oTL As Trendline
    Set oSer = ActiveSheet.ChartObjects(1).Chart.SeriesCollection(1)
    Set oTL = oSer.Trendlines.Add(xlLinear)
    With oTL
        .DisplayEquation = True
        .DisplayRSquared = True
    End With
End Sub

```

UniqueValues Object

The `UniqueValues` object controls the attributes and specifications of a conditional formatting rule that evaluates whether a value is unique or is a duplicate based on a given scope or range of values.

UniqueValues Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

UniqueValues Properties

Name	Returns	Description
<code>AppliesTo</code>	Range	Read-only. Returns the range that is affected by the formatting rule
<code>Borders</code>	Borders	Read-only. Returns a collection that specifies the cell borders for the formatting condition
<code>DupeUnique</code>	<code>xlDupeUnique</code>	Set/Get whether the formatting condition will evaluate for unique values or duplicate values. Use the <code>xlDupeUnique</code> constants
<code>Font</code>	Font	Read-only. Specifies the font formatting attributes for the conditional formatting rule
<code>FormatRow</code>	Boolean	Set/Get the Boolean value specifying if the entire Excel table row should be formatted. The default value is <code>False</code>
<code>Interior</code>	Interior	Read-only. Specifies the Interior formatting attributes for the conditional formatting rule
<code>NumberFormat</code>	Variant	Set/Get the number format applied to a cell if the conditional formatting rule evaluates to true
<code>Priority</code>	Long	Set/Get the priority value of a conditional formatting rule, determining the order of evaluation when other rules are in effect

Table continued on following page

UniqueValues Methods

Name	Returns	Description
PTCondition	Boolean	Read-only. Indicates whether the formatting rule is applied to a PivotTable chart
ScopeType	xlPivotConditionScope	Set/Get the scope of the formatting rule when applied to a PivotTable chart. Use the xlPivotConditionScope constants
StopifTrue	Boolean	Set/Get a Boolean value that determines if additional formatting rules should be applied if the current rule evaluates to True. The default value is True
Type	xlFormatConditionType	Read-only. Returns an xlFormatConditionType constant that specifies the type of conditional formatting being applied

UniqueValues Methods

Name	Parameters	Description
Delete		Deletes the object
ModifyAppliesToRange	Range As Range	Sets the range for which the formatting rule will be applied
SetFirstPriority		Sets the priority value for the formatting rule so that it is evaluated before all other rules on the worksheet
SetLastPriority		Sets the priority value for the formatting rule so that it is evaluated after all other rules on the worksheet

UniqueValues Object Example

```
Sub CreateDuplicateValuesCondition()  
Dim oFormatCondition As UniqueValues  
  
'Add a new Formatting rule  
Set oFormatCondition = Range("A1:A21").FormatConditions.AddUniqueValues  
  
'Find and highlight all duplicate values in the range  
With oFormatCondition  
    .DupeUnique = xlDuplicate  
    .Font.Bold = True  
    .Interior.Color = 7039480  
End With  
End Sub
```

UpBars Object

The UpBars object contains formatting options for up bars on a chart. The parent of the UpBars object is the ChartGroup object. To see if this object exists, use the HasUpDownBars property of the ChartGroup object.

UpBars Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

UpBars Properties

Name	Returns	Description
<code>Format</code>	<code>ChartFormat</code>	Read-only. Returns the <code>ChartFormat</code> object, which controls the line, fill, and effect formatting for the chart area
<code>Name</code>	<code>String</code>	Read-only. Returns the name of the up bars

UpBars Methods

Name	Returns	Description
<code>Delete</code>	<code>Variant</code>	Deletes the up bars
<code>Select</code>	<code>Variant</code>	Selects the up bars in the chart

UsedObjects Collection Object

The `UsedObjects` collection represents the total number of objects currently being used in all open workbooks. Used objects can be worksheets, chart sheets, the workbook itself, and any ActiveX controls placed on worksheets. This object can be referenced through the `Application` object. Note that the `UsedObjects` collection object has no properties or methods outside the typical collection attributes listed at the beginning of this appendix.

UserAccess Collection Object

Represents one user within a possible group of users who have permission to access a range specified by the `AllowEditRange` object. You can refer to a user by using the `Item` property of the `UserAccessList` object. Once referenced, you use the properties of this object to change the user's settings.

UserAccess Collection Properties

Name	Returns	Description
<code>AllowEdit</code>	<code>Boolean</code>	Set/Get whether the user is allowed access to the specified range on a protected worksheet
<code>Name</code>	<code>String</code>	Read-only. Returns the name of the <code>UserAccess</code> object

UserAccess Collection Methods

UserAccess Collection Methods

Name	Description
Delete	Deletes the object

UserAccessList Collection Object

Represents a list of users who have access to a protected range on a worksheet. This object can be accessed via the `AllowEditRange` object after it has been created. Use the `Add` method of this object to add a user to the list, which contains an argument that determines whether or not they need a password to access the range.

Note that the password is set using the `ChangePassword` method of the `AllowEditRange` object. This means that all of the users for an `AllowEditRange` use the same password. Note that this collection only has `Count` and `Item` properties.

UserAccessList Methods

Name	Returns	Parameters	Description
Add	UserAccess	Name As String, AllowEdit As Boolean	Adds a user access list to the collection. Name is the name of the list, and if <code>AllowEdit</code> is <code>True</code> , users on the access list are allowed to edit the editable ranges on a protected worksheet
DeleteAll			Removes all users associated with access to a protected range on a worksheet

UserAccessList Object Example

The following routine loops through all of the `AllowEditRange` objects on a specified worksheet and removes all of the users except for the range `pcNetSales`:

```
Sub DeleteAllUsers()  
Dim oAllowRange As AllowEditRange  
  
    'Loop through all AllowEditRange objects on the active sheet  
    For Each oAllowRange In ActiveSheet.Protection.AllowEditRanges  
  
        'Remove all names from all AllowEditRanges  
        'except for the range whose AllowEditRange Title is pcNetSales  
        If oAllowRange.Title <> "pcNetSales" Then  
            oAllowRange.Users.DeleteAll  
        End If  
    Next oAllowRange  
End Sub
```

Validation Object

The `Validation` object contains properties and methods to represent validation for a range in a worksheet. The `Range` object is the parent of the `Validation` object.

Validation Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Validation Properties

Name	Returns	Description
<code>AlertStyle</code>	Long	Read-only. Returns how the user will be alerted if the range includes invalid data. Uses the <code>xlDVALertStyle</code> constants
<code>ErrorMessage</code>	String	Set/Get the error message to show for data validation
<code>ErrorTitle</code>	String	Set/Get what the title is for the error data validation dialog box
<code>Formula1</code>	String	Read-only. Returns the value, cell reference, or formula used for data validation
<code>Formula2</code>	String	Read-only. Returns the second part of the value, cell reference, or formula used for data validation. The <code>Operator</code> property must be <code>xlBetween</code> or <code>xlNotBetween</code>
<code>IgnoreBlank</code>	Boolean	Set/Get whether a blank cell is always considered valid
<code>IMEMode</code>	Long	Set/Get how the Japanese input rules are described. Use the <code>xlIMEMode</code> constants
<code>InCell Dropdown</code>	Boolean	Set/Get whether a drop-down list of valid values is displayed in the parent range. Used when the <code>Type</code> property is <code>xlValidateList</code>
<code>InputMessage</code>	String	Set/Get the validation input message to prompt the user for valid data
<code>InputTitle</code>	String	Set/Get what the title is for the input data validation dialog box
<code>Operator</code>	Long	Read-only. Returns the operator describing how <code>Formula1</code> and <code>Formula2</code> are used for validation. Uses the <code>xlFormat-ConditionOperator</code> constants
<code>ShowError</code>	Boolean	Set/Get whether the error message will be displayed when invalid data is entered in the parent range
<code>ShowInput</code>	Boolean	Set/Get whether the input message will be displayed when the user chooses one of the cells in the parent range
<code>Type</code>	Long	Read-only. Returns the data validation type for the range. The <code>xlDVType</code> constants can be used (for example, <code>xlValidateDecimal</code> , <code>xlValidateTime</code>)
<code>Value</code>	Boolean	Read-only. Returns if the validation is fulfilled for the range

Validation Methods

Name	Returns	Parameters	Description
Add		Type As XlDVType, [Alert Style], [Operator], [Formula1], [Formula2]	Adds data validation to the parent range. The validation type (Type parameter) must be specified. The type of validation alert (AlertStyle) can be specified with the XlDValertStyle constants. The Operator parameter uses the XlFormatCondition Operator to pick the type of operator to use. The Formula1 and Formula2 parameters pick the data validation formula
Delete			Deletes the validation method for the range
Modify		[Type], [AlertStyle], [Operator], [Formula1], [Formula2]	Modifies the properties associated with the validation. See the properties of the validation object for a description of the parameters

Validation Object Example

```
Sub AddValidation()  
    Dim oValid As Validation  
    Set oValid = Selection.Validation  
    With oValid  
        .Delete  
        .Add Type:=xlValidateWholeNumber, AlertStyle:=xlValidAlertStop, _  
            Operator:=xlBetween, Formula1:="10", Formula2:="20"  
        .ShowInput = False  
        .ShowError = True  
        .ErrorTitle = "Error"  
        .ErrorMessage = "Number must be between 10 and 20"  
    End With  
End Sub
```

VPageBreak Object and the VPageBreaks Collection

The `VPageBreaks` collection contains all of the vertical page breaks in the printable area of the parent object. Each `VPageBreak` object represents a single vertical page break for the printable area of the parent object. Possible parents of the `VPageBreaks` collection are the `Worksheet` and `Chart` objects.

The `VPageBreaks` collection contains one method besides the typical collection attributes. The `Add` method is used to add a `VPageBreak` object to the collection (and vertical page break to the sheet). The `Add` method has a `Before` parameter to specify the range to the right of where the vertical page break will be added.

VPageBreak Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

VPageBreak Properties

Name	Returns	Description
Extent	XlPageBreak Extent	Read-only. Returns whether the vertical page break is full screen or only for the print area
Location	Range	Set/Get the cell where the vertical page break is located. The left edge of the cell is the location of the page break
Type	XlPageBreak	Set/Get whether the page break is automatic or manually set

VPageBreak Methods

Name	Parameters	Description
Delete		Deletes the page break
DragOff	Direction As XlDirection, RegionIndex As Long	Drags the page break out of the printable area. The Direction parameter specifies the direction the page break is dragged. The RegionIndex parameter specifies which print region the page break is being dragged out of

VPageBreak Object and the VPageBreaks Collection Example

```
Sub AddVPageBreaks()
    Dim oCell As Range
    'Loop through all the cells in the first column of the sheet
    For Each oCell In ActiveSheet.UsedRange.Rows(1).Cells
        'If the font size is 16, add a page break to the left of the cell
        If oCell.Font.Size = 16 Then
            ActiveSheet.VPageBreaks.Add oCell
        End If
    Next
End Sub
```

Walls Object

The Walls object contains formatting options for all the walls of a 3D chart. The walls of a 3D chart cannot be accessed individually. The parent of the Walls object is the Chart object.

Walls Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

Walls Properties

Walls Properties

Name	Returns	Description
Format	ChartFormat	Read-only. Returns the ChartFormat object, which controls the line, fill, and effect formatting for the chart area
Name	String	Read-only. Returns the name of the Walls object
PictureType	Variant	Set/Get how an associated picture is displayed on the walls of the 3D chart (for example, stretched, tiled). Use the xlPictureType constants
PictureUnit	Variant	Set/Get how many units a picture represents if the PictureType property is set to xlScale
Thickness	Long	Set/Get the thickness of the wall. Default is 0

Walls Methods

Name	Returns	Description
ClearFormats	Variant	Clears the formatting made on the Walls object
Paste		Pastes a picture from the clipboard
Select	Variant	Selects the walls on the parent chart

Walls Object Example

```
Sub FormatWalls()  
    Dim oWall As Walls  
    Set oWall = ActiveSheet.ChartObjects("Chart 1").Chart.Walls  
    With oWall  
        .Fill.PresetTextured msoTextureCork  
        .Fill.Visible = True  
    End With  
End Sub
```

Watch Object and the Watches Collection Object

The `Watch` object represents one Watch in the Watch window (found on the Formulas tab in the Excel interface). Each Watch can be a cell or cell range you need to keep track of as other data on the worksheet changes. A Watch object is an auditing tool similar to the watches you can create in the VBE. Watches do just that: They keep track of a cell or cell range, allowing you to study changes to those cells when other data on the worksheet changes.

The `Watches` collection contains all the `Watch` objects that have been set in the application.

Watches Collection Methods

Name	Returns	Parameters	Description
Add	Watch	Source As Variant	Adds a range that is tracked when the worksheet is recalculated
Delete			Deletes the object

Watch Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Watch Properties

Name	Returns	Description
Source	Variant	Read-only. Returns the unique name that identifies items that have a <code>SourceType</code> property value of <code>xlSourceRange</code> , <code>xlSourceChart</code> , <code>xlSourcePrintArea</code> , <code>xlSourceAutoFilter</code> , <code>xlSourcePivotTable</code> , or <code>xlSourceQuery</code>

Watch Methods

Name	Description
Delete	Deletes the object

Watch Object Example

The following routine prompts the user for a range, then loops through each cell in the range and adds it to the Watch window. It then displays the Watch window:

```
Sub AddWatches()
    Dim oWatch As Watch
    Dim rng As Range
    Dim rngWatches As Range

    'Prompt the user for a range
    'Suppress the error if they cancel
    On Error Resume Next
        Set rngWatches = Application.InputBox( _
            "Please select a cell or cell range to watch", "Add Watch", , , , , 8)
    On Error GoTo 0

    'If they selected a range
    If Not rngWatches Is Nothing Then
        'Loop through each cell and
        ' add it to the watch list
        For Each rng In rngWatches
            Application.Watches.Add rng
        Next rng
    End If
End Sub
```

WebOptions Object

```
Next rng
End If
'View the watch window based on their answer
Application.CommandBars("Watch Window").Visible = (Not rngWatches Is Nothing)

End Sub
```

WebOptions Object

The `WebOptions` object contains attributes associated with opening or saving web pages. The parent of the `WebOptions` object is the `Workbook` object. The properties set in the `WebOptions` object override the settings of the `DefaultWebOptions` object.

WebOptions Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

WebOptions Properties

Name	Returns	Description
<code>AllowPNG</code>	Boolean	Set/Get whether Portable Network Graphics Format (PNG) is allowed as an output format. PNG is a file format for the lossless, portable, well-compressed storage of images
<code>DownloadComponents</code>	Boolean	Set/Get whether Office components are downloaded to the end user's machine when viewing Excel files in a web browser
<code>Encoding</code>	<code>MsoEncoding</code>	Set/Get the type of code page or character set to save with a document
<code>FolderSuffix</code>	String	Read-only. Returns the suffix name for the support directory created when saving an Excel document as a web page. Language dependent
<code>LocationOfComponents</code>	String	Set/Get the URL or path that contains the Office Web components needed to view documents in a web browser
<code>OrganizeInFolder</code>	Boolean	Set/Get whether supporting files are organized in a separate folder from the document
<code>PixelsPerInch</code>	Long	Set/Get how dense graphics and table cells should be when viewed on a web page
<code>RelyOnCSS</code>	Boolean	Set/Get whether Cascading Style Sheets (CSS) is used for font formatting
<code>RelyOnVML</code>	Boolean	Set/Get whether image files are not created when saving a document with drawn objects. Vector Markup Language is used to create the images on the fly. VML is an XML-based format for high-quality vector graphics on the web

Name	Returns	Description
ScreenSize	MsoScreen Size	Set/Get the target monitor's screen size
Target Browser	MsoTarget Browser	Set/Get the browser version
UseLongFile Names	Boolean	Set/Get whether links are updated every time the document is saved

WebOptions Methods

Name	Description
UseDefault FolderSuffix	Tells Excel to use its default naming scheme for creating supporting folders

WebOptions Object Example

```
Sub SetWebOptions()
    Dim oWO As WebOptions
    Set oWO = ActiveWorkbook.WebOptions
    With oWO
        .ScreenSize = msoScreenSize800x600
        .RelyOnCSS = True
        .UseDefaultFolderSuffix
    End With
End Sub
```

Window Object and the Windows Collection

The `Windows` collection holds the list of windows used in Excel or in a workbook. Each `Window` object represents a single Excel window containing scrollbars and gridlines for the window. The parents of the `Windows` collection can be the `Application` object and the `Workbook` object.

The `Windows` collection has several properties and methods outside the typical collection attributes.

Windows Collection Properties and Methods

Name	Returns	Description
Item	Window	Read-only. Returns a <code>Window</code> object from the <code>Windows</code> collection based on a given <code>Index</code> parameter specifying an index number or a name
SynchScrolling SideBySide	Boolean	Set/Get whether the contents of multiple windows can be simultaneously scrolled when documents are being compared side by side

Table continued on following page

Window Common Properties

Name	Returns	Description
Arrange		Method used to arrange the windows on the screen. Parameters: [ArrangeStyle] As XlArrangeStyle, [ActiveWorkbook], [SynchHorizontal], [SynchVertical]
BreakSideBySide	Boolean	Method used to end Side-by-Side mode. Returns a Boolean value indicating whether the operation was successful
CompareSideBySideWith	Boolean	Method used to open two windows in Side-by-Side mode. Note that you cannot use this method with the Application object or the ActiveWorkbook property. This method takes one parameter: [WindowName] As Variant
ResetPositionsSideBySide		Method used to reset the position of two worksheet windows that are being compared side by side

Window Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

Window Properties

Name	Returns	Description
ActiveCell	Range	Read-only. Returns the cell in the window where the cursor is
ActiveChart	Chart	Read-only. Returns the currently selected chart in the window. If no chart is currently selected, nothing is returned
ActivePane	Pane	Read-only. Returns the active pane in the window
ActiveSheet	Object	Read-only. Returns the active sheet in the window
ActiveSheetView	Object	Read-only. Returns a view of the active sheet in the window
AutoFilterDataGrouping	Boolean	Set/Get whether the auto filter for date grouping is currently displayed in the specified window
Caption	Variant	Set/Get the caption that appears in the window
DisplayFormulas	Boolean	Set/Get whether formulas are displayed in the window. Not valid in a Chart sheet
DisplayGridlines	Boolean	Set/Get whether worksheet gridlines are displayed

Name	Returns	Description
DisplayHeadings	Boolean	Set/Get whether row and column headings are displayed. Not valid in a Chart sheet
DisplayHorizontalScrollBar	Boolean	Set/Get whether the horizontal scrollbar is displayed in the window
DisplayOutline	Boolean	Set/Get whether outline symbols are displayed
DisplayRightToLeft	Boolean	Set/Get whether the window contents are displayed from right to left. Valid only with languages that support right-to-left text
DisplayRuler	Boolean	Set/Get whether a ruler is displayed for the specified window
DisplayVerticalScrollBar	Boolean	Set/Get whether the vertical scrollbar is displayed in the window
DisplayWhitespace	Boolean	Set/Get whether whitespace is displayed
DisplayWorkbookTabs	Boolean	Set/Get whether workbook tabs are displayed
DisplayZeros	Boolean	Set/Get whether zero values are displayed. Not valid with Chart sheets
EnableResize	Boolean	Set/Get whether a user can resize the window
FreezePanels	Boolean	Set/Get whether split panes are frozen. Not valid with Chart sheets
GridlineColor	Long	Set/Get the color of the gridlines. Use the RGB function to create the color value
GridlineColorIndex	XIColorIndex	Set/Get the color of the gridlines. Use the XIColorIndex constants or an index value in the current color palette
Height	Double	Set/Get the height of the window
Index	Long	Read-only. Returns the spot in the collection where the current object is located
Left	Double	Set/Get the distance from the left edge of the client area to the window's left edge
OnWindow	String	Set/Get the name of the procedure to run whenever a window is activated

Table continued on following page

Window Properties

Name	Returns	Description
Panes	Panes	Read-only. Returns the panes that are contained in the window
Range Selection	Range	Read-only. Returns the selected range of cells or objects in the window
ScrollColumn	Long	Set/Get the column number of the leftmost column in the window
ScrollRow	Long	Set/Get the row number of the topmost row in the window
Selected Sheets	Sheets	Read-only. Returns all the selected sheets in the window
Selection	Object	Read-only. Returns the selected object in the window
SheetViews	SheetViews	Read-only. Returns the SheetViews object for a given window
Split	Boolean	Set/Get whether the window is split into panes
SplitColumn	Long	Set/Get at which column number the window split is going to be located
Split Horizontal	Double	Set/Get where the horizontal split of window will be located, in points
SplitRow	Long	Set/Get at which row number the window split is going to be located
SplitVertical	Double	Set/Get where the vertical split of window will be located, in points
TabRatio	Double	Set/Get how big a workbook's tab is, as a ratio of a workbook's tab area width to the window's horizontal scrollbar width
Top	Double	Set/Get the distance from the top edge of the client area to the window's top edge
Type	XlWindow Type	Read-only. Returns the window type
UsableHeight	Double	Read-only. Returns the maximum height that the window can be
UsableWidth	Double	Read-only. Returns the maximum width that the window can be
View	XlWindow View	Set/Get the view in the window (for example, xlNormalView and xlPageBreakPreview)
Visible	Boolean	Set/Get whether the window is visible

Name	Returns	Description
VisibleRange	Range	Read-only. Returns the range of cells that are visible in the current window
Width	Double	Set/Get the width of the window
WindowNumber	Long	Read-only. Returns the number associated with a window. Typically used when the same workbook is opened twice (for example, <code>MyBook.xlsx:1</code> and <code>MyBook.xlsx:2</code>)
WindowState	XlWindowState	Set/Get the state of the window: minimized, maximized, or normal
Zoom	Variant	Set/Get the percentage of window zoom

Window Methods

Name	Returns	Parameters	Description
Activate	Variant		Sets focus to the window
ActivateNext	Variant		Activates the next window in the z-order
ActivatePrevious	Variant		Activates the previous window in the z-order
Close	Boolean	[SaveChanges], [Filename], [RouteWorkbook]	Closes the window. Set <code>SaveChanges</code> to <code>True</code> to automatically save changes in the window's workbook. If <code>SaveChanges</code> is <code>False</code> , then all changes are lost. The <code>Filename</code> parameter can be used to specify the filename to save to. <code>RouteWorkbook</code> is used to automatically route the workbook onto the next recipient, if applicable
LargeScroll	Variant	[Down], [Up], [ToRight], [ToLeft]	Causes the document to scroll a certain direction a screenful at a time, as specified by the parameters
NewWindow	Window		Creates and returns a new window

Table continued on following page

Window Methods

Name	Returns	Parameters	Description
PointsToScreenPixelsX	Long	Points As Long	Converts the horizontal document coordinate Points parameter to screen coordinate pixels
PointsToScreenPixelsY	Long	Points As Long	Converts the vertical document coordinate Points parameter to screen coordinate pixels
PrintOut	Variant	[From], [To], [Copies], [Preview], [ActivePrinter], [PrintToFile], [Collate], [PrToFileName]	Prints out the document in the window. The printer, number of copies, collation, and whether a print preview is desired can be specified with the parameters. Also, the sheets can be printed to a file using the PrintToFile and PrToFileName parameters. The From and To parameters can be used to specify the range of printed pages
PrintPreview	Variant	[EnableChanges]	Displays the current workbook in the window in a print preview mode. Set the EnableChanges parameter to False to disable the Margins and Setup buttons, hence not allowing the viewer to modify the page setup
RangeFromPoint	Object	x As Long, y As Long	Returns the shape or range located at the x and y coordinates. Returns nothing if there is no object at the x, y coordinates
ScrollIntoView		Left As Long, Top As Long, Width As Long, Height As Long, [Start]	Scrolls the spot specified by the Left, Top, Width, and Height parameters to either the upper-left corner of the window (Start = True) or the lower-right corner of the window (Start = False). The Left, Top, Width, and Height parameters are specified in points
ScrollWorkbookTabs	Variant	[Sheets], [Position]	Scrolls through the number of sheets specified by the Sheets parameter, or goes to the sheet specified by the position parameter (xlFirst or xlLast)

Window Object and the Windows Collection Example

Name	Returns	Parameters	Description
SmallScroll	Variant	[Down], [Up], [ToRight], [ToLeft]	Causes the document to scroll a certain direction one document line at a time, as specified by the parameters

Window Object and the Windows Collection Example

```
Sub MinimizeAllWindows()  
    Dim oWin As Window  
    For Each oWin In Windows  
        'Minimize all workbooks that are not hidden  
        If oWin.Visible = True Then  
            oWin.WindowState = xlMinimized  
        End If  
    Next  
End Sub
```

Workbook Object and the Workbooks Collection

The `Workbooks` collection contains the list of open workbooks. A `Workbook` object represents a single workbook. The parent of the `Workbook` is the `Application` object.

Workbooks Properties

Name	Returns	Description
Item	Workbook	Read-only. Returns a single <code>Workbook</code> object from the <code>Workbooks</code> collection based on a given <code>Index</code> parameter specifying an index number or a name

Workbooks Methods

Name	Returns	Parameters	Description
Add	Workbook	[Template]	Adds a new workbook to the collection. Using a template name in the <code>Template</code> parameter can specify a template. Also, the <code>XlWBATemplate</code> constants can be used to open up a type of workbook
CanCheckOut	Boolean	Filename As String	Returns whether Excel can check out a specified workbook from a server

Table continued on following page

Workbooks Methods

Name	Returns	Parameters	Description
Checkout		Filename As String	Returns a specified workbook from a server for editing
Close			Closes the workbook
Open	Workbook	Filename As String, [UpdateLinks], [ReadOnly], [Format], [Password], [WriteRes Password], [IgnoreRead-only Recommended], [Origin], [Delimiter], [Editable], [Notify], [Converter], [AddToMru], [Local], [CorruptLoad]	<p>Opens a workbook specified by the <code>Filename</code> parameter and adds it to the collection. Use the <code>UpdateLinks</code> parameter to choose how links in the file are updated. Set <code>ReadOnly</code> to <code>True</code> to open up the workbook in read-only mode. If the file requires a password, use the <code>Password</code> or <code>WriteResPassword</code> parameters. Set <code>AddToMru</code> to <code>True</code> to add the opening workbook to the recently used files list.</p> <p>If the file to open is a delimited text file, then there are some parameters that can be used. Use the <code>Format</code> parameter to choose the text delimiter character if opening a text file. Use the <code>Origin</code> parameter to choose the code page style of the incoming delimited text file. Use the <code>Delimiter</code> parameter to specify a delimiter if 6 (custom) was chosen for the <code>Format</code> parameter</p>
OpenDatabase	Workbook	Filename As String, [CommandText], [CommandType], [Background Query], [ImportData As]	Returns a <code>Workbook</code> representing a database specified by the <code>Filename</code> parameter. The <code>CommandText</code> and <code>CommandType</code> parameters set the text and the type of the query

Name	Returns	Parameters	Description
OpenText		Filename As String, [Origin], [StartRow], [DataType], [TextQualifier], [ConsecutiveDelimiter], [Tab], [Semicolon], [Comma], [Space], [Other], [OtherChar], [FieldInfo], [TextVisualLayout], [DecimalSeparator], [ThousandsSeparator], [TrailingMinusNumbers], [Local]	Opens the text file in <code>Filename</code> and parses it into a sheet on a new workbook. <code>Origin</code> is used to choose the code page style of the file (XlPlatform constant). <code>StartRow</code> decides the first row to parse. <code>DataType</code> decides if the file is <code>xlDelimited</code> or <code>xlFixedWidth</code> . Set <code>ConsecutiveDelimiter</code> to <code>True</code> to treat consecutive delimiters as one. Set <code>Tab</code> , <code>Semicolon</code> , <code>Comma</code> , <code>Space</code> , or <code>Other</code> to <code>True</code> to pick the delimiter character. Use the <code>DecimalSeparator</code> and <code>ThousandsSeparator</code> to pick the numeric characters to use
OpenXML	Workbook	Filename As String, [Stylesheets], [LoadOption]	Returns an XML file in Microsoft Excel. Use the <code>Stylesheets</code> parameter to specify which XSLT stylesheet processing instructions to apply

Workbook Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Workbook Properties

Name	Returns	Description
ActiveChart	Chart	Read-only. Returns the active chart in the workbook
ActiveSheet	Object	Read-only. Returns the active sheet (chart or workbook) in the workbook
AutoUpdateFrequency	Long	Set/Get how often a shared workbook is updated automatically, in minutes
AutoUpdateSaveChanges	Boolean	Set/Get whether changes made to a shared workbook are visible to other users whenever the workbook is automatically updated

Table continued on following page

Workbook Properties

Name	Returns	Description
Builtin Document Properties	Document Properties	Read-only. Returns a collection holding all the built-in properties of the workbook. Things like title, subject, author, and number of words in the workbook can be accessed from this object
Calculation Version	Long	Read-only. Returns the version number of Excel that was last used to recalculate the Excel spreadsheet
ChangeHistory Duration	Long	Set/Get how far back, in days, a shared workbook's change history is visible
Charts	Sheets	Read-only. Returns the charts in the workbook
Check Compatibility	Boolean	Set/Get whether the compatibility checker runs automatically when the workbook is saved
CodeName	String	Read-only. Returns the name of the workbook that was set at design time in the VBE
Colors	Variant	Parameters: [Index]. Set/Get the color palette colors for the workbook. There are 56 possible colors in the palette
CommandBars	CommandBars	Read-only. Returns an object to manipulate the command bars in Excel
Conflict Resolution	XlSave Conflict Resolution	Set/Get how shared workbook conflicts are resolved when they are being updated (for example, xlLocalSessionChanges means that the local user's changes are always accepted)
Connections		Read-only. Establishes a connection between the workbook and an external datasource, refreshing the data without prompting the user
Connections Disabled	Boolean	Specifies whether the connections between the workbook and external datasources are disabled
Container	Object	Read-only. Returns the object that contains the workbook, if applicable
ContentType Properties	Meta Properties	Read-only. Returns a MetaProperties collection that describes the metadata stored in the workbook
CreateBackup	Boolean	Read-only. Returns whether a backup file is created whenever the workbook is saved
Custom Document Properties	Document Properties	Read-only. Returns a collection holding all the user-defined properties of the workbook
CustomViews	CustomViews	Read-only. Returns the collection of custom views in a workbook
CustomXMLParts	Custom XMLParts	Read-only. Returns a CustomXMLParts collection that represents the custom XML in the XML data store

Name	Returns	Description
Date1904	Boolean	Set/Get whether the 1904 date system is used in the workbook
Default PivotTableStyle	TableStyle	Set/Get the TableStyle that is to be used as the default style for PivotTables
Default TableStyle	TableStyle	Set/Get the TableStyle that is to be used as the default style for tables
Display Drawing Objects	xlDisplay Drawing Objects	Set/Get if shapes are displayed, placeholders are displayed, or shapes are hidden
DisplayInk Comments	Boolean	Set/Get whether ink comments are displayed in the workbook
Document Inspectors	Document Inspectors	Read-only. Returns a DocumentInspectors collection that represents the Document Inspector modules for the specified workbook
DocumentLibrary Versions	Document Library Versions	Read-only. Returns a collection that represents the versions of a shared workbook that has versioning enabled and that is stored in a document library on a server
DoNotPrompt ForConvert	Boolean	Set/Get whether a user should be prompted to convert the workbook if the workbook contains features not supported by previous versions of Excel
EnableAuto Recover	Boolean	Set/Get whether the option to save changed files, of all formats, on a timed interval is switched on
Encryption Provider	String	Set/Get the name of the algorithm encryption provider used to encrypt workbooks
Envelope Visible	Boolean	Set/Get whether the envelope toolbar and e-mail composition header are visible
Excel4Intl MacroSheets	Sheets	Read-only. Returns the collection of Excel 4.0 international macro sheets in the workbook
Excel4Macro Sheets	Sheets	Read-only. Returns the collection of Excel 4.0 macro sheets in the workbook
Excel8 Compatibility Mode	Boolean	Read-only. Returns whether the workbook is in compatibility mode
FileFormat	XlFile Format	Read-only. Returns the file format of the workbook
Final	Boolean	Set/Get indicator tagging the workbook as final

Table continued on following page

Workbook Properties

Name	Returns	Description
ForceFull	Boolean	Determines whether full calculations are performed Calculation
FullName	String	Read-only. Returns the path and filename of the workbook
FullNameURL Encoded	String	Read-only. Returns the name of the object, including its path on disk, as a string
HasPassword	Boolean	Read-only. Returns whether the workbook has a protection password
HasRouting Slip	Boolean	Set/Get whether the workbook has a routing slip. Use with the <code>RoutingSlip</code> object
HasVBProject	Boolean	Read-only. Returns whether the workbook has an attached VBA project
Highlight Changes OnScreen	Boolean	Set/Get whether changes in a shared workbook are visibly highlighted
IconSets	IconSets	Read-only. Used to filter data in a workbook based on a cell icon from the <code>IconSet</code> collection
InactiveList Border	Boolean	Set/Get whether list borders are visible when a list is not active
IsAddin	Boolean	Set/Get whether the current workbook is running as an Add-In
IsInplace	Boolean	Read-only. Returns whether the workbook is being edited as an object (<code>True</code>) or in Microsoft Excel (<code>False</code>)
KeepChange History	Boolean	Set/Get whether changes are tracked in a shared workbook
ListChangesOn NewSheet	Boolean	Set/Get whether a separate worksheet is used to display changes of a shared workbook
Mailer	Mailer	Read-only. Returns a <code>Mailer</code> object
MultiUser Editing	Boolean	Read-only. Returns whether a workbook is being shared
Name	String	Read-only. Returns the filename of the workbook
Names	Names	Read-only. Returns the collection of named ranges in the workbook
Password	String	Set/Get the password that must be supplied to open the specified workbook
Password Encryption Algorithm	String	Read-only. Returns the algorithm used by Excel to encrypt passwords for the specified workbook

Name	Returns	Description
Password EncryptionFile Properties	Boolean	Read-only. Returns whether Excel encrypts file properties for the specified password-protected workbook
Password EncryptionKey Length	Long	Read-only. Returns the key length of the algorithm that Excel uses when encrypting passwords for the specified workbook
Password Encryption Provider	String	Read-only. Returns the name of the algorithm encryption provider that Excel uses when encrypting passwords for the specified workbook
Path	String	Read-only. Returns the file path of the workbook
Path doubled	String	Read-only. Returns the file path of the workbook
Permission	Permission	Read-only. Represents the permission settings for the workbook
PersonalView ListSettings	Boolean	Set/Get whether the filter and sort settings for lists are included in the user's personal view of the shared workbook
PersonalView PrintSettings	Boolean	Set/Get whether a user's view of the workbook includes print settings
PrecisionAs Displayed	Boolean	Set/Get whether the precision of numbers in the workbook are as displayed in the cells. Used for calculations
Protect Structure	Boolean	Read-only. Returns whether the sheet order cannot be changed in the workbook
Protect Windows	Boolean	Read-only. Returns whether the workbook windows are protected
Publish Objects	Publish Objects	Read-only. Returns access to an object used to publish objects in the workbook as web pages
ReadOnly	Boolean	Read-only. Returns whether the workbook is in read-only mode
ReadOnly Recommended	Boolean	Read-only. Returns whether the user is prompted with a message recommending that you open the workbook as read-only
Remove Personal Information	Boolean	Set/Get whether personal information can be removed from the specified workbook
Research	Research	Read-only. Represents the Research service for the workbook

Table continued on following page

Workbook Properties

Name	Returns	Description
Revision Number	Long	Read-only. Returns how many times a shared workbook has been saved while open
Routed	Boolean	Read-only. Returns whether a workbook has been routed to the next recipient
RoutingSlip	RoutingSlip	Read-only. Returns access to a <code>RoutingSlip</code> object that can be used to add a routing slip for the workbook. Use with the <code>HasRoutingSlip</code> property
Saved	Boolean	Set/Get whether a workbook does not have changes that need saving
SaveLink Values	Boolean	Set/Get whether values linked from external sources are saved with the workbook
ServerPolicy	Server Policy	Read-only. Represents a policy specified for a workbook stored on a server running Office SharePoint Server
ServerViewable Items	Server Viewable Items	Read-only. Returns a collection of various Excel objects with which a developer can interact through Excel Services
SharedWorkspace	Shared Workspace	Read-only. Represents the Document Workspace in which a specified document is located
Sheets	Sheets	Read-only. Returns the collection of sheets in a workbook (Chart or Worksheet)
ShowConflict History	Boolean	Set/Get whether the sheet containing conflicts related to shared workbooks is displayed
ShowPivotChart ActiveFields	Boolean	Set/Get whether the PivotChart Filter Pane is visible
ShowPivot Table FieldList	Boolean	Set/Get whether the <code>PivotTable</code> field list can be shown
Signatures	Signatures	Read-only. Returns digital signatures for the workbook
SmartDocument	Smart Document	Read-only. Represents the settings for a <code>SmartDocument</code> solution
SmartTag Options	SmartTag Options	Read-only. Returns the options that can be performed with a <code>SmartTag</code>
Styles	Styles	Read-only. Returns the collection of styles associated with the workbook

Name	Returns	Description
Sync	Sync	Read-only. Provides programmatic access for documents that are part of a Document Workspace
TableStyles	TableStyles	Read-only. Allows users to return and manipulate the style in use by the workbook
Template Remove ExtData	Boolean	Set/Get whether all the external data references are removed after a workbook is saved as a template
Theme	OfficeTheme	Read-only. Returns the theme applied to the current workbook
UpdateLinks	XlUpdate Links	Set/Get the workbook's setting for updating embedded OLE links
UpdateRemote References	Boolean	Set/Get whether remote references are updated for the workbook
UserStatus	Variant	Read-only. Returns the name of the current user
VBA Signed	Boolean	Read-only. Returns whether the VBA Project for the workbook has been digitally signed
VBProject	VBProject	Read-only. Returns access to the VBE and associated project
WebOptions	WebOptions	Read-only. Returns an object allowing manipulation of web-related properties of the workbook
Windows	Windows	Read-only. Returns the collection of windows that make up the workbook
Worksheets	Sheets	Read-only. Returns the collection of worksheets that make up the workbook
WritePassword	String	Set/Get the write password of a workbook
WriteReserved	Boolean	Read-only. Returns whether the workbook can be modified
Write ReservedBy	String	Read-only. Returns the name of the person with write permission to the workbook
XMLMaps	XMLMaps	Read-only. Returns the XMLMaps in a given workbook
XMLNamespaces	XML Namespaces	Read-only. Returns the XMLNamespaces in a given workbook

Workbook Methods

Name	Returns	Parameters	Description
Accept AllChanges		[When], [Who], [Where]	Accepts all the changes made by other people in a shared workbook
Activate			Activates the workbook
AddTo Favorites			Adds the workbook shortcut to the Favorites folder
ApplyTheme		Filename As String	Applies a specified theme
BreakLink		Name As String, Type As XlLinkType	Converts formulas linked to other Excel sources or OLE sources to values
CanCheckIn	Boolean		Set/Get whether Excel can check in a specified workbook to a server
ChangeFile Access		Mode As Xl FileAccess, [Write Password], [Notify]	Changes access permissions of the workbook to the one specified by the Mode parameter. If necessary, the WritePassword can be specified. Set Notify to True to have the user notified if the file cannot be accessed
ChangeLink		Name As String, NewName As String, Type	Changes the link from the workbook specified by the Name parameter to the NewName workbook. Type chooses the type of link (for example, OLE or Excel)
CheckIn		[SaveChanges], [Comments], [MakePublic], [VersionType]	Performs a check-in or undo-check-out of the working copy on the server
Close		[SaveChanges], [Filename], [Route Workbook]	Closes the workbook. Set SaveChanges to True to automatically save changes in the workbook. If SaveChanges is False, then all changes are lost. The Filename parameter can be used to specify the filename to save to. RouteWorkbook is used to automatically route the workbook onto the next recipient, if applicable
DeleteNumber Format		NumberFormat As String	Deletes the number format in the NumberFormat parameter from the workbook

Name	Returns	Parameters	Description
Enable Connections			Enable data connections for a user
EndReview			Ends the review of a file that has been sent for review
Exclusive Access	Boolean		Gives the current user exclusive access to a shared workbook
ExportAs Fixed Format		Type As xlFixedFormatType, Filename, Quality, IncludeDocProperties, IgnorPrintAreas, From, To, OpenAfterPublish, FixedFormatExtClassPtr	Publish a workbook in either PDF or XPS
Follow Hyperlink		Address As String, [SubAddress], [NewWindow], [AddHistory], [ExtraInfo], [Method], [HeaderInfo]	Opens up the appropriate application with the URL specified by the Address parameter. Set NewWindow to True to open up a new window for the hyperlink. Use the ExtraInfo and Method parameters to send more information to the hyperlink (say, for an ASP page). The Method parameter uses the MsoExtraInfoMethod constants
ForwardMailer			Used only in a Macintosh environment. Consult the language reference help included with Microsoft Office Macintosh Edition
GetWorkflow Tasks	Workflow Tasks		Returns the collection of WorkflowTask objects for the specified workbook
GetWorkflow Templates	Workflow Templates		Returns the collection of WorkflowTemplate objects for the specified workbook
Highlight Changes Options		[When], [Who], [Where]	Set/Get when changes are viewed in a shared workbook (When), whose workbook changes can be viewed (Who), and the range that the changes should be put in (Where). Use the xlHighlighChangesTime constants with the When parameter

Table continued on following page

Workbook Methods

Name	Returns	Parameters	Description
LinkInfo	Variant	Name As String, LinkInfo As XlLinkInfo, [Type], [EditionRef]	Returns the link details mentioned in the LinkInfo parameter for the link specified by the Name parameter. Use the Type parameter with the XlLinkInfoType constants to pick the type of link that will be returned
LinkSources	Variant	[Type]	Returns the array of linked documents, editions, and DDE and OLE servers in a workbook. Use the Type parameter with the XlLinkInfoType constants to pick the type of link that will be returned
LockServer File			Locks a workbook on the server to prevent edits and modifications
Merge Workbook		Filename	Merges the changes from the Filename workbook into the current workbook
NewWindow	Window		Opens up a new window with the current workbook
OpenLinks		Name As String, [ReadOnly], [Type]	Opens the Name link and supporting documents. Set ReadOnly to True to open the documents as read-only. Use the Type parameter with the XlLinkInfoType constants to pick the type of link that will be returned
PivotCaches	Pivot Caches		Returns the collection of PivotTable caches in the workbook
Post		[DestName]	Posts the workbook into a Microsoft Exchange public folder

Name	Returns	Parameters	Description
PrintOut		[From], [To], [Copies], [Preview], [Active Printer], [PrintToFile], [Collate], [PrToFileName], [IgnorePrintAreas]	Prints out the workbook. The printer, number of copies, collation, and whether a print preview is desired can be specified with the parameters. Also, the sheets can be printed to a file using the PrintToFile and PrToFileName parameters. The From and To parameters can be used to specify the range of printed pages
PrintPreview		[Enable Changes]	Displays the current workbook in a print preview mode. Set the EnableChanges parameter to False to disable the Margins and Setup buttons, hence not allowing the viewer to modify the page setup
Protect		[Password], [Structure], [Windows]	Protects the workbook from user changes. A protect Password can be specified. Set the Structure parameter to True to protect the relative position of the sheets. Set Windows to True to protect the workbook windows
Protect Sharing		[Filename], [Password], [WriteRes Password], [ReadOnly Recommended], [Create Backup], [Sharing Password], [FileFormat]	Protects and saves the workbook for sharing. The file is saved to the Filename parameter with the optional passwords in the Password, WriteResPassword, and SharingPassword parameters. Set ReadOnlyRecommended to True to display a message to the user every time the workbook is opened. Set CreateBackup to True to create a backup of the saved file

Table continued on following page

Workbook Methods

Name	Returns	Parameters	Description
PurgeChangeHistoryNow		Days As Long, [Sharing Password]	Deletes the entries in the change log for the shared workbook. The Days parameter specifies how many days back to delete the entries. A SharingPassword may be required
RecheckSmartTags			Does a foreground SmartTag check. Any data that was not annotated before will now be annotated
RefreshAll			Refreshes any external data source's data into the workbook
RejectAllChanges		[When], [Who], [Where]	Rejects all the changes in a shared workbook
ReloadAs		Encoding As MsoEncoding	Reopens the workbook using the web page-related Encoding parameter
RemoveDocumentInformation		RemoveDocInfoType As xlRemoveDocInfoType	Removes information specified using one of the xlRemoveDocInfoType constants
RemoveUser		Index As Long	Disconnects the user (specified by the user index in the Index parameter) from a shared workbook
Reply			Replies to the sender of the sent workbook. Valid only in the Macintosh Edition of Excel
ReplyAll			Replies to the sender and all recipients of the sent workbook. Valid only in the Macintosh Edition of Excel
ReplyWithChanges		[ShowMessage]	E-mails a notification to the author of a workbook telling them that a reviewer has completed review of the workbook
ResetColors			Resets the colors in the color palette to the default colors
Route			Routes the workbook using the routing slip

Name	Returns	Parameters	Description
RunAuto Macros		Which As XlRunAuto Macro	Runs the auto macro specified by the Which parameter
Save			Saves the workbook
SaveAs		Filename, FileFormat, Password, WriteRes Password, ReadOnly Recommended, CreateBackup, AccessMode, [Conflict Resolution], [AddToMru], [Text Codepage], [TextVisual Layout], [Local]	Saves the workbook as FileName. The type of file to be saved can be specified with the FileFormat parameter. The file can be saved with the optional passwords in the Password and WriteResPassword parameters. Set ReadOnly Recommended to True to display a message to the user every time the workbook is opened. Set CreateBackup to True to create a backup of the saved file. Use AccessMode to choose how the workbook is accessed (for example, xlShared, xlExclusive). Use the ConflictResolution parameter to decide how shared workbooks resolve change conflicts. Set the AddToMru parameter to True to add the workbook to the recently opened files list
SaveAsXMLData		Filename As String, Map As XMLMap	Exports the data that has been mapped to the specified XML schema map to an XML data file
SaveCopyAs		[Filename]	Saves a copy of the workbook as the FileName
SendFaxOver Internet		[Recipients], [Subject], [ShowMessage]	Sends a worksheet as a fax to the specified recipients
SendForReview		[Recipients], [Subject], [ShowMessage], [Include Attachment]	Sends a workbook in an e-mail message for review to the specified recipients

Table continued on following page

Workbook Methods

Name	Returns	Parameters	Description
SendMail		Recipients, [Subject], [Return Receipt]	Sends the workbook through the default mail system. The recipient or recipients and subject can be specified with the parameters. Set ReturnReceipt to True to request a return receipt
SendMailer		FileFormat, Priority	The SendMailer method is used only on the Macintosh. For information about this keyword, consult the language reference Help included with Microsoft Office Macintosh Edition
SetLinkOnData		Name As String, [Procedure]	Runs the procedure in the Procedure parameter whenever the DDE or OLE link in the Name parameter is updated
SetPassword Encryption Options		[Password Encryption Provider], [Password Encryption Algorithm], [Password EncryptionKey Length], [Password Encryption File Properties]	Sets the options for encrypting workbooks using passwords
ToggleForms Design			Toggles into design mode when using Forms controls
Unprotect		[Password]	Unprotects the workbook with the password, if necessary
Unprotect Sharing		[Sharing Password]	Unprotects the workbook from sharing and saves the workbook
UpdateFrom File			Reloads the current workbook from the file if the file is newer than the workbook
UpdateLink		[Name], [Type]	Updates the link specified by the Name parameter. Use the Type parameter with the XLLinkInfoType constants to pick the type of link that will be returned

Name	Returns	Parameters	Description
WebPage Preview			Previews the workbook as a web page
XmlImport		URL As String, ImportMap As XMLMap, Overwrite, Destination	Imports an XML data file to an established XMLMap in the workbook
XmlImportXml	XmlImport XmlResult	Data As String, ImportMap As XMLMap, Destination	Imports an XML data string to an established XMLMap in the workbook

Workbook Events

Name	Parameters	Description
Activate		Triggered when the workbook is activated
AddinInstall		Triggered when the workbook is opened as an Add-In
Addin Uninstall		Triggered when the workbook is opened as an Add-In is uninstalled
AfterXML Export	Map As XMLMap, URL As String, Result As xlXmlExportResult	Triggered after XML data is exported
AfterXML Import	Map As XmlMap, IsRefresh As Boolean, Result as XLXmlImportResult	Triggered after XML data is refreshed or imported
BeforeClose	Cancel As Boolean	Triggered just before the workbook closes. Set the Cancel parameter to True to cancel the closing
BeforePrint	Cancel As Boolean	Triggered just before the workbook is printed. Set the Cancel parameter to True to cancel the printing
BeforeSave	SaveAsUI As Boolean, Cancel As Boolean	Triggered just before the workbook is saved. Set the Cancel parameter to True to cancel the saving. Set the SaveAsUI to True for the user to be prompted with the Save As dialog box
BeforeXML Export	Map As XMLMap, URL As String, Result As xlXmlExportResult	Triggered before XML data is exported

Table continued on following page

Workbook Events

Name	Parameters	Description
BeforeXML Import	Map As XmlMap, URL As String, IsRefresh As Boolean, Result as XmlImportResult, Cancel As Boolean	Triggered before XML data is refreshed or imported
Deactivate		Triggered when the workbook loses focus
NewSheet	Sh As Object	Triggered when a new sheet is created in the workbook. The Sh parameter passes in the new sheet
Open		Triggered when the workbook is opened
PivotTable Close Connection	ByVal Target As PivotTable	Triggered when a PivotTable report closes the connection to its data source. Target is the selected PivotTable
PivotTable Open Connection	ByVal Target As PivotTable	Triggered when a PivotTable report opens the connection to its data source. Target is the selected PivotTable
RowSet Complete	Description As String, Sheet As String, Success As Boolean	Triggered when an OLAP drillthrough or rowset action has been completed
SheetActivate	Sh As Object	Triggered when a sheet is activated in the workbook. The Sh parameter passes in the activated sheet
SheetBefore DoubleClick	Sh As Object, Target As Range, Cancel As Boolean	Triggered when a sheet is about to be double-clicked. The sheet and the potential double-click spot are passed into the event. The double-click action can be canceled by setting the Cancel parameter to True
SheetBefore RightClick	Sh As Object, Target As Range, Cancel As Boolean	Triggered when a sheet is about to be right-clicked. The sheet and the potential right-click spot are passed into the event. The right-click action can be canceled by setting the Cancel parameter to True
Sheet Calculate	Sh As Object	Triggered when a sheet is recalculated passing in the recalculated sheet
SheetChange Range	Sh As Object, Target As	Triggered when the contents of a cell are changed in any worksheet in the workbook. For example, this can be triggered by entering new data, clearing the cell, or deleting a row/column. <i>Not</i> triggered when inserting rows/columns
Sheet Deactivate	Sh As Object	Triggered when a sheet loses focus. Passes in the sheet

Name	Parameters	Description
SheetFollowHyperlink	Sh As Object, Target As Hyperlink	Triggered when the user clicks a hyperlink on a sheet. Passes in the sheet and the clicked hyperlink
SheetPivotTableUpdate	Sh As Object, Target As PivotTable	Triggered when the sheet of the PivotTable report has been updated
SheetSelectionChange	Sh As Object, Target As Range	Triggered when the user selects a different cell on the sheet. Passes in the new range and the sheet where the change occurred
SyncEvent	SyncEventType As MsoSyncEventType	Triggers when the local copy of a worksheet that is part of a Document Workspace is synchronized with the copy on the server
WindowActivate	Wn As Window	Triggered when a workbook window is activated (brought up to the front of other workbook windows). The workbook and the window are passed in
WindowDeactivate	Wn As Window	Triggered when a workbook window loses focus. The related workbook and the window are passed in
WindowResize	Wn As Window	Triggered when a workbook window is resized. The resized workbook and window are passed into the event

WorkbookConnection Object

The `WorkbookConnection` object manages all external data connections in a workbook. Each time a `QueryTable`, `ListObject`, or `PivotCache` that points to external data is created, a new instance of a `WorkbookConnection` object is created. You can also create a standalone `WorkbookConnection` object, one that is not associated with any external data container. You can create a new `WorkbookConnection` object using the `Add` or `AddFromFile` methods of the `Workbook.Connections` collection. See Chapter 21 for examples of how to use the `WorkbookConnection` object.

Worksheet Object and the Worksheets Collection

The `Worksheets` collection holds the collection of worksheets in a workbook. The `Workbook` object is always the parent of the `Worksheets` collection. The `Worksheets` collection only holds the worksheets. The `Worksheet` objects in the `Worksheets` collection can be accessed using the `Item` property. The name of the worksheet can be specified as either a parameter to the `Item`'s parameter or an index number describing the position of the worksheet in the workbook (from left to right).

Worksheets Collection Properties and Methods

The `Worksheet` object allows access to all of the attributes of a specific worksheet in Excel. This includes worksheet formatting and other worksheet properties. The `Worksheet` object also exposes events that can be used programmatically.

The `Worksheets` collection has a few properties and methods besides the typical collection attributes. These are listed in the following table.

Worksheets Collection Properties and Methods

Name	Returns	Description
<code>HPageBreaks</code>	<code>HPageBreaks</code>	Read-only. Returns a collection holding all the horizontal page breaks associated with the <code>Worksheets</code> collection
<code>Visible</code>	<code>Variant</code>	Set/Get whether the worksheets in the collection are visible. Also can set this to <code>xlVeryHidden</code> to prevent a user from making the worksheets in the collection visible
<code>VPageBreaks</code>	<code>VPageBreaks</code>	Read-only. Returns a collection holding all the vertical page breaks associated with the <code>Worksheets</code> collection
<code>Add</code>		Method. Parameters: <code>[Before]</code> , <code>[After]</code> , <code>[Count]</code> , <code>[Type]</code> . Adds a worksheet to the collection. You can specify where the worksheet goes by choosing which sheet object will be before the new worksheet object (<code>Before</code> parameter) or after the new worksheet (<code>After</code> parameter). The <code>Count</code> parameter decides how many worksheets are created
<code>Copy</code>		Method. Parameters: <code>[Before]</code> , <code>[After]</code> . Adds a new copy of the currently active worksheet to the position specified by the <code>Before</code> or <code>After</code> parameters
<code>Delete</code>		Method. Deletes all the worksheets in the collection
<code>FillAcrossSheets</code>		Method. Parameters: <code>Range As Range</code> , <code>Type</code> . Copies the range specified by the <code>Range</code> parameter across all the other worksheets in the collection. Use the <code>Type</code> parameter to pick what part of the range is copied (for example, <code>xlFillWithContents</code> , <code>xlFillWithFormulas</code>)
<code>Move</code>		Method. Parameters: <code>[Before]</code> , <code>[After]</code> . Moves the current worksheet to the position specified by the parameters
<code>PrintOut</code>		Method. Parameters: <code>[From]</code> , <code>[To]</code> , <code>[Copies]</code> , <code>[Preview]</code> , <code>[ActivePrinter]</code> , <code>[PrintToFile]</code> , <code>[Collate]</code> , <code>[PrToFileName]</code> , <code>[IgnorePrintAreas]</code> . Prints the worksheets in the collection. The printer, number of copies, collation, and whether a print preview is desired can be specified with the parameters. Also, the sheets can be printed to a file using the <code>PrintToFile</code> and <code>PrToFileName</code> parameters. The <code>From</code> and <code>To</code> parameters can be used to specify the range of printed pages

Name	Returns	Description
PrintPreview		Method. Parameters: [EnableChanges]. Displays the current worksheet in the collection in a print preview mode. Set the EnableChanges parameter to False to disable the Margins and Setup buttons, hence not allowing the viewer to modify the page setup
Select		Method. Parameters: [Replace]. Selects the current worksheet in the collection

Worksheet Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

Worksheet Properties

Name	Returns	Description
AutoFilter	AutoFilter	Read-only. Returns an AutoFilter object if filtering is turned on
AutoFilter Mode	Boolean	Set/Get whether AutoFilter drop-down arrows are currently displayed on the worksheet
Cells	Range	Read-only. Returns the cells in the current worksheet
Circular Reference	Range	Read-only. Returns the cell range that contains the first circular reference on the worksheet
CodeName	String	Read-only. Returns the name of the worksheet set at design time in the VBE
Columns	Range	Read-only. Returns a range of the columns in the current worksheet
Comments	Comments	Read-only. Returns the collection of comments in the worksheet
Consolidation Function	Xl Consolidation Function	Read-only. Returns the type of consolidation being used in the worksheet (for example, xlSum, xlMax, xlAverage)
Consolidation Options	Variant	Read-only. Returns a one-dimensional array containing three elements of Booleans. The first element describes whether the labels in the top row are used; the second element describes whether the labels in the leftmost column are used; and the third element describes whether links are created to the source data
Consolidation Sources	Variant	Read-only. Returns the array of strings that describe the source sheets for the current worksheet's consolidation

Table continued on following page

Worksheet Properties

Name	Returns	Description
Custom Properties	Custom Properties	Read-only. Returns the identifier information associated with a worksheet
DisplayPage Breaks	Boolean	Set/Get whether page breaks are displayed
DisplayRight ToLeft	Boolean	Set/Get whether the worksheet contents are displayed from right to left. Valid only with languages that support right-to-left text
EnableAuto Filter	Boolean	Set/Get whether the AutoFilter arrows are enabled when a worksheet is user interface-only protected
Enable Calculation	Boolean	Set/Get whether Excel will automatically recalculate the worksheet as necessary
EnableFormat Conditions Calculation	Boolean	Set/Get whether conditional formats will occur automatically as needed
EnableOut lining	Boolean	Set/Get whether outlining symbols are enabled when a worksheet is user interface-only protected
EnablePivot Table	Boolean	Set/Get whether PivotTable controls and related actions are enabled when a worksheet is user interface-only protected
Enable Selection	XlEnable Selection	Set/Get what objects can be selected when a worksheet is protected (for example, xlNoSelection and xlNoRestrictions)
FilterMode	Boolean	Read-only. Returns whether a worksheet is in a filter mode
HPageBreaks	HPageBreaks	Read-only. Returns a collection holding all the horizontal page breaks associated with the Worksheet
Hyperlinks	Hyperlinks	Read-only. Returns the collection of hyperlinks in the worksheet
Index	Long	Read-only. Returns the spot in the parent collection where the current worksheet is located
ListObjects	ListObjects	Returns a ListObjects object
MailEnvelope	MsoEnvelope	Set/Get the e-mail header for a document
Name	String	Set/Get the name of the worksheet
Names	Names	Read-only. Returns the collection of ranges with names in the worksheet
Next	Object	Read-only. Returns the next sheet in the workbook (from left to right) as an object
Outline	Outline	Read-only. Returns an object to manipulate an outline in the worksheet

Name	Returns	Description
PageSetup	PageSetup	Read-only. Returns an object to manipulate the page setup properties for the worksheet
Previous	Object	Read-only. Returns the previous sheet in the workbook (from right to left) as an object
Protect Contents	Boolean	Read-only. Returns whether the worksheet and everything in it is protected from changes
Protect DrawingObjects	Boolean	Read-only. Returns whether the shapes in the worksheet can be modified (<code>ProtectDrawingObjects = False</code>)
Protection	Protection	Read-only. Returns the protection options of the worksheet
Protection Mode	Boolean	Read-only. Returns whether protection has been applied to the user interface. Even if a worksheet has user interface protection on, any VBA code associated with the worksheet can still be accessed
Protect Scenarios	Boolean	Read-only. Returns whether the worksheet scenarios are protected
QueryTables	QueryTables	Read-only. Returns the collection of query tables associated with the worksheet
Range	Range	Read-only. Parameters: <code>Ce111</code> , [<code>Ce112</code>]. Returns a Range object as defined by the <code>Ce111</code> and, optionally, <code>Ce112</code> parameters
Rows	Range	Read-only. Returns a Range object containing the rows of the current worksheet
ScrollArea	String	Sets the A1-style reference string describing the range in the worksheet that can be scrolled. Cells not in the range cannot be selected
Shapes	Shapes	Read-only. Returns all the shapes contained by the worksheet
SmartTags	SmartTags	Read-only. Returns the identifier for the specified cell
Sort	Sort	Controls the attributes and specifications of a sort of the AutoFilter object
Standard Height	Double	Read-only. Returns the default height of the rows in the worksheet, in points
Standard Width	Double	Read-only. Returns the default width of the columns in the worksheet, in points
Tab	Tab	Read-only. Returns the Tab object for the selected chart or worksheet

Table continued on following page

Worksheet Methods

Name	Returns	Description
TransitionExp Eval	Boolean	Set/Get whether to evaluate expressions using Lotus 1-2-3 rules in the worksheet
TransitionForm Entry	Boolean	Set/Get whether formula entries can be entered using Lotus 1-2-3 rules
Type	XlSheetType	Read-only. Returns the worksheet type (for example, xlWorksheet, xlExcel4MacroSheet, xlExcel4IntlMacroSheet)
UsedRange	Range	Read-only. Returns the range in the worksheet that is being used
Visible	XlSheet Visibility	Set/Get whether the worksheet is visible. Also, set this to xlVeryHidden to prevent a user from making the worksheet visible
VPageBreaks	VPageBreaks	Read-only. Returns a collection holding all the vertical page breaks associated with the worksheet

Worksheet Methods

Name	Returns	Parameters	Description
Activate			Activates the worksheet
Calculate			Calculates all the formulas in the worksheet
ChartObjects	Object	[Index]	Returns either a chart object (ChartObject) or a collection of chart objects (ChartObjects) in a worksheet
CheckSpelling		[CustomDictionary], [IgnoreUppercase], [AlwaysSuggest], [SpellLang]	Checks the spelling of the text in the worksheet. A custom dictionary can be specified (CustomDictionary), all uppercase words can be ignored (IgnoreUppercase), and Excel can be set to display a list of suggestions (AlwaysSuggest)
CircleInvalid			Circles the invalid entries in the worksheet
ClearArrows			Clears out all the tracer arrows in the worksheet
ClearCircles			Clears all the circles around invalid entries in a worksheet

Name	Returns	Parameters	Description
Copy		[Before], [After]	Adds a new copy of the worksheet to the position specified at the Before or After parameter
Delete			Deletes the worksheet
Evaluate	Variant	Name	Evaluates the Name string expression as if it were entered into a worksheet cell
ExportAsFixedFormat		Type As Variant, FileName, Quality, IncludeDoc Properties, IgnorePrintAreas, From, To, OpenAfterPublish	Exports a file to a format specified by using the xlFixedFormatType constants
Move		[Before], [After]	Moves the worksheet to the position specified by the parameters
OLEObjects	Object	[Index]	Returns either a single OLE object (OLEObject) or a collection of OLE objects (OLEObjects) for a worksheet
Paste		[Destination], [Link]	Pastes the contents of the clipboard into the worksheet. A specific destination range can be specified with the Destination parameter. Set Link to True to establish a link to the source of the pasted data. Either the Destination or the Link parameter can be used
PasteSpecial		[Format], [Link], [DisplayAsIcon], [IconFileName], [IconIndex], [IconLabel], [NoHTMLFormatting]	Pastes the clipboard contents into the current worksheet. The format of the clipboard data can be specified with the string Format parameter. Set Link to True to establish a link to the source of the pasted data. Set DisplayAsIcon to True to display the pasted data as an icon and the IconFileName, IconIndex, and IconLabel to specify the icon and label. A destination range must be already selected in the worksheet

Table continued on following page

Worksheet Methods

Name	Returns	Parameters	Description
PivotTables	Object	[Index]	Returns either a single PivotTable report (PivotTable) or a collection of PivotTable reports (PivotTables) for a worksheet
PivotTable Wizard	Pivot Table	[SourceType], [SourceData], [TableDestination], [TableName], [RowGrand], [ColumnGrand], [SaveData], [HasAutoFormat], [AutoPage], [Reserved], [BackgroundQuery], [OptimizeCache], [PageFieldOrder], [PageFieldWrapCount], [ReadData], [Connection]	Creates a PivotTable report. The SourceType uses the XLPivotTableSourceType constants to specify the type of SourceData being used for the PivotTable. TableDestination holds the range in the parent worksheet where that report will be placed. TableName holds the name of the new report. Set RowGrand or ColumnGrand to True to show grand totals for rows and columns, respectively. Set HasAutoFormat to True for Excel to format the report automatically when it is refreshed or changed. Use the AutoPage parameter to set if a page field is created automatically for consolidation. Set BackgroundQuery to True for Excel to query the data source asynchronously. Set OptimizeCache to True for Excel to optimize the cache when it is built. Use the PageFieldOrder with the xlOrder constants to set how new page fields are added to the report. Use the PageFieldWrapCount to set the number of page fields in each column or row. Set ReadData to True to copy the data from the external database into a cache. Finally, use the Connection parameter to specify an ODBC connection string for the PivotTable's cache
PrintOut		[From], [To], [Copies], [Preview], [ActivePrinter], [PrintToFile], [Collate], [PrToFileName], [IgnorePrintAreas]	Prints out the worksheet. The printer, number of copies, collation, and whether a print preview is desired can be specified with the parameters. Also, the sheets can be printed to a file using the PrintToFile and PrToFileName parameters. The From and To parameters can be used to specify the range of printed pages

Name	Returns	Parameters	Description
PrintPreview		[EnableChanges]	Displays the worksheet in a print preview mode. Set the <code>EnableChanges</code> parameter to <code>False</code> to disable the <code>Margins</code> and <code>Setup</code> buttons, hence not allowing the viewer to modify the page setup
Protect		[Password], [DrawingObjects], [Contents], [Scenarios], [User InterfaceOnly], [AllowFormatting Cells], [Allow FormattingColumns], [AllowFormatting Rows], [Allow InsertingColumns], [AllowInserting Rows], [Allow Inserting Hyperlinks], [AllowDeleting Columns], [Allow DeletingRows], [AllowSorting], [AllowFiltering], [AllowUsing PivotTables]	Protects the worksheet from changes. A case-sensitive <code>Password</code> can be specified. Also specifies whether shapes are protected (<code>DrawingObjects</code>), whether the entire contents are protected (<code>Contents</code>), or whether only the user interface is protected (<code>UserInterfaceOnly</code>)
ResetAllPage Breaks			Resets all the page breaks in the worksheet
SaveAs		Filename As String, [FileFormat], [Password], [WriteResPassword], [ReadOnly Recommended], [CreateBackup], [AddToMru], [Text Codepage], [Text VisualLayout], [Local]	Saves the worksheet as <code>FileName</code> . The type of file to be saved can be specified with the <code>FileFormat</code> parameter. The file can be saved with the optional passwords in the <code>Password</code> and <code>WriteResPassword</code> parameters. Set <code>ReadOnlyRecommended</code> to <code>True</code> to display a message to the user every time the worksheet is opened. Set <code>CreateBackup</code> to <code>True</code> to create a backup of the saved file. Set the <code>AddToMru</code> parameter to <code>True</code> to add the worksheet to the recently opened files list

Table continued on following page

Worksheet Events

Name	Returns	Parameters	Description
Scenarios	Object	[Index]	Returns either a single scenario (Scenario) or a collection of scenarios (Scenarios) for a worksheet
Select		[Replace]	Selects the worksheet
SetBackgroundPicture		Filename As String	Sets the worksheet's background to the picture specified by the File Name parameter
ShowAllData			Displays all of the data that is currently filtered
ShowDataForm			Displays the data form that is part of the worksheet
Unprotect		[Password]	Deletes the protection set up for a worksheet. If the worksheet was protected with a password, the password must be specified now
XmlDataQuery	Range	[XPath] As String, [Selection Namespaces], [Map]	Represents cells mapped to a particular XPath
XmlMapQuery	Range	[XPath] As String, [Selection Namespaces] [Map]	Represents cells mapped to a particular XPath

Worksheet Events

Name	Parameters	Description
Activate		Triggered when a worksheet is made to have focus
BeforeDoubleClick	Target AsRange, Cancel AsBoolean	Triggered just before a user double-clicks a worksheet. The cell closest to the point double-clicked in the worksheet is passed into the event procedure as Target. The double-click action can be canceled by setting the Cancel parameter to True
BeforeRightClick	Target as Range, Cancel AsBoolean	Triggered just before a user right-clicks a worksheet. The cell closest to the point right-clicked in the worksheet is passed into the event procedure as Target. The right-click action can be canceled by setting the Cancel parameter to True
Calculate		Triggered after the worksheet is recalculated
Change	Target As Range	Triggered when the worksheet cell values are changed. The changed range is passed into the event procedure as Target

Name	Parameters	Description
Deactivate		Triggered when the worksheet loses focus
Follow Hyperlink	Target As Hyperlink	Triggered when a hyperlink is clicked on the worksheet. The hyperlink that was clicked is passed into the event procedure as Target
PivotTable Update	ByVal Target As PivotTable	Triggered when a PivotTable report is updated on a worksheet
Selection Change	Target As Range	Triggered when the selection changes in a worksheet. The new selected range is passed into the event procedure as Target

WorksheetFunction Object

The `WorksheetFunction` object allows access to Excel worksheet functions via VBA. The parent of the `WorksheetFunction` object is the `Application` object.

WorksheetFunction Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

WorksheetFunction Methods

The methods of the `WorksheetFunction` object are actually individual Excel functions that do not have VBA equivalents. The Excel functions listed here constitute the methods of the `WorksheetFunction` object. You can explore these functions in detail by going to Excel's Help and entering the keywords "List of Functions" in the Search box.

<input type="checkbox"/> AccrInt	<input type="checkbox"/> AveDev	<input type="checkbox"/> Bin2Hex
<input type="checkbox"/> AccrIntM	<input type="checkbox"/> Average	<input type="checkbox"/> Bin2Oct
<input type="checkbox"/> Acos	<input type="checkbox"/> AverageIf	<input type="checkbox"/> BinomDist
<input type="checkbox"/> Acosh	<input type="checkbox"/> AverageIfs	<input type="checkbox"/> Ceiling
<input type="checkbox"/> AmorDegrc	<input type="checkbox"/> BahtText	<input type="checkbox"/> ChiDist
<input type="checkbox"/> AmorLinc	<input type="checkbox"/> BesselI	<input type="checkbox"/> ChiInv
<input type="checkbox"/> And	<input type="checkbox"/> BesselJ	<input type="checkbox"/> ChiTest
<input type="checkbox"/> Asc	<input type="checkbox"/> BesselK	<input type="checkbox"/> Choose
<input type="checkbox"/> Asin	<input type="checkbox"/> BesselY	<input type="checkbox"/> Clean
<input type="checkbox"/> Asinh	<input type="checkbox"/> BetaDist	<input type="checkbox"/> Combin
<input type="checkbox"/> Atan2	<input type="checkbox"/> BetaInv	<input type="checkbox"/> Complex
<input type="checkbox"/> Atanh	<input type="checkbox"/> Bin2Dec	<input type="checkbox"/> Confidence

WorksheetFunction Methods

- | | | |
|-------------------------------------|-------------------------------------|--------------------------------------|
| <input type="checkbox"/> Convert | <input type="checkbox"/> Disc | <input type="checkbox"/> FTest |
| <input type="checkbox"/> Correl | <input type="checkbox"/> DMax | <input type="checkbox"/> Fv |
| <input type="checkbox"/> Cosh | <input type="checkbox"/> DMin | <input type="checkbox"/> FVSchedule |
| <input type="checkbox"/> Count | <input type="checkbox"/> Dollar | <input type="checkbox"/> GammaDist |
| <input type="checkbox"/> CountA | <input type="checkbox"/> DollarDe | <input type="checkbox"/> GammaInv |
| <input type="checkbox"/> CountBlank | <input type="checkbox"/> DollarFr | <input type="checkbox"/> GammaLn |
| <input type="checkbox"/> CountIf | <input type="checkbox"/> DProduct | <input type="checkbox"/> Gcd |
| <input type="checkbox"/> CountIfs | <input type="checkbox"/> DStDev | <input type="checkbox"/> GeoMean |
| <input type="checkbox"/> CoupDayBs | <input type="checkbox"/> DStDevP | <input type="checkbox"/> GeStep |
| <input type="checkbox"/> CoupDays | <input type="checkbox"/> DSum | <input type="checkbox"/> Growth |
| <input type="checkbox"/> CoupDaysNc | <input type="checkbox"/> Duration | <input type="checkbox"/> HarMean |
| <input type="checkbox"/> CoupNcd | <input type="checkbox"/> DVar | <input type="checkbox"/> Hex2Bin |
| <input type="checkbox"/> CoupNum | <input type="checkbox"/> DVarP | <input type="checkbox"/> Hex2Dec |
| <input type="checkbox"/> CoupPcd | <input type="checkbox"/> EDate | <input type="checkbox"/> Hex2Oct |
| <input type="checkbox"/> Covar | <input type="checkbox"/> Effect | <input type="checkbox"/> HLookup |
| <input type="checkbox"/> CritBinom | <input type="checkbox"/> EoMonth | <input type="checkbox"/> HypGeomDist |
| <input type="checkbox"/> CumIPmt | <input type="checkbox"/> Erf | <input type="checkbox"/> IfError |
| <input type="checkbox"/> CumPrinc | <input type="checkbox"/> ErfC | <input type="checkbox"/> ImAbs |
| <input type="checkbox"/> DAverage | <input type="checkbox"/> Even | <input type="checkbox"/> Imaginary |
| <input type="checkbox"/> Days360 | <input type="checkbox"/> ExponDist | <input type="checkbox"/> ImArgument |
| <input type="checkbox"/> Db | <input type="checkbox"/> Fact | <input type="checkbox"/> ImConjugate |
| <input type="checkbox"/> DbcS | <input type="checkbox"/> FactDouble | <input type="checkbox"/> ImCos |
| <input type="checkbox"/> DCount | <input type="checkbox"/> FDist | <input type="checkbox"/> ImDiv |
| <input type="checkbox"/> DCountA | <input type="checkbox"/> Find | <input type="checkbox"/> ImExp |
| <input type="checkbox"/> Ddb | <input type="checkbox"/> FindB | <input type="checkbox"/> ImLn |
| <input type="checkbox"/> Dec2Bin | <input type="checkbox"/> FInv | <input type="checkbox"/> ImLog10 |
| <input type="checkbox"/> Dec2Hex | <input type="checkbox"/> Fisher | <input type="checkbox"/> ImLog2 |
| <input type="checkbox"/> Dec2Oct | <input type="checkbox"/> FisherInv | <input type="checkbox"/> ImPower |
| <input type="checkbox"/> Degrees | <input type="checkbox"/> Fixed | <input type="checkbox"/> ImProduct |
| <input type="checkbox"/> Delta | <input type="checkbox"/> Floor | <input type="checkbox"/> ImReal |
| <input type="checkbox"/> DevSq | <input type="checkbox"/> Forecast | <input type="checkbox"/> ImSin |
| <input type="checkbox"/> DGet | <input type="checkbox"/> Frequency | <input type="checkbox"/> ImSqrt |

<input type="checkbox"/> ImSub	<input type="checkbox"/> Median	<input type="checkbox"/> Pmt
<input type="checkbox"/> ImSum	<input type="checkbox"/> Min	<input type="checkbox"/> Poisson
<input type="checkbox"/> Index	<input type="checkbox"/> MInverse	<input type="checkbox"/> Power
<input type="checkbox"/> Intercept	<input type="checkbox"/> MIrr	<input type="checkbox"/> Ppmt
<input type="checkbox"/> InRate	<input type="checkbox"/> MMult	<input type="checkbox"/> Price
<input type="checkbox"/> Ipmt	<input type="checkbox"/> Mode	<input type="checkbox"/> PriceDisc
<input type="checkbox"/> Irr	<input type="checkbox"/> MRound	<input type="checkbox"/> PriceMat
<input type="checkbox"/> IsErr	<input type="checkbox"/> MultiNomial	<input type="checkbox"/> Prob
<input type="checkbox"/> IsError	<input type="checkbox"/> NegBinomDist	<input type="checkbox"/> Product
<input type="checkbox"/> IsEven	<input type="checkbox"/> NetworkDays	<input type="checkbox"/> Proper
<input type="checkbox"/> IsLogical	<input type="checkbox"/> Nominal	<input type="checkbox"/> Pv
<input type="checkbox"/> IsNA	<input type="checkbox"/> NormDist	<input type="checkbox"/> Quartile
<input type="checkbox"/> IsNonText	<input type="checkbox"/> NormInv	<input type="checkbox"/> Quotient
<input type="checkbox"/> IsNumber	<input type="checkbox"/> NormSDist	<input type="checkbox"/> Radians
<input type="checkbox"/> IsOdd	<input type="checkbox"/> NormSInv	<input type="checkbox"/> RandBetween
<input type="checkbox"/> Ispmt	<input type="checkbox"/> NPer	<input type="checkbox"/> Rank
<input type="checkbox"/> IsText	<input type="checkbox"/> Npv	<input type="checkbox"/> Rate
<input type="checkbox"/> Kurt	<input type="checkbox"/> Oct2Bin	<input type="checkbox"/> Received
<input type="checkbox"/> Large	<input type="checkbox"/> Oct2Dec	<input type="checkbox"/> Replace
<input type="checkbox"/> Lcm	<input type="checkbox"/> Oct2Hex	<input type="checkbox"/> ReplaceB
<input type="checkbox"/> LinEst	<input type="checkbox"/> Odd	<input type="checkbox"/> Rept
<input type="checkbox"/> Ln	<input type="checkbox"/> OddFPrice	<input type="checkbox"/> Roman
<input type="checkbox"/> Log	<input type="checkbox"/> OddFYield	<input type="checkbox"/> Round
<input type="checkbox"/> Log10	<input type="checkbox"/> OddLPrice	<input type="checkbox"/> RoundDown
<input type="checkbox"/> LogEst	<input type="checkbox"/> OddLYield	<input type="checkbox"/> RoundUp
<input type="checkbox"/> LogInv	<input type="checkbox"/> Or	<input type="checkbox"/> RSq
<input type="checkbox"/> LogNormDist	<input type="checkbox"/> Pearson	<input type="checkbox"/> RTD
<input type="checkbox"/> Lookup	<input type="checkbox"/> Percentile	<input type="checkbox"/> Search
<input type="checkbox"/> Match	<input type="checkbox"/> PercentRank	<input type="checkbox"/> SearchB
<input type="checkbox"/> Max	<input type="checkbox"/> Permut	<input type="checkbox"/> SeriesSum
<input type="checkbox"/> MDeterm	<input type="checkbox"/> Phonetic	<input type="checkbox"/> Sinh
<input type="checkbox"/> MDuration	<input type="checkbox"/> Pi	<input type="checkbox"/> Skew

WorksheetFunction Object Example

- | | | |
|--------------------------------------|-------------------------------------|------------------------------------|
| <input type="checkbox"/> Sln | <input type="checkbox"/> SumX2PY2 | <input type="checkbox"/> Var |
| <input type="checkbox"/> Slope | <input type="checkbox"/> SumXMY2 | <input type="checkbox"/> VarP |
| <input type="checkbox"/> Small | <input type="checkbox"/> Syd | <input type="checkbox"/> Vdb |
| <input type="checkbox"/> SqrtPi | <input type="checkbox"/> Tanh | <input type="checkbox"/> VLookup |
| <input type="checkbox"/> Standardize | <input type="checkbox"/> TBillEq | <input type="checkbox"/> Weekday |
| <input type="checkbox"/> StDev | <input type="checkbox"/> TBillPrice | <input type="checkbox"/> WeekNum |
| <input type="checkbox"/> StDevP | <input type="checkbox"/> TBillYield | <input type="checkbox"/> Weibull |
| <input type="checkbox"/> StEyx | <input type="checkbox"/> TDist | <input type="checkbox"/> WorkDay |
| <input type="checkbox"/> Substitute | <input type="checkbox"/> Text | <input type="checkbox"/> Xirr |
| <input type="checkbox"/> Subtotal | <input type="checkbox"/> TInv | <input type="checkbox"/> Xnpv |
| <input type="checkbox"/> Sum | <input type="checkbox"/> Transpose | <input type="checkbox"/> YearFrac |
| <input type="checkbox"/> SumIf | <input type="checkbox"/> Trend | <input type="checkbox"/> YieldDisc |
| <input type="checkbox"/> SumIifs | <input type="checkbox"/> Trim | <input type="checkbox"/> YieldMat |
| <input type="checkbox"/> SumProduct | <input type="checkbox"/> TrimMean | <input type="checkbox"/> ZTest |
| <input type="checkbox"/> SumSq | <input type="checkbox"/> TTest | |
| <input type="checkbox"/> SumX2MY2 | <input type="checkbox"/> USDollar | |

WorksheetFunction Object Example

In this example, an array of numbers is passed to the `Max` worksheet function to determine the biggest number in the array:

```
Sub GetBiggest()  
    Dim oWSF As WorksheetFunction  
    Dim vaArray As Variant  
    Set oWSF = Application.WorksheetFunction  
    vaArray = Array(10, 20, 13, 15, 56, 12, 8, 45)  
    MsgBox "Biggest is " & oWSF.Max(vaArray)  
End Sub
```

WorksheetView Object

The `WorksheetView` object contains various properties that determine how certain values and objects on a given worksheet are displayed. Along with the common properties of `Application`, `Creator`, and `Parent`, the `WorksheetView` object contains the following properties.

WorksheetView Properties

Name	Returns	Description
Display Formulas	Boolean	Set/Get whether the formulas are displayed on the worksheet
Display Gridlines	Boolean	Set/Get whether the gridlines are displayed on the worksheet
DisplayHeadings	Boolean	Set to <code>True</code> to display both row and column headings. Set to <code>False</code> to display no headings
DisplayOutline	Boolean	Set/Get whether outline symbols are displayed on the worksheet
DisplayZeros	Boolean	Set/Get whether the zero values are displayed on the worksheet
Sheet	Object	Read-only. Returns the sheet associated with the specified <code>WorksheetView</code> object

XmlDataBinding Object

The `XMLDataBinding` object represents the connection to the data source for an XML Map. Along with the common properties of `Application`, `Creator`, and `Parent`, the `XMLDataBinding` object contains the `SourceURL` property. The `SourceURL` property returns a string variable that represents the path to the XML data file or the URL that provides the source data for the specified data binding. See Chapter 12 for examples of how to use the `XMLDataBinding` object.

XmlDataBinding Methods

Name	Returns	Parameters	Description
ClearSettings			Clears all settings for the current object
LoadSettings		Url as String	Loads a set of settings
Refresh	<code>xlXmlImportResult</code>		Refreshes all data

XmlMap Object and the XMLMaps Collection

The `XMLMap` object represents an XML Map that has been added to a workbook. The `XMLMaps` collection contains all of the `XMLMap` objects within a workbook. Along with the common collection attributes, the `XMLMaps` collection has an `Add` method that allows you to add a new `XMLMap` object to the collection.

XMLMap Common Properties

XMLMap Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

XmlMap Properties

Name	Returns	Description
<code>AdjustColumnWidth</code>	Boolean	Set this to <code>True</code> to automatically adjust column widths when data is refreshed
<code>AppendOnImport</code>	Boolean	Set this to <code>True</code> and imported data will be appended to current data. Otherwise, imported data will be overwritten
<code>DataBindingBinding</code>	XmlData	Read-only. Returns an <code>XmlDataBinding</code> object that represents the binding associated with the specified schema map
<code>IsExportable</code>	Boolean	Read-only. Determines if the current data is exportable
<code>Name</code>	String	Get/Set the name of a given <code>XMLMap</code> object
<code>PreserveColumnFilter</code>	Boolean	Set/Get whether the column filters are persisted
<code>PreserveNumberFormatting</code>	Boolean	Set/Get whether number formatting is persisted
<code>RootElementName</code>	String	Read-only. Returns a string representing the root element name in the XML tree
<code>RootElementNamespace</code>	XmlName space	Read-only. Returns a <code>XmlNamespace</code> object that represents the namespace of the current root element
<code>SaveDataSourceDefinition</code>	Boolean	Set/Get whether the data source information is saved
<code>Schemas</code>	XmlSchemas	Read-only. Returns a collection of <code>XmlSchema</code> objects that have been applied to the current workbook
<code>ShowImportExportValidationErrors</code>	Boolean	Set this to <code>True</code> and any validation errors will be displayed during import and export operations
<code>WorkbookConnection</code>	Workbook Connection	Read-only. Returns an object that manages the external connections to the source XML data

XmlMap Methods

Name	Returns	Parameters	Description
<code>Delete</code>			Deletes the current <code>XmlMap</code> object
<code>Export</code>	<code>XlXmlExportResult</code>	<code>Url As String, [Overwrite]</code>	Exports the current <code>XmlMap</code> object

Name	Returns	Parameters	Description
ExportXml	XlXmlExportResult	Data As String	Exports the current XmlMap object as XML that can be persisted to a file
Import	XlXmlImportResult	Url As String, [Overwrite]	Imports an XmlMap object
ImportXml	XlXmlImportResult	XmlData As String, [Overwrite]	Imports an XmlMap object from XML

See Chapter 12 for examples of how to use the `XMLMap` object.

XmlNamespace Object and the XMLNameSpacesCollection

The `XMLNamespace` object represents an XML namespace that has been added to a workbook. The `XMLNameSpaces` collection contains all of the `XMLNamespace` objects within a workbook. Along with the common collection attributes, the `XMLNameSpaces` collection contains a `Value` property that returns the actual namespace name. The `XMLNameSpaces` collection only has one method. This method is the `InstallManifest`, which installs a specified XML expansion pack (a collection of files that add custom displays and actions to your Excel workbook).

XMLNamespace Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

XMLNamespace Properties

Name	Returns	Description
Prefix	String	Read-only. Returns the prefix for the specified namespace
Uri	String	Read-only. Returns the Uniform Resource Identifier (URI) for the specified namespace

XmlSchema Object and the XmlSchemas Collection

The `XMLSchema` object represents an XML schema contained by the `XMLMap` object. The `XMLNameSchemasCollection` contains all of the `XMLSchema` objects within a workbook. `XMLNameSchemasCollection` has only the common collection.

XMLSchema Common Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

XMLSchema Properties

XMLSchema Properties

Name	Returns	Description
Name	String	Read-only. Returns the user-friendly name for the XML schema
Namespace	XMLNamespace	Read-only. Returns an XMLNamespace object representing the target namespace for the specified schema
XML	String	Returns the XML string that makes up the content for the specified schema

XPath Object

The XPath object represents an XPath expression mapped to a range or list. For more information on XPath expressions and how to use them with Excel VBA, refer to Chapter 12.

XPath Common Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

XPath Properties

Name	Returns	Description
Map	XMLMap	Read-only. Returns an XMLMap object representing the schema map that contains the specified XPath object
Repeating	Boolean	Read-only. Get whether the given XPath object is mapped to an XML list
Value	String	Set/Get the string that makes up the XPath expression

XPath Methods

Name	Parameters	Description
Clear		Clears all XPath schema information for the mapped range. Note that this method does not clear the data from the cells mapped to the specified XPath
SetValue	Map As XMLMap, XPath as String, Selection NameSpace, Repeating	Maps the specified XPath object to a ListColumn object or Range collection

B

VBE Object Model

Officially known as “*Microsoft Visual Basic for Applications Extensibility 5.3*,” the VBE object library provides access to the code and forms within an application, and to the various objects that compose the *Visual Basic Integrated Development Environment* (VBIDE). By default, this object library is *not* included in the list of referenced libraries for new projects. In order to use the objects referred to in this chapter, a reference to the Microsoft Visual Basic for Applications Extensibility 5.3 library must be created using the Tools ⇄ References menu in the VBE.

Many of the objects in the VBE object model have the same names as objects in the Excel object model. To distinguish the libraries and to ensure that you have the object from the VBE library, you need to include the VBIDE library name in any Dim statements you may use:

```
Dim oWinVB As VBIDE.Window      'Always gives a VBE Window
Dim oWinXL As Excel.Window      'Always gives an XL Window
Dim oWin As Window              'Gives an XL Window
```

All of the applications in Office 2007 share the same development environment—the VBE. The code and forms that belong to each Excel workbook, Word document, Access database, or PowerPoint presentation (that is, the “host document”) are grouped into Visual Basic projects (the VBProject object). There is one project for each host document. Outlook has a single Project, which “belongs” to the application.

Links between the Excel and VBE Object Models

A number of properties of Excel objects provide links to the VBE object model. Similarly, a number of properties in the VBE object model provide a link back into Excel. Many of the code examples in this appendix and in Chapter 26 use the links outlined in the following tables.

Excel to VBE

Excel Property	Resulting VBE Item
<code>Application.VBE</code>	VBE object
<code>Workbook.VBProject</code>	VBProject object
<code>Workbook.CodeName</code>	The name of the workbook-level VBComponent in the workbook's VBProject, usually "ThisWorkbook" in English versions of Excel 2007
<code>Worksheet.CodeName</code>	The name of the sheet-level VBComponent in the workbook's VBProject, usually "Sheet1", "Chart1", and so forth in English versions of Excel 2007

VBE to Excel

VBE Property	Resulting Excel item
<code>VBProject.FileName</code>	The full name of the workbook, if the VBProject is an Excel workbook project and the workbook has been saved.
<code>VBComponent.Properties("Name")</code>	The filename of the workbook, if the VBComponent is the workbook-level item (for example, "ThisWorkbook") or the name of the sheet for sheet-level VBComponents.
<code>VBComponent.Properties("<Other Properties>")</code>	The properties associated with the Excel object to which VBComponent applies (if any).

Common Properties and Methods

Most of the objects in the VBE object library have the following common properties. To avoid redundancy, these properties will be listed for each object, but will not be explained.

Name	Returns	Description
Collection		Read-only. Returns the collection to which an object belongs. For example, a Reference object belongs to the References collection. The Collection property is used for objects that belong to collections.
Parent		Read-only. Returns the object to which an object belongs. For example, a References collection belongs to a VBProject object. The Parent property is used for objects that do not belong to collections.
VBE	VBE	Read-only. Returns the Visual Basic Editor object, which is analogous to the Application object in the Excel object model.

Most of the objects in the VBE object model are contained in associated collections. The collection object is usually the plural form of the associated object. For example, the `Windows` collection holds a collection of `Window` objects. For simplicity, each object and associated collection are grouped together under the same heading. The common properties and methods of the collection objects are the same as in the Excel object model, and are listed in Appendix A. Only unique properties, methods, or events are mentioned for each object.

AddIn Object and Add-Ins Collection

Not to be confused with Excel's `AddIn` object, VBE Add-Ins are DLLs that conform to Microsoft's Component Object Model architecture and are more commonly known as *COM Add-Ins*. These Add-Ins are typically created using C++, Visual Basic, or .NET. If you have any installed, they can be found under the VBE's Add-Ins menu and can be loaded and unloaded using the Add-Ins ⇄ Add-In Manager menu item.

The following table defines the `Collection` and VBE properties.

Add-In Properties

Name	Returns	Description
<code>Connect</code>	Boolean	Whether the COM Add-In is currently connected (active). Can be set to <code>True</code> to load and run the Add-In. Similar to the <code>Installed</code> property of an Excel Add-In.
<code>Description</code>	String	The text that appears in the Description box of the VBE Add-In Manager.
<code>Guid</code>	String	Read-only. Returns the globally unique identifier for the Add-In. The <code>Guid</code> is created by Excel/VB when the Add-In is compiled.
<code>Object</code>		In the Add-In's <code>OnConnection</code> method, it can expose an object to the VBE (typically the root class of its object model, if it has one). You can then use the Add-In's <code>Object</code> property to access this object and, through it, the rest of the Add-In's object model. Few Add-Ins currently expose an <code>Object</code> .
<code>ProgId</code>	String	Read-only. Returns the program ID for the Add-In, which is composed of the name of the Add-In project and the name of the connection class (usually a connection designer). For example, if you have an Add-In project called <code>MyAddin</code> and an Add-In Designer class called <code>dsrMyConnection</code> , the <code>ProgId</code> will be <code>"MyAddin.dsrMyConnection"</code> .

Add-Ins Collection Methods

Name	Returns	Description
Item	AddIn	Parameters: Item As Variant. Returns an Add-In associated with the item. The parameter can be either a number or the ProgID of the Add-In (for example, MyAddin.dsrMyConnection).
Update		Updates the list of available COM Add-Ins from the Registry. This should only need to be used if you are compiling an Add-In through code (for example, using VBProject.MakeCompiledFile).

The following example iterates through all the Add-Ins registered for use in the VBE and prints information about those that are active:

```
Sub ListRunningAddins()  
    'Define as a VBE Addin, not an Excel one  
    Dim oAddin As VBIDE.Addin  
  
    'Loop through the VBE's addins  
    For Each oAddin In Application.VBE.AddIns  
  
        'Is it active (i.e. connected)?  
        If oAddin.Connect Then  
  
            'Yes, so show it's ID and description  
            Debug.Print oAddin.ProgID, oAddin.Description  
        End If  
    Next  
End Sub
```

Note that VBE Add-Ins do not have a property to provide their name, as shown in the list in the Add-In Manager dialog.

CodeModule Object

The `CodeModule` object contains all of the code for a single `VBComponent` (such as a `Module`, `UserForm`, `Class Module`, the Excel workbook, or an Excel sheet). There is only ever one `CodeModule` for a component—its methods and properties enable you to locate, identify, modify, and add lines of code to a project's components. There can be more than one procedure of the same name in a module, if they are `Property` procedures:

```
Dim msSelection As String  
Property Get TheSelection() As String  
    TheSelection = msSelection  
End Property  
  
Property Let TheSelection(NewString As String)  
    MsSelection = NewString  
End Property
```


Hence, to uniquely identify a procedure, you need to supply both its name (`TheSelection` in this example) and the type of procedure you're looking for (`vbext_pk_Get` for Property Get, `vbext_pk_Let` for Property Let, `vbext_pk_Set` for Property Set, or `vbext_pk_Proc` for Subs and Functions). The `ProcOfLine` property provides this information for a given line number — the name of the procedure is the return value of the function, and the type of procedure is returned in the variable you supply to its `ProcKind` argument. It is one of the few properties in the whole of Office 2007 that returns values by modifying the arguments passed to it.

The `Parent` and `VBE` properties are defined at the beginning of this section (the `Parent` of a `CodeModule` being the `VBComponent`).

CodeModule Properties

Name	Returns	Description
<code>CodePane</code>	<code>CodePane</code>	Read-only. Returns the active <code>CodePane</code> for the module. If there is no visible <code>CodePane</code> , one is created and displayed. Note that a <code>CodeModule</code> can have up to two code panes, but there is no <code>CodePanes</code> collection for them!
<code>CountOfDeclarationLines</code>	<code>Long</code>	<p>Read-only. Returns the number of lines at the top of the module used for <code>Dim</code>, <code>Type</code>, and <code>Option</code> statements. If there are any such items at the top of the module, any comments following them are considered to be part of the following procedure, not the declarations. The following has two declaration lines:</p> <pre>Option Explicit Dim msSelection As String 'My Comment Sub ProcedureStart ()</pre> <p>If no such statements exist, comments appearing at the top of the module are counted as declaration lines, if they are followed by a blank line. The following has one declaration line:</p> <pre>'My Comment Sub ProcedureStart ()</pre> <p>If the comment is immediately followed by the procedure, it is included in the procedure's lines, so the following has no declaration line:</p> <pre>'My Comment Sub ProcedureStart ()</pre>
<code>CountOfLines</code>	<code>Long</code>	Read-only. Returns the total number of lines of code in the module, with line continuations counted as separate lines.
<code>Lines</code>	<code>String</code>	Read-only. Parameters: <code>StartLine</code> As Long, <code>Count</code> As Long. Returns a block of code, starting from <code>Startline</code> and continuing for <code>Count</code> lines.

Table continued on following page

CodeModule Methods

Name	Returns	Description
Name	String	(Hidden) Read-only. Returns the name of the associated VBComponent.
ProcBody Line	Long	Read-only. Parameters: ProcName As String, ProcKind As vbext_ProcKind. Returns the line number of the start of the procedure, not including any preceding comments — that is, it gives the line number of the Sub, Function, or Property statement.
ProcCount Line	Long	Read-only. Parameters: ProcName As String, ProcKind As vbext_ProcKind. Returns the number of lines used by the procedure, including preceding comments, up to the End Sub, End Function, or End Property statement.
ProcOfLine	String	Read-only. Parameters: Line As Long [in], ProcKind As Long [out]. Returns the name of the procedure that a line is located within. The ProcKind argument is also modified to return the type of procedure (Sub/Function, Property Let, Get or Set). This is usually the first property to be called; the name and type returned from this are then used in calls to the other methods.
ProcStart Line	Long	Read-only. Parameters: ProcName As String, ProcKind As vbext_ProcKind. Returns the line number of the start of the procedure, including comments. Hence, ProcBodyLine - ProcStartLine gives you the number of preceding comment lines.

CodeModule Methods

Name	Returns	Parameters	Description
AddFromFile		FileName As String	Reads code from a text file and adds it to the end of the code module. It does not check if the names of procedures read from a file already exist in the module.
AddFromString		String As String	Adds code from a string to the end of the code module.
CreateEvent Proc	Long	EventName As String, ObjectName As String	Creates an empty event procedure in a module, filling in the event parameters for you. Cannot be used on standard modules, as they do not support events. The ObjectName must be a valid object for the class module, and the EventName must be a valid event for that object.

Name	Returns	Parameters	Description
DeleteLines		StartLine As Long, Count As Long	Deletes lines from a code module, starting at StartLine, for Count lines.
Find	Boolean	Target As String, StartLine As Long, StartColumn As Long, EndLine As Long, EndColumn As Long, WholeWord As Boolean, MatchCase As Boolean, PatternSearch As Boolean	Locates a string within a code module, or section of a code module. It provides the same functionality as the VBE's Find dialog.
InsertLines		Line As Long, String As String	Adds code from a string into the middle of a code module, inserting the code before the Line given.
ReplaceLines		Line As Long, String As String	Adds code from a string into the middle of a code module, replacing the Line given.

There are a number of CodeModule examples in Chapter 26. The following example identifies the procedure for a given line and displays its type, name, and line count:

```

Sub WhichProc()
    Dim lLine As Long, iProcKind As Long, lLineCount As Long
    Dim sProc As String, sMsg As String
    Dim oActiveCM As VBIDE.CodeModule

    lLine = CLng(InputBox("Which line?"))

    'Cancelled?
    If lLine = 0 Then Exit Sub

    'Get the currently active code module
    Set oActiveCM = Application.VBE.ActiveCodePane.CodeModule

    'Get the name and type of the procedure at
    'that line - iProcKind is filled in
    sProc = oActiveCM.ProcOfLine(lLine, iProcKind)

    If sProc = "" Then
        'We didn't get a name, so you must be in the Declarations section
        sMsg = "You are in the Declarations section"
        lLineCount = oActiveCM.CountOfDeclarationLines
    Else
        sMsg = "You are in "

        'Display the type of the procedure...

```

```
Select Case iProcKind
    Case vbext_pk_Proc
        sMsg = sMsg & "Sub or Function procedure"
    Case vbext_pk_Get
        sMsg = sMsg & "Property Get procedure"
    Case vbext_pk_Let
        sMsg = sMsg & "Property Let procedure"
    Case vbext_pk_Set
        sMsg = sMsg & "Property Set procedure"
End Select

'... its name ...
sMsg = sMsg & " " & sProc & ""

'... and how many lines it has.
lLineCount = oActiveCM.ProcCountLines(sProc, iProcKind)
End If

'Display the message
MsgBox sMsg & vbCrLf & "which has " & lLineCount & " lines."
End Sub
```

CodePane Object and CodePanels Collection

A `CodePane` is a view of a `CodeModule`, providing you with access to the interaction layer between the developer and the code being edited. Most VBE Add-Ins use this layer to identify the line in which `CodePane` is currently being edited and then modify the code at the line, using `CodeModule`'s methods and properties. Note that there can be more than one `CodePane` for a `CodeModule` (for example, by splitting a code window into two panes with the horizontal splitter bar).

The following tables define the `Collection` and `VBE` properties.

CodePane Properties

Name	Returns	Description
<code>CodeModule</code>	<code>CodeModule</code>	Read-only. Returns the <code>CodeModule</code> that contains the code being viewed in the <code>CodePane</code> .
<code>CodePaneView</code>	<code>vbext_CodePaneView</code>	Read-only. Returns whether the <code>CodePane</code> is set to show one procedure at a time, or a full-module view with separator lines between procedures.
<code>CountOfVisibleLines</code>	<code>Long</code>	Read-only. Returns the number of lines visible in the <code>CodePane</code> . This and the <code>TopLine</code> property can be used to center a line in the <code>CodePane</code> window (see following example).
<code>TopLine</code>	<code>Long</code>	The <code>CodeModule</code> line number of the first line visible in the <code>CodePane</code> window.
<code>Window</code>	<code>Window</code>	Read-only. Returns the <code>Window</code> object containing the <code>CodePane(s)</code> .

CodePane Methods

Name	Parameters	Description
GetSelection	StartLine As Long, StartColumn As Long, EndLine As Long, EndColumn As Long	Used to retrieve the currently selected text. All of the arguments are passed ByRef and are modified within the procedure to return the selection. All arguments are required, but it is only required to pass arguments for those items you want to retrieve. For example, to get only the start line, you can use: Dim lStart As Long Application.VBE.ActiveCodePane.GetSelection lStart, 0, 0, 0
SetSelection	StartLine As Long, StartColumn As Long, EndLine As Long, EndColumn As Long	Used to set the position of the currently selected text. A program would typically read the selection using GetSelection, modify the code, then set the selection back again using SetSelection. See the PrintProcedure routine in Chapter 26 for an example.
Show		Opens and displays the CodePane, making it active.

The CodePanes collection contains all of the open CodePane objects in the VBE.

CodePanes Collection Properties

Name	Returns	Description
Current	CodePane	Hidden. Read-only. Returns the currently active CodePane, and is the same as Application.VBE.ActiveCodePane.

Chapter 26 contains many CodePane examples. The following example identifies the current selection and centers it in the CodePane window:

```
Sub CenterSelectionInWindow()
    Dim oCP As VBIDE.CodePane
    Dim lStartLine As Long, lEndLine As Long
    Dim lVisibleLines As Long, lNewTop As Long

    'Get the active CodePane
    Set oCP = Application.VBE.ActiveCodePane

    'Using the CodePane object...
    With oCP
        'Get the start and end lines of the selection
        .GetSelection lStartLine, 0, lEndLine, 0

        'How many lines fit in the window?
        lVisibleLines = .CountOfVisibleLines

        'So what should the new top line be?
```

CommandBarEvents Object

```
lNewTop = (lStartLine + lEndLine - lVisibleLines) \ 2

'Set the window to display code from that line
.TopLine = lNewTop
End With
End Sub
```

CommandBarEvents Object

Within the VBE, the `OnAction` property of a command bar button has no effect—the routine named in this property is *not* run when the button is clicked. Instead, the VBE object model provides you with the `CommandBarEvents` object, which hooks into whichever command bar button you tell it to, either your own custom buttons or built-in items, and raises events for the button's actions. In Office 2007 it only raises the `Click` event, and hence provides exactly the same functionality as Excel's `OnAction` and the `Click` event of the `CommandBarButton`. It was introduced for Excel 97 and is now rarely used.

CommandBarEvents Events

Name	Parameters	Description
Click	CommandBarControl As Object, handled As Boolean, Cancel Default As Boolean	<p>Triggered when a hooked command bar button is clicked. The <code>CommandBarControl</code> is passed to the event.</p> <p>A single control can be hooked by many <code>CommandBarEvents</code> objects. The events are fired in reverse order of setting up (most recently set up fires first). An event handler can set the <code>handled</code> flag to <code>True</code> to tell subsequent handlers that the event has already been processed.</p> <p>The <code>CommandBarEvents</code> object can also be used to hook into built-in menu items. If you want to handle the event through code, you can set the <code>CancelDefault</code> flag to <code>True</code> to stop the menu's normal action.</p>

To demonstrate the use of `CommandBarEvents`, in a class module called `CBarEvents`, add the following code:

```
Public WithEvents oCBEvents As VBIDE.CommandBarEvents

'Hook into the Click event for the menu item
Private Sub oCBEvents_Click(ByVal CommandBarControl As Object, _
    handled As Boolean, CancelDefault As Boolean)

    Debug.Print "Clicked " & CommandBarControl.Caption
End Sub
```

In a normal module, add the following code:

```
'Declare a collection to hold all the instances of our events class
Dim ocolMenus As New Collection

Sub AddMenus()

    'Declare some CommandBar items
    Dim oBar As CommandBar
    Dim oBtn1 As CommandBarButton, oBtn2 As CommandBarButton

    'And an object to hold instances of your events class
    Dim oCBE As CBarEvents

    'Get the VBE's menu bar
    Set oBar = Application.VBE.CommandBars("Menu Bar")

    'Add a menu item to it
    Set oBtn1 = oBar.Controls.Add(Type:=msoControlButton, temporary:=True)
    oBtn1.Caption = "Menu1"
    oBtn1.Style = msoButtonCaption

    'Create a new instance of your CommandBarEvent handler
    Set oCBE = New CBarEvents

    'Link your CommandBarEvent handler to the menu item you just created
    Set oCBE.oCBEEvents = Application.VBE.Events.CommandBarEvents(oBtn1)

    'And add the instance of your event handler to the collection
    ocolMenus.Add oCBE

    'Repeat for a second menu
    Set oBtn2 = oBar.Controls.Add(Type:=msoControlButton, temporary:=True)
    oBtn2.Caption = "Menu2"
    oBtn2.Style = msoButtonCaption

    Set oCBE = New CBarEvents
    Set oCBE.oCBEEvents = Application.VBE.Events.CommandBarEvents(oBtn2)
    ocolMenus.Add oCBE
End Sub
```

When you run the `AddMenus` routine, two menus are added to the VBE standard menu bar, which both use your `CommandBarEvents` handling class to hook into their `Click` event. When you click each of the menu items, the Immediate window displays the menu's caption.

Events Object

The `Events` object is a high-level container for the VBE's event model. In Office 2007, it contains event objects associated with clicking a command bar button and adding or removing references. The VBE extensibility model is based on the Visual Basic extensibility model, which contains a much richer set of events.

Events Properties

Name	Returns	Description
CommandBarEvents	CommandBarEvents	Read-only. Parameters: <code>CommandBarControl</code> . Performs the linking required to hook a <code>CommandBarEvents</code> object to a specific command bar button.
ReferencesEvents	ReferencesEvents	Read-only. Parameters: <code>VBProject</code> . Performs the linking required to hook a <code>ReferencesEvents</code> object to a specific project.

Examples of the Events object are included in the `CommandBarEvents` section.

LinkedWindows Collection

The `LinkedWindows` collection contains all the docked windows in the VBE workspace. COM Add-Ins written in VB6 (but not .NET) can add their own windows to this collection. Within the Office environment, you are limited to docking or undocking the built-in windows. Note that if you undock, then dock a built-in window, it does *not* go back to its original position.

LinkedWindows Collection Methods

Name	Description
Add	Method. Parameters: <code>Window As Window</code> . Docks the specified window.
Remove	Method. Parameters: <code>Window As Window</code> . Undocks the specified window.

Property Object and Properties Collection

Every `VBComponent` in a project has a `Properties` collection. The properties contained in the collection correspond to the items shown in the Properties window of the VBE. For each `VBComponent` that corresponds to the Excel objects, the `Properties` collection of the `VBComponent` also includes many of the properties of the Excel object.

The following tables define the `Collection`, `Parent`, and `VBE` properties.

Property Properties

Name	Returns	Description
<code>IndexedValue</code>	<code>Variant</code>	Parameters: <code>Index1</code> , [<code>Index2</code>], [<code>Index3</code>], [<code>Index4</code>]. The <code>Value</code> of the <code>Property</code> can be an array of up to four indices. The <code>IndexedValue</code> can be used to read a single item in the returned array.

Name	Returns	Description
Name	String	Read-only. Returns the name of the property, and is also used to refer to a specific property.
NumIndices	Integer	Read-only. If the value of the Property is an array, NumIndices returns the number of indices (dimensions) in the array. If not an array, it returns 0.
Object	Object	Used to obtain a reference to the object returned by the Property, if any.
Value	Variant	The value of the Property.

It is easy to get confused between the many types of Name property of a VBComponent, which are summarized in the following table.

Syntax	Refers to
Worksheet.CodeName	The code name of the VBComponent (read-only).
VBComponent.Name	The code name of the VBComponent (read/write).
VBComponent.Properties ("CodeName")	The code name of the VBComponent (read-only). (This was the only reliable way to change a worksheet's CodeName in Excel 97.)
VBComponent.Properties ("_CodeName")	The code name of the VBComponent (read/write).
VBComponent.Properties ("Name")	The name of the worksheet (read/write).
VBComponent.Properties ("Name").Name	"Name".

This simple example identifies the workbook containing a given VBComponent:

```
Sub IdentifyWorkbook()
    Dim oBk As Workbook

    'Get the workbook containing a given VBComponent
    Set oBk =
Application.VBE.ActiveVbProject.VBComponents("Sheet1").Properties("Parent").Object

    MsgBox oBk.Name
End Sub
```

Reference Object and References Collection

A `Reference` is a link from your `VBProject` to an external file, which may be an object library (for example, linking to the Word object library), a control (for example, Windows Common Controls), an ActiveX DLL, or another `VBProject`. By creating a reference to the external object, you can implement early binding, meaning that the referenced objects run in the same memory area, all the links are evaluated at compile time, and Excel provides ToolTip programming help when working with the referenced objects.

When you run your application on another machine, it may not have all the objects that your application requires. The `Reference` object and `References` collection provide access to these references, allowing you to check that they are all present and working before you try to use them.

The tables that follow define the `Collection` and VBE properties.

Reference Properties

Name	Returns	Description
<code>BuiltIn</code>	<code>Boolean</code>	Read-only. Returns if the reference is built-in or added by the developer. The Visual Basic for Applications and Microsoft Excel 12.0 Object Library references are built-in and cannot be removed.
<code>Description</code>	<code>String</code>	Read-only. Returns the description of the reference, which is the text shown in the Object Browser.
<code>FullPath</code>	<code>String</code>	Read-only. Returns the path to the workbook, DLL, OCX, TLB, or OLB file that is the source of the reference.
<code>Guid</code>	<code>String</code>	Read-only. Returns the globally unique identifier for the reference.
<code>IsBroken</code>	<code>Boolean</code>	Read-only. Returns <code>True</code> if the reference is broken (is not available on the machine).
<code>Major</code>	<code>Long</code>	Read-only. Returns the major version number of the referenced file.
<code>Minor</code>	<code>Long</code>	Read-only. Returns the minor version number of the referenced file.
<code>Name</code>	<code>String</code>	Read-only. Returns a short name for the reference (for example, VBA or Excel).
<code>Type</code>	<code>vbext_RefKind</code>	Read-only. Returns the reference type, <code>vbext_rk_TypeLib</code> for DLLs and so on, or <code>vbext_rk_Project</code> for other <code>VBProjects</code> .

References Collection Methods

Name	Returns	Description
AddFromFile	Reference	Method. Parameters: FileName As String. Adds a reference between the VBProject and a specific file. This should only be used to create references between workbooks.
AddFromGuid	Reference	Method. Parameters: Guid As String, Major As Long, Minor As Long. Adds a reference between the VBProject and a specific DLL, Typelib, and so forth. A library's file-name, location, and version may change over time, but its Guid is guaranteed to be constant. Hence, when adding a reference to a DLL, Typelib, and so on, the Guid should be used. If you require a specific version of the DLL, you can request the major and minor version numbers.
Remove		Method. Parameters: Reference As Reference. Removes a reference from the VBProject.

The `References` collection provides two events, which you can use to detect when items are added to or removed from the collection. You could use this, for example, to create a Top 10 References dialog, by using `Application` events to detect when a workbook is opened or created and hooking into the `References` collection of the workbook's `VBProject` events to detect when a particular `Reference` is added to a project. You could maintain a list of these and display them in a dialog box, similar to the existing `Tools` ⇄ `References` dialog in the VBE (but without all the clutter).

References Collection Events

Name	Parameters	Description
ItemAdded	Reference As VBIDE.Reference	Triggered when a <code>Reference</code> is added to the <code>VBProject</code> being watched.
ItemRemoved	Reference As VBIDE.Reference	Triggered when a <code>Reference</code> is removed from the <code>VBProject</code> being watched.

Putting the `References` collection to use, this example checks for broken references and alerts the user:

```
Function HasMissingRefs() As Boolean
    Dim oRef As VBIDE.Reference

    'Loop through all the references for the project
    For Each oRef In ThisWorkbook.VBProject.References

        'Is it missing?
        If oRef.IsBroken Then

            'Yes - show different messages for workbook and DLL references
            If oRef.Type = vbext_rk_Project Then
```

End FunctionReferencesEvents Object

```
        MsgBox "Could not find the workbook " & oRef.FullPath & _  
            ", which is required by this application."  
    Else  
        MsgBox "This application requires the object library '" & _  
            oRef.Description & "', which has not been installed."  
    End If  
  
    'Return that there are some missing references  
    HasMissingRefs = True  
End If  
Next
```

End FunctionReferencesEvents Object

In a similar manner to the way in which the `CommandBarEvents` object provides the `Click` event for a command bar, the `ReferencesEvents` object provides two events related to the `References` collection of a `VBProject`. The `ReferencesEvents` object appears to be redundant—all of the events it handles are also included in the `References` object of a `VBProject`. The only difference (apart from the definition) is that the `ReferencesEvents` object works with a `VBProject` object instead of the `References` collection of a `VBProject`. Note that a `VBProject` is compiled when a `Reference` is added or removed, resulting in the loss of any variables and instances of classes. Hence, a `VBProject` cannot monitor its own `References` events.

ReferencesEvents Events

Name	Parameters	Description
<code>ItemAdded</code>	<code>Reference As VBIDE.Reference</code>	Triggered when a <code>Reference</code> is added to the <code>VBProject</code> being watched.
<code>ItemRemoved</code>	<code>Reference As VBIDE.Reference</code>	Triggered when a <code>Reference</code> is removed from the <code>VBProject</code> being watched.

VBComponent Object and VBComponents Collection

The `VBComponents` collection contains all the modules, class modules (including code-behind worksheets), and `UserForms` in a `VBProject`; they are all different types of `VBComponent`. Every `VBComponent` has a `CodeModule` to store its code, and some `VBComponents` (such as a `UserForm`) have a graphical development interface, called its `Designer`. Through the `Designer`, you can modify the graphical elements of the `VBComponent`, such as adding controls to a `UserForm`.

This section defines the `Collection` and `VBE` properties.

VBComponent Properties

Name	Returns	Description
CodeModule	CodeModule	Read-only. Returns the <code>CodeModule</code> for the component, used to store its VBA code.
Designer		Read-only. Returns the <code>Designer</code> object for the component, which provides access to the design-time graphical elements of the component.
DesignerID	String	Read-only. Returns an identifier for the <code>Designer</code> , so you know what sort of <code>Designer</code> it is. For example, a UserForm's designer ID is <code>Forms.Form</code> .
DesignerWindow	Window	Read-only. Returns a <code>Window</code> object, representing the <code>Window</code> displaying the <code>Designer</code> . (Shown as a method in the Object Browser, as it opens the <code>Window</code> if not already open.)
HasOpenDesigner	Boolean	Read-only. Identifies if the component's <code>Designer</code> is open.
Name	String	The name of the <code>VBComponent</code> .
Properties	Properties	Read-only. Returns the component's <code>Properties</code> collection, providing access to the items shown in the Property window, and to many of the associated Excel object's properties if the <code>VBComponent</code> represents the code behind an Excel object. See the <code>Property</code> object in this appendix for more information.
Saved	Boolean	Read-only. Returns whether the contents of the <code>VBComponent</code> has changed since the last save. It is analogous to an Excel workbook's <code>Saved</code> property, but applies to each component individually.
Type	<code>vbext_ComponentType</code>	Read-only. Returns the type of the component: Normal module: <code>vbext_ct_StdModule</code> Class module: <code>vbext_ct_ClassModule</code> UserForm: <code>vbext_ct_MSForm</code> Excel object: <code>vbext_ct_Document</code> All other types: <code>vbext_ct_ActiveXDesigner</code>

VBComponent Methods

Name	Parameters	Description
Activate		Displays the <code>VBComponent</code> 's main window (code module or designer) and sets the focus to it.
Export	FileName As String	Saves the component as a file, separate from the workbook.

VBComponents Collection Methods

Name	Returns	Description
Add	VBComponent	Parameters: ComponentType. Adds a new, built-in VBComponent to the project. The ComponentType can be one of vbext_ct_StdModule, vbext_ct_ClassModule, or vbext_ct_MSForm.
AddCustom	VBComponent	Parameters: ProgId. Adds a new, custom VBComponent to the project. The result is always of type vbext_ct_ActiveXDesigner. It seems that custom VB components can only be added to ActiveX DLL projects and not to Excel workbook projects.
Import	VBComponent	Parameters: FileName. Adds a new VBComponent to the project from a file (usually a previously exported VBComponent).
Remove		Parameters: VBComponent. Removes a VBComponent from a project.

Many of the examples in this section and in Chapter 26 use the VBComponent object and its properties and methods. The example that follows exports a UserForm from the workbook containing the code, imports it into a new workbook, and renames it. It then adds a standard module, fills in some code to show the form, then calls the routine to show the form in the new workbook:

```
Sub CopyAndShowUserForm()  
    Dim oNewBk As Workbook, oVBC As VBIDE.VBComponent  
  
    'Create a new workbook  
    Set oNewBk = Workbooks.Add  
  
    'Export a UserForm from this workbook to disk  
    ThisWorkbook.VBProject.VBComponents("UserForm1").Export "c:\temp.frm"  
  
    'Import the UserForm into the new workbook  
    Set oVBC = oNewBk.VBProject.VBComponents.Import("c:\temp.frm")  
  
    'Rename the UserForm  
    oVBC.Name = "MyForm"  
  
    'Add a standard module to the new workbook  
    Set oVBC = oNewBk.VBProject.VBComponents.Add(vbext_ct_StdModule)  
  
    'Add some code to the standard module, to show the form  
    oVBC.CodeModule.AddFromString _  
        "Sub ShowMyForm()" & vbCrLf & _  
        "    MyForm.Show" & vbCrLf & _  
        "End Sub" & vbCrLf  
  
    'Close the code pane the Excel opened when you added code to the module
```

```

oVBC.CodeModule.CodePane.Window.Close

'Delete the exported file
Kill "c:\temp.frm"

'Run the new routine to show the imported UserForm
Application.Run oNewBk.Name & "!ShowMyForm"
End Sub

```

VBE Object

The **VBE** object is the top-level object in the **VBIDE** object library, and hence is analogous to the **Application** object in the Excel library. Its main jobs are to act as a container for the **VBIDE**'s command bars, add-ins, windows, and so on, and to provide information about the objects currently being modified by the user. Unfortunately, the **VBE** object does not expose any of the **VBIDE**'s options settings (code settings, edit formats, error handling, and so on), nor does it provide any editing events (such as selecting a different project, or adding or deleting lines of code).

VBE Properties

Name	Returns	Description
ActiveCodePane	CodePane	Returns or sets the CodePane currently being edited by the user. Typically used to identify which object is being worked on, or to force the user to work with a specific code pane.
ActiveVBProject	VBProject	Returns or sets the VBProject selected in the Project Explorer window. If the Project Explorer is showing a VBComponent selected, this property returns the VBProject containing the component.
ActiveWindow	Window	Read-only. Returns the active Window , which may be a code pane, designer, or one of the VBIDE windows (such as Project Explorer, Immediate window, and so on).
Addins	Addins	Read-only. Returns a collection of all the COM Add-Ins registered for use in the VBIDE . See the AddIn object for more information.
CodePanels	CodePanels	Read-only. Returns a collection of all the open CodePanels in the VBIDE . See the CodePane object for more information.
CommandBars	Command Bars	Read-only. Returns a collection of all the command bars in the VBIDE .
Events	Events	Read-only. Returns an object containing all the events in the VBIDE . See the Events object for more information.
MainWindow	Window	Read-only. Returns a Window object representing the main window of the VBIDE .

Table continued on following page

VBProject Object and VBProjects Collection

Name	Returns	Description
SelectedVBComponent	VbComponent	Read-only. Returns the VbComponent object that is shown as selected in the Project Explorer window. Note that this usually, but not always, corresponds to the ActiveCodePane.
VBProjects	VBProjects	Read-only. Returns a collection of all the VBProjects in the VBIDE, both Excel workbooks and ActiveX DLLs.
Version	String	Read-only. Returns the version number of the Extensibility library (shows 6.05 for Office 2007).
Windows	Windows	Read-only. Returns a collection of all the open windows in the VBIDE. See the Windows object for more information.

Most of the examples in this section and in Chapter 26 include the VBE's properties. The following line displays the VBE:

```
Application.VBE.MainWindow.Visible = True
```

VBProject Object and VBProjects Collection

A VBProject represents all of the code for a workbook, including code behind sheets, modules, class modules, and UserForms. In the Developer edition of Office 2002 (and only in more recent versions when installed by upgrading from Office 2002), a VBProject can also be a standalone project, compiled as an ActiveX DLL.

VBProject Common Properties

This section defines the Collection and VBE properties.

VBProject Properties

Name	Returns	Description
BuildFileName	String	For ActiveX DLLs only, gets or sets the name of the DLL file to compile the project into.
Description	String	For ActiveX DLLs only, the description of the DLL as it will appear in the Tools ⇄ References list.
FileName	String	Read-only. For workbook projects, returns the full name of the workbook. For ActiveX DLL projects, returns the name of the source code version of the project *.vba. If the file has not been saved, a run-time error occurs if you try to read this property.

Name	Returns	Description
HelpContextID	Long	Identifies the default help-file context ID for the project.
HelpFile	String	Gets or sets the help file for a project. Each of the UserForms and controls within the project can be assigned a context ID to show a page from this help file.
Mode	Vbext_ VBAMode	Read-only. Returns the VBProject's operation mode (Design, Run, or Break). Note that a VBProject can have different execution modes (for example, an ActiveX COM Add-In project can be running while you are in Design mode on a different project).
Name	String	The name of the project.
Protection	Vbext_ Project Protection	Read-only. Returns whether the project is locked for viewing. Locked projects only expose their VBProject object. Any attempt to navigate below the VBProject level results in an error. Note that if a VBProject is set to Protected, but is unprotected by the user during a session, its Protection property shows as vbext_pp_none for the remainder of that session.
References	References	Read-only. Returns the collection of References for the VBProject. See the References object for more information.
Saved	Boolean	Read-only. Returns whether the VBProject has been changed since the last save. For Excel projects, this should agree with the workbook's Saved property.
Type	Vbext_ Project Type	Read-only. Returns the type of project — host project (an Excel workbook, Word document, Access database, and so on) or an ActiveX DLL project.
VBComponents	VBComponents	Read-only. Returns the collection of VBComponents in the project. See the VBComponent object for more information.

VBProject Methods

Name	Parameters	Description
MakeCompiledFile		For ActiveX DLL projects only. Compiles the project and makes the DLL file.
SaveAs	FileNameAs String	For ActiveX DLL projects only. Saves the project file.

VBProjects Collection Methods

Name	Returns	Description
Add	VBProject	Method. Parameters: Type. Adds a new project to the VBE. Can only successfully add standalone (ActiveX DLL) projects using this method.
Remove		Method. Parameters: lpc As VBProject. Removes a VBProject from the VBE. Can only be used for ActiveX DLL projects.

Most of the examples in this section use the `VBProject` object and its properties. This example lists the names of all the `VBComponents` in all the unlocked projects in the VBE:

```
Sub PrintComponents()  
    Dim oVBP As VBIDE.VBProject  
    Dim oVBC As VBIDE.VBComponent  
  
    'Loop through all the projects in the VBE  
    For Each oVBP In Application.VBE.VBProjects  
  
        'If the project is not protected...  
        If oVBP.Protection = vbext_pp_none Then  
  
            '... loop through its components  
            For Each oVBC In oVBP.VBComponents  
                Debug.Print oVBP.Name & "." & oVBC.Name  
            Next  
        End If  
    Next  
End Sub
```

Window Object and Windows Collection

The `Window` object represents a single window in the VBE, including the VBE's main window, the built-in Project Explorer, Immediate, Debug, and Watch windows, and so on, as well as all open `CodePanels` and `Designer` windows.

Window Common Properties

The `Collection` and `VBE` properties are defined at the beginning of this section.

Window Properties

Name	Returns	Description
Caption	String	Read-only. Returns the caption of the <code>Window</code> , as shown in its title bar.

Name	Returns	Description
Height	Long	The height of the <code>Window</code> , in twips (1 twip = 1/20 points). Does not affect docked windows.
HWnd	Long	Hidden. Read-only. Returns a handle to the <code>Window</code> , for use in Windows API calls.
Left	Long	The left edge of the <code>Window</code> on the screen, in twips (1 twip = 1/20 points). Does not affect docked windows.
LinkedWindowFrame	Window	Read-only. Multiple windows can be linked together in the VBE (for example, while docking them). This property returns another <code>Window</code> that represents the frame surrounding the docked windows. Returns <code>Nothing</code> if the window is not linked.
LinkedWindows	LinkedWindows	Read-only. Returns a collection of windows linked to the <code>Window</code> (for example, when docked).
Top	Long	The top of the <code>Window</code> on the screen, in twips (1 twip = 1/20 points). Does not affect docked windows.
Type	vbext_ WindowType	Read-only. Returns the window type, such as <code>CodePane</code> , <code>Immediate window</code> , <code>Main window</code> , and so on.
Visible	Boolean	Gets or sets whether or not the window is visible.
Width	Long	The width of the <code>Window</code> , in twips (1 twip = 1/20 points). Does not affect docked windows.
WindowState	vbext_ WindowState	The <code>Window</code> state.

Window Methods

Name	Description
Close	Closes the window.
SetFocus	Opens and activates the window, displays it, and gives it the focus.

Windows Collection Methods

Name	Returns	Description
CreateToolWindow	Window	Parameters: <code>AddInInst</code> , <code>ProgId</code> , <code>Caption</code> , <code>GuidPosition</code> , <code>DocObj</code> . This method is only used when creating COM Add-Ins using VB6, to create a dockable window in the VBE.

Windows Collection Methods

This example closes all code and designer windows in the VBE:

```
Sub CloseAllCodeWindows()  
    Dim oWin As VBIDE.Window  
  
    'Loop through all the open windows in the VBE  
    For Each oWin In Application.VBE.Windows  
  
        'Close the window, depending on its type  
        Select Case oWin.Type  
            Case vbext_wt_Browser, vbext_wt_CodeWindow, vbext_wt_Designer  
  
                'Close the Object Browser, code windows and designer windows  
                Debug.Print "Closed '" & oWin.Caption & "' window."  
                oWin.Close  
  
            Case Else  
                'Don't close any other windows  
                Debug.Print "Kept '" & oWin.Caption & "' window open."  
        End Select  
    Next  
End Sub
```



Office 2007 Object Model

Common Properties with Collections and Associated Objects

Most of the objects in the Office object model have objects with associated collections. The collection object is usually the plural form of the associated object. For example, the `CommandBars` collection holds a collection of `CommandBar` objects. For simplicity, all the objects and associated collections are grouped together under the same heading.

In most cases, the purpose of the collection object is only to hold a collection of the same objects. The common properties of the collection objects are listed in the following section. Only unique properties, methods, or events are mentioned in each object section.

Common Collection Properties

Name	Returns	Description
<code>Application</code>	<code>Application</code>	Read-only. Returns a reference to the application owning the current object.
<code>Count</code>	<code>Long</code>	Read-only. Returns the number of objects in the collection.
<code>Creator</code>	<code>Long</code>	Read-only. Returns a <code>Long</code> number that describes which application the object was created in. Macintosh only.
<code>Parent</code>	<code>Object</code>	Read-only. The <code>Parent</code> object is the container object of the collection object. For example, <code>Workbooks.Parent</code> returns a reference to the <code>Application</code> object.

Common Object Properties

Objects also have some common properties. To avoid redundancy, the common properties of all objects are listed next. They will be mentioned in each object description as existing but are only defined here.

Name	Returns	Description
Application	Application	Read-only. Returns a reference to the application owning the current object.
Creator	Long	Read-only. Returns a Long number that describes which application the object was created in. Macintosh only.
Parent	Object	Read-only. The container object of the current object. For example, in Excel Shapes (1) .Parent may return a reference to a Worksheet object, since a Worksheet object is one of the possible containers of a Shapes object.

Office Objects and Their Properties and Events

The objects are listed in alphabetical order. Each object has a general description of the object and possible parent objects. This is followed by a table format of each of the object's properties and methods. The last section of each object describes some code examples of the object's use.

BulletFormat2 Object

The `BulletFormat2` object exposes the properties and methods used to configure the formatting options of bullets, such as color and size.

BulletFormat2 Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
Character	Boolean	Set/Get the Unicode character value that is used for bullets in the specified text.
Font	Font2	Read-only. Returns a <code>Font2</code> object that represents character formatting for a given <code>BulletFormat2</code> object.
Number	Long	Read-only. Returns the bullet number of a paragraph.
RelativeSize	Single	Set/Get the size of a given bullet relative to the size of the first text character in the paragraph.
StartValue	Long	Set/Get the beginning value of a bulleted list.

Name	Returns	Description
Style	MsoNumberedBulletStyle	Get/Set an MsoNumberedBulletStyle constant that defines the style of a bullet.
Type	MsoBulletType	Get/Set an MsoBulletType constant that defines the type of bullet used.
UseTextColor	Boolean	Set/Get whether the specified bullets are set to the color of the first text character in the paragraph.
UseTextFont	Boolean	Set/Get whether the specified bullets are set to the font of the first text character in the paragraph.
Visible	Boolean	Set/Get whether the specified bullets are visible.

BulletFormat2 Methods

Name	Returns	Parameters	Description
Picture		Filename	Sets the graphics file to be used for bullets in a bulleted list. Valid graphics files include: .bmp, .cdr, .cgm, .drw, .dxf, .emf, .eps, .gif, .jpg, .jpeg, .pcd, .pct, .pcx, .pict, .png, .tga, .tiff, .wmf, and .wpg.

COMAddinObject and the COMAddins Collection Object

COMAddins object represents a single COM Add-In in the Microsoft Office host application, and is also a member of COMAddins collection. COMAddins are custom solutions for use with several Office applications like Excel, Access, Word, and Outlook developed in any language (VB, C++, or J++) that supports COM (Component Object Model) components. The COMAddins collection is a list of all COMAddins objects for a Microsoft Office host application, in this case Excel.

COMAddins Collection Properties

The Application, Count, Creator, and Parent properties are defined at the beginning of this appendix.

COMAddins Collection Methods

Name	Returns	Parameters	Description
Item	COMAddIn	Index as Variant	Returns a member of the specified COMAddins collection.
Update			Updates the contents of the COMAddins collection from the list of Add-Ins stored in the Windows registry.

COMAddinProperties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
Connect	Boolean	Set/Get the state of the connection for the specified <code>COMAddIn</code> object.
Description	String	Set/Get a descriptive String value for the specified <code>COMAddIn</code> object.
Guid	String	Read-only. Returns the globally unique class identifier (Guid) for the specified <code>COMAddIn</code> object.
Object	Object	Set/Get the object that is the basis for the specified <code>COMAddIn</code> object. Used primarily to communicate with other <code>COMAddins</code> .
ProgId	String	Read-only. Returns the programmatic identifier (ProgID) for the specified <code>COMAddIn</code> object.

COMAddin Object Example

The following routine loops through the list of `COMAddins` and displays its relevant information in a table on `Sheet1` of the workbook containing the code:

```
Sub COMAddinInfo()  
  
    Dim lRow As Long  
    Dim oCom As COMAddIn  
  
    ' Set up the headings on Sheet1 of this workbook  
    With Sheet1.Range("A1:D1")  
        .Value = Array("Guid", "ProgId", "Creator", "Description")  
        .Font.Bold = True  
        .HorizontalAlignment = xlCenter  
    End With  
  
    ' Loop through the COMAddins collection and place  
    ' its information in cells below the headings  
    If Application.COMAddIns.Count Then  
        For Each oCom In Application.COMAddIns  
  
            With Sheet1.Range("A2")  
                .Offset(lRow, 0).Value = oCom.GUID  
                .Offset(lRow, 1).Value = oCom.ProgID  
                .Offset(lRow, 2).Value = oCom.Creator  
                .Offset(lRow, 3).Value = oCom.Description  
            End With  
            lRow = lRow + 1  
        Next oCom  
    End If  
End Sub
```



```
' Autofit the table
Sheet1.Range("A1:D1").EntireColumn.AutoFit

End Sub
```

CommandBar Object and the CommandBars Collection Object

The `CommandBar` object holds the properties and methods for a specific `CommandBar` in the `CommandBars` collection. The properties and methods are similar to the `CommandBars` collection, but only apply to the individual `CommandBar` referenced. The `CommandBars` collection contains a list of all `CommandBars` (known as `Toolbars` to most users) in the container application. Use `CommandBars(Index)` to return a reference to a specific `CommandBar`, as in the following example:

```
Dim oBar As CommandBar
Set oBar = CommandBars("Wrox")
```

CommandBars Collection Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>ActionControl</code>	<code>CommandBarControl</code>	Read-only. Returns the <code>CommandBarControl</code> object whose <code>OnAction</code> property is set to the running procedure. If the running procedure was not initiated by a command bar control, this property returns <code>Nothing</code> .
<code>ActiveMenuBar</code>	<code>CommandBar</code>	Read-only. Returns a <code>CommandBar</code> object that represents the active menu bar in the container application. This almost always returns the application's <code>Worksheet</code> menu bar.
<code>AdaptiveMenus</code>	<code>Boolean</code>	Set/Get whether adaptive (abbreviated) menus are enabled.
<code>DisableAskAQuestionDropDown</code>	<code>Boolean</code>	Set/Get whether the Answer Wizard drop-down menu is enabled. When set to <code>True</code> , the drop-down disappears from the menu bar.
<code>DisableCustomize</code>	<code>Boolean</code>	Set/Get whether toolbar customization is disabled. When <code>True</code> , the <code>Customize</code> command becomes disabled on the <code>Tools</code> menu and disappears from the <code>Toolbar</code> 's shortcut (right-click) menu.
<code>DisplayFonts</code>	<code>Boolean</code>	Set/Get whether the font names in the <code>Font</code> box are displayed in their actual fonts. Recommend setting this to <code>False</code> on older computer systems with fewer resources.
<code>DisplayKeysInTooltips</code>	<code>Boolean</code>	Set/Get whether shortcut keys are displayed in the <code>ToolTips</code> for each command bar control. This property has no effect on Excel's command bars.

Table continued on following page

CommandBars Collection Methods

Name	Returns	Description
Display Tooltips	Boolean	Set/Get whether ScreenTips are displayed whenever the user positions the pointer over command bar controls.
Item	CommandBar	Read-only. Returns a CommandBar object from the CommandBars collection with the Index value specified by the Index parameter. Index can also be a string representing the name of the CommandBar.
Large Buttons	Boolean	Set/Get whether the toolbar buttons displayed are larger than normal size.
Menu Animation Style	MsoMenu Animation	Set/Get the animation type of all CommandBarPopup controls (menus) in the CommandBars collection.

CommandBars Collection Methods

Name	Returns	Parameters	Description
Add	CommandBar	Name, Position, MenuBar, Temporary	Creates a new command bar and adds it to the collection of command bars.
ExecuteMso		idMso	Executes a given control identified by the idMso parameter.
FindControl	CommandBar Control	Type, Id, Tag, Visible	Returns a single CommandBarControl object that fits a specified criterion based on the parameters.
FindControls	CommandBar Controls	Type, Id, Tag, Visible	Returns a series of CommandBarControl objects in a collection that fits the specified criteria based on the parameters.
GetEnabled Mso	Boolean	idMso	Returns True if the control identified by the idMso parameter is enabled.
GetImagedMso	IPictureDisp	idMso, Width, Height	Returns an IPictureDisp object of the control image identified by the idMso parameter.
GetLabelMso	Variant	idMso	Returns the label of the control identified by the idMso parameter as a String.
GetPressed Mso	Boolean	idMso	Returns whether a given toggle Button control identified by the idMso parameter is pressed.
GetScreen tipMso	Variant	idMso	Returns the ScreenTip of the control identified by the idMso parameter as a String.

Name	Returns	Parameters	Description
GetSupertip Mso	Variant	idMso	Returns the supertip of the control identified by the idMso parameter as a String.
GetVisible Mso	Boolean	idMso	Returns True if the control identified by the idMso parameter is visible.
ReleaseFocus		idMso	Releases the user interface focus from all command bars.

CommandBars Collection Events

Name	Parameters	Description
OnUpdate		The OnUpdate event is recognized by the CommandBar object and all command bar controls. Due to the large number of OnUpdate events that can occur during normal usage, Excel developers should exercise caution when using this event.

CommandBar Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

Name	Returns	Description
Adaptive Menu	Boolean	Set/Get whether short menus are shown.
BuiltIn	Boolean	Read-only. Returns True if the specified command bar or command bar control is a built-in command bar or control of the container application. Returns False if it's a custom command bar or control, or if it's a built-in control whose OnAction property has been set.
Context	String	Set/Get a string that determines where a command bar will be saved. The string is defined and interpreted by the application.
Controls	CommandBar Controls	Read-only. Returns a CommandBarControl object that represents all the controls on a command bar.
Enabled	Boolean	Set/Get whether the CommandBar is enabled. Setting this property to True causes the name of the command bar to appear in the list of available command bars.
Height	Long	Set/Get the height of the CommandBar.
Index	Long	Read-only. Returns the index number for a CommandBar in the CommandBars collection.

Table continued on following page

CommandBar Methods

Name	Returns	Description
Left	Long	Set/Get the distance (in pixels) of the left edge of the command bar relative to the screen.
Name	String	Set/Get the name of the CommandBar.
NameLocal	String	Set/Get the name of a built-in command bar as it's displayed in the language version of the container application, or the name of a custom command bar.
Position	MsoBar Position	Set/Get the position of the command bar.
Protection	MsoBar Protection	Set/Get the way a command bar is protected from user customization.
RowIndex	Long	Set/Get the docking order of a command bar in relation to other command bars in the same docking area. Can be an integer greater than 0, or either of the following MsoBar Row constants: msoBarRowFirst or msoBarRowLast.
Top	Long	Set/Get the distance (in points) from the top of the command bar to the top edge of the screen.
Type	MsoBar Type	Read-only. Returns the type of command bar.
Visible	Boolean	Set/Get whether the command bar is visible. The Enabled property for a command bar must be set to True before the visible property is set to True.
Width	Long	Set/Get the width (in pixels) of the specified command bar.

CommandBar Methods

Name	Returns	Parameters	Description
Delete			Deletes the specified CommandBar from the CommandBars collection.
Find Control	CommandBar Control	Type, Id, Tag, Visible, Recursive	Returns a CommandBar that fits the specified criteria.
Reset			Resets a built-in CommandBar to its default configuration.
ShowPopup		x, y	Displays the CommandBar as a shortcut menu at specified coordinates or at the current pointer coordinates.

CommandBarButton Object

A `CommandBarButton` is any button or menu item on any `CommandBar`. You access a specific `CommandBarButton` by referencing the `CommandBar` it's located in and by using `Controls(Index)`. `Index` can either be the `CommandBarButton` object's number position on the menu or toolbar, or its `Caption`.

For example, you can refer to the first control on a `CommandBar` called "Wrox" using:

```
CommandBars("Wrox").Controls(1)
```

Or:

```
CommandBars("Wrox").Controls("Member Info")
```

CommandBarButton Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>BeginGroup</code>	Boolean	Set/Get whether the specified <code>CommandBarButton</code> appears at the beginning of a group of controls on the command bar.
<code>BuiltIn</code>	Boolean	Read-only. Returns <code>True</code> if the specified command bar or command bar control is a built-in command bar or control of the container application. Returns <code>False</code> if it's a custom command bar or control, or if it's a built-in control whose <code>OnAction</code> property has been set.
<code>BuiltInFace</code>	Boolean	Set/Get whether the face of the <code>CommandBarButton</code> control is its original built-in face. This property can only be set to <code>True</code> , which will reset the face to the built-in face.
<code>Caption</code>	String	Set/Get the caption text of the <code>CommandBarButton</code> .
<code>DescriptionText</code>	String	Set/Get the description for a <code>CommandBarButton</code> . The description is not displayed to the user, but it can be useful for documenting the behavior of the control for other developers.
<code>Enabled</code>	Boolean	Set/Get whether the <code>CommandBarButton</code> object is enabled.
<code>FaceId</code>	Long	Set/Get the <code>Id</code> number for the face of the <code>CommandBarButton</code> .
<code>Height</code>	Long	Set/Get the height of the <code>CommandBarButton</code> .
<code>HelpContextId</code>	Long	Set/Get the Help context <code>Id</code> number for the Help topic attached to the <code>CommandBarButton</code> .

Table continued on following page

CommandBarButton Properties

Name	Returns	Description
HelpFile	String	Set/Get the filename for the Help topic for the CommandBarButton.
HyperlinkType	MsoCommandBarButtonHyperlinkType	Set/Get the type of hyperlink associated with the specified CommandBarButton.
Id	Long	Read-only. Returns the ID for a built-in CommandBarButton.
Index	Long	Read-only. Returns the index number for a CommandBarButton in the CommandBars collection.
IsPriorityDropped	Boolean	Read-only. Returns whether the CommandBarButton is currently dropped from the menu or toolbar, based on usage statistics and layout space. (Note that this is not the same as the control's visibility, as set by the Visible property.) A CommandBarButton with Visible set to True will not be immediately visible on a Personalized menu or toolbar if IsPriorityDropped is True.
Left	Long	Read-only. Returns the horizontal position of the CommandBarButton (in pixels) relative to the left edge of the screen. Returns the distance from the left side of the docking area.
Mask	IPictureDisp	Returns an IPictureDisp object representing the mask image of a CommandBarButton object. The mask image determines what parts of the button image are transparent.
OLEUsage	MsoControlOLEUsage	Set/Get the OLE client and OLE server roles in which a CommandBarButton will be used when two Microsoft Office applications are merged.
OnAction	String	Set/Get the name of a Visual Basic procedure that will run when the user clicks or changes the value of a CommandBarButton.
Parameter	String	Set/Get a string that an application can use to execute a command.
Picture	IPictureDisp	Set/Get an IPictureDisp object representing the image of the CommandBarButton.
Priority	Long	Set/Get the priority of a CommandBarButton.
ShortcutText	String	Set/Get the shortcut key text displayed next to the CommandBarButton control when the button appears on a menu, submenu, or shortcut menu.
State	MsoButtonState	Set/Get the appearance of the CommandBarButton.

Name	Returns	Description
Style	MsoButton Style	Set/Get the way a CommandBarButton is displayed.
Tag	String	Set/Get information about the CommandBarButton — for example, data to be used as an argument in procedures.
TooltipText	String	Set/Get the text displayed in the CommandBarButton's ScreenTip.
Top	Long	Read-only. Returns the number of pixels from the top edge of a given command bar and the top edge of the application window.
Type	MsoControl Type	Read-only. Returns the specified command bar control's type name.
Visible	Boolean	Set/Get whether the CommandBarButton is visible.
Width	Long	Set/Get the width (in pixels) of the specified CommandBarButton.

CommandBarButton Methods

Name	Returns	Parameters	Description
Copy	CommandBar Control	Bar, Before	Copies a CommandBarButton to an existing command bar.
CopyFace			Copies the face of a CommandBarButton to the Clipboard.
Delete		Temporary	Deletes the specified CommandBarButton from its collection. Set Temporary to True to delete the control for the current session only — the application will display the control again in the next session.
Execute			Runs the procedure or built-in command assigned to the specified CommandBarButton. For custom controls, use the OnAction property to specify the procedure to be run.
Move	CommandBar Control	Bar, Before	Moves the specified CommandBarButton to an existing command bar.
PasteFace			Pastes the contents of the Clipboard onto a CommandBarButton.

Table continued on following page

CommandBarButton Events

Name	Returns	Parameters	Description
Reset			Resets a built-in <code>CommandBarButton</code> to its default configuration, or resets a built-in <code>CommandBarButton</code> to its original function and face.
SetFocus			Moves the keyboard focus to the specified <code>CommandBarButton</code> . If the control is disabled or isn't visible, this method will fail.

CommandBarButton Events

Name	Parameters	Description
Click	ByVal Ctrl As <code>CommandBarButton</code> , CancelDefault As <code>Boolean</code>	Triggered when a user clicks a <code>CommandBarButton</code> . <code>Ctrl</code> denotes the control that initiated the event. <code>CancelDefault</code> is <code>False</code> if the default behavior associated with the <code>CommandBarButton</code> control occurs, unless cancelled by another process or <code>Add-In</code> .

CommandBarComboBox Object

This object represents a drop-down list, custom edit box, or `ComboBox` (combination of the first two) control on any `CommandBar`. These types of controls only appear on the command bar — when it's either floating or docked at either the top or bottom of the Application window.

CommandBarComboBox Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
BeginGroup	<code>Boolean</code>	Set/Get whether the specified <code>CommandBarComboBox</code> appears at the beginning of a group of controls on the command bar.
BuiltIn	<code>Boolean</code>	Read-only. Returns <code>True</code> if the specified command bar or command bar control is a built-in command bar or control of the container application. Returns <code>False</code> if it's a custom command bar or control, or if it's a built-in control whose <code>OnAction</code> property has been set.
Caption	<code>String</code>	Set/Get the caption text of the <code>CommandBarComboBox</code> .
DescriptionText	<code>String</code>	Set/Get the description for a <code>CommandBarComboBox</code> . The description is not displayed to the user, but it can be useful for documenting the behavior of the control for other developers.

CommandBarComboBox Properties

Name	Returns	Description
DropDownLines	Long	Set/Get the number of lines in a <code>CommandBarComboBox</code> . The <code>ComboBox</code> control must be a custom control. Note that an error occurs if you attempt to set this property for a <code>ComboBox</code> control that's an edit box or a built-in <code>ComboBox</code> control.
DropDownWidth	Long	Set/Get the width (in pixels) of the list for the specified <code>CommandBarComboBox</code> . Note that an error occurs if you attempt to set this property for a built-in control.
Enabled	Boolean	Set/Get whether the <code>CommandBarComboBox</code> object is enabled.
Height	Long	Set/Get the height of the <code>CommandBarComboBox</code> .
HelpContextId	Long	Set/Get the Help context Id number for the Help topic attached to the <code>CommandBarComboBox</code> .
HelpFile	String	Set/Get the file name for the Help topic for the <code>CommandBarComboBox</code> .
Id	Long	Read-only. Returns the ID for a built-in <code>CommandBarComboBox</code> .
Index	Long	Read-only. Returns a <code>Long</code> representing the index number for the <code>CommandBarComboBox</code> object in the <code>CommandBars</code> collection.
IsPriorityDropped	Boolean	Read-only. Returns whether the <code>CommandBarComboBox</code> is currently dropped from the menu or toolbar, based on usage statistics and layout space. (Note that this is not the same as the control's visibility, as set by the <code>Visible</code> property.) A <code>CommandBarComboBox</code> with <code>Visible</code> set to <code>True</code> will not be immediately visible on a <code>Personalized</code> menu or toolbar if <code>IsPriorityDropped</code> is <code>True</code> .
Left	Long	Read-only. Returns the horizontal position of the <code>CommandBarComboBox</code> (in pixels) relative to the left edge of the screen. Returns the distance from the left side of the docking area.
List	String	Set/Get a specified item in the <code>CommandBarComboBox</code> . Read-only for built-in <code>CommandBarComboBox</code> controls. Required parameter: <code>Index</code> as <code>Long</code> .
ListCount	Long	Read-only. Returns the number of list items in a <code>CommandBarComboBox</code> .
ListHeaderCount	Long	Set/Get the number of list items in a <code>CommandBarComboBox</code> that appear above the separator line. Read-only for built-in <code>ComboBox</code> controls.

Table continued on following page

CommandBarComboBox Methods

Name	Returns	Description
ListIndex	Long	Set/Get the index number of the selected item in the list portion of the CommandBarComboBox. If nothing is selected in the list, this property returns 0.
OLEUsage	MsoControlOLEUsage	Set/Get the OLE client and OLE server roles in which a CommandBarComboBox will be used when two Microsoft Office applications are merged.
OnAction	String	Set/Get the name of a Visual Basic procedure that will run when the user clicks or changes the value of a CommandBarComboBox.
Parameter	String	Set/Get a string that an application can use to execute a command.
Priority	Long	Set/Get the priority of a CommandBarComboBox.
Style	MsoComboStyle	Set/Get the way a CommandBarComboBox control is displayed. Can be either of the following MsoComboStyle constants: msoComboLabel or msoComboNormal.
Tag	String	Set/Get information about the CommandBarComboBox — for example, data to be used as an argument in procedures.
Text	String	Set/Get the text in the display or edit portion of the CommandBarComboBox control.
TooltipText	String	Set/Get the text displayed in the CommandBarComboBox's ScreenTip.
Top	Long	Read-only. Returns the distance (in pixels) from the top edge of the CommandBarComboBox to the top edge of the screen.
Type	MsoControlType	Read-only. Returns the type of CommandBarComboBox.
Visible	Boolean	Set/Get whether the CommandBarComboBox is visible.
Width	Long	Set/Get the width (in pixels) of the specified CommandBarComboBox.

CommandBarComboBox Methods

Name	Returns	Parameters	Description
AddItem		Text as String, Index as Variant	Adds a list item to the specified CommandBarComboBox. The combo box control must be a custom control and must be a drop-down list box or a combo box. This method will fail if it's applied to an edit box or a built-in ComboBox control.

Name	Returns	Parameters	Description
Clear			Removes all list items from a <code>CommandBarComboBox</code> (drop-down list box or combo box) and clears the text box (edit box or combo box). This method will fail if it's applied to a built-in command bar control.
Copy	<code>CommandBar control</code>	<code>Bar, Before</code>	Copies a <code>CommandBarComboBox</code> to an existing command bar.
Delete		<code>Temporary</code>	Deletes the specified <code>CommandBarComboBox</code> from its collection. Set <code>Temporary</code> to <code>True</code> to delete the control for the current session only—the application will display the control again in the next session.
Execute			Runs the procedure or built-in command assigned to the specified <code>CommandBarComboBox</code> . For custom controls, use the <code>OnAction</code> property to specify the procedure to be run.
Move	<code>CommandBar Control</code>	<code>Bar, Before</code>	Moves the specified <code>CommandBarComboBox</code> to an existing command bar.
RemoveItem		<code>Index As Long</code>	Removes a specified item from a <code>CommandBarComboBox</code> .
Reset			Resets a built-in <code>CommandBarComboBox</code> to its default configuration, or resets a built-in <code>CommandBarComboBox</code> to its original function and face.
SetFocus			Moves the keyboard focus to the specified <code>CommandBarComboBox</code> . If the control is disabled or isn't visible, this method will fail.

CommandBarComboBox Events

Name	Parameters	Description
Change	<code>ByVal Ctrl As CommandBar ComboBox</code>	Triggered when the end user changes the selection in a <code>CommandBarComboBox</code> .

CommandBarControl Object and the CommandBarControls Collection Object

The `CommandBarControl` object represents a generic control on a `CommandBar`. A control usually consists of a `CommandBarButton`, a `CommandBarComboBox`, or a `CommandBarPopup`. When using one of these controls, you can work with them directly using their own object reference. Doing so will yield all of the properties and methods specific to that control.

Use the `Control` object when you are unsure which type of `CommandBar` object you are working with, or when using controls other than the three mentioned earlier. Most of the methods and properties for the `CommandBarControl` object can also be accessed via the `CommandBarButton`, `CommandBarComboBox`, and `CommandBarPopup` controls.

The `CommandBarControl` collection object holds all of the controls on a `CommandBar`. This collection's name can only be seen when declaring it as a variable type. You can access all the controls for a `CommandBar` directly, using:

```
CommandBars (Index) .Controls
```

Where `Index` can either be a number representing its position on the list of `CommandBars` or a string representing the `Name` of the `CommandBar`.

CommandBarControls Collection Properties

The `Application`, `Creator`, `Count`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Parameters	Description
Item	CommandBarControl	Index	Returns a <code>CommandBarControl</code> object from the <code>CommandBarControls</code> collection.

CommandBarControls Collection Methods

Name	Returns	Parameters	Description
Add	CommandBarControl	Type, Id, Parameter, Before, Temporary	Creates a new <code>CommandBarControl</code> object and adds it to the collection of controls on the specified command bar.

CommandBarControl Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
BeginGroup	Boolean	Set/Get whether the specified <code>CommandBarControl</code> appears at the beginning of a group of controls on the command bar.
BuiltIn	Boolean	Read-only. Returns <code>True</code> if the specified command bar or command bar control is a built-in command bar or control of the container application. Returns <code>False</code> if it's a custom command bar or control, or if it's a built-in control whose <code>OnAction</code> property has been set.
Caption	String	Set/Get the caption text of the <code>CommandBarControl</code> .
DescriptionText	String	Set/Get the description for a <code>CommandBarControl</code> . The description is not displayed to the user, but it can be useful for documenting the behavior of the control for other developers.
Enabled	Boolean	Set/Get whether the <code>CommandBarControl</code> object is enabled.
Height	Long	Set/Get the height of the <code>CommandBarControl</code> .
HelpContextId	Long	Set/Get the Help context Id number for the Help topic attached to the <code>CommandBarControl</code> .
HelpFile	String	Set/Get the file name for the Help topic for the <code>CommandBarControl</code> .
Id	Long	Read-only. Returns the Id for a built-in <code>CommandBarControl</code> .
Index	Long	Read-only. Returns a <code>Long</code> representing the index number for the <code>CommandBarControl</code> object in the <code>CommandBarControls</code> collection.
IsPriorityDropped	Boolean	Read-only. Returns whether the <code>CommandBar</code> control is currently dropped from the menu or toolbar based on usage statistics and layout space. (Note that this is not the same as the control's visibility, as set by the <code>Visible</code> property.) A <code>CommandBarControl</code> with <code>Visible</code> set to <code>True</code> will not be immediately visible on a Personalized menu or toolbar if <code>IsPriorityDropped</code> is <code>True</code> .
Left	Long	Read-only. Returns the horizontal position of the <code>CommandBarControl</code> (in pixels) relative to the left edge of the screen. Returns the distance from the left side of the docking area.
OLEUsage	<code>MsoControlOLEUsage</code>	Set/Get the OLE client and OLE server roles in which a <code>CommandBarControl</code> will be used when two Microsoft Office applications are merged.

Table continued on following page

CommandBarControl Methods

Name	Returns	Description
OnAction	String	Set/Get the name of a Visual Basic procedure that will run when the user clicks or changes the value of a CommandBarControl.
Parameter	String	Set/Get a string that an application can use to execute a command.
Priority	Long	Set/Get the priority of a CommandBarControl.
Tag	String	Set/Get information about the CommandBarControl — for example, data to be used as an argument in procedures.
Tooltip Text	String	Set/Get the text displayed in the CommandBarControl object's ScreenTip.
Top	Long	Read-only. Returns the distance (in pixels) from the top edge of the CommandBarControl to the top edge of the screen.
Type	MsoControl Type	Read-only. Returns the type of CommandBarControl.
Visible	Boolean	Set/Get whether the CommandBarControl is visible.
Width	Long	Set/Get the width (in pixels) of the specified CommandBarControl.

CommandBarControl Methods

Name	Returns	Parameters	Description
Copy	CommandBar Control	Bar, Before	Copies a CommandBarControl to an existing command bar.
Delete		Temporary	Deletes the specified CommandBarControl from its collection. Set <code>Temporary</code> to <code>True</code> to delete the control for the current session only — the application will display the control again in the next session.
Execute			Runs the procedure or built-in command assigned to the specified CommandBarControl. For custom controls, use the <code>OnAction</code> property to specify the procedure to be run.
Move	CommandBar Control	Bar, Before	Moves the specified CommandBarControl to an existing command bar.

Name	Returns	Parameters	Description
Reset			Resets a built-in <code>CommandBarControl</code> to its default configuration, or resets a built-in <code>CommandBarControl</code> to its original function and face.
SetFocus			Moves the keyboard focus to the specified <code>CommandBarControl</code> . If the control is disabled or isn't visible, this method will fail.

CommandBarPopup Object

This object represents a menu or submenu on a `CommandBar`, which can contain other `CommandBar` controls within them.

Because `CommandBarPopup` controls can have other controls added to them, they are in effect a separate `CommandBar`. For example, assuming the first control on a custom `CommandBar` named `Wrox` is a `CommandBarPopup` control, the following code can be used to reference and treat the control as if it were just another `CommandBar`:

```
Dim oBar as CommandBar
Set oBar = CommandBars("Wrox").Controls(1).CommandBar
```

To reference the same control as a `CommandBarPopup`:

```
Dim ctl As CommandBarPopup
Set ctl = CommandBars("Wrox").Controls(1)
```

CommandBarPopup Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
BeginGroup	Boolean	Set/Get whether the specified <code>CommandBarPopup</code> appears at the beginning of a group of controls on the command bar.
BuiltIn	Boolean	Read-only. Returns <code>True</code> if the specified command bar or command bar control is a built-in command bar or control of the container application. Returns <code>False</code> if it's a custom command bar or control, or if it's a built-in control whose <code>OnAction</code> property has been set.
Caption	String	Set/Get the caption text of the <code>CommandBarPopup</code> .
CommandBar	CommandBar	Read-only. Returns a <code>CommandBar</code> object that represents the menu displayed by the specified popup control.

Table continued on following page

CommandBarPopup Properties

Name	Returns	Description
Controls	CommandBarControls	Read-only. Returns a <code>CommandBarControls</code> object that represents all the controls on a command bar popup control.
Description Text	String	Set/Get the description for a <code>CommandBarPopup</code> . The description is not displayed to the user, but it can be useful for documenting the behavior of the control for other developers.
Enabled	Boolean	Set/Get whether the <code>CommandBarPopup</code> object is enabled.
Height	Long	Set/Get the height of the <code>CommandBarPopup</code> .
HelpContext Id	Long	Set/Get the Help context Id number for the Help topic attached to the <code>CommandBarPopup</code> .
HelpFile	String	Set/Get the file name for the Help topic for the <code>CommandBarPopup</code> .
Id	Long	Read-only. Returns the Id for a built-in <code>CommandBarPopup</code> .
Index	Long	Read-only. Returns a <code>Long</code> representing the index number for the <code>CommandBarPopup</code> object in the <code>CommandBars</code> collection.
IsPriority Dropped	Boolean	Read-only. Returns whether the <code>CommandBarPopup</code> is currently dropped from the menu or toolbar, based on usage statistics and layout space. (Note that this is not the same as the control's visibility, as set by the <code>Visible</code> property.) A <code>CommandBarPopup</code> with <code>Visible</code> set to <code>True</code> will not be immediately visible on a Personalized menu or toolbar if <code>IsPriorityDropped</code> is <code>True</code> .
Left	Long	Read-only. Returns the horizontal position of the <code>CommandBarPopup</code> (in pixels) relative to the left edge of the screen. Returns the distance from the left side of the docking area.
OLEMenu Group	MsoOLEMenuGroup	Set/Get the menu group that the specified <code>CommandBarPopup</code> belongs to when the menu groups of the OLE server are merged with the menu groups of an OLE client. Read-only for built-in controls.
OLEUsage	MsoControlOLEUsage	Set/Get the OLE client and OLE server roles in which a <code>CommandBarPopup</code> will be used when two Microsoft Office applications are merged.
OnAction	String	Set/Get the name of a Visual Basic procedure that will run when the user clicks or changes the value of a <code>CommandBarPopup</code> .
Parameter	String	Set /Get a string that an application can use to execute a command.

Name	Returns	Description
Priority	Long	Set/Get the priority of a <code>CommandBarPopup</code> .
Tag	String	Set/Get information about the <code>CommandBarPopup</code> — for example, data to be used as an argument in procedures.
TooltipText	String	Set/Get the text displayed in the <code>CommandBarPopup</code> object's <code>ScreenTip</code> .
Top	Long	Read-only. Returns the distance (in pixels) from the top edge of the <code>CommandBarPopup</code> to the top edge of the screen.
Type	<code>MsoControlType</code>	Read-only. Returns the type of <code>CommandBarPopup</code> .
Visible	Boolean	Set/Get whether the <code>CommandBarPopup</code> is visible.
Width	Long	Set/Get the width (in pixels) of the specified <code>CommandBarPopup</code> .

CommandBarPopup Methods

Name	Returns	Parameters	Description
Copy	<code>CommandBarControl</code>	<code>Bar, Before</code>	Copies a <code>CommandBarPopup</code> to an existing command bar.
Delete		<code>Temporary</code>	Deletes the specified <code>CommandBarPopup</code> from its collection. Set <code>Temporary</code> to <code>True</code> to delete the control for the current session only — the application will display the control again in the next session.
Execute			Runs the procedure or built-in command assigned to the specified <code>CommandBarPopup</code> . For custom controls, use the <code>OnAction</code> property to specify the procedure to be run.
Move	<code>CommandBarControl</code>	<code>Bar, Before</code>	Moves the specified <code>CommandBarPopup</code> to an existing command bar.
Reset			Resets a built-in <code>CommandBarPopup</code> to its default configuration, or resets a built-in <code>CommandBarPopup</code> to its original function and face.
SetFocus			Moves the keyboard focus to the specified <code>CommandBarPopup</code> . If the control is disabled or isn't visible, this method will fail.

CustomTaskPane Object

CustomTaskPane Object

The `CustomTaskPane` object acts as the container for a single custom-created task pane. This container can hold any task pane created in a language that supports COM and allows the creation of DLL files. Note that Microsoft VBA does not support the creation of custom task panes.

CustomTaskPane Properties

The `Application` property is defined at the beginning of this appendix.

Name	Returns	Description
Content Control		Read-only. Returns the ActiveX control instance displayed in the frame of the <code>CustomTaskPane</code> object.
Dock Position	<code>MsoCTPDockPosition</code>	Set/Get the position of the <code>CustomTaskPane</code> object; using one of the <code>MsoCTPDockPosition</code> constants.
DockPosition Restrict	<code>MsoCTPDockPositionRestrict</code>	Set/Get the orientation restriction of a given <code>CustomTaskPane</code> object using one of the <code>MsoCTPDockPositionRestrict</code> constants.
Height	<code>Long</code>	Set/Get the height of the <code>CustomTaskPane</code> object.
Title	<code>String</code>	Read-only. Returns the title of the <code>CustomTaskPane</code> object.
Visible	<code>Boolean</code>	Set/Get whether the <code>CustomTaskPane</code> object is displayed.
Width	<code>Long</code>	Set/Get the width of the <code>CustomTaskPane</code> object.
Window	<code>Window</code>	Read-only. Returns the parent window of the <code>CustomTaskPane</code> object.

CustomTaskPane Methods

Name	Returns	Parameters	Description
Delete			Deletes the active <code>CustomTaskPane</code> object.

CustomTaskPane Events

Name	Parameters	Description
<code>DockPositionStateChange</code>	<code>ByVal CustomTaskPaneInst</code>	Triggered when the end user changes the docking position of the active <code>CustomTaskPane</code> object.
<code>VisibleStateChange</code>	<code>ByVal CustomTaskPaneInst</code>	Triggered when the end user changes the visibility of the active <code>CustomTaskPane</code> object.

CustomXMLNode Object and the CustomXMLNodes Collection Object

The `CustomXMLNode` object is a new object in Office 2007, and is designed to provide functionality similar to that found in the `IXMLDOMNode` interface when working with custom XML parts. This object and its properties and methods allow users to find and extract a single node in an XML document without relying on the DOM interface found in MSXML. Whereas the `CustomXMLNode` object represents a single node in an XML document, the `CustomXMLNodes` collection object represents a collection of `CustomXMLNode` objects.

CustomXMLNodes Collection Properties

In addition to the common properties found in most collections, the `CustomXMLNodes` collection object also contains an `Item` property that returns the index number of a single `CustomXMLNode` object in the collection.

CustomXMLNode Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Attributes</code>	<code>CustomXMLNodes</code>	Read-only. Returns a <code>CustomXMLNodes</code> collection representing the attributes of the current element in the current node.
<code>BaseName</code>	<code>String</code>	Read-only. Returns the name of the specified node without the namespace prefix.
<code>ChildNodes</code>	<code>CustomXMLNodes</code>	Read-only. Returns a collection of nodes containing all of the child elements of specified node.
<code>FirstChild</code>	<code>CustomXMLNode</code>	Read-only. Returns a <code>CustomXMLNode</code> object corresponding to the first child element of the current node.
<code>LastChild</code>	<code>CustomXMLNode</code>	Read-only. Returns a <code>CustomXMLNode</code> object corresponding to the last child element of the current node.
<code>NamespaceURI</code>	<code>String</code>	Read-only. Returns the unique address identifier for the namespace of the given node.
<code>NextSibling</code>	<code>CustomXMLNode</code>	Read-only. Returns a <code>CustomXMLNode</code> object corresponding to next sibling node (element, comment, or processing instruction) of the current node.
<code>NodeType</code>	<code>MsoCustomXMLNodeType</code>	Read-only. Returns an <code>MsoCustomXMLNodeType</code> constant for a given node, specifying the node's type.
<code>NodeValue</code>	<code>String</code>	Set/Get the value of a given node.
<code>OwnerDocument</code>	<code>Object</code>	Read-only. Returns the object representing the Microsoft Office Excel workbook, Microsoft Office PowerPoint presentation, or the Microsoft Office Word document associated with a given node.

Table continued on following page

CustomXMLNode Methods

Name	Returns	Description
OwnerPart	CustomXMLPart	Read-only. Returns the CustomXMLPart object associated with a given node.
ParentNode	CustomXMLNode	Read-only. Returns the parent element node of a given node current node. If the current node is at the root level, the property returns Nothing.
PreviousSibling	CustomXMLNode	Read-only. Returns a CustomXMLNode object corresponding to the previous sibling node (element, comment, or processing instruction) of the current node.
Text	String	Set/Get the text for a given node.
XML	String	Read-only. Returns the XML representation of the specified node and its children.
XPath	String	Read-only. Returns a String with the canonicalized XPath for the specified node.

CustomXMLNode Methods

Name	Returns	Parameters	Description
AppendChildNodes		Name, NamespaceURI, NodeType, NodeValue	Appends a single node as the last child under the context element node.
AppendChildSubtree		XML	Adds a subtree as the last child under the context element node.
Delete			Deletes a specified node and all of its children.
HasChildNodes	Boolean		Returns True if the specified node has child element nodes.
InsertNodesBefore		Name, NamespaceURI, NodeType, NodeValue, NextSibling	Inserts a new node just before the context node.
InsertSubtreeBefore		XML, NextSibling	Inserts the specified subtree into the location just before the context node.
RemoveChild		Child	Removes the specified child node from the tree.
ReplaceChildNode		OldNode, Name, NamespaceURI, NodeType, NodeValue	Removes the specified child node and replaces it with a different node in the same location.

CustomXMLNode, CustomXMLNodes, and CustomXMLPart Example

Name	Returns	Parameters	Description
ReplaceChild Subtree		XML, OldNode	Removes the specified node and replaces it with a different subtree in the same location.
SelectNodes		Xpath	Selects a collection of nodes matching an XPath expression.
SelectSingle Node		Xpath	Selects a single node from a collection matching an XPath expression.

CustomXMLNode, CustomXMLNodes, and CustomXMLPart Example

The following routine adds a custom XML part to the active workbook and traverses the XML part using the new CustomXMLNodes collection:

```
Sub Add_Traverse_CustomXMLPart()  
Dim oCustomPart As CustomXMLPart  
Dim oCustomNode As CustomXMLNode  
Dim oCustomNodes As CustomXMLNodes  
  
'Add a Custom XML Part from a file and then load  
Set oCustomPart = ActiveWorkbook.CustomXMLParts.Add  
oCustomPart.Load "C:\EmployeeSales.xml"  
  
'Return all nodes for the employee who has invoice amount over 3000  
Set oCustomNodes = oCustomPart.SelectNodes("//Employee[InvoiceAmount>3000]")  
For Each oCustomNode In oCustomNodes  
    Debug.Print oCustomNode.Text  
Next  
  
'Delete the Custom XML Part  
oCustomPart.Delete  
End Sub
```

To run this code, enter the following XML into Notepad and save as C:\EmployeeSales.xml:

```
<?xml version="1.0"?>  
<EmployeeSales>  
  <Employee>  
    <Empid>2312</Empid>  
    <FirstName>Mike</FirstName>  
    <LastName>Alexander</LastName>  
    <InvoiceNumber>100</InvoiceNumber>  
    <InvoiceAmount>2300</InvoiceAmount>  
  </Employee>  
  
  <Employee>  
    <Empid>24601</Empid>  
    <FirstName>Stephen</FirstName>  
    <LastName>Bullen</LastName>
```

CustomXMLPart Object and the CustomXMLParts Collection Object

```
<InvoiceNumber>200</InvoiceNumber>
<InvoiceAmount>3211</InvoiceAmount>
</Employee>
</EmployeeSales>
```

CustomXMLPart Object and the CustomXMLParts Collection Object

The `CustomXMLPart` object allows you to programmatically work with any XML document you integrate into your Excel workbook as a custom XML part. The `CustomXMLParts` collection object represents a collection of `CustomXMLPart` objects.

CustomXMLParts Collection Properties

In addition to the common properties found in most collections, the `CustomXMLNodes` object also contains an `Item` property that returns the index number of a single `CustomXMLPart` object in the collection.

CustomXMLParts Collection Methods

Name	Returns	Parameters	Description
Add	CustomXMLPart	XMLSchemaCollection	Adds a new custom XML part to a file.
SelectByID	CustomXMLPart	ID	Selects a custom XML part by matching a GUID.
SelectByNamespace	CustomXMLParts	NamespaceURI	Selects a custom XML part by matching a Namespace URI.

CustomXMLParts Collection Events

Name	Parameters	Description
PartAfterAdd	NewPart	Triggered after a new <code>CustomXMLPart</code> object is added to the current file.
PartAfterLoad	Part	Triggered after a <code>CustomXMLPart</code> object is loaded.
PartBeforeDelete	OldPart	Triggered before a <code>CustomXMLPart</code> object is deleted.

CustomXMLPart Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
BuiltIn	Boolean	Read-only. Specifies whether the CustomXMLPart is built-in.
Document Element	CustomXML Node	Read-only. Returns the root element of a bound region of data in a document.
Errors	CustomXML Validation Errors	Read-only. Provides access to any XML validation errors via the CustomXMLValidationErrors object.
Id	String	Read-only. Returns the GUID assigned to the specified CustomXMLPart object.
Namespace Manager	CustomXML Prefix Mappings	Read-only. Returns the set of namespace prefix mappings used against the specified CustomXMLPart object.
NamespaceURI	String	Read-only. Returns the unique address identifier for the namespace of the CustomXMLPart object.
Schema Collection	CustomXML Schema Collection	Set/Get the CustomXMLSchemaCollection object representing the set of schemas attached to a bound region of data in a document.
XML	String	Read-only. Returns the XML representation of the specified CustomXMLPart object.

CustomXMLPart Methods

Name	Returns	Parameters	Description
AddNode		Parent, Name, NamespaceURI, NextSibling, NodeType, NodeValue	Adds a node to the XML tree within a CustomXMLPart object.
Delete			Deletes the specified CustomXMLPart object from the IXMLDataStore interface.
Load		FilePath	Populates a given CustomXMLPart object using an existing XML file.
LoadXML		XML	Populates a given CustomXMLPart object using an XML string.
SelectNodes		XPath	Selects a collection of nodes from a CustomXMLPart object.
SelectSingle Node		XPath	Selects a single node from a CustomXMLPart object.

CustomXMLPart Events

Name	Returns	Parameters	Description
NodeAfterDelete		OldNode, OldParentNode, OldNextSibling, InUndoRedo	Triggered after a node within the specified CustomXMLPart object is deleted.
NodeAfterInsert		NewNode, InUndoRedo	Triggered after a node within the specified CustomXMLPart object is inserted.
NodeAfterReplace		OldNode, NewNode, InUndoRedo	Triggered after a node within the specified CustomXMLPart object is replaced.

CustomXMLPrefixMapping Object and the CustomXMLPrefixMappings Collection Object

The `CustomXMLPrefixMapping` object allows you to programmatically work with both the namespaces and the namespace prefixes within your custom XML parts. The `CustomXMLPrefixMappings` collection object represents a collection of `CustomXMLPrefixMapping` objects.

CustomXMLPrefixMappings Collection Properties

In addition to the common properties found in most collections, the `CustomXMLPrefixMappings` object also contains an `Item` property that returns the index number of a single `CustomXMLPrefixMapping` object in the collection.

CustomXMLPrefixMappings Collection Methods

Name	Returns	Parameters	Description
AddNamespace		Prefix, NamespaceURI	Adds a custom namespace and prefix mapping.
LookupNamespace	String	Prefix	Returns the namespace corresponding to a specified prefix.
LookupPrefix	String	NamespaceURI	Returns the prefix corresponding to a specified namespace.

CustomXMLPrefixMapping Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

CustomXMLSchema Object and the CustomXMLSchemaCollection Object

Name	Returns	Description
NamespaceURI	String	Read-only. Returns the unique address identifier for the namespace of the CustomXMLPrefixMapping object.
Prefix	String	Read-only. Returns the prefix name for the namespace of the CustomXMLPrefixMapping object.

CustomXMLSchema Object and the CustomXMLSchemaCollection Object

The CustomXMLSchema object allows you to programmatically work with the XML schemas for your custom XML parts. The CustomXMLSchemaCollection object represents a collection of CustomXMLSchema objects.

CustomXMLSchemaCollection Properties

In addition to the common properties found in most collections, the CustomXMLPrefixMappings object also contains an Item property that returns the index number of a single CustomXMLPrefixMapping object in the collection, and a NamespaceURI property that returns the unique address identifier for the namespace of the CustomXMLSchemaCollection object.

CustomXMLSchemaCollection Methods

Name	Returns	Parameters	Description
Add	CustomXML Schema	NamespaceURI, Alias, FileName, InstallForAllUsers	Adds one or more schemas to a schema collection.
AddCollectionCollection		Schema	Adds an existing schema collection to the current schema collection.
Validate	Boolean		Allows you to check whether the schemas in a schema collection conform to the syntactic rules of XML and the rules for a specified vocabulary.

CustomXMLSchema Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

Name	Returns	Description
Location	String	Read-only. Returns the location of a specified schema file.
NamespaceURI	String	Read-only. Returns the unique address identifier for the namespace of the CustomXMLPrefixMapping object.

CustomXMLSchema Methods

Name	Returns	Parameters	Description
Delete			Deletes the specified schema from the CustomXMLSchema collection.
Reload			Reloads a schema from a specified schema file.

CustomXMLValidationError Object and the CustomXMLValidationErrors CollectionObject

The `CustomXMLValidationError` object represents a single error triggered when validating an operation against the schema for your custom XML part. This object is typically used to display and manage any errors that may occur when programming against your custom XML parts. The `CustomXMLValidationErrors` collection object represents a collection of `CustomXMLValidationError` objects.

CustomXMLValidationErrors Collection Properties

In addition to the common properties found in most collections, the `CustomXMLValidationErrors` object also contains an `Item` property that returns the index number of a single `CustomXMLValidationError` object in the collection.

CustomXMLValidationErrors Collection Methods

Name	Returns	Parameters	Description
Add		Node, Error Name, Error Text, Cleared OnUpdate	Adds a <code>CustomXMLValidationError</code> object containing an XML validation error to the <code>CustomXMLValidationErrors</code> collection.

CustomXMLValidationError Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
ErrorCode	Long	Read-only. Returns the validation error number.
Name	String	Read-only. Returns the name of the error.
Node	CustomXML Node	Read-only. Returns the node in which the error occurred. If no node is bound to the error, this property returns <code>Nothing</code> .
Text	String	Read-only. Returns the plain language text associated with the <code>CustomXMLValidationError</code> object.

Name	Returns	Description
Type	MsoCustomXMLValidation ErrorType	Read-only. Returns an MsoCustomXMLValidation ErrorType constant indicating the type of error generated.

CustomXMLValidationError Methods

Name	Description
Delete	Deletes the specified CustomXMLValidationError object representing the data validation error.

DocumentInspector Object and the DocumentInspectors CollectionObject

The `DocumentInspector` object allows you to programmatically inspect and fix an Excel workbook just as you would via the Document Inspector dialog box. You can programmatically call upon any one of the inspection modules found in the Document Inspector dialog box by specifying that module's index value in the `DocumentInspectors` collection, as demonstrated in the example code for this section.

Here are the built-in inspection modules for Excel 2007 and their corresponding index values:

- Inspect for custom XML data stored with the document: Use index value 1
- Inspect the workbook for information in headers and footers: Use index value 2
- Inspect the workbook for hidden rows and columns: Use index value 3
- Inspect the workbook for hidden worksheets: Use index value 4
- Inspect the workbook for invisible objects: Use index value 5

You will note that the first two modules in the Document Inspector dialog box (Comments and Annotations, Document Properties and Personal Information) are not shown here. That is because they are not available via `DocumentInspectors` collection; instead, their functionality is available through the `RemoveDocumentInformation` method of the `Workbook` object.

DocumentInspectors Collection Properties

In addition to the common properties found in most collections, the `DocumentInspectors` object also contains an `Item` property that returns the index number of a single `DocumentInspector` object in the collection.

DocumentInspector Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

DocumentInspector Methods

Name	Returns	Description
Description	Long	Read-only. Returns the description of the specified DocumentInspector object.
Name	String	Read-only. Returns the module name of the specified DocumentInspector object.

DocumentInspector Methods

Name	Parameters	Description
Fix	Status, Results	Performs an action on specific information or document properties as defined by the specified DocumentInspector object.
Inspect	Status, Results	Inspects a document for the information or document properties defined by a given DocumentInspector object.

DocumentInspector Object Example

```
Sub InspectMyDocument()  
Dim oInspector As DocumentInspector  
Dim cnstStatus As MsoDocInspectorStatus  
Dim strResult As String  
  
'Set the inspector to inspect for hidden worksheets  
Set oInspector = ActiveWorkbook.DocumentInspectors(4)  
oInspector.Inspect cnstStatus, strResult  
  
'If inspection produced no results, notify user and exit procedure  
If cnstStatus <> 1 Then  
MsgBox "Inspection produced no results"  
Exit Sub  
End If  
  
'Give user the option to delete the found hidden worksheets  
'If User chooses to delete worksheets, then remove worksheet  
'If User chooses not to delete worksheets, then exit procedure  
  
Select Case MsgBox(strResult & vbCrLf & "Remove Hidden Worksheets?", vbYesNo)  
  
Case Is = vbYes  
oInspector.Fix cnstStatus, strResult  
MsgBox strResult  
Exit Sub  
  
Case Is = vbNo  
MsgBox "Hidden worksheets will not be removed"  
Exit Sub  
  
End Select  
End Sub
```

DocumentLibraryVersion Object and the DocumentLibraryVersions Collection Object

The `DocumentLibraryVersion` object represents a single saved version of a shared document that has versioning enabled and that is stored in a document library on the server. Each `DocumentLibraryVersion` object is a member of the active document's `DocumentLibraryVersions` collection. The `DocumentLibraryVersions` collection object represents a collection of `DocumentLibraryVersion` objects.

DocumentLibraryVersions Collection Properties

In addition to the common properties found in most collections, the `DocumentInspectors` object also contains an `Item` property that returns the index number of a single `DocumentInspector` object in the collection, and the `IsVersioningEnabled` property that indicates whether the document library in which the active document is saved on the server is configured to create a backup copy, or version, each time the file is edited on the web site.

DocumentLibraryVersion Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Comments</code>	<code>String</code>	Read-only. Returns any optional comments associated with the specified version of the shared document.
<code>Index</code>	<code>Long</code>	Returns a <code>Long</code> representing the index number for an object in the collection. Read-only.
<code>Modified</code>	<code>Variant</code>	Read-only. Returns the date and time the document was last saved to the server.
<code>ModifiedBy</code>	<code>String</code>	Returns the name of the user who last saved the specified version of the shared document to the server. Read-only <code>String</code> .

DocumentLibraryVersion Methods

Name	Returns	Description
<code>Delete</code>		Removes a document library version from the <code>DocumentLibraryVersions</code> collection.
<code>Open</code>	<code>Object</code>	Opens the specified version of the shared document from the <code>DocumentLibraryVersions</code> collection in Read-only mode.
<code>Restore</code>	<code>Object</code>	Restores a previous saved version of a shared document from the <code>DocumentLibraryVersions</code> collection.

DocumentProperty Object and the DocumentProperties Collection Object

The `DocumentProperty` object represents a single property in the `DocumentProperties` collection. The property can be either a built-in or custom property. Use `BuiltinDocumentProperties` or `CustomDocumentProperties` to reference a single `DocumentProperty`.

The `DocumentProperties` collection object represents all of the `Document Properties` listed in the host application's `Summary` and `Custom` tabs of the `Properties` command (File menu) for a document. The document would be the `Workbook` object in Excel and the `Document` object in Word.

The `DocumentProperties` collection consists of two distinct types: Built-in properties and Custom properties. Built-in properties are native to the host application and are found on the `Summary` tab of the `Properties` command. Custom properties are those created by the user for a particular document and are found on the `Custom` tab of the `Properties` command.

It's important to note that when accessing `DocumentProperties` for a document, you must use either the `BuiltinDocumentProperties` property for properties native to the host application, or the `CustomDocumentProperties` property for properties created by the user. Strangely enough, `BuiltinDocumentProperties` and `CustomDocumentProperties` are not found in the Office object model, but are part of the host application's model. In other words, you will not find these two properties within the `DocumentProperties` or `DocumentProperty` objects of Microsoft Office 2007.

To access the built-in author document property, you can use the index value of the built-in document property. For example, you can get the document author by using:

```
MsgBox ActiveWorkbook.BuiltinDocumentProperties(3).Value
```

You can also assign values to document properties as such:

```
ActiveWorkbook.BuiltinDocumentProperties(3).Value = "Mike Alexander"
```

The following is a list of the available built-in document properties and their corresponding index values:

- Title: 1
- Subject: 2
- Author: 3
- Keywords: 4
- Comments: 5
- Template: 6
- Last author: 7
- Revision number: 8
- Application name: 9
- Last print date: 10

- Creation date: 11
- Last save time: 12
- Total editing time: 13
- Number of pages: 14
- Number of words: 15
- Number of characters: 16
- Security: 17
- Category: 18
- Format: 19
- Manager: 20
- Company: 21
- Number of bytes: 22
- Number of lines: 23
- Number of paragraphs: 24
- Number of slides: 25
- Number of notes: 26
- Number of hidden slides: 27
- Number of multimedia clips: 28
- Hyperlink base: 29
- Number of characters (with spaces): 30
- Content type: 31
- Content status: 32
- Language: 33
- Document version: 34

Keep in mind that several of the built-in properties are specific to certain host applications. For example, the `Number of Paragraphs` property is native to Microsoft Word, and any attempt to reference it from another application will result in a run-time error.

DocumentProperties Collection Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Parameters	Description
Item	Document Property	Index as Variant	Index can also be a string representing the <code>DocumentProperty</code> 's name.

DocumentProperties Collection Methods

DocumentProperties Collection Methods

Name	Returns	Parameters	Description
Add	Document Property	Name As String, LinkToContent As Boolean, Type, Value, LinkSource	Creates a new custom document property. You can only add a new document property to the custom DocumentProperties collection.

DocumentProperty Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

Name	Returns	Description
LinkSource	String	Set/Get the source of a linked custom document property.
LinkToContent	Boolean	Returns whether the custom document property is linked to the content of the container document.
Name	String	Set/Get the name of the Document property.
Type	MsoDoc Properties	Set/Get the document property type. Read-only for built-in document properties; Read/Write for custom document properties.
Value	Variant	Set/Get the value of a document property. If the container application doesn't define a value for one of the built-in document properties, reading the Value property for that document property causes an error.

DocumentProperty Methods

Name	Parameters	Description
Delete		Removes a custom document property.

EncryptionProvider Object

The EncryptionProvider object allows access to the methods for setting up permissions, applying encryption and decryption, and controlling user authentication. Microsoft Office provided a certain amount of storage for Add-In-specific information to store whatever information you need to encrypt, decrypt, apply rights, and display permission setup or authentication user interfaces.

EncryptionProvider Methods

Name	Returns	Parameters	Description
Authenticate	Long	ParentWindow, EncryptionData, Permission Mask	Determines whether the user has the proper permissions to open the encrypted document.
CloneSession	Long	SessionHandle	Creates a second, working copy of the EncryptionProvider object's encryption session for a file that is about to be saved.
Decrypt Stream	Encryption Provider	SessionHandle, StreamName, Encrypted Stream, Unencrypted Stream	Decrypts and returns a stream of encrypted data for a document. This method is the inverse of Encrypt Stream method and converts encrypted data back into pure (unencrypted) data.
Encrypt Stream	Encryption Provider	SessionHandle, StreamName, Unencrypted Stream, Encrypted Stream	Encrypts and returns a stream of data for a document.
EndSession	Encryption Provider	SessionHandle	Ends the current encryption session.
GetProvider Detail	VARIANT	Encryption Provider Detail	Queries the EncryptionProvider object for information such as download URL, implementation algorithm, and cipher mode.
NewSession	Long	ParentWindow	Creates a new encryption session.
Save	Long	SessionHandle, EncryptionData	Saves an encrypted document.
ShowSettings		SessionHandle, ParentWindow, ReadOnly, Remove	Used to display any dialogs and encryption settings required for access to the current document.

FileDialog Object

This object is now a more structured and more flexible alternative to both the `GetSaveAsFilename` and `GetOpenFilename` methods. It includes the ability to customize the action button (for example, the Save button in Save As dialog) and choose from a list of different dialog types (above and beyond Open and

FileDialog Properties

Save As), adds more flexibility when using custom file types or filters (for example, "*.bil"), and allows you to set a default view that the user will see when the dialog appears (for example, Detail or Large Icon views).

Note that some of the properties and methods for this object depend on the `MsoFileDialogType` chosen in the `FileDialogType` property. For example, the following will encounter an error when attempting to use the `Add` method of the `Filters` property with the `msoFileDialogSaveAs` dialog type:

```
Application.FileDialog(msoFileDialogSaveAs).Filters.Add _  
    "Billing Files", "*.bil", 1
```

FileDialog Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>AllowMultiSelect</code>	Boolean	Set/Get whether the user is allowed to select multiple files from a file dialog box.
<code>ButtonName</code>	String	Set/Get the text that is displayed on the action button of a file dialog box. By default, this property is set to the standard text for the type of file dialog box.
<code>DialogType</code>	<code>MsoFileDialogType</code>	Read-only. Returns an <code>MsoFileDialogType</code> constant representing the type of file dialog box that the <code>FileDialog</code> object is set to display.
<code>FilterIndex</code>	Long	Set/Get the default file filter of a file dialog box. The default filter determines which types of files are displayed when the file dialog box is first opened.
<code>Filters</code>	<code>FileDialogFilters</code>	Returns a <code>FileDialogFilters</code> collection.
<code>InitialFileName</code>	String	Set/Get the path and/or file name that is initially displayed in a file dialog box.
<code>InitialView</code>	<code>MsoFileDialogView</code>	Set/Get an <code>MsoFileDialogView</code> constant representing the initial presentation of files and folders in a file dialog box.
<code>Item</code>	String	Read-only. Returns the text associated with the <code>FileDialog</code> object.
<code>SelectedItem</code>	<code>FileDialogSelectedItem</code>	Returns a <code>FileDialogSelectedItem</code> collection. This collection contains a list of the paths of the files that a user selected from a file dialog box displayed using the <code>Show</code> method of the <code>FileDialog</code> object.
<code>Title</code>	String	Set/Get the title of a file dialog box displayed using the <code>FileDialog</code> object.

FileDialog Methods

Name	Returns	Description
Execute		FileDialog objects of type <code>msoFileDialogOpen</code> or <code>msoFileDialogSaveAs</code> , carries out a user's action right after the <code>Show</code> method is invoked.
Show	Long	Displays a file dialog box. Returns a <code>Long</code> indicating whether the user pressed the action button (-1) or the cancel button (0). When the <code>Show</code> method is called, no more code will execute until the user dismisses the file dialog box. With <code>Open</code> and <code>Save As</code> dialog boxes, use the <code>Execute</code> method right after the <code>Show</code> method to carry out the user's action.

FileDialogFilter Object and the FileDialogFilters Collection Object

The `FileDialogFilter` is a single filter in the `FileDialogFilters` collection. To reference an individual filter, use:

```
Application.FileDialog(msoFileDialogOpen).Filters(lIndex)
```

The `FileDialogFilters` collection object represents all the filters shown in the new `FileDialog` object, including custom filters created using the `Add` method of the `Filters` property for the `FileDialog` object.

Note that filters created using the `Add` method of the `Filters` property do not appear in the standard `Open` and `Save As` dialogs.

FileDialogFilters Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

FileDialogFilters Collection Methods

Name	Returns	Parameters	Description
Add	FileDialog Filter	Description As String, Extensions As String, Position	Adds a new file filter to the list of filters in the <code>Files</code> of type drop-down list box in the File dialog box. Returns a <code>FileDialogFilter</code> object that represents the newly added file filter.
Clear			Removes all the file filters in the <code>FileDialogFilters</code> collection.

Table continued on following page

FileDialogFilter Properties

Name	Returns	Parameters	Description
Delete		filter	Removes a specified file filter from the <code>FileDialogFilters</code> collection.
Item	<code>FileDialogFilter</code>	Index As Long	Returns the specified <code>FileDialogFilter</code> object from a <code>FileDialogFilters</code> collection.

FileDialogFilter Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
Description	String	Read-only. Returns the description displayed in the file dialog box of each <code>Filter</code> object as a <code>String</code> value.
Extensions	String	Read-only. Returns a <code>String</code> value containing the extensions that determine which files are displayed in a file dialog box for each <code>Filter</code> object.

FileDialogSelectedItems Collection Object

This collection returns all of the chosen items in a `FileDialog`. It consists of more than one item when the `AllowMultiSelect` property of the `FileDialog` object is set to `True`, unless the `msoFileDialogSaveAs` `FileDialog` is used (where only one item is always returned). The `FileDialogSelectedItems` collection is a collection of strings.

FileDialogSelectedItems Collection Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

FileDialogSelectedItems Collection Methods

Name	Returns	Parameters	Description
Item	String	Index as long	Returns the path of one of the files that the user selected from a file dialog box that was displayed using the <code>Show</code> method of the <code>FileDialog</code> object.

FileTypes Object

The `FileTypes` object represents a set of file types you want to search for when using the `FileSearch` object. You will note that although the `FileTypes` object is meant to be used with the `FileSearch` object, the `FileSearch` object is no longer part of the Office 2007 object model.

FileTypes Collection Properties

The `Application` and `Creator` properties are defined at the beginning of this appendix.

Name	Returns	Parameters	Description
<code>Item</code>	<code>MsoFileType</code>	<code>Index as Long</code>	Read-only. Returns a value that indicates which file type will be searched for by the <code>Execute</code> method of the <code>FileSearch</code> object.

FileTypes Collection Methods

Name	Parameters	Description
<code>Add</code>	<code>FileType As MsoFileType</code>	Adds a new file type to a file search.
<code>Remove</code>	<code>Index As Long</code>	Removes the specified file type from the <code>FileTypes</code> collection.

Font2 Object

The `GlowFormat` object exposes the various properties used to configure the font attributes for an Office object.

Font2 Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Allcaps</code>	<code>Boolean</code>	Set/Get whether font is formatted as all capital letters.
<code>Autorotate Numbers</code>	<code>Boolean</code>	Set/Get whether the numbers in a numbered list should be rotated when the text is rotated.
<code>Baseline Offset</code>	<code>Single</code>	Set/Get a single defining the horizontal offset of the selected font.
<code>Bold</code>	<code>Boolean</code>	Set/Get whether the font should be bold.
<code>Caps</code>	<code>MsoTextCaps</code>	Set/Get an <code>MsoTextCaps</code> constant specifying that the text should be capitalized.
<code>DoubleStrike Through</code>	<code>Boolean</code>	Set/Get whether the font is formatted as double strikethrough text.
<code>Embeddable</code>	<code>Boolean</code>	Read-only. Returns whether the font can be embedded in a page.

Table continued on following page

Font2 Properties

Name	Returns	Description
Embedded	Boolean	Read-only. Returns whether the font is embedded in a page.
Equalize	Boolean	Set/Get whether the text for a selection should be spaced equal distances apart.
Fill	FillFormat	Read-only. Returns the fill format for a font.
Glow	GlowFormat	Read-only. Returns the value indicating whether the font is displayed as a glow effect.
Highlight	ColorFormat	Read-only. Returns the value indicating whether the font is displayed as highlighted.
Italic	Boolean	Set/Get whether the text for a selection is italic.
Kerning	Single	Set/Get the amount of spacing between text characters.
Line	LineFormat	Read-only. Returns the format of a line.
Name	String	Set/Get the font to use for a selection.
NameAscii	String	Set/Get the font used for Latin text (characters with character codes from 0 (zero) through 127).
NameComplex Script	String	Set/Get the complex script font name used for mixed language text.
NameFarEast	String	Set/Get an East Asian font name.
NameOther	String	Set/Get the font used for characters whose character set numbers are greater than 127.
Reflection	ReflectionFormat	Read-only. Returns the type of reflection format for the selection of text.
Shadow	ShadowFormat	Read-only. Returns the type of shadow effect for the selection of text.
Size	Single	Read-only. Returns the size of the font.
Smallcaps	Boolean	Set/Get whether small caps should be used with the selection of text.
SoftEdgeFormat	MsoSoftEdgeType	Set/Get the type of soft edge effect used in a selection of text.
Spacing	Single	Set/Get the value specifying the spacing between characters in a selection of text. Read/write.
Strike	MsoTextStrike	Set/Get the strike format used for a selection of text.
StrikeThrough	Boolean	Set/Get whether strikethrough formatting should be used.

Name	Returns	Description
Subscript	Boolean	Set/Get whether subscript formatting should be used.
Underline Color	ColorFormat	Read-only. Returns the color of the underline for the selected text.
Underline Style	MsoText Underline Type	Set/Get whether underline formatting should be used.
WordArt Format	MsoPreset Text Effect	Set/Get the text effect for the selected text.

GlowFormat Object

The `GlowFormat` object exposes the properties used to configure the glow effect around Office graphics.

GlowFormat Properties

The `Application` and `Creator` properties are defined at the beginning of this appendix.

Name	Returns	Description
Color	ColorFormat	Read-only. Returns a <code>ColorFormat</code> object representing the color of text formatted as glow.
Radius	Single	Sets/Get the radius value of the glow effect.

GradientStop Object and the GradientStops Collection Object

A `GradientStop` object represents one endpoint in a series of sections that make up a color gradient. This object can be used to add and remove gradient color stops, effectively customizing the color gradient for a particular graphic, shape, or object. The `GradientStops` collection object represents all of the `GradientStop` objects that make up a color gradient.

GradientStops Collection Properties

The `Application`, `Count`, and `Creator` properties are defined at the beginning of this appendix.

Name	Returns	Description
Item	GradientStop	Read-only. Returns a <code>GradientStop</code> from a <code>GradientStops</code> collection using an index value or a name.

GradientStops Collection Methods

GradientStops Collection Methods

Name	Returns	Parameters	Description
Delete		Index	Removes a gradient stop.
Insert		RGB, Position, Transparency, Index	Adds a stop to a gradient.

GradientStop Properties

The `Application` and `Creator` properties are defined at the beginning of this appendix.

Name	Returns	Description
Color	ColorFormat	Read-only. Returns a <code>ColorFormat</code> object representing the color of the gradient stop.
Position	Single	Set/Get the position of a stop within the gradient. This is expressed as a percentage.
Transparency	Single	Set/Get the opacity of the gradient fill. This is expressed as a percentage.

IAssistance Object

The `IAssistance` object allows developers to display help topics through the Office Help Viewer. The `IAssistance` object is returned by the `Assistance` property of the `Application` object, allowing developers to call specific help files by either passing a Help ID or a keyword to the help system. For instance:

This code uses the `SearchHelp` method to open the Office Help Viewer, and displays all topics relating to calculated fields in pivot tables:

```
Application.Assistance.SearchHelp "PivotTable Calculated Field"
```

Here, you are using the `ShowHelp` method to pass a specific Help ID to display help on corrupted workbooks:

```
Application.Assistance.ShowHelp "22261"
```

IAssistance Methods

Name	Parameters	Description
ClearDefaultContext	Helpid	Clears the default help topic previously defined in the <code>SetDefaultContext</code> method.
SearchHelp	Query, Scope	Performs a search from the Office Help Viewer based on one or more keywords. Keywords can be words or phrases.

IBlogExtensibility and IBlogPictureExtensibility Objects

Name	Parameters	Description
SetDefaultContext	Helpid	Sets a default help topic that allows a specified topic to display when the user presses F1 or clicks the Help button in a dialog box.
ShowHelp	Helpid, Scope	Displays the help topic specified by its ID. The Scope will determine the application namespace that is used. The following scopes are available within the Microsoft Office applications: Access, Excel, Outlook, PowerPoint, Project, Publisher, SharePoint Designer, Visio, and Word. By default, the scope is set to the current application's namespace if a Null string ("") is passed as a scope parameter.

IBlogExtensibility and IBlogPictureExtensibility Objects

Both the `IBlogExtensibility` object and the `IBlogPicturesExtensibility` object are new in Office 2007. These objects provide the interfaces that allow users to interact with, and publish to, blog providers via Microsoft Word.

IBlogExtensibility Methods

Name	Returns	Parameters	Description
BlogProviderProperties		BlogProvider, FriendlyName, Category Support, Padding, NoCredentials	Returns information about the provider.
GetCategories		Account, ParentWindow, Document, username, Password, Categories()	Returns the list of blog categories for an account, allowing Word to populate its category drop-down lists.
GetRecentPosts		Account, ParentWindow, Document, userName, Password, PostTitles(), PostDates(), PostIDs()	Returns the list of the last 15 blogs in the Open Existing Post dialog. Note that this method does not actually return the blog post contents.

Table continued on following page

IBlogExtensibility Methods

Name	Returns	Parameters	Description
GetUserBlogs		Account, ParentWindow, Document, userName, Password, BlogNames(), BlogIDs(), BlogURLs()	Returns the list and details of user blogs associated with the specified account.
Open		Account, PostID, ParentWindow, userName, Password, xHTML, Title, DatePosted, Categories()	Opens the blog specified by the blog ID.
PublishPost		Account, ParentWindow, Document, username, Password, xHTML, Title, DateTime, Categories(), Draft, PostID, PublishMessage	Transfers the current post of the blog provider so that it can be published.
RepublishPost		Account, ParentWindow, Document, username, Password, PostID, DateTime, Categories(), Draft, PublishMessage	Transfers the current post of the blog provider so it can be republished.
SetupBlog Account		Account, ParentWindow, Document, NewAccount, ShowPictureUI	Called from the Choose Account dialog when the provider's name is chosen in the Blog Host dropdown, or when the user requests to change a provider's account in the Blog Accounts dialog box.

IBlogPictureExtensibility Methods

Name	Returns	Parameters	Description
BlogPicture Provider Properties		BlogPicture Provider, FriendlyName	Enables picture providers to offer themselves as an upload location for blog pictures.
CreatePicture Account		Account, BlogProvider, ParentWindow, Document, userName, Password	Allows a picture provider to display the user interface needed to guide the user through setting up a picture account.
Create Picture Account		Account, ParentWindow, Document, userName, Password, Image, PictureURI	Posts a picture object to its final destination in a blog.

ICTPFactory Object

When an external application is used to create an instance of a CustomTaskPane object in an Add-In and implements the CTPFactoryAvailable method, the CTPFactoryAvailable method passes an ICTPFactory object to the Add-In. From here, the ICTPFactory object is used to create the task pane by employing the CreateCTP method.

ICTPFactory Methods

Name	Returns	Parameters	Description
CreateCTP	Custom TaskPane	CTPaxID, CTPtitle, CTP ParentWindow	Creates an instance of a custom task pane.

ICustomTaskPaneConsumer Object

The ICustomTaskPaneConsumer object acts as an interface, providing access to its only method, CTPFactoryAvailable. This method creates an instance of a custom task pane by passing a CTPFactory object to an ActiveX Add-In that can then use that object to create the custom task pane.

ICustomTaskPaneConsumer Methods

Name	Parameters	Description
CTPFactory Available	CTPFactory Inst	Passes a CTPFactory object to an ActiveX Add-In that can then be used when creating a custom task pane.

IDocumentInspector Object

The `IDocumentInspector` object provides an interface that can be used to access the methods of custom Document Inspector modules. Note that the `IDocumentInspector` object is designed to be used by developers of Document Inspector modules and cannot be used with Visual Basic for Applications (VBA).

IDocumentInspector Methods

Name	Parameters	Description
Fix	Doc, Hwnd, Status, Result	Performs some action on specific information items or document properties as defined by the custom Document Inspector module.
GetInfo	Name, Desc	Returns information about a given custom Document Inspector module.
Inspect	Doc, Status Result, Action	Evaluates specific information items or document properties as defined by the custom Document Inspector module.

IRibbonControl Object

The `IRibbonControl` object allows developers to pass information to and from a given Ribbon UI control's callback procedure. Review Chapter 14 for a detailed look at how the `IRibbonControl` object is used in Excel.

IRibbonControl Properties

Name	Returns	Description
Context	Window	Read-only. Returns the active window containing the control that triggers the callback procedure.
Id	String	Read-only. Returns the unique identifier for a given control. This ID is specified within the custom UI XML part used to create the custom interface.
Tag	String	Read-only.

IRibbonExtensibility Object

The `IRibbonExtensibility` object provides the interface that allows COM Add-Ins to customize the Ribbon UI.

IRibbonExtensibility Methods

Name	Returns	Parameters	Description
<code>GetCustomUI</code>	<code>String</code>	<code>RibbonID</code>	Loads the XML markup, either from an XML customization file or from XML markup embedded in the procedure that customizes the Ribbon user interface.

IRibbonUI Object

When a host application that contains a custom UI XML part starts, the `onLoad` callback procedure is called, returning an `IRibbonUI` object that points to the Ribbon UI. You can then use the `IRibbonUI` object to either invalidate control caches or perform an immediate refresh of the user interface.

IRibbonUI Methods

Name	Returns	Parameters	Description
<code>Invalidate</code>			Forces the recaching of response values from callback procedures for all controls. For each callback an Add-In implements, the response values are cached. From there, the cached values are used instead of recalling the procedure. The cached values remain in place until the Add-In signals that the cached values are invalid by using the <code>Invalidate</code> method, at which time the callback procedure is again called and the return response is cached.
<code>InvalidateControl</code>		<code>ControlID</code>	Forces the recaching of response values from callback procedures for a single control.

LanguageSettings Object

Returns information about the language settings currently being used in the host application. These are read-only and can affect how data is viewed and edited in certain host applications.

LanguageSettings Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>LanguageID</code>	<code>Long</code>	Read-only. Returns the locale identifier (LCID) for the install language, the user interface language, or the Help language.
<code>LanguagePreferredForEditing</code>	<code>Boolean</code>	Read-only. Returns <code>True</code> if the value for the <code>msoLanguageID</code> constant has been identified in the Windows registry as a preferred language for editing.

MetaProperty Object and the MetaProperties Collection Object

The `MetaProperties` collection object represents a collection of properties describing the metadata stored in a document. Each single property in the `MetaProperties` collection object is represented by its own `MetaProperty` object.

MetaProperties Collection Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Item</code>	<code>MetaProperty</code>	Read-only. Returns a single <code>MetaProperty</code> object based on the name or index number of the property.
<code>SchemaXML</code>	<code>String</code>	Read-only. Returns schema XML that provides information about various metadata properties of a document, such as type information and restrictions.

MetaProperties Collection Methods

Name	Returns	Parameters	Description
<code>GetItemByInternalName</code>	<code>MetaProperty</code>	<code>InternalName</code>	Returns a given property's value using its name, as opposed to its index number.
<code>Validate</code>	<code>String</code>		Validates all properties in a given <code>MetaProperties</code> collection against a schema.

MetaProperty Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Id</code>	String	Read-only. Returns the ID of a given <code>MetaProperty</code> object.
<code>IsReadOnly</code>	Boolean	Read-only. Returns whether a given <code>MetaProperty</code> is read-only.
<code>IsRequired</code>	Boolean	Read-only. Returns whether a given <code>MetaProperty</code> is required.
<code>Name</code>	String	Read-only. Returns the name of the <code>MetaProperty</code> object.
<code>Type</code>	<code>MsoMetaPropertyType</code>	Read-only. Returns the data type of the given <code>MetaProperty</code> object.
<code>Value</code>	Variant	Set/Get the value of a given <code>MetaProperty</code> object.

MetaProperty Methods

Name	Returns	Parameters	Description
<code>Validate</code>	String		Validates a single property against a schema.

MsoEnvelope Object

This Office object allows you to send data from a host application using an Outlook mail item without having to reference and connect to the Outlook object model. Using the `Item` property of this object allows access to a host of Outlook features not available through the `SendMail` feature, such as Voting Options, CC and BCC fields, Body Formatting choices (HTML, rich text, and plain text), and much more.

Note that the `MsoEnvelope` object sends the document as inline (formatted) text. It does not attach the document to an e-mail, though you can add attachments using the `Attachments` property of the `MailItem` object, which you can access via this object's `Item` property. For Excel, this object can only be accessed through a `Worksheet` or a `Chart` object, which means it only sends those objects (and not the entire workbook). Similar to the `SendMail` feature in Excel, except that this exposes a `CommandBar` object associated with this feature and allows for the setting of `Introduction` text.

The properties you set are saved with the document or workbook and are therefore persistent.

MsoEnvelope Properties

The `Parent` property is defined at the beginning of this appendix.

MsoEnvelope Events

Name	Returns	Description
CommandBars	CommandBars	Read-only. Returns a <code>CommandBars</code> collection.
Introduction	String	Set/Get the introductory text that is included with a document that is sent using the <code>MsoEnvelope</code> object. The introductory text is included at the top of the document in the e-mail.
Item	MailItem	Read-Only. Returns a <code>MailItem</code> object that can be used to send the document as an e-mail.

MsoEnvelope Events

Name	Description
EnvelopeHide	Triggered when the user interface that corresponds to the <code>MsoEnvelope</code> object is hidden.
EnvelopeShow	Triggered when the user interface that corresponds to the <code>MsoEnvelope</code> object is displayed.

NewFile Object

Represents a new document listing in the Task Pane of the host application. In Excel, this object allows you to add workbooks to any of the five sections in the Task Pane: Open a Workbook, New, New from existing workbook, New from template, or the bottom section (which has no name). When clicking added workbooks in the New, New from existing workbook, or New from template sections, Excel creates a copy of the file by default unless you override it using the `Action` parameter of the `Add` method.

NewFile Properties

The `Application` and `Creator` properties are defined at the beginning of this appendix.

NewFile Methods

Name	Returns	Parameters	Description
Add	Boolean	FileName As String, Section, DisplayName, Action	Adds a new item to the New Item task pane.
Remove	Boolean	FileName As String, Section, DisplayName, Action	Removes a new item to the New Item task pane.

The ODSOColumn Object and the ODSOColumns Collection Object

The `ODSOColumn` object represents a single field in a Mail Merge Data Source, while the `ODSOColumns` object represents a set of data fields (columns) in a Mail Merge Data Source. Note that these objects cannot be implemented at this time. These objects require that the `OfficeDataSourceObject` be referenced via the `Application` object of the host application. No `OfficeDataSourceObject` exists in any of the `Application` objects in Microsoft Office 2007.

The ODSOFilter Object and the ODSOFilters Collection Object

The `ODSOFilter` object represents a single Filter in the ODSO (Office Data Source Object) Filters collection, while the `ODSOFilters` object represents a set of filters applied to a Mail Merge Data Source. Filters are essentially queries that restrict which records are returned when a Mail Merge is performed. Note that these objects cannot be implemented at this time. These objects require that the `OfficeDataSourceObject` be referenced via the `Application` object of the host application. No `OfficeDataSourceObject` exists in any of the `Application` objects in Microsoft Office 2007.

OfficeDataSourceObject Object

This object represents a data source when performing a Mail Merge operation and allows you to return a set of records that meet specific criteria. Note that this object cannot be implemented at this time. This object requires that the `OfficeDataSourceObject` be referenced via the `Application` object of the host application. No `OfficeDataSourceObject` exists in any of the `Application` objects in Microsoft Office 2007.

OfficeTheme Object

The `OfficeTheme` object exposes the properties that control the color, font, and effects in a given Office theme.

OfficeTheme Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>ThemeColorScheme</code>	<code>ThemeColorScheme</code>	Read-only. Returns a <code>ThemeColorScheme</code> object that exposes the color scheme of a given Office theme.
<code>ThemeEffectScheme</code>	<code>ThemeEffectScheme</code>	Read-only. Returns a <code>ThemeEffectScheme</code> object that exposes the effect scheme of a given Office theme.
<code>ThemeFontScheme</code>	<code>ThemeFontScheme</code>	Read-only. Returns a <code>ThemeFontScheme</code> object that exposes the font scheme of a given Office theme.

ParagraphFormat2 Object

The `ParagraphFormat2` object exposes various properties that control alignment, spacing, and other paragraph formatting options.

ParagraphFormat2 Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Alignment</code>	<code>MsoParagraphAlignment</code>	Set/Get the alignment of the paragraph.
<code>BaselineAlignment</code>	<code>MsoBaselineAlignment</code>	Set/Get the vertical position of fonts in a paragraph.
<code>Bullet</code>	<code>BulletFormat2</code>	Read-only. Returns a <code>BulletFormat2</code> object for the paragraph.
<code>FarEastLineBreakLevel</code>	<code>Boolean</code>	Set/Get the East Asian line break control level for the specified paragraph.
<code>FirstLineIndent</code>	<code>Single</code>	Set/Get the first line indent or hanging indent.
<code>HangingPunctuation</code>	<code>Boolean</code>	Set/Get whether hanging punctuation is enabled for the specified paragraphs.
<code>IndentLevel</code>	<code>Integer</code>	Set/Get the indent level assigned to text in the selected paragraph.
<code>LeftIndent</code>	<code>Single</code>	Set/Get the left indent value for the specified paragraphs.
<code>LineRuleAfter</code>	<code>Boolean</code>	Set/Get whether line spacing after the last line in each paragraph is set to a specific number of points or lines.
<code>LineRuleBefore</code>	<code>Boolean</code>	Set/Get whether line spacing before the first line in each paragraph is set to a specific number of points or lines.
<code>LineRuleWithin</code>	<code>Boolean</code>	Set/Get whether line spacing between base lines is set to a specific number of points or lines.
<code>RightIndent</code>	<code>Single</code>	Set/Get the right indent (in points) for the specified paragraphs.
<code>SpaceAfter</code>	<code>Single</code>	Set/Get the amount of spacing (in points) after the specified paragraph.
<code>SpaceBefore</code>	<code>Single</code>	Set/Get the spacing (in points) before the specified paragraphs.
<code>SpaceWithin</code>	<code>Single</code>	Set/Get the amount of space between base lines in the specified paragraph, in points or lines.
<code>TabStops</code>	<code>TabStops2</code>	Read-only. Returns a <code>TabStops2</code> collection that represents all the custom tab stops for the specified paragraphs.

Name	Returns	Description
TextDirection	MsoText Direction	Set/Get the text direction for the specified paragraph.
WordWrap	Boolean	Set/Get whether the application wraps the Latin text in the middle of a word in the specified paragraphs.

Permission Object

Use the `Permission` object to restrict permissions to the active document, and to return or set specific permissions settings.

Permission Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
Document Author	String	Set/Get the name, in e-mail form, of the author of the active document.
Enabled	Boolean	Set/Get a <code>Boolean</code> value that indicates whether permissions are enabled on the active document.
Enable Trusted Browser	Boolean	Determines if a user can view a document with restricted permission in a web browser if client application is not installed.
Item	User Permission	Read-only. Returns a <code>UserPermission</code> object that is a member of the <code>Permission</code> collection. The <code>UserPermission</code> object associates a set of permissions on the active document with a single user and an optional expiration date.
Permission FromPolicy	Boolean	Read-only. Returns a <code>Boolean</code> value indicating if a permission has been applied to the active document.
Policy Description	String	Read-only. Returns a description of the permission policy that is applied the currently active document.
PolicyName	String	Read-only. Returns the name indicating the permission policy that is currently applied to the active document.
Request PermissionURL	String	When users need additional permission for the current document, this property can contain a web address or e-mail address for the person to contact.
StoreLicenses	Boolean	Set/Get a <code>Boolean</code> value that indicates whether the user's license to view the active document should be cached to allow offline viewing when the user cannot connect to a rights management server.

Permission Methods

Name	Returns	Parameters	Description
Add	User Permission	UserID, Permission, Expiration Date	Creates a new set of permissions on the active document for the specified user.
ApplyPolicy		Filename	Applies the specified permission policy to the active document.
RemoveAll			Removes all <code>UserPermission</code> objects from the <code>Permission</code> collection of the active document.

PolicyItem Object and the ServerPolicy Collection Object

The `ServerPolicy` object represents a policy specified for a document type stored on a server running Office SharePoint Server 2007. Each `ServerPolicy` object contains its own collection of `PolicyItem` objects, each one representing the individual definition settings for one policy item in the active document. Policy items are distinct conditions defined for a document stored on a server running Office SharePoint Server 2007.

ServerPolicy Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
BlockPreview	Boolean	Read-only. Indicates whether you can preview the items using a given server policy.
Description	String	Read-only. Returns the full description of the server policy, including its definition and purpose.
Id	String	Read-only. Returns the ID of a given server policy.
Item	PolicyItem	Read-only. Returns a <code>PolicyItem</code> object representing one policy item.
Name	String	Read-only. Returns the name of a given server policy.
Statement	String	Read-only. Returns the information specified in the server policy statement. This information is displayed as a <i>business bar</i> notification in the Office client application when a document affected by the policy is opened.

PolicyItem Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Data</code>	<code>String</code>	Read-only. Returns the information used to implement a given policy item.
<code>Description</code>	<code>String</code>	Read-only. Returns the description of the current state of a given policy item.
<code>Id</code>	<code>String</code>	Read-only. Returns the ID of a given policy item.
<code>Name</code>	<code>String</code>	Read-only. Returns the name of a given policy item.

ReflectionFormat Object

The `ReflectionFormat` object represents the reflection effect of a given Office graphic.

ReflectionFormat Properties

The `Application` property is defined at the beginning of this appendix.

Name	Returns	Description
<code>Type</code>	<code>MsoReflectionType</code>	Set/Get the reflection type using one of the <code>MsoReflectionType</code> constants.

Ruler2 Object

The `Ruler2` object, returned by the `Ruler2` property of the `TextFrame2` object, represents the ruler for the text in a given shape or text style. This object exposes the tab stops and the indentation settings for text outline levels.

Ruler2 Properties

The `Application` property is defined at the beginning of this appendix.

Name	Returns	Description
<code>Levels</code>	<code>RulerLevels2</code>	Read-only. Returns a <code>RulerLevels2</code> object, exposing outline text formatting.
<code>TabStops</code>	<code>TabStops2</code>	Read-only. Returns the <code>TabStops2</code> collection, exposing the tab stops for the specified text.

RulerLevel2 Object and the RulerLevels2 Collection Object

RulerLevel2 Object and the RulerLevels2 Collection Object

The `RulerLevel2` object exposes information about the first-line and left indent for text at a particular outline level. The `RulerLevels2` collection object will always contain five `RulerLevel2` objects—one for each of the available outline levels.

RulerLevels2 Collection Properties

The `Application` and `Parent` properties are defined at the beginning of this appendix.

RulerLevels2 Collection Methods

Name	Returns	Parameters	Description
<code>Item</code>	<code>RulerLevel2</code>	<code>Index</code>	Read-only. Returns one of the five <code>RulerLevel2</code> objects contained in the <code>RulerLevels2</code> collection.

RulerLevel2 Properties

The `Application`, `Creator` and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>FirstMargin</code>	<code>Single</code>	Set/Get the first-line indent for a given outline level.
<code>LeftMargin</code>	<code>Single</code>	Set/Get the left indent for a given outline level.

ScopeFolder Object and the ScopeFolders Collection Object

Both the `ScopeFolder` object and the `ScopeFolders` collection can be analyzed to determine whether they will be used in a search by the `FileSearch` object. Any `ScopeFolder` you want used in a search is added to the `SearchFolders` collection using the `AddToSearchFolders` method of the `ScopeFolder` object. You will note that although these objects are meant to be used with the `FileSearch` object, the `FileSearch` object is no longer part of the Office 2007 object model.

ScopeFolders Collection Properties

The `Application`, `Count`, and `Creator` properties are defined at the beginning of this appendix.

Name	Returns	Parameters	Description
<code>Item</code>	<code>ScopeFolder</code>	<code>Index</code>	Returns a <code>ScopeFolder</code> object that represents a subfolder of the parent object.

ScopeFolder Properties

The `Application` and `Creator` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Name</code>	<code>String</code>	Read-only. Returns the name of the <code>ScopeFolder</code> object.
<code>Path</code>		Read-only. Returns the full path for a given <code>ScopeFolder</code> object.
<code>ScopeFolders</code>		Read-only. Returns a <code>ScopeFolders</code> collection representing the subfolders of the parent <code>ScopeFolder</code> object.

ScopeFolder Methods

Name	Description
<code>AddToSearchFolders</code>	Adds a <code>ScopeFolder</code> object to the <code>SearchFolders</code> collection.

SearchFolders Collection Object

Represents all of the folders used in a File Search (by the `FileSearch` object). `SearchFolders` consist of `ScopeFolder` objects (with the corresponding `ScopeFolders` collection), which are simply folders. Use the `Add` method of the `SearchFolders` object to add `ScopeFolder` objects to its collection. You will note that although the `SearchFolders` object is meant to be used with the `FileSearch` object, the `FileSearch` object is no longer part of the Office 2007 object model.

SearchFolders Collection Common Properties

The `Application`, `Count`, and `Creator` properties are defined at the beginning of this appendix.

SearchFolders Collection Properties

Name	Returns	Parameters	Description
<code>Item</code>	<code>Scope Folder</code>	<code>Index</code>	Returns a <code>ScopeFolder</code> object that represents a subfolder of the parent object.

SearchFolders Collection Methods

The `Application` and `Creator` properties are defined at the beginning of this appendix.

Name	Parameters	Description
<code>Add</code>	<code>ScopeFolder</code>	Adds a search folder to a file search.
<code>Remove</code>	<code>Index</code>	Removes a specified object from the collection.

SearchScope Object and the SearchScopes Collection Object

The `SearchScope` object represents an individual top-level area in the `SearchScopes` collection object that can be searched when using the `FileSearch` object. The `SearchScopes` collection object contains the list of top-level searchable areas when performing a File Search using the `FileSearch` object. Top-level areas include My Computer, My Network Places, Outlook (folders), and Custom, if available. You will note that although these objects are meant to be used with the `FileSearch` object, the `FileSearch` object is no longer part of the Office 2007 object model.

SearchScopes Collection Properties

The `Application`, `Count`, and `Creator` properties are defined at the beginning of this appendix.

Name	Returns	Parameters	Description
<code>Item</code>	<code>SearchScope</code>	<code>Index as Long</code>	Returns a <code>SearchScope</code> object that corresponds to an area in which to perform a file search, such as local drives or Microsoft Outlook folders.

SearchScope Properties

The `Application` and `Creator` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>ScopeFolder</code>	<code>ScopeFolder</code>	Read-only. Returns a <code>ScopeFolder</code> object.
<code>Type</code>	<code>MsoSearchIn</code>	Read-only. Returns a value that corresponds to the type of <code>SearchScope</code> object. The type indicates the area in which the <code>Execute</code> method of the <code>FileSearch</code> object will search for files.

SharedWorkspace Object

The `SharedWorkspace` property returns a `SharedWorkspace` object that allows the developer to add the active document to a Microsoft Windows SharePoint Services document workspace on the server, and to manage other objects in the shared workspace.

SharedWorkspace Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Connected</code>	<code>Boolean</code>	Read-only. Indicates if the current document is saved in and connected to a workspace.

Name	Returns	Description
Files	Shared Workspace Files	Read-only. Returns a collection of SharedWorkspaceFiles.
Folders	Shared Workspace Folders	Read-only. Returns a collection of SharedWorkspaceFolders.
LastRefreshed	Date/Time	Read-only. Returns a date and time indicating the last time the Refresh was most recently called.
Links	Shared Workspace Links	Read-only. Returns a collection of SharedWorkspaceLinks.
Members	Shared Workspace Members	Read-only. Returns a collection of SharedWorkspaceMembers.
Name	String	Set/Get the name of the object. Read/Write.
SourceURL	String	Read-only. Designates the location of a shared document.
Tasks	Share dWorkspace Tasks	Read-only. Returns a collection of SharedDocumentTasks.
URL	String	Read-only. Returns the URL of the shared workspace.

SharedWorkspace Methods

Name	Parameters	Description
CreateNew	URL, Name	Creates a new SharedWorkspace object.
Delete		Deletes a SharedWorkspace object from the collection.
Disconnect		Disconnects from the shared workspace.
Refresh		Refreshes the current copy of a document from the shared workspace.
RemoveDocument		Removes the current document from the shared workspace.

SharedWorkspaceFile Object and the SharedWorkspaceFiles Collection Object

The SharedWorkspaceFile object represents a file that has been saved in a shared document workspace. This shared document workspace would typically be a SharePoint server. The SharedWorkspaceFiles collection object represents multiple SharedWorkspaceFile objects.

SharedWorkspaceFiles Collection Properties

SharedWorkspaceFiles Collection Properties

The Application, Count, Creator, and Parent properties are defined at the beginning of this appendix.

Name	Returns	Description
Item	SharedWorkSpaceFile	Read-only. Returns a SharedWorkSpaceFile object from the Files collection of the SharedWorkSpace object.
ItemCountExceeded	Boolean	Read-only. Returns whether the number of files allowed in the shared workspace has been exceeded.

SharedWorkspaceFiles Collection Methods

Name	Returns	Parameters	Description
Add	SharedWorkspaceFile	FileName, ParentFolder, OverwriteifFileAlreadyExists, KeepInSync	Adds a file to the document library in a shared workspace.

SharedWorkspaceFile Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

Name	Returns	Description
CreatedBy	String	Read-only. Returns the name of creator of the shared workspace object .
CreatedDate	Variant	Read-only. Returns the date on which the shared workspace object was created.
ModifiedBy	String	Read-only. Returns the name of the member who last modified the shared workspace object.
ModifiedDate	Variant	Read-only. Returns the date on which the shared workspace object was last modified.
URL	String	Read-only. Returns the URL and filename of a given shared workspace folder.

SharedWorkspaceFile Methods

Name	Description
Delete	Deletes a SharedWorkSpaceFile object.

SharedWorkspaceFolder Object and the SharedWorkspaceFolders Collection Object

The `SharedWorkspaceFolder` object represents a single subfolder within the main document library folder of a shared workspace. The `SharedWorkspaceFolders` object is a collection of multiple `SharedWorkspaceFolder` objects.

SharedWorkspaceFolders Collection Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Item</code>	<code>SharedWorkspaceFolder</code>	Read-only. Returns a <code>SharedWorkspaceFolder</code> object from the <code>Folders</code> collection of the <code>SharedWorkspace</code> object.
<code>ItemCountExceeded</code>	<code>Boolean</code>	Read-only. Returns whether the number of folders allowed in the shared workspace has been exceeded.

SharedWorkspaceFolders Collection Methods

Name	Parameters	Description
<code>Add</code>	<code>FolderName</code> , <code>ParentFolder</code>	Adds a folder to the document library in a shared workspace.

SharedWorkspaceFolder Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>FolderName</code>	<code>String</code>	Read-only. Returns the name of a given subfolder in the document library in a shared workspace.

SharedWorkspaceFolder Methods

Name	Parameters	Description
<code>Delete</code>	<code>DeleteEvenIfFolderContainsFiles</code> As <code>Boolean</code>	Deletes a given subfolder and all of its files if specified.

SharedWorkspaceLink Object and the SharedWorkspaceLinks Collection Object

The `SharedWorkspaceLink` object is used to manage links to additional documents and information of interest to the members who are collaborating on the documents in the shared workspace site. The `SharedWorkspaceLinks` object represents a collection of `SharedWorkspaceLink` objects on a given shared workspace.

SharedWorkspaceLinks Collection Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Item</code>	<code>SharedWorkspaceLink</code>	Read-only. Returns a <code>SharedWorkspaceLink</code> object from the <code>Links</code> collection of the <code>SharedWorkSpace</code> object.
<code>ItemCountExceeded</code>	<code>Boolean</code>	Read-only. Returns whether the number of links allowed in the shared workspace has been exceeded.

SharedWorkspaceLinks Collection Methods

Name	Parameters	Description
<code>Add</code>	<code>URL</code> , <code>Description</code> , <code>Notes</code>	Adds a link to the list of links in a shared workspace.

SharedWorkspaceLink Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>CreatedBy</code>	<code>String</code>	Read-only. Returns the name of creator of the shared workspace object.
<code>CreatedDate</code>	<code>Variant</code>	Read-only. Returns the date on which the shared workspace object was created.
<code>Description</code>	<code>String</code>	Set/Get some descriptive text for the specified <code>SharedWorkspaceLink</code> or <code>SharedWorkspaceTask</code> .
<code>ModifiedBy</code>	<code>String</code>	Read-only. Returns the name of member who last modified the shared workspace object.
<code>ModifiedDate</code>	<code>Variant</code>	Read-only. Returns the date on which the shared workspace object was last modified.

Name	Returns	Description
Notes	String	Set/Get the optional notes associated with a workspace link.
URL	String	Set/Get the URL for a given shared workspace link.

SharedWorkspaceLink Methods

Name	Parameters	Description
Delete		Deletes the specified link.
Save	QueryName	Saves changes to the specified link.

SharedWorkspaceMember Object and the SharedWorkspaceMembers Collection Object

The `SharedWorkspaceMember` object is used to manage users and their rights on a given shared workspace site. The `SharedWorkspaceMembers` object represents a collection of users who have the right to work with shared documents on a given shared workspace site.

SharedWorkspaceMembers Collection Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
Item	Shared Workspace Member	Read-only. Returns a <code>SharedWorkspaceMember</code> object from the <code>Members</code> collection of the <code>SharedWorkSpace</code> object.
ItemCount Exceeded	Boolean	Read-only. Returns whether the number of members allowed in the shared workspace has been exceeded.

SharedWorkspaceMembers Collection Methods

Name	Parameters	Description
Add	Email, DomainName, DisplayName, Role	Adds a user to the list of members in a shared workspace.

SharedWorkspaceMember Properties

SharedWorkspaceMember Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>DomainName</code>	<code>String</code>	Read-only. Returns the domain and username for a given shared workspace user.
<code>Email</code>	<code>String</code>	Read-only. Returns the e-mail address for a given shared workspace user.
<code>Name</code>	<code>String</code>	Read-only. Returns the name of a given shared workspace user.

SharedWorkspaceMember Methods

Name	Description
<code>Delete</code>	Deletes a given shared workspace user.

SharedWorkspaceTask Object and the SharedWorkspaceTasks Collection Object

The `SharedWorkspaceTask` object is used to manage tasks assigned to users on a given shared workspace site. The `SharedWorkspaceTasks` object represents a collection of tasks.

SharedWorkspaceTasks Collection Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Item</code>	<code>SharedWorkspaceTask</code>	Read-only. Returns a <code>SharedWorkspaceTask</code> object from the <code>Tasks</code> collection of the <code>SharedWorkSpace</code> object.
<code>ItemCountExceeded</code>	<code>Boolean</code>	Read-only. Returns whether the number of tasks allowed in the shared workspace has been exceeded.

SharedWorkspaceTasks Collection Methods

Name	Returns	Parameters	Description
<code>Add</code>	<code>SharedWorkspaceTask</code>	<code>Title</code> , <code>Status</code> , <code>Priority</code> , <code>Assignee</code> , <code>Description</code> , <code>DueDate</code>	Adds a task to the list of tasks in a shared workspace.

SharedWorkspaceTask Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>AssignedTo</code>	String	Set/Get the name of the user to which the task is assigned.
<code>CreatedBy</code>	String	Read-only. Returns the name of creator of the task.
<code>CreatedDate</code>	Variant	Read-only. Returns the date on which the task was created.
<code>Description</code>	String	Set/Get an optional description value for the task.
<code>DueDate</code>	Variant	Set/Get the date and time the task is due.
<code>ModifiedBy</code>	String	Read-only. Returns the name of the member who last modified the task.
<code>ModifiedDate</code>	Variant	Read-only. Returns the date on which the task was last modified.
<code>Priority</code>	MsoSharedWorkspaceTaskPriority	Set/Get the value setting the priority for the task.
<code>Status</code>	MsoSharedWorkspaceTaskStatus	Set/Get the value setting the status of the task.
<code>Title</code>	String	Set/Get the value setting the status for the task.

SharedWorkspaceTask Methods

Name	Description
<code>Delete</code>	Deletes the current task.
<code>Save</code>	Saves changes to the current task.

Signature Object and the SignatureSet Collection Object

The `Signature` object represents a digital signature attached to a document. A document can contain multiple `Signature` objects, which are held in a `SignatureSet` collection.

Digital signatures are electronic versions of handwritten signatures. Digital signatures protect users from opening documents that could contain macro viruses, and they also protect authors by ensuring that the contents of the documents remain unchanged. When you digitally sign a document, an encrypted key is added to the signature. When other users change the document, a message appears informing them that they do not have the key to unlock the signature. This causes the document to lose its signature.

This object is currently not accessible in Microsoft Excel, though it is available through the `Document` object in Microsoft Word and the `Presentation` object in Microsoft PowerPoint.

SignatureSet Collection Properties

SignatureSet Collection Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>CanAddSignatureLine</code>	Boolean	Read-only. Indicates whether a signature line can be added to the document.
<code>Item</code>	Signature	Read-only. Returns a <code>Signature</code> object corresponding to one of the digital signatures with which the document is currently signed.
<code>ShowSignaturesPane</code>	Boolean	Set/Get whether the signature task pane is displayed.
<code>Subset</code>	<code>MsoSignatureSubset</code>	Set/Get the <code>MsoSignatureSubset</code> that is to be used as a filter on the available <code>Signature</code> objects for a document.

SignatureSet Collection Methods

Name	Returns	Parameters	Description
<code>AddNonVisibleSignature</code>	Signature	<code>varSigProv</code>	Creates a signature packet when digitally signing a document.
<code>AddSignatureLine</code>	Signature	<code>varSigProv</code>	Adds lines to a document where signatures are collected.

Signature Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>CanSetup</code>	Boolean	Read-only. Indicates whether the user can set the properties of the <code>Signature</code> object.
<code>Details</code>	<code>SignatureInfo</code>	Read-only. Returns information about the signature, such as whether the certificate associated with the signature has expired, whether the signature is valid, and whether the signature is read-only.
<code>IsSignatureLine</code>	Boolean	Read-only. Returns whether this is a signature line.
<code>IsSigned</code>	Boolean	Read-only. Returns whether the digital certificate that corresponds to the <code>Signature</code> object has been signed successfully.
<code>Setup</code>	<code>SignatureSetup</code>	Read-only. Returns the <code>SignatureSetup</code> object, exposing the various properties of the signature packet.

Name	Returns	Description
Signature LineShape	Object	Read-only. Returns <i>Shape</i> object that makes up the signature line for the associated <i>Signature</i> object.
SortHint	Long	Read-only. Returns the value representing the sort order of the signatures in a packet with multiple signatures.

Signature Methods

Name	Parameters	Description
Delete		Deletes the specified signature from the <i>SignatureSet</i> collection.
ShowDetails		Displays the details related to a signature packet.
Sign	<i>varSigImg</i> , <i>VarDelSugg</i> <i>Signer</i> , <i>varDelSugg</i> <i>SignerLine2</i> , <i>VarDelSugg</i> <i>SignerEmail</i>	Creates a signature packet.

SignatureInfo Object

The *SignatureInfo* object exposes the properties and methods used to create a digital or in-document signature.

SignatureInfo Properties

The *Application* and *Creator* properties are defined at the beginning of this appendix.

Name	Returns	Description
Certificate Verification Results	Certificate Verification Results	Read-only. Returns the results from the verification of a digital certificate.
Content Verification Results	Content Verification Results	Read-only. Returns the results from the verification of the hash contents of a signed document.
IsCertificate Expired	Boolean	Read-only. Returns whether a given digital certificate is expired.
IsCertificate Revoked	Boolean	Read-only. Returns whether a given digital certificate is revoked.

Table continued on following page

SignatureInfo Methods

Name	Returns	Description
IsCertificateUntrusted	Boolean	Read-only. Returns whether a given digital certificate comes from an untrusted source.
IsValid	Boolean	Read-only. Returns whether a signature was successfully validated.
ReadOnly	Boolean	Read-only. Returns whether the <code>SignatureInfo</code> object is read-only.
SignatureComment	String	Set/Get the comments for a signature packet.
SignatureImage	IPictureDisp	Set/Get the value of the image used to sign the document.
SignatureProvider	String	Read-only. Identifies the installed signature provider Add-In.
SignatureText	String	Set/Get the text used to sign the document.

SignatureInfo Methods

Name	Parameters	Description
GetCertificateDetail	Certdet	Displays a specified certificate detail value as defined by the passed <code>CertificateDetail</code> constant.
GetSignatureDetail	sigdet	Displays a specified signature detail value as defined by the passed <code>SignatureDetail</code> constant.
SelectCertificateDetailByThumbprint	bstr Thumbprint	Displays a dialog box containing information about a digital certificate following verification of the user from a thumbprint.
SelectSignatureCertificate	ParentWindow	Displays a dialog box allowing users to select a signature certificate to use for signing the document.
ShowSignatureCertificate	ParentWindow	Displays the selected or default digital certificate.

SignatureProvider Object

The `SignatureProvider` object represents a signature provider Add-In implemented via a custom COM Add-In. Note that signature providers cannot be implemented in VBA.

SignatureProvider Methods

Name	Returns	Parameters	Description
Generate Signature LineImage	IPictureDisp	Siglnimg, psigsetup, psiginfo, xmlDsigStream	Gets the signature line image.
GetProvider Detail	Variant	sigprovdet	Queries the signature provider for various details
HashStream	Byte	QueryContinue, Stream	Allows a signature provider Add-In to create a hash value for the document that you can use to determine if the document contents were tampered with after digital signing.
Notify Signature Added		ParentWindow, psigsetup, psiginfo	Used to display a dialog box informing the user that the signing process has completed, and providing additional functionality for the Add-In.
ShowSignature Added		ParentWindow, psigsetup, psiginfo	Provides a signature provider Add-In the opportunity to display details about a signed signature line and display additional stored information, such as a secure time-stamp.
Show Signature Details		ParentWindow, psigsetup, XmlDsigStream, pcontverres, pserverres	Provides a signature provider Add-In the opportunity to display details about a signature.
Show Signature Setup		ParentWindow, psigsetup	Provides a signature provider Add-In the opportunity to display the Signature Setup dialog box to the user.
ShowSigning Ceremony		ParentWindow, psigsetup, psiginfo	Provides a signature provider Add-In the opportunity to display the Signature dialog box to users, allowing them to specify their identity and then be authenticated.
SignXmlDsig		QueryContinue, psigsetup, psiginfo, XmlDsigStream	Used to sign the XMLDSIG template. XMLDSIG is a standards-based signature format (www.w3.org/tr/xmlsig-core/), verifiable by third parties. This is the default format for signatures in Office 2007.

Table continued on following page

SignatureSetup Object

Name	Returns	Parameters	Description
VerifyXmlDsig		QueryContinue, psigsetup, psiginfo, XmlDsigStream, pcontverres, pcertverres	Verifies a signature based on the signed state of the document and the legitimacy of the certificate used for signing.

SignatureSetup Object

The `SignatureSetup` object exposes the various properties used to set up a signature packet.

SignatureSetup Properties

The `Application` and `Creator` properties are defined at the beginning of this appendix.

Name	Returns	Description
AdditionalXml	String	Set/Get any XML data added to the signature during setup.
AllowComments	Boolean	Set/Get whether the signer can enter comments.
Id	String	Read-only. Returns the ID of the signature provider for a document.
ReadOnly	Boolean	Read-only. Returns whether the <code>SignatureSetup</code> object is read-only.
ShowSignDate	Boolean	Set/Get whether the document signed date should be displayed.
Signature Provider	String	Read-only. Identifies the installed signature provider Add-In.
Signing Instructions	String	Set/Get the instructions for signing the document.
Suggested Signer	String	Set/Get the name of the principal signer.
Suggested SignerEmail	String	Set/Get the e-mail address of the principal signer.
Suggested SignerLine2	String	Set/Get the additional information on the principal signer, such as title, address, phone, and so on.

SmartDocument

The `SmartDocument` object is used to manage the XML expansion pack attached to the currently active document.

SmartDocument Properties

The `Application` and `Creator` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>SolutionID</code>	<code>String</code>	Set/Get the ID identifying the XML expansion pack attached to the active document.
<code>SolutionURL</code>	<code>String</code>	Set/Get the absolute URL that links to the XML expansion pack attached to the active document.

SmartDocument Methods

Name	Parameters	Description
<code>PickSolution</code>	<code>ConsiderAll Schemas</code>	Displays a dialog box that allows the user to choose an available XML expansion pack to attach to the active document.
<code>RefreshPane</code>		Refreshes the <code>Document Actions</code> task pane for the active document.

SoftEdgeFormat Object

The `SoftEdgeFormat` object represents the soft edges effect of a given Office graphic.

ReflectionFormat Properties

The `Application` and `Creator` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Type</code>	<code>MsoSoftEdgeType</code>	Set/Get the soft edge type using one of the <code>MsoSoftEdgeType</code> constants.

Sync Object

Use the `Sync` object to manage the synchronization of the local and server copies of a shared document stored in a Windows SharePoint Services document workspace. The `Status` property returns important information about the current state of synchronization. Use the `GetUpdate` method to refresh the sync status. Use the `LastSyncTime`, `ErrorType`, and `WorkspaceLastChangedBy` properties to return additional information.

Sync Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Sync Methods

Name	Returns	Description
ErrorType	MsoSync ErrorType	Read-only. Returns the <code>MsoSyncErrorType</code> constant indicating the type of document synchronization error that most recently occurred.
LastSyncTime	Variant	Read-only. Returns the date and time when the local copy of the active document was last synchronized.
Status	MsoSync StatusType	Read-only. Returns the <code>MsoSyncStatusType</code> constant indicating the status of the most recent synchronization.
WorkspaceLast ChangedBy	String	Read-only. Returns the name of the user who last saved changes to the server copy of a shared document.

Sync Methods

Name	Parameters	Description
GetUpdate		Compares the local version of the shared document to the version on the server.
OpenVersion	SyncVersion Type	Opens a different version of the shared document along side the currently open local version.
PutUpdate		Updates the server copy of the shared document with the local copy.
Resolve Conflict	SyncConflict Resolution	Resolves conflicts between the local and server copies of a shared document.
Unsuspend		Resumes synchronization between the local copy and the server copy of a shared document.

TabStop2 Object and the TabStops2 Collection Object

The `TabStop2` object represents a single numerically indexed tab stop along the ruler. The `TabStops2` collection object contains all of `TabStop2` objects represented in a given document.

TabStops2 Collection Properties

The `Application`, `Creator`, `Count`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
Default Spacing	Sing	Set/Get the default spacing between tab stops.

TabStops2 Collection Methods

Name	Returns	Parameters	Description
Add		Type, Position	Adds a new tab stop.
Item	TabStop2	Index	Returns the individual TabStop2 object matching a specified index value.

TabStop2 Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

Name	Returns	Description
Position	Single	Set/Get the position of a given tab stop relative to the left margin.
Type	MsoTabStopType	Set/Get the type of TabStop2 object by using one of the MsoTabStopType constants.

TabStop2 Methods

Name	Description
Clear	Removes the specified custom tab stop.

TextColumn2 Object and the TextColumns2 Collection Object

The TextColumn2 object represents a single text column. Multiple TextColumn2 objects are contained within the TextColumns2 collection. As of this writing, the TextColumns2 collection is not available in the Office object model. This will most assuredly be corrected in a future version of Office.

TextColumn2 Properties

The Application and Creator properties are defined at the beginning of this appendix.

Name	Returns	Description
Number	Integer	Set/Get the index number of the specified column.
Spacing	Single	Set/Get the spacing between text columns.
TextDirection	MsoTextDirection	Set/Get the direction of the text in the text column.

TextRange2 Object

The `TextRange2` object exposes the text in a text frame, as well as the properties and methods that control the alignment and anchoring of the text frame.

TextRange2 Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>BoundHeight</code>	Single	Read-only. Returns the height of the bounding box, which represents the perimeter immediately surrounding the text.
<code>BoundLeft</code>	Single	Read-only. Returns the left coordinate of the bounding box, which represents the perimeter immediately surrounding the text.
<code>BoundTop</code>	Single	Read-only. Returns the top coordinate of the bounding box, which represents the perimeter immediately surrounding the text.
<code>BoundWidth</code>	Single	Read-only. Returns the width of the bounding box, which represents the perimeter immediately surrounding the text.
<code>Characters</code>	<code>TextRange2</code>	Read-only.
<code>Font</code>	Font	Returns a <code>Font</code> , which exposes the character formatting for the <code>TextRange2</code> object.
<code>LanguageID</code>	<code>MsoLanguageID</code>	Set/Get the language value for the <code>TextRange2</code> object. Use one of the <code>MsoLanguageID</code> constants.
<code>Length</code>	Long	Read-only. Returns the length of the text range.
<code>Lines</code>	<code>TextRange2</code>	Read-only. Returns a specified subset of text lines.
<code>ParagraphFormat</code>	<code>ParagraphFormat</code>	Returns a <code>ParagraphFormat</code> , which exposes the paragraph formatting for the <code>TextRange2</code> object.
<code>Paragraphs</code>	<code>TextRange2</code>	Read-only. Returns a specified subset text paragraphs.
<code>Runs</code>	<code>TextRange2</code>	Read-only. Returns a specified subset of text runs (a range of characters that share the same font attributes).
<code>Sentences</code>	<code>TextRange2</code>	Read-only. Returns a specified subset of text sentences.
<code>Start range.</code>	Long	Read-only. Returns the starting point for the specified text
<code>Text</code>	String	Set/Get the text in the specified text range.
<code>Words</code>	<code>TextRange2</code>	Read-only. Returns a <code>TextRange2</code> object that represents a subset of text matching the parameters passed in the expression <code>Words(Start, Length)</code> . The <code>Start</code> parameter specifies the first word in the returned range. The <code>Length</code> parameter specifies the number of words to be returned.

TextRange2 Methods

Name	Returns	Parameters	Description
AddPeriods			Adds period (.) punctuation to the text contained in a given TextRange2 object.
ChangeCase		Type	Changes the case of a TextRange2 object to one of MsoTextChangeCase constants.
Copy			Copies a TextRange2 object.
Cut			Removes a portion or all of the text from a range of text.
Delete			Deletes a TextRange2 object.
Find		FindWhat, AfterMatchCase, WholeWords	Searches a TextRange2 object for a subset of text.
InsertAfter		NewText	Inserts text to the right of existing text in the TextRange2 object.
InsertBefore		NewText	Inserts text to the left of existing text in the TextRange2 object.
InsertSymbol		FontName, ChartNumber, Unicode	Inserts a symbol from the specified font set.
Item		Index	Gets the range of text specified by the index number from the TextRange2 object.
LtrRun			Returns a TextRange2 object that represents the specified subset of left-to-right text runs (a range of characters that share the same font attributes).
Paste			Pastes the contents of the Clipboard into the TextRange2 object.
PasteSpecial		Format	Replaces the text range with the contents of the Clipboard in the format specified.
RemovePeriods			Removes all period (.) punctuation from the text in the TextRange2 object.

Table continued on following page

ThemeColor Object

Name	Returns	Parameters	Description
Replace		FindWhat, ReplaceWhate, After, MatchCase, WholeWords	Finds specific text in a text range, replaces the found text with a specified string, and returns a TextRange2 object that represents the first occurrence of the found text. Returns nothing if no match is found.
RotateBounds		X1, Y1, X2, Y2, X3, Y3, X4, Y4	Gets the coordinates of the vertices of the text bounding box for the specified text range.
RtlRun			Returns a TextRange2 object that represents the specified subset of right-to-left text runs. A text run consists of a range of characters that share the same font attributes.
Select			Selects the TextRange2 object.
TrimText			Removes the white space on the left and right sides of the text in the TextRange2 object.

ThemeColor Object

The `ThemeColor` object exposes the properties that can be used to configure the color in the color scheme of a given theme.

ThemeColor Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
RGB	MsoRGBType	Set/Get the value of a color in the color scheme of a given theme.
ThemeColor Scheme Index	MsoTheme ColorScheme Index	Read-only. Returns a <code>MsoThemeColorSchemeIndex</code> constant representing the index value of the color scheme for a given theme.

ThemeColorsScheme Object

The `ThemeColorScheme` object exposes the properties and methods for working with the color scheme of a given theme.

ThemeColorsScheme Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

ThemeColorsScheme Methods

Name	Returns	Parameters	Description
Colors	ThemeColor	Index	Gets a ThemeColor object that represents a color in the color scheme of a given theme.
GetCustomColor	MsoRGBType	Name	Gets a value that represents a color in the color scheme of a given theme.
Load		FileName	Loads a specified color scheme from a file.
Save		FileName	Saves a specified color scheme to a file.

ThemeEffectScheme Object

The ThemeEffectScheme object exposes the properties and methods for working with the effect scheme of a given theme.

ThemeEffectScheme Properties

The Application, Creator, and Parent properties are defined at the beginning of this appendix.

ThemeEffectScheme Methods

Name	Parameters	Description
Load	FileName	Loads a specified effects scheme from a file.

ThemeFont Object and the ThemeFonts Collection Object

The ThemeFont represents a container for the font schemes in a given theme. The ThemeFonts object represents a collection of major and minor fonts in the font scheme of a given theme.

ThemeFonts Collection Properties

The Application, Count, Creator, and Parent properties are defined at the beginning of this appendix.

ThemeFonts Collection Methods

Name	Returns	Parameters	Description
Item	ThemeFont	Index	Returns one of the three language fonts contained in the ThemeFonts collection.

ThemeFont Properties

ThemeFont Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Name</code>	<code>String</code>	Sets/Gets the name of a font in the font scheme of a given theme.

ThemeFontScheme Object

The `ThemeFontScheme` object exposes the properties and methods for working with the font scheme of a given theme.

ThemeFontScheme Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>MajorFont</code>	<code>ThemeFonts</code>	Read-only. Returns the font settings for the Headings in a document.
<code>MinorFont</code>	<code>ThemeFonts</code>	Read-only. Returns the font settings for the body of a document.

ThemeFontScheme Methods

Name	Parameters	Description
<code>Load</code>	<code>FileName</code>	Loads a specified font scheme from a file.
<code>Save</code>	<code>FileName</code>	Saves a specified font scheme to a file.

UserPermission

The `UserPermission` object is used to assign permissions for the active document on a per-user basis with per-user expiration dates. This object represents a member of the active document's `Permission` collection.

UserPermission Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
ExpirationDate	Variant	Set/Get an optional expiration date on the permissions assigned to the user associated with a given <code>UserPermission</code> object.
Permission	MsoPermission	Set/Get an <code>MsoPermission</code> constant, representing the permissions assigned to the user associated with a given <code>UserPermission</code> object.
UserId	String	Read-only. Returns the e-mail address of the user whose permissions on the active document are determined by the specified <code>UserPermission</code> object.

UserPermission Methods

Name	Description
Remove	Removes the current <code>UserPermission</code> object from the collection of <code>Permissions</code> .

WebPageFont Object and the WebPageFonts Collection Object

The `WebPageFont` object represents which fixed and proportional font and size are used when the host application's documents are saved as web pages. Microsoft Excel and Microsoft Word also use these settings when you open a web page within the application, but the settings only take effect when the web page being opened cannot display its own font settings, or when no font information is contained within the HTML code.

Be aware that the `FixedWidthFont` and `ProportionalFont` properties will accept any valid `String` and `FixedWidthFontSize`, and `ProportionalFontSize` will accept any valid `Single` value. For example, the following will not encounter an error, even though they aren't valid font and size settings:

```
Application.DefaultWebOptions.Fonts(msoCharacterSetEnglishWesternEuropean
OtherLatinScript).ProportionalFont = "XXXXXXXXX"

Application.DefaultWebOptions.Fonts(msoCharacterSetEnglishWesternEuropean
OtherLatinScript).ProportionalFontSize = 1200
```

An error will occur when the application attempts to use these settings.

The `WebPageFonts` collection object can be referenced using the `Fonts` property of the `DefaultWebOptions` property in the host's `Application` object, like so:

```
Set oWebPageFonts = Application.DefaultWebOptions.Fonts
```

WebPageFonts Collection Properties

Note that the `count` property of the `WebPageFonts` collection object always returns 0, even though there are 12 `WebPageFont` objects (character sets) in the collection.

WebPageFonts Collection Properties

The `Application`, `Count`, and `Creator` properties are defined at the beginning of this appendix.

Name	Returns	Parameters	Description
<code>Item</code>	<code>WebPageFont</code>	<code>Index</code> as <code>MsoCharacterSet</code>	Returns a <code>WebPageFont</code> object from the <code>WebPageFonts</code> collection for a particular value of <code>MsoCharacterSet</code> .

WebPageFont Properties

The `Application` and `Creator` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>FixedWidthFont</code>	<code>String</code>	Set/Get the fixed-width font setting in the host application.
<code>FixedWidthFontSize</code>	<code>Single</code>	Set/Get the fixed-width font size setting (in points) in the host application.
<code>ProportionalFont</code>	<code>String</code>	Set/Get the proportional font setting in the host application.
<code>ProportionalFontSize</code>	<code>Single</code>	Set/Get the proportional font size setting (in points) in the host application.

WorkflowTask Object and the WorkflowTasks Collection Object

The `WorkflowTask` object is used to manage tasks assigned to users for a SharePoint workflow process. The `WorkflowTasks` object represents a collection of tasks.

WorkflowTasks Collection Properties

The `Application`, `Count`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Item</code>	<code>WorkflowTask</code>	Read-only. Returns the <code>WorkflowTask</code> object matching a specified index value.

WorkflowTask Properties

The `Application`, `Creator`, and `Parent` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>AssignedTo</code>	String	Read-only. Returns the name of the user to which the task is assigned.
<code>CreatedBy</code>	String	Read-only. Returns the name of creator of the task.
<code>CreatedDate</code>	Variant	Read-only. Returns the date on which the task was created.
<code>Description</code>	String	Read-only. Returns an optional description value for the task.
<code>DueDate</code>	Variant Date	Read-only. Returns the date and time the task is due.
<code>Id</code>	String	Read-only. Returns the ID of the SharePoint list item.
<code>ListID</code>	String	Read-only. Returns the ID of the list containing the workflow task.
<code>Name</code>	String	Read-only. Returns the name of the workflow task.
<code>WorkflowID</code>	String	Read-only. Returns the ID of the workflow associated with a workflow task.

WorkflowTask Methods

Name	Returns	Description
<code>Show</code>	Long	Displays a workflow task edit user interface for the specified <code>WorkflowTask</code> object.

WorkflowTemplate Object and the WorkflowTemplates Collection Object

The `WorkflowTemplate` object represents a single workflow in the collection of workflows that may be available for the current document. The collection for workflows is represented by the `WorkflowTemplates` object.

WorkflowTemplates Collection Properties

The `Application`, `Count`, and `Creator` properties are defined at the beginning of this appendix.

Name	Returns	Description
<code>Item</code>	Workflow Template	Read-only. Returns the <code>WorkflowTemplate</code> object matching a specified index value.

WorkflowTemplate Properties

WorkflowTemplate Properties

The `Application` and `Creator` properties are defined at the beginning of this appendix.

Name	Returns	Description
Description	String	Read-only. Returns the description value for the workflow template.
Document Library Name	String	Read-only. Returns the name of the document library associated with the workflow template.
Document LibraryURL	String	Read-only. Returns the URL of the document library where workflow templates are stored.
Id	String	Read-only. Returns the ID of the template used to create a workflow instance.
Name	String	Read-only. Returns the name of the <code>WorkflowTemplate</code> object.

WorkflowTemplate Methods

Name	Returns	Parameters	Description
Show	Long		Displays a configuration user interface for the specified <code>WorkflowTemplate</code> object.

Index

A

- AboveAverage **object**, **636–638, 753**
- AboveBelow **property**, AboveAverage **object**, **637**
- Abs **function**, **156**
- absolute recording**, **7–8**
- Accept **property**, CalloutFormat **object**, **680**
- AcceptAllChanges **method**, Workbook **object**, **944**
- Access, and ADO**
 - connecting to Access, **448–449**
 - inserting, updating, and deleting records with plain text SQL, **452–454**
 - overview, **419–423**
 - retrieving data using plain text queries, **449–450**
 - retrieving data using stored queries, **451–452**
- Action **object**, **638–639**
- ActionControl **property**, CommandBars **collection**, **999**
- Actions **collection**, **638–639**
- Activate **event**
 - Chart **object**, **694**
 - Workbook **object**, **951**
 - Worksheet **object**, **962**
- Activate **method**
 - Chart **object**, **690**
 - overview, **93–94**
 - Range **object**, **858**
 - VBcomponent **object**, **987**
 - Window **object**, **933**
 - Workbook **object**, **944**
 - Worksheet **object**, **958**
- ActivateMso **method**, **317**
- ActivateNext **method**, Windows **object**, **933**
- ActivatePrevious **method**, Windows **object**, **933**
- Active Office language**, **538**
- active properties**, **64–65**
- Active Server pages (ASP)**, **532**
- ActiveCell **property**
 - Application **object**, **643**
 - Windows **collection**, **930**
- ActiveChart **object**, **186**
- ActiveChart **property**
 - Application **object**, **643**
 - Window **object**, **930**
 - Workbooks **collection**, **937**
- ActiveCodePane **property**, VBE, **989**
- ActiveConnection **property**
 - Command **object**, **457**
 - Recordset **object**, **441–442, 446, 463**
- ActiveMenuBar **property**, CommandBars **collection**, **999**
- ActiveMicrosoftApp **method**, Application **object**, **656**
- ActivePane **property**, Windows **object**, **930**
- ActivePrinter **property**, Application **object**, **643**
- ActiveSheet **property**
 - Application **object**, **644**
 - Window **object**, **930**
 - Workbooks **collection**, **937**
- ActiveSheet.UsedRange**, **107, 684, 726, 759, 925**
- ActiveSheetView **property**, Windows **object**, **930**
- ActiveVbProject **property**, VBE, **989**
- ActiveWindow **property**
 - Application **object**, **644**
 - VBE, **989**
- ActiveWorkbook **property**, **65, 644**
- ActiveX category**, **496**
- ActiveX **controls**
 - CheckBox **control**, **212**
 - dynamic, **216–219**
 - Option Button **controls**, **212–214**

ActiveX controls (continued)

ActiveX controls (continued)

- and running macros, 13–15
- Scrollbar control, 211
- Spin Button control, 211–212

ActiveX Data Objects (ADO)

- Command object
 - collections, 447
 - methods, 446–447
 - properties, 446
- Connection object, 437–441
 - collections, 441–443
 - events, 440–441
 - methods, 439–440
 - properties, 437–438
- Recordset object, 441–445
 - collections, 445
 - events, 445
 - methods, 443–445
 - properties, 441–443
- SELECT Statement, 432–434
- Structured Query Language (SQL) overview, 431–435
 - DELETE statement, 435
 - INSERT statement, 434
 - UPDATE statement, 434–435
- using in Excel applications, 447–448
- using with Access, 448–454
 - connecting to Access, 448–449
 - inserting, updating, and deleting records with plain text SQL, 452–454
 - retrieving data using plain text queries, 449–450
 - retrieving data using stored queries, 451–452
- using with non-standard data sources, 463–468
 - inserting and updating records in workbooks, 466–467
 - querying text files, 467–468
 - querying workbooks, 464–466
- using with SQL Server, 454–463
 - connecting to SQL Server, 455–456
 - disconnected recordsets, 461–463
 - multiple recordsets, 460–461
 - stored procedures, 456–460

ActiveX Data Objects Multidimensional (ADO MD), 522–523

ActiveX settings category, 495–496

- ActiveXControl **property**, SmartTagAction **object**, 894
- AdaptiveMenu **property**, CommandBar **object**, 1001
- AdaptiveMenus **property**, CommandBars **collection**, 999

Add button, 15, 252, 405

- Add Charts **method**, Charts **collection**, 686
- Add fields **method**, 166
- Add **method**
 - AllowEditRanges collection, 641
 - CalculatedMembers collection, 678

- CommandBarControls collection, 1010
- CommandBars collection, 1000
- CustomXMLParts collection, 1020
- CustomXMLSchemaCollection object, 1023
- CustomXMLValidationErrors collection, 1024
- DocumentProperties collection, 1030
- FileDialogFilters collection, 1033
- FileTypes collection, 1035
- LinkedWindows Collection, 982
- NewFile object, 1046
- Permission object, 1050
- QueryTables collection, 846
- Scenarios collection, 871
- SearchFolders collection, 1053
- SeriesCollection object, 873
- ServerViewableItems collection, 878
- SharedWorkspaceFiles collection, 1056
- SharedWorkspaceFolders collection, 1057
- SharedWorkspaceMembers collection, 1059
- SharedWorkspaceTasks collection, 1060
- Sheets collection, 892
- SortFields collection, 897
- TabStops2 collection, 1069
- UserAccessList object, 922
- Validation object, 924
- VBcomponents collection, 988
- VBprojects collection, 992
- Watches collection, 927
- Workbooks collection, 935

- AddCallout **method**, Shapes **collection**, 880
- AddChart **method**, 177, 184, 187, 880
- AddCollectionCollection **method**, CustomXMLSchema **collection**, 1023
- AddComment **method**, Range **object**, 858
- AddConnector **method**, Shapes **collection**, 880
- AddCurve **method**, Shapes **collection**, 880
- AddCustom **method**, VBcomponents **collection**, 988
- AddCustomList **method**, Application **object**, 656
- AddFormControl **method**, Shapes **collection**, 880
- AddFromFile **method**
 - CodeModule object, 976
 - Reference collection, 985
 - Workbook.Connections collection, 953
- AddFromGuid **method**, Reference **collection**, 985
- AddFromString **method**, 597
- AddFromString **method**, CodeModule **object**, 976
- add-in **class**, 388, 396, 401, 407
- Add-In **object**, 639
- AddIn **object**, 640, 973–974
- AddIndent **property**
 - CellFormat object, 682
 - Range object, 852
 - Style object, 901

AddInInstall **event**, **381, 951**

Add-ins

- closing, 375–376
- code changes, 376
- creating, 374–375
- hiding code, 374
- installing, 379–381
- interface changes, 377–379
- removing from Add-ins list, 381–382
- saving changes, 377

Add-ins category, **494, 498**

AddIns **collection**, **639, 640**

Add-ins dialog box, **295–297, 373, 379, 381, 382, 385, 386, 395, 396, 398, 405, 575, 588, 598**

AddIns **property**, Application **object**, **644**

Add-ins tab, **319, 335**

AddInUninstall **event**, **381, 951**

Addin.xlam **file**, **379**

AddItem **method**, **150, 216, 218, 1008**

AdditionalXml **property**, SignatureSetup **object**, **1066**

AddLabel **method**, Shapes **collection**, **880**

AddLine **method**, Shapes **collection**, **880**

AddMenu **procedure**, **579**

AddNamespace **method**, CustomXMLPrefixMapping **collection**, **1022**

AddNode **method**, CustomXMLPart **object**, **1021**

AddNonVisibleSignature **method**, SignatureSet **collection**, **1062**

AddOLEObject **method**, Shapes **collection**, **880**

AddPeriods **method**, TextRange2 **object**, **1071**

AddPicture **method**, Shapes **collection**, **880**

AddPolyline **method**, Shapes **collection**, **881**

AddReplacement **method**, AutoCorrect **object**, **666**

Address **property**, **111, 128, 760, 761, 852**

AddressLocal **property**, Range **object**, **852**

AddShape **method**, Shapes **collection**, **881**

AddSignatureLine **method**, SignatureSet **collection**, **1062**

AddTextbox **method**, Shapes **collection**, **881**

AddTextEffect **method**, Shapes **collection**, **881**

AddToFavorites **method**, Workbook **object**, **944**

AddToSearchFolders **method**, ScopeFolder **object**, **1053**

AddToTable **parameter**, **166, 832**

AdjustColumnWidth **property**, QueryTable **object**, **847**

Adjustments **object**, **640–641**

Adjustments **property**

- Shape **object**, **881**

- ShapeRange **object**, **887**

ADO (ActiveX Data Objects)

Command **object**

- collections, 447
- methods, 446–447
- properties, 446

Connection **object**, 437–441

- collections, 441–443
- events, 440–441
- methods, 439–440
- properties, 437–438

Recordset **object**, 441–445

- collections, 445
- events, 445
- methods, 443–445
- properties, 441–443

SELECT **Statement**, 432–434

Structured Query Language (SQL) **overview**, 431–435

- DELETE **statement**, 435

- INSERT **statement**, 434

- UPDATE **statement**, 434–435

using in Excel applications, 447–448

using with Access, 448–454

- connecting to Access, 448–449

- inserting, updating, and deleting records with plain text SQL, 452–454

- retrieving data using plain text queries, 449–450

- retrieving data using stored queries, 451–452

using with non-standard data sources, 463–468

- inserting and updating records in workbooks, 466–467

- querying text files, 467–468

- querying workbooks, 464–466

using with SQL Server, 454–463

- connecting to SQL Server, 455–456

- disconnected recordsets, 461–463

- multiple recordsets, 460–461

- stored procedures, 456–460

ADO MD (ActiveX Data Objects Multidimensional), **522–523**

ADO **object library**, **419, 448**

ADO **object model**, **436**

ADO Recordset **object**, **179**

AdParamInput **parameter**, **446**

AdParamInputOutput **parameter**, **446**

AdParamOutput **parameter**, **446**

AdParamReturn **value parameter**, **446**

AdStateConnecting **object**, **438**

Advanced button, **128, 157, 555**

Advanced Filter, **128, 141, 142, 156–160, 559**

AdvancedFilter **method**, **158, 559–560, 858**

ADVAPI32.DLL **file**, **602**

After **parameter**, **686**

AfterCalculate **event**, Application **object**, **661**

AfterRefresh **event**, **528, 851**

AfterXMLExport event, Workbook object

- AfterXMLExport **event**, Workbook **object**, **951**
- AfterXMLImport **event**, Workbook **object**, **951**
- AlertBeforeOverwriting **property**, Application **object**, **644**
- AlertStyle **property**, Validation **object**, **923**
- Align **method**, ShapeRange **object**, **889**
- Alignment **property**
 - ParagraphFormat2 **object**, **1048**
 - TextEffectFormat **object**, **907**
 - TickLabels **object**, **914**
- Allcaps **property**, Font2 **object**, **1035**
- AllowComments **property**, SignatureSetup **object**, **1066**
- AllowEdit **property**
 - Range **object**, **852**
 - UserAccess **collection**, **921**
- AllowEditRange **object**, **641**, **642**, **921**, **922**
- AllowEditRanges **collection**, **641**
- AllowMultiSelect **property**, **237**, **1032**
- AllowPNG **property**, WebOptions **object**, **928**
- AlternativeText **property**
 - Shape **object**, **881**
 - ShapeRange **object**, **887**
- AltStartupPath **property**, Application **object**, **644**
- AlwaysUseClearType **property**, Application **object**, **644**
- Analysis Services**, **508**, **509**, **512**, **514–518**, **520**, **522**, **523**
- AND operator**, **157**
- Angle **property**, CalloutFormat **object**, **680**
- AnswerWizard **property**, Application **object**, **644**
- API, Windows **object**. *See* Windows API
- API **functions**, **595**, **602**, **605–607**, **609**, **611**, **615**, **619**, **620**, **634**
- Appearance **parameter**, **691**, **706**, **708**, **794**, **796**
- AppendChildNodes **method**, CustomXMLNode **object**, **1018**
- AppendChildSubtree **method**, CustomXMLNode **object**, **1018**
- AppendOnImport **property**, XmlMap **object**, **968**
- AppExcel **object**, **643**
- Application add-ins**, **494**
- Application **events**, **trapping**, **363**, **365**
- Application **method**, **63**
- Application **object**
 - active properties, **64–65**
 - Caller **property**, **74–75**
 - display alerts, **65**
 - Evaluate **method**, **66–68**
 - events, **661–663**
 - example, **664**
 - globals, **63–64**
 - InputBox **function**, **68–69**
 - methods, **656–661**
 - OnKey **method**, **72–73**
 - OnTime **method**, **71–72**
 - properties, **63**, **64**, **643–656**
 - screen updating, **66**
 - SendKeys **method**, **70–71**
 - StatusBar **property**, **70**
 - worksheet functions, **73–74**
- Application **property**, **635**, **995**, **996**
- Application window**, **373**
- Application.Caller**, **75**, **133**, **215**, **333**
- Applications Extensibility library**, **571**
- Application.StatusBar**, **70**, **288**, **323**, **324**, **326**, **329**, **339**, **340**
- Application.VBE Excel **property**, **972**
- AppliesTo **property**
 - AboveAverage **object**, **637**
 - Top10 **object**, **915**
 - UniqueValues **object**, **919**
- Apply **method**
 - Shape **object**, **884**
 - ShapeRange **object**, **890**
 - Sort **object**, **143**, **897**
- ApplyChartTemplate **method**, Chart **object**, **690**
- ApplyDataLabels **method**
 - Chart **object**, **690**
 - Series **object**, **875**
- ApplyFilter **method**, AutoFilter **object**, **668**
- ApplyLayout **method**, **184**, **690**
- ApplyNames **method**, Range **object**, **858**
- ApplyOutlineStyles **method**, Range **object**, **858**
- ApplyPictToEnd **property**, Series **object**, **873**
- ApplyPictToSides **property**, Series **object**, **873**
- ApplyPolicy **method**, Permission **object**, **1050**
- ApplyTheme **method**, Workbook **object**, **944**
- Appointment **item object**, **413**
- ArabicModes **property**, SpellingOptions **collection**, **900**
- ARange **object**, **93**
- ArbitraryXMLSupportAvailable **property**, Application **object**, **644**
- Areas**, **8**, **113–115**, **247**, **313**, **501**, **537**, **652**, **664**, **665**, **692**, **754**, **827**, **852**, **1054**
- Areas collection**, **113–114**, **664–665**
- Areas property**, Range **object**, **852**
- Arg1**, **202**, **203**, **366**, **659–661**, **692**, **694**, **695**, **738**, **864**
- Arg2**, **202–204**, **366**, **659–661**, **692**, **694**, **695**, **738**, **864**
- ArRange **method**, Windows **object**, **930**
- Array **function**, **56**, **87**, **109**, **167**, **216**, **279**
- arrays**, **55–58**
 - converting charts to use, **193**, **194**
 - defining chart series with, **190–193**
 - dynamic, **58**, **120**, **279**
 - multi-dimensional arrays, **57**
 - storing, **130–131**
- Asian language**, **143**

- AskToUpdateLinks **property**, Application **object**, 644
- ASP (Active Server pages)**, 532
- AssignedTo **property**
- SharedWorkspaceTask **object**, 1061
 - WorkflowTask **object**, 1077
- Assistance **property**, Application **object**, 644
- Assistant **property**, Application **object**, 644
- attributes, XML**, 242–243
- Attributes **property**, CustomXMLNode **object**, 1017
- Auditing add-in**, 295, 313
- Auditing group**, 295–297, 301
- Auditing tab**, 296, 304
- Auditing.xlam **file**, 296
- Auditing.xlam **workbook**, 303
- Authenticate **method**, EncryptionProvider **object**, 1031
- Auto page **parameter**, 836
- AutoAttach **property**, CalloutFormat **object**, 680
- AutoComplete **feature**, 648
- AutoComplete **method**, Range **object**, 858
- AutoCorrect **features**, 665
- AutoCorrect **object**, 665–667
- AutoCorrect **property**, Application **object**, 644
- AutoExpand **property**, AutoCorrect **object**, 665
- AutoFill **method**, Range **object**, 858
- AutoFillFormulasInLists **property**, AutoCorrect **object**, 665
- AutoFilter dialog box**, 148
- AutoFilter drop-downs**, 338
- AutoFilter **feature**
- adding combo boxes, 149–153
 - date format problems, 151–152
 - getting exact date, 152–153
 - copying visible rows, 153–154
 - Date Custom Filter, 148–149
 - Filter **object**, 148
 - finding visible rows, 154–156
- AutoFilter **method**, 147, 148, 151, 152, 559, 667, 859
- AutoFilter **object**, 143, 147–148, 667–669, 746
- AutoFilter operation**, 338
- AutoFilter **property**, Worksheet **object**, 955
- AutoFilterDataGrouping **property**, Windows **object**, 930
- AutoFilterMode **property**, Worksheet **object**, 955
- AutoFit **method**, Range **object**, 859
- AutoFormatAsYouTypeReplaceHyperlinks **property**, Application **object**, 644
- AutoLength **property**, CalloutFormat **object**, 680
- AutoMargins **property**, TextFrame **object**, 908
- AutomaticLength **method**, CalloutFormat **object**, 681
- automation**, 161, 253, 265, 383–391, 393–399, 401, 403, 405, 407, 409, 411, 412, 415–419, 611, 643, 644, 792
- Automation Add-Ins**. *See also* COM Add-in Designer
- IDExtensibility2 interface, 388–394
 - registering with Excel, 385–386
 - in Registry, 385–386
 - through Excel user interface, 385
 - using VBA, 385
 - using, 386–387
 - in VBA, 387
 - in worksheet, 386–387
- Automation add-ins dialog box**, 385, 395
- Automation button**, 385
- AutomationSecurityAutomationSecurity **property**, Application **object**, 644
- AutoNumber field**, 452
- AutoOutline **method**, Range **object**, 859
- AutoPercentEntry **property**, Application **object**, 644
- AutoRecover **object**, 669–670
- AutoRecover **option**, 645, 670
- AutoRecover **property**, Application **object**, 645
- AutoRecover setting**, 669
- AutoRepublish **property**, PublishObject **object**, 845
- AutorotateNumbers **property**, Font2 **object**, 1035
- AutoScaleFont **property**
- AxisTitle **object**, 674
 - ChartArea **object**, 697
 - TickLabels **object**, 914
- AutoScaling **property**, Chart **object**, 687
- AutoShapeType **property**
- Shape **object**, 881
 - ShapeRange **object**, 887
- AutoSize **property**, TextFrame **object**, 909
- AutoUpdateFrequency **property**, Workbook **object**, 937
- AutoUpdateSaveChanges **property**, Workbook **object**, 937
- Axes **method**, Chart **object**, 690
- Axis **object**, 670–674, 690, 738, 756, 913
- AxisGroup **property**
- Axis **object**, 671
 - Series **object**, 873
- AxisTitle **object**, 674–675
- AxisTitle **property**, Axis **object**, 671
- AZ button**, 142
- ## B
- BackColor **property**, 698, 830
- Background ODBC query, 786
- Background OLEDB query, 789
- background query, 474, 512, 936
- Background query option, 512
- BackgroundChecking **property**, 743
- BackgroundQuery **argument**, 475
- BackgroundQuery **property**, QueryTable **object**, 847

BackgroundStyle property

BackgroundStyle **property**

- Shape object, 881
- ShapeRange object, 887

BackWall **property**, Chart **object**, 687

Backward **parameter**, 917

Backward **property**, Trendline **object**, 918

Backward2 **property**, Trendline **object**, 918

BarShape **property**

- Chart object, 687
- Series object, 873

BaselineAlignment **property**, ParagraphFormat2 **object**, 1048

BaselineOffset **property**, Font2 **object**, 1035

BaseName **property**, CustomXMLNode **object**, 1017

BaseUnit **property**, Axis **object**, 671

BaseUnitIsAuto **property**, Axis **object**, 671

BasicCode **parameter**, 660

Before double click, 202

Before **parameter**, 686, 758

BeforeClose **event**, 16, 206, 207, 338, 351, 377, 951

BeforeDoubleClick **event**, 215, 366, 367, 694, 962

BeforePrint **event**, 207, 363, 951

BeforeRefresh **event**, 528, 851

BeforeRightClick **event**, 200, 217, 343, 695, 962

BeforeSave **event**, 16, 951

BeforeXMLExport **event**, Workbook **object**, 951

BeforeXMLImport **event**, Workbook **object**, 952

BeginGroup **property**

- CommandBarButton object, 1003
- CommandBarComboBox object, 1006
- CommandBarControl object, 1011
- CommandBarPopup object, 1013

BevelBottomDepth **property**, ThreeDFormat **object**, 911

BevelBottomInset **property**, ThreeDFormat **object**, 911

BevelBottomType **property**, ThreeDFormat **object**, 911

BevelTopDepth **property**, ThreeDFormat **object**, 911

BevelTopInset **property**, ThreeDFormat **object**, 911

BevelTopType **property**, ThreeDFormat **object**, 911

BlackWhiteMode **property**

- Shape object, 881
- ShapeRange object, 887

block If **structure**, 48–49

BlockPreview **property**, ServerPolicy **object**, 1050

BlogPictureProviderProperties **method**,
IBlogPictureExtensibility **object**, 1041

BlogProviderProperties **method**,
IBlogExtensibility **object**, 1039

Blur **property**, ShadowFormat **object**, 879

BOF **property**, Recordset **object**, 442

Bold **property**, Font2 **object**, 1035

bookmark, 426, 427

BOOL C **data type**, 604

Border **object**, 676–677

Border **property**

- Axis object, 671
- AxisTitle object, 674
- CalloutFormat object, 680
- SeriesLines object, 877
- Trendline object, 918

BorderAround **method**, Range **object**, 859

Borders **property**

- AboveAverage object, 637
- CellFormat object, 682
- Range object, 853
- Style object, 901
- TableStyleElement object, 906
- Top10 object, 915
- UniqueValues object, 919

BottomRight Cell **property**, Shape **object**, 881

BoundColumn **property**, 279

BoundHeight **property**, TextRange2 **object**, 1070

BoundLeft **property**, TextRange2 **object**, 1070

BoundTop **property**, TextRange2 **object**, 1070

BoundWidth **property**, TextRange2 **object**, 1070

<box ...> contents </box> container **control**, 301

box **element**, 298

boxStyle **attribute**, 301

Break key, 5

BreakLink **method**, Workbook **object**, 944

BreakSideBySide **method**, Windows **object**, 930

Browse button, 379, 406, 492

BubbleSizes **property**, Series **object**, 874

Build Freeform **method**, Shapes **collection**, 881

Build **property**, Application **object**, 645

BuildFileName **property**, VBProject **object**, 990

built-in command bars, 322–325

BuiltIn **property**

- CommandBar object, 1001
- CommandBarButton object, 1003
- CommandBarComboBox object, 1006
- CommandBarControl object, 1011
- CommandBarPopup object, 1013
- CustomXMLPart object, 1021
- Reference object, 984
- Style object, 901
- TableStyle object, 905

built-in shortcut menus, 338

BuiltInDocumentProperties **property**, Workbook **object**, 938

BuiltInFace **property**, CommandBarButton **object**, 1003

Bullet **property**, ParagraphFormat2 **object**, 1048

BulletFormat2 **object**, 996–997

bUseDATE **function**, 542

BUseDATE **function**, 543

<button .../> **control type**, 299

Button **parameter**, 695

<buttonGroup ...> contents </buttonGroup>
 container **control**, 301
 ButtonName **property**, FileDialog **object**, 1032
 bWinToDate **function**, 566–567
 bWinToNum **function**, 565–566
 ByRef **parameter**, 303
 BYTE * C **data type**, 604
 BYTE C **data type**, 604

C

C control **events**, 368
 C controlEvents **class module**, 369
 CalcFor **property**
 AboveAverage **object**, 637
 Top10 **object**, 915
 Calculate **event**
 Chart **object**, 695
 Worksheet **object**, 962
 Calculate **method**
 Application **object**, 656
 Range **object**, 859
 Worksheet **object**, 958
 CalculateBeforeSave **property**, Application **object**, 645
 CalculatedFields **collection**, 170–171, 677, 812
 CalculatedItems **collection**, 176, 678, 822–823
 CalculatedMember **object**, 678–679
 CalculatedMembers **collection**, 678, 679–680
 CalculateFull **method**, Application **object**, 656
 CalculateFullRebuild **method**, Application **object**, 657
 CalculateUnitAsyncQueries Done **method**, Application **object**, 657
 Calculation **property**, 645
 CalculationDone **event**, Application **object**, 661
 CalculationInterruptKey **property**, Application **object**, 645
 CalculationState **property**, Application **object**, 645
 CalculationVersion **property**
 Application **object**, 645
 Workbooks **object**, 938
 Call **statement**, 36, 38
 Caller **property**, 74–75, 134, 210, 645
 Callout **property**
 Shape **object**, 882
 ShapeRange **object**, 887
 CalloutFormat **object**, 680–682
 CanAddSignatureLine **property**, SignatureSet **collection**, 1062
 Cancel **parameter**, 200, 661, 663, 694, 695, 952
 Cancel **property**, 281
 CancelRefresh **method**, QueryTable **object**, 851
 CanCheckIn **method**, Workbook **object**, 944
 CanCheckOut **method**, Workbook **object**, 935
 CanPlaySounds **property**, Application **object**, 645
 CanRecordSounds **property**, Application **object**, 645
 CanSetup **property**, Signature **object**, 1062
 CapitalizeNamesOfDays **property**, AutoCorrect **object**, 666
 CApp **events**, 363–365
 Caps **property**, Font2 **object**, 1035
 Caption **property**
 Action **object**, 638
 Application **object**, 645
 AxisTitle **object**, 674
 and built-in command bars, 324
 Characters **object**, 684
 CommandBarButton **object**, 1003
 CommandBarComboBox **object**, 1006
 CommandBarControl **object**, 1011
 CommandBarPopup **object**, 1013
 and new menus, 332
 and pop-up menus, 341
 and user form creation, 275, 276
 and UserForm controls, 369
 Window **object**, 930, 992
 Catalog **object**, 455–458, 517–520, 522, 523
 CategoryNames **property**, Axis **object**, 671
 CategoryType **property**, Axis **object**, 671
 CBool **function**, 542
 CByte **function**, 541
 CChart **events**, 367
 CCur **function**, 541
 CDate **function**, 541
 CDb1 **function**, 541
 CDec **function**, 541
 Cell **command**, 340–342
 Cell **command bar**, 340–342
 Cell format **object**, 682–684
 Cell shortcut menu, 342, 344
 Cell1 **parameter**, 852
 Cell2 **parameter**, 852
 CellDragAndDrop **property**, Application **object**, 645
 CellFormat **object**, 682–684
 Cells **property**, 51, 97–101, 102, 123, 133, 175, 645, 853, 955
 CEmployee **class module**, 357
 CentimetersToPoints **method**, Application **object**, 657
 CertificateVerificationResults **property**, SignatureInfo **object**, 1063
 CFormChanger **class**, 624–625
 CFormResizer **class**, 627, 628–634
 CFreezeForm **class**, 618, 619
 Chain Pivot item **object**, 823

chain reaction

chain reaction, 201

Change **event**, 200, 201, 206, 209, 211, 237, 284, 286, 287, 576, 962, 1009

ChangeCase **method**, TextRange2 **object**, 1071

ChangeFileAccess **method**, Workbook **object**, 944

ChangeHistoryDuration **property**, Workbook **object**, 938

ChangeLink **method**, Workbook **object**, 944

ChangePassword **method**, 642, 922

ChangeScenario **methods**, Scenario **object**, 872

ChangingCells **property**, Scenario **object**, 871

Char C **data type**, 604

char FAR * C **data type**, 604

char huge * C **data type**, 604

char NEAR * C **data type**, 604

Character **property**, BulletFormat2 **object**, 996

Characters **method**, TextFrame **object**, 909

Characters **object**, 684–685

Characters **property**

AxisTitle **object**, 674

Range **object**, 853

TextRange2 **object**, 1070

chart **events**, 199, 202–204, 355, 365, 694, 695

chart labels, 181, 195–196, 690

Chart **object**

and embedded charts, 185–190

events, 694–695

example, 696

methods, 690–694, 707–708

properties, 687–690, 706–707

Chart **property**

Shape **object**, 882

ShapeRange **object**, 887

chart series with arrays, defining, 190

chart sheet, 85, 90, 95, 181, 184, 186, 196, 199, 422, 423, 692, 704, 710, 903

Chart title, 182, 186, 190, 193, 674, 696, 709, 710

Chart tools, 313

Chart type **property**, 184, 192

ChartArea **object**, 696–697, 838

ChartArea **property**, Chart **object**, 687

ChartColorFormat **object**, 698

ChartData **sub procedure**, 421, 422

ChartFillFormat **object**, 698–700

ChartFormat **object**, 700–701

ChartGroup **object**, 691, 701–704, 740, 758, 824, 839, 877, 920

ChartGroups **collection**, 701, 702, 704

ChartGroups **method**, Chart **object**, 691

Chartobject **collection**, 83, 177, 181, 187, 704, 705, 708

ChartObject **object**, 706–708

ChartObjects **collection**, 196, 704–706, 708

ChartObjects **method**

Chart **object**, 691

Worksheet **object**, 958

charts

chart labels, 195–196

chart sheets

adding using VBA code, 184–185

recorded macro, 184

controls on, 220–221

converting to use arrays, 193

defining chart series with arrays, 190–193

determining ranges used in, 194–195

editing data series in, 181, 187–190

embedded, 185–187

adding using VBA code, 186–187

using macro recorder, 186

Charts **collection**, 83, 181, 184, 196, 685, 686–687, 696, 704

Charts **property**

Application **object**, 645

Workbook **object**, 938

ChartStyle **property**, Chart **object**, 687

ChartTitle **method**, 710

ChartTitle **object**, 184–187, 708–710

ChartTitle **property**, Chart **object**, 687

ChartType **property**

Chart **object**, 687

Series **object**, 874

ChartView **object**, 710

ChartWizard **method**, Chart **object**, 691

ChDrive **statement**, 60

CheckAbort **method**, Application **object**, 657

<checkbox .../> **control type**, 299

CheckBox **control**, 212, 276

CheckBoxState **property**, SmartTagAction **object**, 894

CheckCompatibility **property**, Workbook **object**, 938

CheckIn **method**, Workbook **object**, 944

CheckOut **method**, Workbook **object**, 936

CheckSpelling **method**

Application **object**, 657

Chart **object**, 691

Range **object**, 859

Worksheet **object**, 958

Child **property**

Shape **object**, 882

ShapeRange **object**, 888

ChildNodes **property**, CustomXMLNode **object**, 1017

CInt **function**, 189, 541

CircularReference **property**, Worksheet **object**, 955

Class Module **collection**, 360–363

class modules

collection of userform controls, 368–370

creating collections, 359–363

- creating objects, 356–357
- embedded chart events, 365–367
- encapsulation, 363
- property procedures, 357–359
- referencing classes across projects, 370–371
- trapping Application events, 363–365
- wrapping API calls in, 611–616
- class_Initialize **event**, **576, 577**
- class_Initialize **procedure**, **579, 582**
- class_Terminate **event**, **577**
- classes**, **606–609**
- classReferences.xlsm, **370, 371**
- classUserForm **controls.xlsm**, **370, 371**
- Clear **method**
 - CellFormat object, 683
 - ChartArea object, 697
 - CommandBarComboBox object, 1009
 - FileDialogFilters collection, 1033
 - Range object, 859
 - SortFields collection, 897
 - TableStyleElement object, 906
 - TabStop2 object, 1069
 - XPath object, 970
- ClearArrows **method**, Worksheet **object**, **958**
- ClearCircles **method**, Worksheet **object**, **958**
- ClearComments **method**, Range **object**, **859**
- ClearContents **method**
 - ChartArea object, 697
 - Range object, 859
- ClearDefaultContext **method**, IAssistance **object**, **1038**
- ClearFormats **method**
 - ChartArea object, 697
 - Range object, 859
 - Series object, 876
 - Trendline object, 918
 - Walls object, 926
- ClearNotes **method**, Range **object**, **859**
- ClearOutline **method**, Range **object**, **859**
- ClearSettings **method**, XmlDataBinding **object**, **967**
- ClearToMatchStyle **method**, Chart **object**, **691**
- Click **event**, **153–155, 157, 209, 210, 212, 217, 218, 219, 277, 281, 286, 378, 398, 399, 400, 550, 576–578, 594, 597, 619, 980**
- Click **event procedure**, **157, 210, 212, 219**
- ClipboardFormats **property**, Application **object**, **646**
- CLng **function**, **541**
- CloneSession **method**, EncryptionProvider **object**, **1031**
- Close button**, **281–282, 596, 624**
- Close **method**
 - Connection object, 440
 - Recordset object, 444
 - Window object, 933, 993
 - Workbook object, 82, 83, 936, 944
- CloseMode **parameter**, **282**
- CLSID **property**, Add-In, **639**
- CMenuHandler **class**, **577, 580, 589, 594**
- code, hiding**, **363**
- code modules**, **VBE, 9–10**
- Code Window popup menu**, **580, 589**
- CodeModule **object**, **574, 974–978**
- CodeModule property**
 - CodePane object, 978
 - VBComponent object, 987
- CodeName **property**
 - Chart object, 687
 - and VBComponent, 573
 - Workbook object, 938
 - Worksheet object, 955
- CodePane **object**, **574, 978–980**
- CodePane **property**, CodeModule **object**, **975**
- CodePanes **property**, **VBE, 989**
- CodePaneView **property**, CodePane **object**, **978**
- collection **methods**, **636**
- Collection **object**, **359, 741**
- collection properties**, **635**
- collections**, **22–23**
- color index**, **51, 204, 749**
- Color **object**, **713, 728, 749**
- Color **property**
 - Border object, 677
 - Borders collection, 676
 - GlowFormat object, 1037
 - GradientStop object, 1038
 - Tab object, 903
- ColorFormat **object**, **710–711**
- ColorIndex **property**
 - Border object, 677
 - Borders collection, 676
 - Interior object, 26
 - Tab object, 903
- Colors **method**, ThemeColorsScheme **object**, **1073**
- Colors **property**, Workbook **object**, **938**
- ColorScale **object**, **711–713, 749**
- ColorScaleCriteria **collection**, **713**
- ColorScaleCriteria **object**, **711, 712**
- ColorStop **object**, **714**
- ColorStops **collection**, **714**
- column field**, **164, 818**
- Column **object**, **113**
- Column **property**
 - Range object, 853
 - TextFrame2 object, 909
- column width**, **24, 250, 279, 483, 683, 702, 853, 857, 902**
- ColumnCount **property**, **279**

ColumnDifferences method, Range object

- ColumnDifferences **method**, Range **object**, **859**
- ColumnLevels **parameter**, **797**
- columns **attribute**, **301**
- Columns **property**
 - Application object, 646
 - overview, 112–114
 - Range object, 853
 - Worksheet object, 955
- ColumnWidth **property**, **23, 24, 279, 853**
- COM Add-in Designer, 396–408**
 - creating custom task panes, 404–405
 - linking to Excel, 397–401
 - adding CommandBar controls, 398–401
 - overview, 397–398
 - responding to Excel's events, 398
 - linking to multiple Office applications, 407–408
 - showing VBA UserForms as task panes, 405–406
 - using COM Add-In from VBA, 401–403
 - adding Ribbon controls, 402–403
 - overview, 401–402
- COM Add-ins **collection**, **407, 997–999**
- COM Add-ins dialog box, 396, 398, 405**
- COMAddinObject, **997–999**
- COMAddIns **property**, Application **object**, **646**
- ComandBars.ExecuteMso **method**, **316**
- combo boxes, adding, 149–153**
 - date format problems, 151–152
 - getting exact date, 152–153
- `<comboBox ...> contents </comboBox>` **container control, 300**
- ComboBox **control**, **214, 315**
- ComboBox **object**, **150, 216, 218**
- COMDLG32.DLL **file**, **602**
- command bars**
 - built-in, 322–325
 - controls at all levels, 325–330
 - creating new menus, 330–333
 - creating toolbars, 335–338
 - deleting menus, 334
 - passing parameter values, 333–334
 - popup menus, 338–341
 - showing popup command bars, 342–353
 - toolbars, menu bars, and popups, 320–322
- command button, 13–16, 26, 153, 155, 157, 214, 233, 274, 282, 283, 321, 619**
- Command **object**
 - ActiveConnection property, 457
 - CommandText property, 446, 447, 457
 - methods, 446–447
 - parameters collection, 446, 457, 460
 - properties, 446, 459
- Command type, 446, 458, 473–478, 485, 786, 788, 808**
- Command type **property**, **474**
- CommandBar **controls, 398–401**
- CommandBar extensions, for RibbonX, 316**
- CommandBar **object**, **576, 999–1002**
- CommandBar **property**, CommandBarPopup **object**, **1013**
- CommandBarButton **events, 398, 576**
- CommandBarButton **object**, **1003–1006**
- CommandBarComboBox **object**, **1006–1009**
- CommandBarControl **object**, **1010–1013**
- CommandBarControls **collection, 1010–1013**
- CommandBarEvents **object, 980–981**
- CommandBarEvents **property**, Events **object, 982**
- CommandBarPopup **object, 1013–1015**
- CommandBars **collection, 319, 321, 326, 330, 333, 337, 341, 343, 347, 349, 995, 999–1002, 1007, 1014**
- CommandBars **object, 293, 316, 317, 348, 398**
- CommandBars **property**
 - Application object, 646
 - MsoEnvelope object, 1046
 - VBE, 989
 - Workbook object, 938
- CommandBars.Reset, **349**
- CommandButton, **275, 276, 596**
- CommandButton **control, 276**
- CommandLineSafe value, 396**
- Commands list, 556**
- CommandText **argument, 439, 440**
- CommandText operation, 439**
- CommandText **property, 446, 447, 454, 457, 474, 477, 482, 847**
- CommandText type, 440**
- CommandType **property**
 - QueryTable object, 847
 - Recordset object, 446
- CommandTypeEnum **value, 440**
- CommandUnderlines Underlines **property**, Application **object, 646**
- Comment **object, 714–716**
- Comment **property**
 - Range object, 853
 - Scenario object, 871
- comments, XML, 241**
- Comments box, 381, 588**
- Comments **collection, 714–716**
- Comments **property**
 - DocumentLibraryVersion object, 1027
 - Worksheet object, 955
- CompareSideBySide **method**, Windows **object, 930**
- Complex add-in, 390, 394**
- Complex **class, 401**
- Component tab, 389**
- CONCATENATE **function, 74**
- Condition **object, 750, 753**

- Conditions **collection**, 711, 750, 753, 763
- Conditionvalue **object**, 716, 728
- ConflictResolution **property**, Workbook **object**, 938
- Connect **class**, 402
- Connect **optionEnum value**, 439
- Connect **property**, 973, 998
- Connected **property**, SharedWorkspace **object**, 1054
- Connection **events**, 441
- Connection **file**, 470, 484, 490, 785, 788
- Connection **object**
 - collections, 441
 - events, 440–441
 - Execute method, 443, 446, 447
 - methods, 439–440
 - properties, 437–438
 - Recordset **object**, 462
- Connection **parameter**, 836
- Connection **property**, 482, 790, 809, 847
- Connection **string argument**, 439
- Connection **string property**, 439
- Connections button**, 470, 471, 486
- Connections **collection**, 487–489, 490
- Connections dialog box**, 470, 484, 485
- Connections **object**, 716–717
- Connections **property**, Workbook **object**, 938
- ConnectionsDisabled **property**, Workbook **object**, 938
- ConnectionSite **parameter**, 718
- ConnectionSiteCount **property**
 - Shape **object**, 882
 - ShapeRange **object**, 888
- ConnectionString **property**, Connection **object**, 438
- ConnectionTimeout **property**, Connection **object**, 438
- Connector format **object**, 717–719
- Connector **property**
 - Shape **object**, 882
 - ShapeRange **object**, 888
- ConnectorFormat **object**, 717–719
- ConnectorFormat **property**
 - Shape **object**, 882
 - ShapeRange **object**, 888
- Consolidate **method**, Range **object**, 860
- ConsolidationFunction **property**, Worksheet **object**, 955
- ConsolidationOptions **property**, Worksheet **object**, 955
- ConsolidationSources **property**, Worksheet **object**, 955
- Const keyword**, 44
- constants**, 33, 44, 606–609
- ConstrainNumeric **property**, Application **object**, 646
- container **controls**, 298, 300
- Container **property**, Workbook **object**, 938
- Content button**, 489
- Content **property**, Action **object**, 638
- ContentControl **property**, CustomTaskPane **object**, 1016
- ContentTypeProperties **property**, Workbook **object**, 938
- ContentVerificationResults **property**, SignatureInfo **object**, 1063
- Context **property**
 - CommandBar **object**, 1001
 - IRibbonControl **object**, 1042
- ContourColor **property**, ThreeDFormat **object**, 911
- ContourWidth **property**, ThreeDFormat **object**, 911
- <control .../> **control type**, 299
- control **attributes**, 301
- control ID**, 302, 304, 310, 344, 346, 347
- control images**, 305, 316, 324, 354
- control menu**, 282
- control reference**, 304
- control Toolbox**, 12, 496, 579, 580
- control types**, 298, 299, 301, 305, 307, 308, 313, 318, 327
- ControlCharacters **property**, Application **object**, 646
- ControlFormat **object**, 719–721
- ControlFormat **property**, Shape **object**, 882
- ControlFormatType **property**, Shape **object**, 882
- controls**, 209–221
 - ActiveX controls, 210–214
 - CheckBox control, 212
 - dynamic, 216–219
 - Option Button controls, 212–214
 - Scrollbar control, 211
 - Spin Button control, 211–212
 - adding, 209–221, 347, 594, 986
 - on charts, 220–221
 - Forms controls, 214–216
- controls collection**, 321, 322, 330, 332, 337, 341, 369
- controls dialog box**, 12, 13
- controls group**, 12, 209, 495, 496
- Controls **property**
 - CommandBar **object**, 1001
 - CommandBarPopup **object**, 1014
- controlSource **property**, 276
- conversion **functions**, 542
 - Application.Evaluate function, 544–545
 - DateValueUS function, 544–545
 - IsDateUS function, 544
 - overview, 544
- CBool function, 542
- CByte function, 541
- CCur function, 541
- CDate function, 541
- CDbl function, 541
- CDec function, 541
- CInt function, 541

conversion functions (continued)

conversion functions (continued)

- CLng function, 541
- CSng function, 541
- CStr function, 541
- DateValue function, 541
- Format function, 542
- FormatCurrency function, 542
- FormatDateTime function, 542
- FormatNumber function, 542
- FormatPercent function, 542
- IsDate function, 541
- IsNumeric function, 541
- sNumToUS function, 542–543
- Str function, 542
- Val function, 543–544
- ConvertFormula **function**, Application **object**, 560, 657
- ConvertFormulaLocale **function**, 557, 558, 559
- Coordinate **property**, Action **object**, 638
- Copy button**, 297
- Copy command**, 321
- Copy menu**, 298
- Copy method**
 - ChartArea object, 697
 - Charts collection, 686
 - CommandBarButton object, 1005
 - CommandBarComboBox object, 1009
 - CommandBarControl object, 1012
 - CommandBarPopup object, 1015
 - Range object, 25, 860
 - Series object, 876
 - Shape object, 884
 - Sheets collection, 892
 - TextRange2 object, 1071
 - Worksheet object, 85–87, 355, 959
 - Worksheets collection, 954
- Copy statement**, 266
- Copy Variant method**, 705, 794
- CopyFace method**, CommandBarButton **object**, 1005
- CopyFromRecordset method**, 420, 445, 450, 517, 860
- CopyObjectsWithCells property**, Application **object**, 646
- CopyPicture method**
 - Chart object, 691
 - Range object, 860
 - Shape object, 884
- CorrectCapsLock property**, AutoCorrect **object**, 666
- CorrectSentenceCap property**, AutoCorrect **object**, 666
- Count method**, ShapeNodes **collection**, 886
- Count parameter**, 686
- Count property**
 - Addins collection, 635
 - Adjustments object, 640
 - AllowEditRanges collection, 641
 - Areas collection, 664
 - CalculatedMembers collection, 678
 - Characters object, 684
 - Employees collection, 360
 - overview, 995
 - Pages collection, 797
 - Range object, 100, 853
 - Rows object, 112
 - SeriesCollection object, 221
 - Worksheets collection, 53, 375
- COUNTA function**, 130
- CountLarge property**, 100, 853
- CountOfDeclarationLines property**, CodeModule **object**, 975
- CountOfVisibleLines property**, CodePane **object**, 978
- CountOfLines property**, CodeModule **object**, 975
- CREATE GLOBAL CUBE statement**, 522
- Create item method**, 413, 423
- Create method**, 165, 807
- CreateBackup property**, Workbook **object**, 938
- CreateCTP method**, ICTPFactory **object**, 1041
- CreateCubeFile method**, 521–522
- CreatedBy property**
 - SharedWorkspaceFile object, 1056
 - SharedWorkspaceLink object, 1058
 - SharedWorkspaceTask object, 1061
 - WorkflowTask object, 1077
- CreatedDate property**
 - SharedWorkspaceFile object, 1056
 - SharedWorkspaceLink object, 1058
 - SharedWorkspaceTask object, 1061
 - WorkflowTask object, 1077
- CreateEventProc method**, CodeModule **object**, 976
- CreateNames method**, Range **object**, 860
- CreateNew method**, SharedWorkspace **object**, 1055
- CreateParameter method**, Recordset **object**, 446–447
- CreatePictureAccount method**, IBlogPicture **Extensibility object**, 1041
- CreatePivotTable method**, 164–166, 170
- CreateSummary method**, Scenario **objects**, 871
- CreateToolbar**, 336
- CreateToolWindow method**, Window **collection**, 993
- Creator property**, 635, 636, 995, 996
- Criteria range**, 156, 559
- cross tabulation tables**, 161
- Crosses property**, Axis **object**, 671
- CrossesAt property**, Axis **object**, 671
- CSng function**, 541
- CStr function**, 541, 542
- C-style declarations**, interpreting, 603–606
- CSysInfo class**, 619
- CTemp file**, 611
- CTP functionality**, 405

CTP-creation features, 409

CTPFactoryAvailable **method**, ICustomTaskPaneConsumer **object**, 1042

Ctrl **object**, 577

Cube field **methods**, 724

Cube file **method**, 521

CubeField **object**, 721–724, 917

CubeFields **collection**, 721–724

cubes, 508, 512, 514, 516, 518–521, 523

Currency (sealed integer) data type, 43

Currency **object**, 901

CurrentArray **property**, Range **object**, 853

CurrentRegion **property**, 108–110, 123, 422, 853

Cursor type **property**, 463

CursorLocation **property**, Recordset **object**, 442

CursorMovement **property**, Application **object**, 646

CursorPointer **property**, Application **object**, 646

Custom Filter, 148, 755

Custom menu, 334, 343–345, 351, 352, 575, 577

Custom option, 542

Custom **parameter**, 657

Custom **properties collection**, 724

custom task panes, 383, 403–405, 409, 1016

CustomDocumentProperties **property**, Workbook **object**, 938

CustomDrop **method**, CalloutFormat **object**, 681

Customers table, 432–435, 456, 473, 474, 484, 485

CustomersTable.odc **file**, 485

customization, 158, 294, 295, 297, 303, 309, 313, 314, 382, 383, 405, 409, 999, 1002, 1043

Customization button, 158

customized **objects**, 355

CustomLength **method**, CalloutFormat **object**, 681

CustomListCount **property**, Application **object**, 646

CustomOrder **property**, SortFields **object**, 897

CustomProperties **collection**, 724–726

CustomProperties **property**, Worksheet **object**, 956

CustomProperty **object**, 724–726

CustomTaskPane **object**, 1016

customUI, 294–296, 306–313, 316, 403

CustomUI **method**, 317

CustomView **object**, 726–727

CustomViews **collection**, 726–727

CustomViews **property**, Workbook **object**, 938

CustomXMLNode **object**, 1017–1020

CustomXMLNodes **collection**, 1017–1020

CustomXMLPart **object**, 1020–1022

CustomXMLParts **collection**, 317, 1020–1022

CustomXMLParts **property**, Workbook **object**, 938

CustomXMLPrefixMapping **object**, 1022–1023

CustomXMLPrefixMappings **collection**, 1022–1023

CustomXMLSchema **object**, 1023–1024

CustomXMLSchemaCollection **object**, 1023–1024

CustomXMLValidationError **object**, 1024–1025

CustomXMLValidationErrors CollectionObject, 1024–1025

Cut button, 321**Cut method**

Range **object**, 860

Shape **object**, 884

TextRange2 **object**, 1071

CutCopyMode **property**, Application **object**, 646

D

Data Connection Wizard, 509

Data Connections section, 490

Data field, 169, 170

Data field **object**, 144

Data Form **feature**, 158–159, 160

Data group, 182, 469, 470

Data Labels button, 195

data lists, 141–160

Advanced Filter, 156–158

AutoFilter **feature**, 146–156

adding combo boxes, 149–153

AutoFilter **object**, 147–148

copying visible rows, 153–154

Date Custom Filter, 148–149

Filter **object**, 148

finding visible rows, 154–156

creating tables, 144–145

Data Form, 158–159

sorting ranges, 142–144

sorting tables, 145–146

structuring data, 141–142

Data **object**, 557

data strings, reading, 229

Data tab, 128, 142, 146, 157, 469, 484, 508, 716

Databar **methods**, 729

Databar **object**, 716, 727–729

DataBindingBinding **property**, XmlMap **object**, 968

DataEntryMode **property**, Application **object**, 646

DataRowLabel **object**, 729–734

DataRowLabel **property**, 174, 918

DataRowLabels **collection**, 729–734

DataRowLabels **method**, Series **object**, 876

DataRowLists, 344, 345, 349, 352

DataRowRange **property**, 174, 175

DataSeries **method**, Range **object**, 860

DataTable **object**, 734–735

DataTable **property**, Chart **object**, 687

date

format problems, 151

getting exact, 152

date custom filter, 148–149

Date data type

- Date **data type**, 43
- Date Filters**, 148, 818
- DATE function**, 73, 74, 87, 542, 543, 544, 547
- date literals**, 540, 541
- date order**, 538, 542, 551, 552, 555, 556
- Date Order setting**, 538
- Date Picker button**, 403
- Date **property**, **PolicyItem**, 1051
- date separator**, 538, 551, 552
- Date1904 **property**, **Workbook object**, 939
- DateSerial **function**, 73, 87, 152, 153, 413
- DateValue **function**, 87, 541, 544
- DateValueUS **function**, 544, 545
- DDEAppReturnCode **property**, **Application object**, 646
- DDEExecute **method**, **Application object**, 657
- DDEInitiate **method**, **Application object**, 657
- DDEPoke **method**, **Application object**, 657
- DDERequest **method**, **Application object**, 657
- DDETerminate **method**, **Application object**, 658
- Deactivate **event**, 91
 - Chart object, 695
 - Workbook object, 952
 - Worksheet object, 963
- Debug toolbar**, 29, 579
- Decimal **data type**, 43
- DecimalSeparator**, 530, 538, 552, 553, 555, 556, 558, 646, 867
- (Declarations) section**, 39–42, 44, 55–57, 109, 118, 131, 216, 283, 358, 365, 593, 977
- Declare **function**, 584
- Declare **statement**, 602
- DecryptStream **method**, **EncryptionProvider object**, 1031
- Default **property**, **OK button**, 281
- DefaultFilePath **property**, **Application object**, 647
- DefaultPivotTableStyle **property**, **Workbook object**, 939
- DefaultSaveFormat **property**, **Application object**, 647
- DefaultSheetDirection **property**, **Application object**, 647
- DefaultSpacing **property**, **TabStops2 collection**, 1068
- DefaultTableStyle **property**, **Workbook object**, 939
- DefaultWebOptions **object**, 735–737
- DefaultWebOptions **property**, **Application object**, 647
- Definition tab**, 486
- Delete button**, 283
- Delete **method**
 - AboveAverage object, 638
 - AllowEditRange object, 642
 - Axis object, 673
 - AxisTitle object, 675
 - CalculatedMember object, 679
 - Characters object, 685
 - Chart object, 691
 - Charts collection, 686
 - CommandBar object, 1002
 - CommandBarButton object, 1005
 - CommandBarComboBox object, 1009
 - CommandBarControl object, 1012
 - CommandBarPopup object, 1015
 - CustomTaskPane object, 1016
 - CustomXMLNode object, 1018
 - CustomXMLPart object, 1021
 - CustomXMLSchema object, 1024
 - CustomXMLValidationError, 1025
 - deleting menus, 334
 - deleting XML map, 255
 - DocumentLibraryVersion object, 1027
 - DocumentProperty object, 1030
 - FileDialogFilters collection, 1034
 - GradientStops collection, 1038
 - HPageBreak object, 759
 - Hyperlink object, 760
 - PublishObject object, 845
 - PublishObjects collection, 844
 - QueryTable object, 851
 - Range object, 861
 - RecentFile object, 868
 - Series object, 197, 876
 - SeriesLines object, 877
 - ServerViewableItems collection, 878
 - Shape object, 884
 - ShapeNodes Collection, 886
 - ShapeRange object, 890
 - SharedWorkspace object, 1055
 - SharedWorkspaceFile object, 1056
 - SharedWorkspaceFolder, 1057
 - SharedWorkspaceLink object, 1059
 - SharedWorkspaceMember, 1060
 - SharedWorkspaceTask object, 1061
 - Sheets collection, 892
 - Signature object, 1063
 - SmartTag object, 894
 - SortFields collection, 898
 - SoundNote object, 898
 - TableStyle object, 905
 - TextRange2 object, 1071
 - TickLabels object, 914
 - Top10 object, 916
 - Trendline object, 918
 - UniqueValues object, 920
 - UpBars object, 921
 - UserAccess Collection, 922
 - Validation object, 924
 - VPageBreak object, 925
 - Watch object, 927

- Watches Collection, 927
- Worksheet object, 959
- Worksheets collection, 954
- XmlMap object, 968
- DELETE statement, 435, 445, 452, 454**
- DeleteAll method**
 - ServerViewableItems collection, 878
 - UserAccessList object, 922
- DeleteCustomList method, Application object, 658**
- DeleteLines method, CodeModule object, 977**
- DeleteNumberFormat method, Workbook object, 944**
- DeleteReplacement method, AutoCorrect object, 666**
- DeleteText method, ThreeDFormat object, 911**
- deleting**
 - menus, 334
 - records with plain text SQL in Access, 452–454
 - rows, 121–123
- delimiters, 230, 233, 483, 849, 850, 867, 937**
- Delivery property, RoutingSlip object, 870**
- Dependents property, Range object, 853**
- Depth property**
 - ThreeDFormat object, 911
 - TickLabels object, 914
- DepthPercept property, Chart object, 687**
- DESC specifier, 434**
- description attribute, 302**
- Description property**
 - Add-in object, 973
 - COMAddin object, 998
 - Err object, 59
 - FileDialogFilter, 1034
 - IDocumentInspector object, 1026
 - PolicyItem object, 1051
 - Reference object, 984
 - ServerPolicy object, 1050
 - SharedWorkspaceLink object, 1058
 - SharedWorkspaceTask object, 1061
 - VBproject object, 990
 - WorkflowTask object, 1077
 - WorkflowTemplate object, 1078
- Description value, 396**
- DescriptionText property**
 - CommandBarButton object, 1003
 - CommandBarComboBox object, 1006
 - CommandBarControl object, 1011
 - CommandBarPopup object, 1014
- Deselect method, Chart object, 692**
- Design Mode button, 13, 14**
- Design tab, 144, 182, 186, 313**
- Designer class, 397, 407**
- Designer Connect class, 402**
- Designer object, 574–575, 596**
- Designer property, VBcomponent, 987**
- DesignerID property, VBcomponent, 987**
- DesignerWindow property, VBcomponent, 987**
- Destination parameter, 25**
- Destination property, QueryTable object, 847**
- Details property, Signature object, 1062**
- Developer tab, 3, 5–9, 12, 13, 71, 209, 233, 247, 250, 251, 305, 495, 496, 596, 653**
- Diagram property, Shape object, 882**
- DiagramNode property, Shape object, 882**
- Dialog object, 223, 233, 235, 737–738**
- dialog sheets, 12, 74, 209, 549, 608, 686**
- Dialog types, 235**
- DialogBox method, Range object, 861**
- <dialogBoxLauncher ../> control type, 300**
- DialogFilters collection, 235**
- Dialogs collection, 737–738**
- Dialogs property, Application object, 647**
- DialogSelected items collection, 235, 237**
- DialogSelected items object, 235**
- DialogType property, FileDialog object, 1032**
- dictator applications, 312, 313**
- dictator applications, RibbonX in, 312**
- DictLang property, SpellingOptions collection, 900**
- digital certificate, 492, 1062–1064**
- digital signatures, 492, 942, 1061, 1062**
- Dim statement, 38, 39, 42, 43**
- Dim Zip file, 267**
- Dir function, 58, 81, 82**
- direct reference to ranges, 20–21**
- DirectDependents property, Range object, 853**
- Direction parameter, 759**
- Direction property, Speech object, 898**
- DirectPrecedents property, Range object, 853**
- Dirty method, Range object, 861**
- DisableAskAQuestionDropdown property, CommandBars collection, 999**
- DisableCustomize property, CommandBars collection, 999**
- Disconnect method, SharedWorkspace object, 1055**
- disconnected recordsets, 461–463**
- display alerts, 65**
- DisplayAlerts property, 60, 65, 254, 256, 647**
- DisplayAutoCorrectOptions property, AutoCorrect object, 666**
- DisplayBlanksAs property, Chart object, 688**
- DisplayClipboardWindow property, Application object, 647**
- DisplayCommentIndicatorDisplayMode property, Application object, 647**
- DisplayDocumentActionTaskPane property, Application object, 647**

DisplayDocumentInformationPanel property, Application object

- DisplayDocumentInformationPanel **property**, Application **object**, 647
- DisplayDrawingObjects **property**, Workbook **object**, 939
- DisplayEquation **property**, Trendline **object**, 918
- DisplayExcel4 Menus **property**, Application **object**, 647
- DisplayFonts **property**, CommandBars **collection**, 999
- DisplayFormulaAutoComplete **property**, Application **object**, 647
- DisplayFormulaBar **property**, Application **object**, 647
- DisplayFormulas **property**
 - Window object, 930
 - WorksheetView object, 967
- DisplayFullScreen **property**, Application **object**, 647
- DisplayFunctionToolTips **property**, Application **object**, 647
- DisplayGridlines **property**
 - Window object, 930
 - WorksheetView object, 967
- DisplayHeadings **property**
 - Window object, 931
 - WorksheetView object, 967
- DisplayHorizontalScrollBar **property**, Windows **object**, 931
- DisplayInkComments **property**, Workbook **object**, 939
- DisplayInsertOptions **property**, Application **object**, 648
- DisplayKeysInToolTips **property**, CommandBars **collection**, 999
- DisplayNoteIndicator **property**, Application **object**, 648
- DisplayOutline **property**, Worksheet **object**, 967
- DisplayPageBreaks **property**, Worksheet **object**, 956
- DisplayPasteOptions **property**, Application **object**, 648
- DisplayRecentFiles **property**, Application **object**, 648
- DisplayRightToLeft **property**
 - Window object, 931
 - Worksheet object, 956
- DisplayRSquared **property**, Trendline **object**, 918
- DisplayRuler **property**, Windows **object**, 931
- DisplayScrollBars **property**, Application **object**, 648
- DisplaySmartTags **property**, SmartTagOptions **collection**, 895
- DisplayStatusBar **property**, Application **object**, 648
- DisplayToolTips **property**, CommandBars **collection**, 1000
- DisplayUnit **or** DisplayUnitCustom **property**, 672
- DisplayUnit **property**, 671
- DisplayUnitCustom **property**, Axis **object**, 671
- DisplayUnitLabel **object**, 738–739
- DisplayUnitLabel **property**, Axis **object**, 671
- DisplayVerticalScrollBar **property**, Windows **object**, 931
- DisplayWhitespace **property**, Windows **object**, 931
- DisplayWorkbookTabs **property**, Windows **object**, 931
- DisplayXMLSourcePane **method**, Application **object**, 658
- DisplayZeros **property**
 - Window object, 931
 - WorksheetView object, 967
- DISTINCT** keyword, 433
- Distribute **method**, ShapeRange **object**, 890
- DivID **property**, PublishObject **object**, 845
- Do **events**, 289, 531, 536, 593, 619
- Do **statement**, 51, 52
- Do...Loop, 50, 57
- Docked Window, 580
- DockPosition **property**, CustomTaskPane **object**, 1016
- DockPositionRestrict **property**, CustomTaskPane **object**, 1016
- DockPositionStateChange **event**, CustomTaskPane **object**, 1016
- document inspection automation, 503–506
 - DocumentInspectors collection, 505–506
 - RemoveDocumentInformation method, 503–505
- Document Inspector, 501, 503, 505, 939, 1025, 1042
- Document Library Name **property**, Workflow **object**, 1078
- Document LibraryURL **property**, WorkflowTemplate **object**, 1078
- Document **Object Model (DOM)**, 258–262
 - loading XML into DOM document, 259–260
 - traversing and modifying XML files with, 262–265
 - using with ADO to convert Excel data to XML, 260–262
- document security, 491
- document variables, 426–429
- DocumentAuthor **property**, Permission **object**, 1049
- documentControls, 307, 313
- DocumentElement **property**, CustomXMLPart **object**, 1021
- DocumentInspector **object**, 505, 506, 1025–1026
- DocumentInspectors **collection**, 501, 503, 505–506, 1025–1026
- DocumentInspectors **property**, Workbook **object**, 939
- DocumentLibraryVersion **object**, 1027
- DocumentLibraryVersions **collection**, 1027
- DocumentLibraryVersions **property**, Workbook **object**, 939
- DocumentProperties **collection**, 1028–1030
- DocumentProperties **object**, 485, 724
- DocumentProperty **object**, 1028–1030
- Documents.Open filename, 416, 429
- DoEvents **statement**, 289, 593

Do...loop, 50–53DOM (Document **Object Model**), 258–262

- loading XML into DOM document, 259–260
- traversing and modifying XML files with, 262–265
- using with ADO to convert Excel data to XML, 260–262

DomainName **property**, SharedWorkspaceMember **object**, 1060DOMDocument **object**, 259, 260, 262, 269–271DoNotPromptForConvert **property**, Workbook **object**, 939DoubleClick **method**, Application **object**, 658DoubleStrikeThrough **property**, Font2 **object**, 1035**Down menus, 309**DownBars **object**, 740DownloadComponents **property**, WebOptions **object**, 928DownloadURL **property**, SmartTag **object**, 894DragOff **method**, VPageBreak **object**, 925DragOver **event**, Chart **object**, 695DragPlot **event**, Chart **object**, 695Drawing **object**, 693, 844DrilldownLevel **function**, 516Drilled **property**, TreeviewControl **object**, 917Drop **parameter**, 681Drop **property**, CalloutFormat **object**, 680DropDown **control**, 215

DropDown menu buttons, 146

DropDown **object**, 214, 216DropDownLines **property**, CommandBarComboBox **object**, 1007DropDowns **collection**, 214, 216DropDownWidth **property**, CommandBarComboBox **object**, 1007DropLines **methods**, 741DropLines **object**, 740–741DropType **property**, CalloutFormat **object**, 680DueDate **property**

- SharedWorkspaceTask **object**, 1061
- WorkflowTask **object**, 1077

DupeUnique **property**, UniqueValues **object**, 919Duplicate **method**

- Shape **object**, 884
- ShapeRange **object**, 890
- TableStyle **object**, 905

Duration, 964

DWORD C data type, 604

dynamic ActiveX **controls**, 216–219

dynamic arrays, 58, 120, 279

dynamic **controls**, RibbonX

- comboBox **control**, 315
- gallery, 315

dynamicMenu, 300–302, 304, 315, 316

<dynamicMenu .../> **control type**, 300**E**

early binding, 411, 412, 414–420, 430, 598, 984

edit box, 209, 1006–1009

Edit button, 9

Edit menu, 320, 321

Edit Query button, 486

editBox, 300, 302–304, 316

<editBox .../> **control type**, 300EditDirectlyInCell **property**, Application **object**, 648

Editing options, 552

EditingType **property**, ShapeNode **object**, 886EditionOptions **method**, Range **object**, 861EditNow **property**, 760

Editor tab, 40, 64

EditWebPage **property**, QueryTable **object**, 847ElementID **parameter**, 204, 692

elements, XML, 241–242

Elevation **property**, Chart **object**, 688Else **statement**, 20Email **property**, SharedWorkspaceMember **object**, 1060Embeddable **property**, Font2 **object**, 1035

embedded chart events, 355, 365

embedded charts, 185–187

- adding using VBA code, 186–187

- using macro recorder, 186

Embedded OLE **object**, 792Embedded **property**, Font2 **object**, 1036EmbedSmartTags **property**, SmartTagOptions **collection**, 896Employee **object**, 355, 356, 357, 359, 360, 371Employees **collection**, 355, 360, 362EmployeeSales.xml **file**, 247, 248, 257, 264

empty cells, 110, 115, 324, 743, 866

Enable button, 522

EnableAnimations **property**, Application **object**, 648EnableAutoComplete **property**, Application **object**, 648EnableAutoFilter **property**, Worksheet **object**, 956EnableAutoRecover **property**, Workbook **object**, 939EnableCalculation **property**, Worksheet **object**, 956EnableChanges **parameter**, 687, 693EnableConnections **method**, Workbook **object**, 945Enabled **attribute**, 302Enabled **property**

- AutoRecover **object**, 669
- CommandBar **object**, 1001
- CommandBarButton **object**, 1003
- CommandBarComboBox **object**, 1007
- CommandBarControl **object**, 1011
- CommandBarPopup **object**, 1014
- Permission **object**, 1049
- SmartTagRecognizer **object**, 896

EnableEditing property, QueryTable object

- EnableEditing **property**, QueryTable **object**, 847
- EnableEvents **property**, Application **object**, 648
- EnableFormatConditionsCalculation **property**, Worksheet **object**, 956
- EnableKeyCancelKey **property**, Application **object**, 648
- EnableLargeOperationAlerts **property**, Application **object**, 648
- EnableLivePreview **property**, Application **object**, 648
- EnableOutlining **property**, Worksheet **object**, 956
- EnablePivotTable **property**, Worksheet **object**, 956
- EnableRefresh **property**, QueryTable **object**, 847
- EnableResize **property**, Windows **object**, 931
- EnableSelection **property**, Worksheet **object**, 956
- EnableSound **property**, Application **object**, 648
- EnableTrustedBrowser **property**, Permission **object**, 1049
- encapsulation**, 363, 371
- Encapsulation **class modules**, 363
- Encoding **property**, WebOptions **object**, 928
- Encoding text files**, 265
- EncryptionProvider **object**, 1030–1031
- EncryptionProvider **property**, Workbook **object**, 939
- EncryptStream **method**, EncryptionProvider **object**, 1031
- End **function**, 79
- End FunctionReferencesEvents **object**, 986
- End If **statement**, 48
- End **parameter**, 174
- End **property**, 110–111, 115, 123, 358, 361, 401, 408, 613, 614, 617, 620–622, 629, 630, 853, 974
- End **statement**, 365
- End Sub **statement**, 14
- End With **statement**, 281
- EndReview **method**, Workbook **object**, 945
- EndSession **method**, EncryptionProvider **object**, 1031
- English language formulas**, 546
- EntireColumn **property**, Range **object**, 853
- EntireRow **property**, 52, 107, 854
- EnvelopeHide **event**, MsoEnvelope **object**, 1046
- EnvelopeShow **event**, MsoEnvelope **object**, 1046
- EnvelopeVisible **property**, Workbook **object**, 939
- EOF **function**, 226
- EOF **property**, Recordset **object**, 442
- Equalize **property**, Font2 **object**, 1036
- Err **object**, 59, 60, 62, 134
- Error **object**, 441, 741–742
- Error string **property**, 827
- ErrorBar **method**, Series **object**, 876
- ErrorBars **object**, 742–743
- ErrorBars **property**, Series **object**, 874
- ErrorCheckingOptions **collection**, 743–744
- ErrorCheckingOptions **object**, 743
- ErrorCheckingOptions **property**, Application **object**, 649
- ErrorCode **property**, CustomXMLValidationError **object**, 1024
- error-handling, run-time**, 59–62
- ErrorMessage **property**, Validation **object**, 923
- Errors **collection**, 441, 741–742
- Errors **object**, 741
- Errors **property**
 - CustomXMLPart object, 1021
 - Range object, 854
- ErrorTitle **property**, Validation **object**, 923
- ErrorType **property**, Sync **object**, 1068
- Esc**, 5, 201, 281, 646, 692
- Evaluate **method**
 - Application object, 96, 544–545, 560, 658
 - Chart object, 692
 - overview, 66–68
 - Worksheet object, 959
- event list**, 363
- event procedures**, 199–208
 - chart events, 202–204
 - headers and footers, 207–208
 - and running macros, 16–17
 - Workbook object events, 205–207
 - worksheet events, 199–202
 - enable events, 200–201
 - Worksheet_Calculate event, 201–202
- events**, 26–27
- Events **object**, 981–982
- Events **property**, VBE, 989
- Excel 2007 object model**
 - AboveAverage object, 636–638
 - Action object, 638–639
 - Actions collection, 638–639
 - Add-In object, 639
 - Addins collection, 639
 - Adjustments object, 640–641
 - AllowEditRange object, 642
 - AllowEditRanges collection, 641
 - Application object, 642–664
 - events, 661–663
 - example, 664
 - methods, 656–661
 - properties, 643–656
 - Areas collection, 664–665
 - AutoCorrect object, 665–667
 - AutoFilter object, 667–669
 - AutoRecover object, 669–670
 - Axis object and Axes collection, 670–674
 - AxisTitle object, 674–675
 - Border object and Borders collection, 676–677

- CalculatedFields collection, 677
- CalculatedItems collection, 678
- CalculatedMember object, 678–679
- CalculatedMembers collection, 679–680
- CalloutFormat object, 680–682
- CellFormat object, 682–684
- Characters object, 684–685
- Chart object
 - events, 694–695
 - example, 696
 - methods, 690–694
 - properties, 687–690
- ChartArea object, 696–697
- ChartColorFormat object, 698
- ChartFillFormat object, 698–700
- ChartFormat object, 700–701
- ChartGroup object, 701–704
- ChartObject object, 706–708
- ChartObjects collection, 704–706
- Charts collection, 686–687
- ChartTitle object, 708–710
- ChartView object, 710
 - collection methods, 636
 - collection properties, 635
- ColorFormat object, 710–711
- ColorScale object, 711–713
- ColorScaleCriteria collection, 713
- Comment object and Comments collection, 714–716
- ConditionValue object, 716
- Connections object, 716–717
- ConnectorFormat object, 717–719
- ControlFormat object, 719–721
- CubeField object and CubeFields collection, 721–724
- CustomProperty object and CustomProperties collection, 724–726
- CustomView object and CustomViews collection, 726–727
- Databar object, 727–729
- DataLabel object and DataLabels collection, 729–734
- DataTable object, 734–735
- DefaultWebOptions object, 735–737
- Dialog object and Dialogs collection, 737–738
- DisplayUnitLabel object, 738–739
- DownBars object, 740
- DropLines object, 740–741
- Error object and Errors collection, 741–742
- ErrorBars object, 742–743
- ErrorCheckingOptions collection object, 743–744
- FillFormat object, 744–746
- Filter object and Filters collection, 746–747
- Floor object, 747–748
- Font object, 748–749
- FormatColor object, 749–750
- FormatCondition object and FormatConditions collection, 750–753
- FreeformBuilder object, 753–754
- Graphic object, 754–756
- Gridlines object, 756–757
- GroupShapes collection, 757
- HeaderFooter object, 757–758
- HiLoLines object, 758
- HPageBreak object and HPageBreaks collection, 758–759
- Hyperlink object and Hyperlinks collection, 759–761
- Icon object, 761–762
- IconCriterion and IconCriteria collection, 762
- IconSet and IconSets collection, 762–763
- IconSetCondition object, 763–765
- Interior object, 765–767
- IRtdServer object, 767–768
- IRTDUpdateEvent object, 768
- LeaderLines object, 769
- Legend object, 770–771
- LegendEntry object and LegendEntries collection, 771–773
- LegendKey object, 773–774
- LinearGradient object, 774
- LineFormat object, 775–776
- LinkFormat object, 776–777
- ListColumn and ListColumns collection, 777
- ListColumns Common Properties, 777–778
- ListDataFormat object, 778–779
- ListObject object and ListObjects collection, 779–781
- Mailer object, 782
- MultiThreadedCalculation object, 783
- Name object and Names collection, 783–785
 - object properties, 636
- ODBCConnection object, 785–787
- ODBCError object and ODBCErrors collection, 787–788
- OLEDBConnection object, 788–790
- OLEDBError object and OLEDBErrors collection, 790–791
- OLEFormat object, 791–792
- OLEObject object and OLEObjects collection, 792–796
- Outline object, 796–797
- Page object and Pages collection, 797–798
- PageSetup object, 798–801
- Pane object and Panes collection, 802–803
- Parameter object and Parameters collection, 803–804
- Phonetic object and Phonetics collection, 804–805
- PictureFormat object, 806–807
- PivotAxis object, 807
- PivotCache object and PivotCaches collection, 807–811

Excel 2007 object model (continued)

Excel 2007 object model (continued)

- PivotCell object, 811–812
- PivotField object, PivotFields collection, and CalculatedFields collection, 812–819
- PivotFilter object and PivotFilters collection, 819–821
- PivotFormula object and PivotFormulas collection, 821–822
- PivotItem object, PivotItems collection, and CalculatedItems collection, 822–823
- PivotItemList object, 824
- PivotLayout object, 824–825
- PivotLine object, PivotLines collection, and PivotLinesCells collection, 825
- PivotTable object and PivotTables collection, 825–837
- PlotArea object, 838–839
- Point object and Points collection, 839–842
- Protection object, 842–844
- PublishObject object and PublishObjects collection, 844–846
- QueryTable object and QueryTables collection, 846–852
- Range object and Ranges collection object, 852–868
 - methods, 858–868
 - overview, 852
 - properties, 852–857
- Real-Time Data (RTD) object, 870
- RecentFile object and RecentFiles collection, 868–869
- RectangularGradient object, 869
- RoutingSlip object, 869–870
- Scenario object and Scenarios collection, 871–872
- Series object and SeriesCollection collection, 872–877
- SeriesLines object, 877
- ServerViewableItems collection, 878
- ShadowFormat object, 878–879
- Shape object and Shapes collection, 880–885
- ShapeNode object and ShapeNodes collection, 886–887
- ShapeRange object collection, 887–891
- Sheets collection, 891–893
 - overview, 891
 - Sheets common properties, 891
 - Sheets methods, 892–893
 - Sheets properties, 892
- SheetViews object, 893
- SmartTag object and SmartTags collection object, 893–894
- SmartTagAction object and SmartTagActions collection object, 894–895
- SmartTagOptions collection object, 895
- SmartTagReconizer object and SmartTagRecognizers collection object, 895–896
- Sort object, 896–897
- SortField object and SortFields collection, 897–898
- SoundNote object, 898
- Speech object, 898–899
- SpellingOptions collection object, 899–901
- Style object and Styles collection, 901–903
- Tab object, 903–904
- TableStyle object and TableStyles collection object, 904–906
- TableStyleElement object and TableStyleElements collection object, 906
- TextEffectFormat object, 907–908
- TextFrame object, 908–909
- TextFrame2 object, 909–911
- ThreeDFormat object, 911–913
- TickLabels object, 913–915
- Top10 object, 915–917
- TreeViewControl object, 917
- Trendline object and Trendlines collection, 917–919
- UniqueValues object, 919–920
- UpBars object, 920–921
- UsedObjects collection object, 921–922
- UserAccessList collection object, 922
- Validation object, 923–924
- VPageBreak object and VPageBreaks collection, 924–925
- Walls object, 925–926
- Watch object and Watches collection object, 926–928
- WebOptions object, 928–929
- Window object and Windows collection, 929–935
- Workbook object and Workbooks collection, 935–953
 - events, 951–953
 - methods, 935–937, 944–951
 - overview, 935
 - properties, 935–943
- WorkbookConnection object, 953
- Worksheet object, 953–963
 - events, 962–963
 - methods, 958–962
 - properties, 955–958
- WorksheetFunction object, 963–966
- Worksheets collection, 954–955
- WorksheetView object, 966–967
- XmlDataBinding object, 967
- XmlMap object and XMLMaps collection, 967–969
- XmlNameSpace object and XMLNameSpaces collection, 969
- XmlSchema object and XmlSchemas collection, 969–970
- XPath object, 970

Excel chart window (prior to Excel 2007) window, 608

- Excel Object **model**, **21–30**
 - getting help, 27–29
 - Immediate window, 29–30
 - objects, 22–27
 - collections, 22–23
 - events, 26–27
 - methods, 25
 - properties, 23–25
 - EXCEL4 **class**, **608**
 - Excel4IntlMacroSheets **property**
 - Application object, 649
 - Workbook object, 939
 - Excel4MacroSheets **property**
 - Application object, 649
 - Workbook object, 939
 - EXCEL7 **class**, **608**
 - Excel8CompatibilityMode **property**, Workbook **object**, **939**
 - EXCELE **class**, **608**
 - ExclusiveAccess **method**, Workbook **object**, **945**
 - Execute **method**
 - Command object, 447
 - CommandBarButton object, 1005
 - CommandBarComboBox object, 1009
 - CommandBarControl object, 1012
 - CommandBarPopup object, 1015
 - Connection object, 439, 439–440, 446
 - FileDialog object, 235, 1033
 - Recordset object, 447
 - SmartTagAction object, 895
 - ExecuteExcel4Macro **function**, **546, 555, 560, 569, 658**
 - ExecuteMso **method**, **316, 317, 1000**
 - ExecuteOptionEnum **value**, **440**
 - Exit Do **statement**, **52**
 - Exit **statement**, **59**
 - Exit Sub **statement**, **35, 91**
 - ExpandHelp **property**, SmartTagAction **object**, **895**
 - ExpirationDate **property**, UserPermission **object**, **1075**
 - Explosion **property**, Series **object**, **874**
 - Export **file**, **363**
 - Export **method**
 - exporting to XML file, 258
 - VBComponent object, 987
 - XmlMap object, 968
 - ExportAsFixedFormat **method**
 - Chart object, 692
 - Range object, 861
 - Workbook object, 945
 - Worksheet object, 959
 - ExportString **method**, Chart **object**, **692**
 - ExportXml **method**, XmlMap **object**, **969**
 - Extend **method**, Series **object** Collection, **873**
 - ExtendList **property**, Application **object**, **649**
 - extensibility**, **295, 402, 403, 409, 571, 572, 598, 600, 971, 981, 990**
 - eXtensible Markup Language (XML), **239–272**
 - attributes, 242–243
 - comments, 241
 - consuming XML data directly, 246–249
 - elements and root element, 241–242
 - namespaces, 243–245
 - processing instructions, 241
 - using VBA to program Open XML files, 265–272
 - programmatically unzipping Excel containers, 266–267
 - programmatically zipping Excel containers, 267–272
 - using VBA to program XML processes, 253–265
 - Document Object Model (DOM), 258–265
 - programming XML maps, 253–258
 - XPath, 262–264
 - viewing and editing XML documents, 245
 - XML Declaration, 240–241
 - XML maps, 249–253
 - creating, 251–253
 - creating XML schema description, 249–251
 - Extensions **property**, FileDialogFilter, **1034**
 - Extent **property**, VPageBreak **object**, **925**
 - External Content category**, **499–500**
 - External Content section**, **489**
 - external data, managing**, **469–490**
 - External Data user interface, 469–472
 - Get External Data group, 470–471
 - Manage Connections group, 471–472
 - QueryTables, 472–487
 - associated with ListObject, 475–476
 - creating and using connection files, 484–489
 - and parameter queries, 476–479
 - from relational database, 472–475
 - from text file, 482–483
 - from web queries, 479–482
 - security settings, 489–490
 - WorkbookConnection object and Connections collection, 487–489
 - external data sources**, **142, 165, 178, 180, 241, 270, 470, 716**
 - External Excel **workbooks**, **499, 500**
 - Extra Auditing tab**, **296**
 - ExtraInfo **property**, **761**
 - ExtrusionColor **property**, ThreeDFormat **object**, **911**
 - ExtrusionColor Type **property**, ThreeDFormat **object**, **911**
- ## F
- FaceId **property**, **322–324, 330, 341, 1003**
 - FarEastLineBreakLevel **property**, ParagraphFormat2 **object**, **1048**

FeatureInstallInstall property, Application object

FeatureInstallInstall **property**, Application **object**, 649

FetchComplete **event**, 445

FetchRowOverflow **property**, QueryTable **object**, 847

field codes, 426, 428

field names, 123, 141, 145, 157, 261, 285, 420, 422, 429, 432, 433, 445, 450, 466, 467, 512, 832, 847

field properties, 816

FieldNames **property**, QueryTable **object**, 847

FieldOfView **property**, ThreeDFormat **object**, 912

Fields **collection**, Recordset **object**, 445

FileConverters **property**, Application **object**, 649

FileDialog **object**

dialog types, 235

Execute method, 235

FileDialogFilters **property**, 235

FileDialogSelectedItems **collection**, 235

MultiSelect **property**, 236–237

overview, 233–237, 1031–1033

FileDialog **property**, Application **object**, 649

FileDialogFilter **object**, 1033–1034

FileDialogFilters **collection**, 234–236, 235, 1033–1034

FileDialogFilters **property**, 235

FileDialogSelectedItems **collection**, 235, 1034

FileFind **property**, Application **object**, 649

FileFormat **property**, Workbook **object**, 939, 940

filename, getting from a path, 78–80

FileName **property**

PublishObject **object**, 845

VBProject **object**, 990

file number, 223

files, 405, 406, 648

files **folder**, 504, 505

Files **property**, SharedWorkspace **object**, 1055

FileTypes **object**, 1034–1035

Fill **property**

ChartFormat **object**, 700

Font2 **object**, 1036

Shape **object**, 882

ShapeRange **object**, 888

FillAcrossSheets **method**

Sheets **collection**, 892

Worksheets **collection**, 954

FillAdjacent Formulas **property**, QueryTable **object**, 847

FillDown **method**, Range **object**, 861

FillFormat **object**, 700, 710, 744–746

FillFormat **property**, Axis **object** Title, 674

FillLeft **method**, Range **object**, 861

FillRight **method**, Range **object**, 861

FillUp **method**, Range **object**, 861

Filter **button**, 146, 827

Filter **menu**, 347

Filter **object**, 148, 668, 746–747, 1034

Filter **property**, 442–443

FilterDates **procedure**, 151

FilterGroupEnum **constants**, 442

FilterIndex **property**, FileDialog **object**, 1032

FilterMode **property**

AutoFilter **object**, 668

Worksheet **object**, 956

Filters **collection**, 746–747

Filters **property**

AutoFilter **object**, 668

FileDialog **object**, 1032

Final **property**, Workbook **object**, 939

Find Control **method**, CommandBar **object**, 1002

Find **method**

CellFormat **object**, 682

CodeModule **object**, 977

Range **object**, 862

TextRange2 **object**, 1071

FindControl **method**, CommandBars **collection**, 1000

FindControls **method**, CommandBars **collection**, 1000

FindFile **method**, Application **object**, 658

FindFormat **property**

Application **object**, 649

CellFormat **object**, 682

settings, 682

FindNext **method**, Range **object**, 862

FindPrevious **method**, Range **object**, 862

FindWindow **function**, 608, 611

FirstChild **property**, CustomXMLNode **object**, 1017

FirstLineIndent **property**, ParagraphFormat2 **object**, 1048

FitToPages **variant**, 799

FitToPagesTall **property**, 799

FitToPagesWide **property**, 799

FitToPagesWide **variant**, 799

Fix **method**, DocumentInspector **object**, 506, 1026

Fix **method**, IDocumentInspector **object**, 1042

Fixed **method**, 964

FixedDecimal **property**, Application **object**, 649

FixedDecimalPlaces **property**, Application **object**, 649

FixedWidthFont **property**, WebPageFont **object**, 1076

FixedWidthFontSize **property**, WebPageFont **object**, 1076

Flip **method**

Shape **object**, 884

ShapeRange **object**, 890

Floor **object**, 747–748, 964

Floor **property**, Chart **object**, 688

Folder **options**, 5

- FolderName **property**, SharedWorkspaceFolder **object**, 1057
- Folders **property**, SharedWorkspace **object**, 1055
- FolderSuffix **property**, WebOptions **object**, 928
- FollowHyperlink **event**, Worksheet **object**, 963
- FollowHyperlink **method**, Workbook **object**, 534, 945
- Font group**, 295
- Font **object**, 748–749
- Font options**, 674, 734, 751, 772, 805
- Font **property**
 - AboveAverage object, 637
 - AxisTitle object, 674
 - BulletFormat2 object, 996
 - CellFormat object, 683
 - Characters object, 685
 - Range object, 854
 - Style object, 901
 - TableStyleElement object, 906
 - TextRange2 object, 1070
 - TickLabels object, 914
 - Top10 object, 915
 - UniqueValues object, 919
- Font2 **object**, 1035–1037
- FontBold **property**, TextEffectFormat **object**, 907
- FontItalic **property**, TextEffectFormat **object**, 907
- FontName **property**, TextEffectFormat **object**, 907
- FontSize **property**, TextEffectFormat **object**, 907
- Footer dialog box**, 754
- footers**, 201–202, 207, 505, 757, 799, 1025
- For Each...Next loop**, 54–55
- For statement**, 53
- For...Next loop**, 53, 79
- Forecast **method**, 964
- ForeColor **property**
 - ChartFillFormat object, 698
 - ShadowFormat object, 879
- Form button**, 158, 214
- Form command**, 158, 556
- Form **controls**, 209–210, 214, 216, 221, 535
- Format **function**, 87, 152, 228, 542, 546–549, 552, 560
- Format **parameter**, 691, 706, 708, 794, 796
- Format **property**
 - Axis object, 671
 - AxisTitle object, 674
 - ChartArea object, 697
 - Series object, 874
 - SeriesLines object, 877
 - TickLabels object, 914
 - Trendline object, 918
 - UpBars object, 921
 - Walls object, 926
- FormatColor **object**, 749–750
- FormatCondition **object**, 750–753
- FormatConditions **collection**, 750–753
- FormatConditions **property**, Range **object**, 854
- FormatCurrency **function**, 542
- FormatDateTime **function**, 542
- FormatNumber **function**, 542
- FormatPercent **function**, 542
- FormatRow **property**
 - AboveAverage object, 637
 - Top10 object, 915
 - UniqueValues object, 919
- FormFun.xlsm **file**, 624
- FormResizer.xlsm **workbook**, 628
- Forms button**, 12
- Forms **controls**, 12–13, 214–216, 950
- Forms toolbar**, 12, 75, 214, 219, 220
- Formula Hidden **property**, Range **object**, 854
- Formula **parameter**, 657, 821
- Formula **property**, 194, 197, 548, 557, 559, 568, 678, 854, 874
- Formula strings**, 548
- Formula1 **property**, Validation **object**, 923
- Formula2 **property**, Validation **object**, 923
- FormulaArray **property**, Range **object**, 559, 854
- FormulaBarHeight **property**, Application **object**, 649
- FormulaHidden **property**
 - CellFormat object, 683
 - Style object, 901
- FormulaLocal **property**
 - Range object, 854
 - Series object, 874
- FormulaR1C1 **property**
 - Range object, 854
 - Series object, 874
- FormulaR1C1Local **property**
 - Range object, 854
 - Series object, 874
- Formulas tab**, 17, 125, 295
- Formulas table**, 649
- For...Next loop**, 53–54
- Forward **parameter**, 917
- Forward2 **property**, Trendline **object**, 918
- ForwardMailer **method**, Workbook **object**, 945
- Frame **control**, 275, 496
- FreeformBuilder **object**, 753–754
- FreezePanes **property**, Windows **object**, 931
- Frequency **method**, 964
- FriendlyName, 396, 1039, 1041
- From **parameters**, 686
- Front **collection**, 705
- FrontPage Server Extensions**, 526, 536
- FTest **method**, 964

FullName property

FullName **property**

Add-In, 639
SmartTagRecognizer object, 896
Workbook object, 940
FullNameURLEncoded **property**, Workbook **object**, 940
FullPath **property**, Reference **object**, 984
Function **property**, 593, 976
function types, declaring, 44
functions, calling, 35–36
FunctionWizard **method**, Range **object**, 862
Fv **method**, 964
FVSchedule **method**, 964

G

<gallery .../> **control type**, 300
GammaDist **method**, 964
GammaInv **method**, 964
GammaLn **method**, 964
Gap **property**, CalloutFormat **object**, 680
GapDepth **property**, Chart **object**, 688
Gcd **method**, 964
GDI32.DLL **file**, 602
GenerateGetPivotData **property**, Application **object**, 649
GenerateSignatureLineImage **method**,
SignatureProvider **object**, 1065
GenerateTableRefs **property**, Application **object**, 649
GeoMean **method**, 964
GermanPostReform **property**, SpellingOptions **collection**, 900
GeStep **method**, 964
Get External Data group, 470–471
GetCategories **method**, IBlogExtensibility **object**, 1039
GetCertificateDetail **method**, SignatureInfo **object**, 1064
GetChartElementLong **method**, Chart **object**, 692
getContent, 304, 305, 316
GetCustomColor **method**, ThemeColorsScheme, 1073
GetCustomListContents **method**, Application **object**, 658
GetCustomListNum **method**, Application **object**, 658
GetCustomUI **function**, 402, 403
GetCustomUI **method**, IRibbonExtensibility **object**, 1043
GetEnabledMso **method**, 316, 1000
GetFileName **function**, 79
GetImagedMso **method**, CommandBars **collection**, 1000
GetImageMso, 316
GetInfo **method**, IDocumentInspector **object**, 1042
GetInput **function**, 36
GetItemByInternalName **property**, MetaProperties **collection**, 1044
getItemCount, 304, 305, 315
getItemID, 304, 305, 315
getItemImage, 304, 305, 315
getItemLabel, 304, 305, 315
getItemScreenTip, 304, 305, 315
getItemSupertip, 304, 305, 315
getLabel **callback**, 298
GetLabelMso **method**, CommandBars **collection**, 1000
GetObject **function**, 416, 417, 430, 643, 656
GetOpen **filename**, 539, 540, 584, 664
GetOpenFileName **function**, 540, 584, 658
GetPhonetic **method**, Application **object**, 658
GetPhonetic **string**, 658
GetPivotData **method**, 811, 824
GETPIVOTDATA **worksheet function**, 824
GetPressedMso **method**, 316, 1000
GetProviderDetail **method**
EncryptionProvider object, 1031
SignatureProvider object, 1065
GetRecentPosts **method**, IBlogExtensibility **object**, 1039
GetSaveAsFilename **method**, Application **object**, 658
GetScreenTipMso **method**, 316, 1000
getSelectedItemID **callback**, 304
getSelectedItemIndex **callback**, 304
GetSelection **method**, 591, 979
GetSignatureDetail **method**, SignatureInfo **object**, 1064
GetSupertipMso **method**, 316, 1001
GetTempPath **function**, 603
getText, 304
GetUpdate **method**, Sync **object**, 1068
GetUserBlogs **method**, IBlogExtensibility **object**, 1040
GetVisibleMso **method**, CommandBars **collection**, 1001
GetWindowLong **function**, 623
GetWindowRect **function**, 607, 609
GetWorkflowTasks **method**, Workbook **object**, 945
GetWorkflowTemplates **method**, Workbook **object**, 945
Given Action **object**, 638, 639
globals, 63–64
<globals>, 63, 74, 95, 97, 115, 376
Glow **property**
ChartFormat object, 700
Font2 object, 1036
Shape object, 882
ShapeRange object, 888
GlowFormat **object**, 1037
Go button, 385
GoalSeek **method**, Range **object**, 862

Goto **method**, Application **object**, 659
 Gradient **property**, 774, 869
 GradientColorType **property**, ChartFillFormat **object**, 698
 GradientDegree **property**, ChartFillFormat **object**, 698
 GradientStop **object**, 1037–1038
 GradientStops **collection**, 1037–1038
 GradientStyle **property**, ChartFillFormat **object**, 698
 GradientVariant **property**, ChartFillFormat **object**, 699
 Graphic **object**, 754–756
 GridlineColor **property**, Windows **object**, 931
 GridlineColorIndex **property**, Windows **object**, 931
 Gridlines **methods**, 757
 Gridlines **object**, 756–757
 Group **method**
 Range **object**, 862
 ShapeRange **object**, 890
Group mode, 90, 91
grouping, Worksheet **objects**, 87–88
Grouping dialog box, 174
 GroupItems **property**
 Shape **object**, 882
 ShapeRange **object**, 888
 GroupName **property**, 213, 281
groups, controlling using RibbonX, 313–314
 GroupShapes **collection**, 757
 Guid **property**
 Add-in **object**, 973
 COMAddin **object**, 998
 Reference **object**, 984

H

HANDLE C data type, 604
handles, 306, 406, 407, 606–609, 640, 986
 HangingPunctuation **property**, ParagraphFormat2 **object**, 1048
 HarMean **method**, 964
Has3DEffect property, Series **object**, 874
 HasArray **property**, Range **object**, 854
HasAuto format, 836
 HasAxis **property**, Chart **object**, 688
 HasChart **property**
 Shape **object**, 882
 ShapeRange **object**, 888
 HasChildNodes **method**, CustomXMLNode **object**, 1018
 HasDataLabel **property**, 204
 HasDataLabels **property**, Series **object**, 874
 HasDataTable **property**, Chart **object**, 688
 HasDisplayUnitLabel **property**, Axis **object**, 672
 HasErrorBars **property**, Series **object**, 874
 HasFormat **property**, TableStyleElement, 906
 HasFormula **property**, Range **object**, 854
 HashStream **method**, SignatureProvider **object**, 1065
 HasLeaderLines **property**, Series **object**, 874
 HasLegend **property**, Chart **object**, 688
 HasMajorGridlines **property**, Axis **object**, 672
 HasMinorGridlines **property**, Axis **object**, 672
 HasOpenDesigner **property**, VBComponent, 987
 HasPassword **property**, Workbook **object**, 940
 HasRoutingSlip **property**, 869, 940
 HasShortcut **key**, 659
 HasText **property**, TextFrame **object**, 910
 HasTitle **property**
 Axis **object**, 672, 674
 Chart **object**, 688
 ChartTitle **object**, 708
 HasUpDownBars **property**, 701, 920
 HasVBProject **property**, Workbook **object**, 940
HBITMAP C data type, 604
HBRUSH C data type, 604
HCURSOR C data type, 604
HDC C data type, 604
 Header **property**, Sort **object**, 896
Header Row, 155, 347, 350, 378, 777, 779, 780
 HeaderFooter **object**, 757–758
headers, 207–208, 505, 757, 799, 1025
 HeadersFooters **collection**, 757
 HeartbeatInterval **property**, 768
 HebrewModes **property**, SpellingOptions **collection**, 900
 Height **parameters**, 705, 793, 802
 Height **property**
 Application **object**, 650
 Axis **object**, 672
 ChartArea **object**, 697
 CommandBar **object**, 1001
 CommandBarButton **object**, 1003
 CommandBarComboBox **object**, 1007
 CommandBarControl **object**, 1011
 CommandBarPopup **object**, 1014
 CustomTaskPane **object**, 1016
 Range **object**, 854
 Shape **object**, 882
 ShapeRange **object**, 888
 values, 187
 Window **object**, 931, 993
 HeightPercent **property**, Chart **object**, 688
help, 27–29
Help button, 32, 34
Help file, 659
Help method, Application **object**, 659
 HelpContextID **parameter**, 659

HelpContextId property

HelpContextId **property**

- CommandBarButton object, 1003
- CommandBarComboBox object, 1007
- CommandBarControl object, 1011
- CommandBarPopup object, 1014

HelpContextID **property**, VBproject **object**, 991

HelpFile **property**

- CommandBarButton object, 1004
- CommandBarComboBox object, 1007
- CommandBarControl object, 1011
- CommandBarPopup object, 1014
- VBproject object, 991

Hex2Bin **method**, 964

Hex2Dec **method**, 964

Hex2Oct **method**, 964

HFONT C **data type**, 604

HICON C **data type**, 604

Hidden fields **object**, 828

Hidden items **property**, 171

hidden member, 214

Hidden **property**

- Range object, 154, 160, 854
- Scenario object, 871
- TreeviewControl object, 917

Hide button, 6

Hide **method**, 274, 277

hiding names, 131–132

Hierarchize **function**, 516

Highlight **property**, Font2 **object**, 1036

HighlightChangesOnScreen **property**, Workbook **object**, 940

HighlightChangesOptions **method**, Workbook **object**, 945

high-resolution timer, 616, 618, 634

HiLoLines **object**, 758

HINSTANCE C **data type**, 604

Hinstance **property**, Application **object**, 650

HLOCAL C **data type**, 604

HLookup **method**, 964

HMENU C **data type**, 604

HMETAFILE C **data type**, 604

HMODULE C **data type**, 604

Home Directory tab, 532

Home tab, 4, 8, 295

HorizontalAlignment **property**

- AxisTitle object, 674
- CellFormat object, 683
- Range object, 854
- Style object, 901
- TextFrame object, 908

HorizontalAnchor **property**, TextFrame **object**2, 910

HorizontalFlip **property**

- Shape object, 882
- ShapeRange object, 888

HoursPerWeek **property**, 357, 358

HPageBreak **object**, 758–759

HPageBreaks **collection**, 758–759

HPageBreaks **property**

- Charts collection, 686
- Sheets collection, 892
- Worksheets collection, 954, 956

HPALETTE C **data type**, 604

HPEN C **data type**, 604

HRGN C **data type**, 604

HTASK C **data type**, 604

HTML **feature**, 533

HTML **file**, 532

HTML **format**, 480, 533

HTML **page**, 404, 735

HtmlType **property**, PublishObject **object**, 845

HWND C **data type**, 605

Hwnd **property**

- Application object, 650
- GetWindowRect function, 607

Hwnd **property**, Window **object**, 993

Hyperlink **object**, 759, 760, 761

Hyperlink **property**, Shape **object**, 882

Hyperlinks **collection**, 759–761

Hyperlinks **property**

- Range object, 854
- Worksheet object, 956

HyperlinkType **property**, CommandBarButton **object**, 1004

HypGeomDist, 964

I

IAssistance **object**, 1038–1039

IBlogExtensibility **object**, 1039–1041

IBlogPictureExtensibility **object**, 1039–1041

Icon **object**, 761–762

IconCriteria **collection**, 762, 763

IconCriterion **collection**, 762

Icons button, 306

IconSet **collection**, 762–763

IconSet **object**, 762

IconSetCondition **object**, 763–765

IconSets **property**, Workbook **object**, 940

ICTPFactory **object**, 1041

ICustomTaskPaneConsumer **interface**, 404

ICustomTaskPaneConsumer **object**, 1041–1042

Id **attribute**, 302

Id property

CommandBarButton object, 1004
 CommandBarComboBox object, 1007
 CommandBarControl object, 1011
 CommandBarPopup object, 1014
 CustomXMLPart object, 1021
 IRibbonControl object, 1042
 MetaProperty object, 1045
 overview, 322–324
 PolicyItem object, 1051
 Range object, 854
 ServerPolicy object, 1050
 Shape object, 882
 ShapeRange object, 888
 SignatureSetup object, 1066
 WorkflowTask object, 1077
 WorkflowTemplate object, 1078

idMso attribute, 302**IDocumentInspector object, 1042****idQ attribute, 302****IDTExtensibility2 interface, 388–394****IF function, 47, 276, 546****If statements, 47–48****IfError method, 964****IgnoreBlank property, Validation object, 923****IgnoreCaps property, SpellingOptions collection, 900****IgnoreFileNames property, SpellingOptions collection, 900****IgnoreMixedDigits property, SpellingOptions collection, 900****IgnoreRemoteRequests property, Application object, 650****IIf function, 47, 48****ImAbs method, 964****Image attribute, 302****Image control, 233, 235, 237****imageMso attribute, 302****Imaginary method, 964****ImArgument method, 964****ImConjugate method, 964****ImCos method, 964****ImDiv method, 964****IMEMode property, Validation object, 923****ImExp method, 964****ImLn method, 964****ImLog10 method, 964****ImLog2 method, 964****Immediate window, 29–30, 71, 95, 260, 375, 441, 580, 981, 989, 993****Immediate Window button, 29****Implements keyword, 767****implicit conversion, 539, 541, 561****Import file, 363****Import method**

importing data into XML map, 257
 SoundNote object, 898
 VBcomponents collection, 988
 XmlMap object, 969

ImportXml method, XmlMap object, 969**ImPower method, 964****ImProduct method, 964****ImReal method, 964****ImSin method, 964****ImSqrt method, 964****ImSub method, 965****ImSum method, 965****inactive worksheets, 96, 99****InactiveListBorder property, Workbook object, 940****Inbox folder, 413****InCell Dropdown property, Validation object, 923****Inches parameter, 659****InchesToPoints method, Application object, 659****Include Protection property, Style object, 902****IncludeAlignment property, Style object, 901****IncludeBorder property, Style object, 902****IncludeFont property, Style object, 902****IncludeLayout property, Axis object Title, 674****IncludeNumber property, Style object, 902****IncludePatterns property, Style object, 902****IncrementLeft method**

Shape object, 885

ShapeRange object, 890

IncrementOffsetX method, ShadowFormat object, 879**IncrementOffsetY method, ShadowFormat object, 879****IncrementRotation method**

Shape object, 885

ShapeRange object, 890

IncrementRotationHorizontal method, ThreeDFormat object, 912**IncrementRotationVertical method, ThreeDFormat object, 912****IncrementRotationX method, ThreeDFormat object, 913****IncrementRotationY method, ThreeDFormat object, 913****IncrementRotationZ method, ThreeDFormat object, 913****IncrementTop method**

Shape object, 885

ShapeRange object, 890

IndentLevel property

CellFormat object, 683

ParagraphFormat2 object, 1048

Range object, 854

Style object, 902

Index method, 965

Index parameter

Index **parameter**, **636, 640, 664, 666, 721, 771, 801**

Index **property**

- Chart object, 688
 - Clear Contents control, 341
 - CommandBar object, 1001
 - CommandBarButton object, 1004
 - CommandBarComboBox object, 1007
 - CommandBarControl object, 1011
 - CommandBarPopup object, 1014
 - DocumentLibraryVersion object, 1027
 - RecentFile object, 868
 - Scenario object, 871
 - Trendline object, 918
 - Window object, 931
 - Worksheet object, 84, 85, 956
- IndexedValue **property**, Property **object**, **982**
- InitialFileName **property**, FileDialog **object**, **1032**
- Initialization **controls**, **496**
- Initialize **event**, **279, 284, 369, 461, 575, 576, 577, 629, 633**
- Initialize routine**, **617**
- InitializeChart **events**, **367**
- InitialView **property**, FileDialog **object**, **1032**
- input and output**, **30–35**
- constants, 33
 - InputBox, 34–35
 - parameters specified by name, 31–33
 - parameters specified by position, 31
 - return values, 33–34
- Input **statement**, **226, 227**
- InputBox **function**, **36, 62, 68–69, 273**
- InputBox **method**, **68, 69, 659**
- InputMessage **property**, Validation **object**, **923**
- InputTitle **property**, Validation **object**, **923**
- Insert button**, **12, 13, 209**
- Insert dialog box**, **182**
- Insert method**
- Characters object, 685
 - GradientStops collection, 1038
 - Range object, 862
 - ShapeNodes Collection, 886
- INSERT **statement**, **434, 453, 467**
- InsertAfter **method**, TextRange2 **object**, **1071**
- InsertBefore **method**, TextRange2 **object**, **1071**
- InsertIndent **method**, Range **object**, **862**
- InsertLines **method**, CodeModule **object**, **977**
- InsertNodesBefore **method**, CustomXMLNode **object**, **1018**
- InsertSubtreeBefore **method**, CustomXMLNode **object**, **1018**
- InsertSymbol **method**, TextRange2 **object**, **1071**

Inspect button, **501**

Inspect **method**

- IDocumentInspector object, 505, 1026, 1042
- Installed **property**, Add-In, **639**
- installing Add-ins**, **379–381**
- InstallManifest, **969**
- Instancing **property**, **383, 384, 389**
- InStr **function**, **195**
- Int C data type**, **605**
- int FAR * C **data type**, **605**
- IntegralHeight **property**, **634**
- Interactive **property**, Application **object**, **650**
- Intercept **method**, **965**
- Intercept **parameter**, **917**
- Intercept **property**, Trendline **object**, **918**
- InterceptIsAuto **property**, Trendline **object**, **918**
- Interior **object**, **26, 765–767, 774, 869**
- Interior **property**
- AboveAverage object, 637
 - AxisTitle object, 674
 - CellFormat object, 683
 - Range object, 854
 - Style object, 902
 - TableStyleElement object, 906
 - Top10 object, 915
 - UniqueValues object, 919
- InteriorPattern **property**, **869**
- international issues**, **537–569**
- exceptions to rules, 554–560
 - = TEXT() worksheet function, 558–560
 - OpenText function, 555–556
 - pasting text, 557
 - PivotTable calculated fields and items, 557–558
 - SaveAs function, 556
 - ShowDataForm sub procedure, 556
 - web queries, 558
 - helpful functions, 565–568
 - bWinToDate function, 566–567
 - bWinToNum function, 565–566
 - ReplaceHolders function, 568
 - sFormatDate function, 567
 - interacting with Excel, 545–548
 - reading data from Excel, 548
 - rules for working with Excel, 548
 - sending data to Excel, 545–547
 - interacting with users, 549–551
 - displaying data, 549
 - interpreting data, 550
 - paper sizes, 549
 - rules for working with users, 551
 - xxxLocal properties, 550–551

- international options, 552–554
- Office 2007 language settings, 560–565
 - creating multilingual application, 562–564
 - identifying Office UI language settings, 561–562
 - rules for developing multilingual application, 565
 - where text comes from, 560–561
 - working in multilingual environment, 564–565
- regional settings and Office 2007 UI language, 537–538
- regional settings and Windows language, 538–545
 - identifying user's regional settings and Windows language, 538–539
 - VBA conversion functions, 539–545
- International options, 558, 566**
- International property, Application object, 650**
- Internet, 525–536**
 - overview, 526
 - using as communication channel, 533–536
 - using as data source, 527–531
 - opening web pages as workbooks, 528
 - parsing web pages for specific information, 530–531
 - using web queries, 528–530
 - using for storing workbooks, 526–527
 - using to publish results, 531–533
 - creating interactive web pages, 533
 - saving worksheets as web pages, 532–533
 - setting up web server, 532
- Internet Transfer control, 525, 534–536, 601**
- Intersect method, Application object, 115, 659**
- IntRate method, 965**
- Introduction property, MsoEnvelope object, 1046**
- invalidate ContentOnDrop attribute, 302**
- Invalidate method, 309, 1043**
- InvalidateControl method, 309, 310, 1043**
- InvertIfNegative property, Series object, 874**
- IPicture object, 306, 317**
- Ipmt method, 965**
- IQY (Web Query) files, 486–487**
- IQY files, 484, 486, 487**
- IRibbon control, 303–305, 310, 311, 316, 403**
- IRibbon parameter, 309**
- IRibbonControl object, 1042**
- IRibbonExtensibility object, 1043**
- IRibbonUI object, 1043**
- Irr method, 965**
- IRtdServer object, 767–768**
- IRtdServer object, 870**
- IRTDUpdate event, 768**
- IRTDUpdateEvent object, 768**
- IsAddin property, 375, 380–382**
- IsAddin property, Workbook object, 940**
- ISBLANK function, 67**
- IsBoolean function, 541**
- IsBroken property, Reference object, 984**
- IsCertificateExpired property, SignatureInfo object, 1063**
- IsCertificateRevoked property, SignatureInfo object, 1063**
- IsCertificateUntrusted property, SignatureInfo object, 1064**
- IsDate function, 541**
- IsDateUS function, 544**
- IsEmptyWorksheet function, 323, 324**
- IsErr method, 965**
- IsError method, 544, 965**
- IsEven method, 965**
- IsInplace property, Workbook object, 940**
- IsLogical method, 965**
- IsMember property, Boolean Read-only, 820**
- IsNA method, 965**
- IsNameIn workbook, 133, 134, 135**
- IsNonText method, 965**
- IsNumeric function, 541, 965**
- IsOdd method, 965**
- Ispmt method, 965**
- IsPriorityDropped property**
 - CommandBarButton object, 1004
 - CommandBarComboBox object, 1007
 - CommandBarControl object, 1011
 - CommandBarPopup object, 1014
- IsReadOnly property, MetaProperty object, 1045**
- IsRequired property, MetaProperty object, 1045**
- IsSignatureLine property, Signature object, 1062**
- IsSigned property, Signature object, 1062**
- IsText method, 965**
- IsValid property**
 - CalculatedMember object, 678
 - SignatureInfo object, 1064
- Italic buttons, 4, 8**
- Italic property, Font2 object, 1036**
- <item .../> control type, 300**
- Item method**
 - Add-ins collection, 974
 - COMAddins collection, 997
 - FileDialogFilters collection, 1034
 - FileDialogSelectedItems collection, 1034
 - overview, 636
 - RulerLevels2 collection, 1052
 - TabStops2 collection, 1069
 - TextRange2 object, 1071
 - ThemeFonts collection, 1073
- Item property, 101**
 - Adjustments object, 640
 - AllowEditRanges collection, 641
 - Areas collection, 664

Item property (continued)

Item **property (continued)**

Borders collection, 676
CalculatedMembers collection, 678
Chart object, 686
CommandBarControls collection, 1010
CommandBars collection, 1000
DocumentProperties collection, 1029
Errors Collection object, 741
FileDialog object, 1032
FileTypes collection, 1035
GradientStops collection, 1037
MetaProperties collection, 1044
MsoEnvelope object, 1046
Pages collection, 797
Permission object, 1049
Range object, 97, 100
ScopeFolder collection, 1052
SearchFolders collection, 1053
SearchScopes collection, 1054
ServerPolicy object, 1050
SharedWorkspaceFiles collection, 1056
SharedWorkspaceFolders collection, 1057
SharedWorkspaceLinks collection, 1058
SharedWorkspaceMembers collection, 1059
SharedWorkspaceTasks collection, 1060
Sheetviews collection, 893
SignatureSet collection, 1062
UserAccessList object, 921, 922
WebPageFonts collection, 1076
Windows Collection, 929
Workbook object, 935
WorkflowTasks collection, 1076
WorkflowTemplates collection, 1077

ItemAdded event
Reference collection, 985
ReferenceEvent object, 986

ItemCountExceeded property
SharedWorkspaceFiles collection, 1056
SharedWorkspaceFolders collection, 1057
SharedWorkspaceLinks collection, 1058
SharedWorkspaceMembers collection, 1059
SharedWorkspaceTasks collection, 1060

ItemHeight attribute, 302

ItemID parameter, 317

ItemIndex parameter, 317

ItemRemoved event
Reference collection, 985
ReferenceEvent object, 986

Items combo box, 721

Items parameter, 390, 953

Items property, 363

ItemSize attribute, 302

ItemWidth attribute, 302

Iteration property, Application object, 650

IXMLDOMNode object, 262, 264

IXMLDOMNodeList object, 262

J

Justify method, Range object, 862

K

KeepChangeHistory property, Workbook object, 940

KernedPairs property, TextEffectFormat object, 907

KERNEL32.DLL file, 602

Kerning property, Font2 object, 1036

Key parameter, 660

Key property, SortFields object, 897

Keys parameter, 660

keytip attribute, 302

Kill statement, 60, 61

KoreanCombineAux property, SpellingOptions collection, 900

KoreanProcessCompound property, SpellingOptions collection, 900

KoreanUseAutoChangeList property, Spelling Options collection, 900

Kurt method, 965

L

Label controls, 276, 289

Label filters, 820

label group attribute, 302

labelControl, 299

<labelControl .../> control type, 299

Labels formatting, 703

Labels group, 186, 195

language version of Windows, 561

LanguageID property, TextRange2 object, 1070

LanguagePreferredForEditing property, Language Settings object, 1044

LanguageSettings object, 1043–1044

LanguageSettings property, Application object, 650

LanguageID property, LanguageSettings object, 1044

LARGE INTEGER C data type, 605

Large method, 965

LargeButtons property, CommandBars collection, 1000

LargeOperationCellThousandCount property, Application object, 650

LargeScroll method, Windows object, 933

Last cell, 8, 105, 106, 110, 123

LastChild property, CustomXMLNode object, 1017

- LastRefreshed **property**, SharedWorkspace **object**, 1055
- LastSyncTime **property**, Sync **object**, 1068
- late binding, 412–414
- Launcher button, 300
- Layout **option**, 837
- Layouts group, 182
- LBound **functions**, 57, 109, 190
- Lcm **method**, 965
- LeaderLines **object**, 769, 771, 772
- LeaderLines **property**, Series **object**, 874
- Left **property**
- Application object, 650
 - Axis object, 672
 - AxisTitle object, 675
 - ChartArea object, 697
 - CommandBar object, 1002
 - CommandBarButton object, 1004
 - CommandBarComboBox object, 1007
 - CommandBarControl object, 1011
 - CommandBarPopup object, 1014
 - Range object, 855
 - Shape object, 882
 - ShapeRange object, 888
 - Window object, 931, 993
- LeftIndent **property**, ParagraphFormat2 **object**, 1048
- Legend **object**, 770–771
- Legend **property**, Chart **object**, 688
- LegendEntries **collection**, 771–773
- LegendEntry **object**, 771–773
- LegendKey **object**, 773–774
- Len **function**, 54, 79
- Length **parameter**, 681, 684
- Length **property**
- CalloutFormat object, 680
 - TextRange2 object, 1070
- Level **object**, 520
- Levels **property**, Ruler2 **object**, 1051
- LibraryPath **property**, Application **object**, 650
- LightAngle **property**, ThreeDFormat **object**, 912
- Line charts, 671, 702
- Line Input **statement**, 226, 227
- Line **property**
- ChartFormat object, 700
 - Font2 object, 1036
 - Shape object, 882
 - ShapeRange object, 888
- LinearGradient **object**, 774, 869
- LineFormat **object**, 700, 775–776
- LineRuleAfter **property**, ParagraphFormat2 **object**, 1048
- LineRuleBefore **property**, ParagraphFormat2 **object**, 1048
- LineRuleWithin **property**, ParagraphFormat2 **object**, 1048
- Lines **property**
- CodeModule object, 975
 - TextRange2 object, 1070
- LinEst **method**, 965
- LineStyle **property**
- Border collection, 677
 - Borders collection, 676
- LinkedCell **property**, 211
- LinkedWindowFrame **property**, Window **object**, 993
- LinkedWindows **collection**, 982
- LinkedWindows **property**, Window **object**, 993
- LinkFormat **object**, 776–777
- LinkFormat **property**, Shape **object**, 883
- LinkInfo **method**, Workbook **object**, 946
- Links **property**, SharedWorkspace **object**, 1055
- LinkSource **property**, DocumentProperty **object**, 1030
- LinkSources **method**, Workbook **object**, 946
- LinkToContent **property**, DocumentProperty **object**, 1030
- List **controls**, 326, 327
- List **field**, 157, 827
- List **files**, 55
- list management tools, 141
- List **object**, 145, 147, 148, 151–154, 164, 256, 469, 472, 475, 476, 484, 487, 489, 499, 778–781, 804
- List **property**, 150, 216, 279, 285, 338, 1007
- ListAll **controls**, 279, 341
- ListAllFaces, 329
- ListBox **control**, 276
- ListChangesOnNewSheet **property**, Workbook **object**, 940
- ListColumn **collection**, 777
- ListColumn **methods**, 778
- ListColumn **object**, 777, 778
- ListColumns **collection**, 777–778
- ListColumns **object**, 777
- ListControls **function**, 326
- ListCount **property**, CommandBarComboBox **object**, 1007
- ListDataFormat **object**, 778–779
- ListFirstLevel **controls**, 322, 323, 327, 338, 340
- ListHeaderCount **property**, CommandBarComboBox **object**, 1007
- ListHeaderRows **property**, Range **object**, 855
- ListID **property**, WorkflowTask **object**, 1077
- ListIndex **property**, 152, 216, 237, 338, 1008
- Listing **files**, 55
- ListNames **method**, Range **object**, 862
- ListObject **collection**
- methods, 781
 - properties, 779–780
 - QueryTables associated with, 475–476

ListObject object

- ListObject **object**, 143, 145–148, 155, 160, 779–781, 846
 - ListObject **property**
 - QueryTable object, 847
 - Range object, 855
 - ListObjects **collection**, 779–781
 - ListObjects **property**, Worksheet **object**, 956
 - ListPopups, 340
 - ListRows **collection**, 781
 - ListRows **object**, 782
 - Lists items**, 830
 - ListSelection **property**, SmartTagAction **object**, 895
 - Ln **method**, 965
 - Load **method**
 - CustomXMLPart object, 1021
 - ThemeColorsScheme object, 1073
 - ThemeEffectScheme object, 1073
 - ThemeFontScheme object, 1074
 - Load **statement**, 273
 - LoadBehavior value**, 396
 - loadImage callback**, 306
 - LoadPicture **function**, 235, 306
 - LoadPictureGDI **function**, 306
 - LoadSettings **method**, XmlDataBinding **object**, 967
 - LoadXML **method**, CustomXMLPart **object**, 1021
 - Local **parameter**, 555, 556, 560
 - LocalConnection **property**, 790, 809
 - Locals Window**, 580
 - Location **property**
 - CustomXMLSchema object, 1023
 - VPageBreak object, 925
 - LocationInTable **property**, Range **object**, 855
 - LocationOfComponents **property**, WebOptions **object**, 928
 - LocationXLChart **method**, Chart **object**, 692
 - LockAspectRatio **property**
 - Shape object, 883
 - ShapeRange object, 888
 - Locked **property**
 - CellFormat object, 683
 - Range object, 855
 - Scenario object, 871
 - Shape object, 883
 - Style object, 902
 - LockServerFile **method**, Workbook **object**, 946
 - Log **method**, 965
 - Log10 **method**, 965
 - LogBase **property**, Axis **object**, 672
 - LogEst **method**, 965
 - LogInv **method**, 965
 - LogNormDist **method**, 965
 - LONG C **data type**, 605
 - Long **method**, 692
 - Lookup **method**, 965
 - Lookup Namespace **method**, CustomXMLPrefixMapping **collection**, 1022
 - LookupPrefix **method**, CustomXMLPrefixMapping **collection**, 1022
 - Loop **statement**, 51, 52
 - looping, 50–55
 - Do...loop, 50–53
 - For Each...Next loop, 54–55
 - For...Next loop, 53–54
 - LPARAM C **data type**, 605
 - LPCSTR C **data type**, 605
 - LPCTSTR C **data type**, 605
 - lpRect **parameter**, 607
 - LPSTR C **data type**, 605
 - LPTSTR C **data type**, 605
 - LPVOID C **data type**, 605
 - LRESULT C **data type**, 605
 - LtrRun **method**, TextRange2 **object**, 1071
- ## M
- Macro dialog box, 3, 4, 6, 7, 9, 12, 151, 216, 377
 - Macro options, 659
 - Macro parameter, 660
 - macro recorder, 2–17
 - embedded charts using, 186
 - other ways to run macros, 11–17
 - ActiveX controls, 13–15
 - event procedures, 16–17
 - Forms controls, 12–13
 - Quick Access Toolbar, 15–16
 - worksheet buttons, 12
 - recording macros, 2–6
 - macro security, 5
 - Personal Macro Workbook, 5–6
 - running macros, 6–8
 - absolute and relative recording, 7–8
 - shortcut keys, 6–7
 - Visual Basic Editor (VBE), 8–11
 - code modules, 9–10
 - procedures, 10
 - Project Explorer, 10–11
 - Properties window, 11
 - Macro Security button, 5
 - Macro Settings, 5
 - Macro settings category, 497
 - Macro settings dialog, 571
 - macro settings, Trust Center user interface, 497–498
 - Macro-Enabled workbook, 5
 - MacroOptions **method**, Application **object**, 659

- Macros button, 6, 9**
- MailEnvelope **property**
 - Chart object, 688
 - Worksheet object, 956
- Mailer **object, 782**
- Mailer **property, Workbook object, 940**
- MailLogoff **method, Application object, 659**
- MailLogon **method, Application object, 659**
- MailSession **property, Application object, 650**
- MailSystem **property, Application object, 650**
- MaintainConnection **property, 790, 847**
- MainWindow **property, VBE, 989**
- Major **property, Reference object, 984**
- MajorFont **property, ThemeFontScheme object, 1074**
- MajorGridlines **property, Axis object, 672**
- MajorTickMark **property, Axis object, 672**
- MajorUnit **property, Axis object, 672**
- MajorUnitIsAuto **property, Axis object, 672**
- MajorUnitScale **property, Axis object, 672**
- MakeCompiledFile **method, VBProject object, 991**
- Manage Connections group, 471–472**
- Map **property, XPath object, 970**
- MapPaperSize **property, Application object, 651**
- maps, XML. See XML, XML maps**
- MarginBottom **property**
 - TextFrame object, 908
 - TextFrame2 object, 910
- MarginLeft **property**
 - TextFrame object, 908
 - TextFrame2 object, 910
- MarginRight **property**
 - TextFrame object, 908
 - TextFrame2 object, 910
- MarginTop **property**
 - TextFrame object, 908
 - TextFrame2 object, 910
- MarkerBackgroundColor **property, Series object, 874**
- MarkerBackgroundColorIndex **property, Series object, 874**
- MarkerForegroundColor **property, Series object, 874**
- MarkerForegroundColorIndex **property, Series object, 874**
- MarkerSize **property, Series object, 875**
- MarkerStyle **property, Series object, 875**
- Mask **property, CommandBarButton object, 1004**
- MatchCase **property, Sort object, 896**
- MathCoprocessorAvailable **property, Application object, 651**
- Max **property, 285, 287**
- MaxChange **property, Application object, 651**
- Maximize box, 623**
- Maximum **property, 868**
- MaximumScale **property, Axis object, 672**
- MaxIterations **property, Application object, 651**
- maxLength **attribute, 302**
- MDX (Multidimensional Expression)**
 - behind OLAP-based pivot tables, 512–517
 - creating MDX log, 515–517
 - deciphering MDX queries, 515
- MDX **property, Range object, 855**
- MDX query, 512–518, 521–523**
- MDX statement, 513, 515, 516**
- Me keyword, 152**
- MeasurementUnit **property, Application object, 651**
- Members **property, SharedWorkspace object, 1055**
- <menu ...> contents </menu> container
 - control, 301**
- menu bars, 320–322, 579**
- Menu item, Click events, 575**
- MenuAnimationStyle **property, CommandBars collection, 1000**
- MenuKey menu, 654**
- menus**
 - creating, 330–333
 - deleting, 334
 - popup, 338–341
- menuSeparator, 300, 301, 303**
- <menuSeparator .../> **control type, 300**
- Merge **method**
 - Range object, 863
 - Scenarios collection, 871
- MergeArea **property, Range object, 855**
- MergeCells **property**
 - CellFormat object, 683
 - Range object, 855
 - Style object, 902
- MergeWorkbook **method, Workbook object, 946**
- Message Bar, 494, 496–500**
- Message box, 24, 32, 78, 356, 403, 505, 513, 550, 609**
- Message **property, RoutingSlip object, 870**
- MetaProperties **collection, 1044–1045**
- MetaProperty **object, 1044–1045**
- methods, 25**
- Microsoft Access. See Access, and ADO**
- Microsoft Access database, 448, 449**
- Microsoft Internet controls, 530**
- Microsoft Office button, 3, 5, 373, 374, 379, 380, 489, 491**
- Microsoft Outlook. See Outlook**
- Microsoft Word. See Word**
- Mid **function, 54, 195**
- Min **function, 212**
- Mini toolbar, 317**
- Minimize box, 623**
- MinimumScale **property, Axis object, 672**

Minor property, Reference object

- Minor **property**, Reference **object**, 984
- MinorFont **property**, ThemeFontScheme **object**, 1074
- MinorGridlines **property**, Axis **object**, 672
- MinorTickMark **property**, Axis **object**, 672
- MinorUnit **property**, Axis **object**, 672
- MinorUnitIsAuto **property**, Axis **object**, 672
- MinorUnitScale **property**, Axis **object**, 672
- MOD **function**, 73
- Mod operator**, 73, 74, 204
- Mode **property**, VBproject **object**, 991
- Modeless UserForm**, 288–291
- Modified **property**, DocumentLibraryVersion **object**, 1027
- ModifiedBy **property**
 - DocumentLibraryVersion **object**, 1027
 - SharedWorkspaceFile **object**, 1056
 - SharedWorkspaceLink **object**, 1058
 - SharedWorkspaceTask **object**, 1061
- ModifiedDate **property**
 - SharedWorkspaceFile **object**, 1056
 - SharedWorkspaceLink **object**, 1058
 - SharedWorkspaceTask **object**, 1061
- Modify button**, 15, 493
- Modify **method**, Validation **object**, 924
- ModifyAppliesToRange **method**
 - AboveAverage **object**, 638
 - Top10 **object**, 916
 - UniqueValues **object**, 920
- ModifyKey **method**, SortFields **object**, 898
- module-level **variable**, 42, 45, 72, 91, 307, 309, 404, 405, 576
- Modules **collection**, 83
- MouseAvailable **property**, Application **object**, 651
- MouseDown **event**, Chart **object**, 695
- MouseMove **event**, Chart **object**, 695
- MouseUp **event**, Chart **object**, 695
- Move **method**
 - Chart **object**, 686, 692
 - CommandBarButton **object**, 1005
 - CommandBarComboBox **object**, 1009
 - CommandBarControl **object**, 1012
 - CommandBarPopup **object**, 1015
 - Recordset **object**, 444
 - Sheets **collection**, 893
 - Worksheet **object**, 85–87, 959
 - Worksheets **collection**, 954
- MoveAfterReturn **property**, Application **object**, 651
- MoveAfterReturnDirection **property**, Application **object**, 651
- MPR.DLL file**, 602
- MSDN documentation**, 606, 623
- MSDN library**, 634
- MSDN Library menu**, 602
- MSForms **Control**, 579
- MSForms **Control Group**, 579
- MSForms DragDrop**, 580
- MSForms MPC**, 579
- MSForms **object library**, 594
- MSForms Palette**, 579
- MSForms Toolbox**, 580
- MsgBox **function**, 30, 33, 62, 273, 561
- MsgBox **statements**, 360, 363
- msoBarPopup, 343
- msoBarTypeMenuBar **constant**, 321
- msoBarTypeNormal **constant**, 321
- msoBarTypePopup, 321, 338–340
- MsoEnvelope **object**, 1045–1046
- msoFileDialogFilePicker **constant**, msoFileDialog **object**, 234
- msoFileDialogFolderPicker **constant**, msoFileDialog **object**, 234
- msoFileDialogOpen **constant**, msoFileDialog **object**, 234
- msoFileDialogSaveAs **constant**, msoFileDialog **object**, 234
- MsoSync **event type**, 663
- Multi page control**, 496
- multi-column ListBox**, 279
- multi-dimensional arrays**, 57
- Multidimensional Expression (MDX)**
 - behind OLAP-based pivot tables, 512–517
 - creating MDX log, 515–517
 - deciphering MDX queries, 515
- MultiLevel **property**, TickLabels **object**, 914
- multilingual application**, 562–564
- multilingual environment**
 - allowing extra space, 564
 - using Excel's objects, 564
 - using RibbonX, 565
 - using SendKeys, 565
- MultiNomial**, 965
- multiple recordsets**, 460–461
- MultiSelect**, 236–238, 658
- MultiSelect **property**, 236–237
- MultiThreadedCalculation **object**, 651, 783
- MultiThreadedCalculationCalculation **property**, Application **object**, 651
- MultiUserEditing **property**, Workbook **object**, 940
- My Network checkbox**, 493
- MyMap **object**, 257
- MyTable **element**, 241–243
- MyTableroot **element**, 242

N**Name box, 3, 125, 126, 185, 186****Name dialog box, 652****Name Manager dialog box, 125, 126, 129, 132**Name **object, 126, 127, 132, 134, 135, 137, 783–785**Name **parameter, 603, 692, 833, 835, 841, 868, 901, 917**Name **property**

Action object, 639

Add-In, 639

Application object, 651

AxisTitle object, 675

CalculatedMember object, 678

Chart object, 688

ChartArea object, 697

CodeModule object, 976

CommandBar object, 1002

CustomXMLValidationError, 1024

DocumentProperty object, 1030

Employee object, 356, 360

Font2 object, 1036

MetaProperty object, 1045

Name object, 127, 137

NewMonth object, 87

OLEObjects collection, 218

PolicyItem object, 1051

Property object, 983

QueryTable object, 847

Range object, 128, 132–135, 855

RecentFile object, 868

Reference object, 984

Scenario object, 871

ScopeFolder object, 1053

Series object, 875

SeriesLines object, 877

ServerPolicy object, 1050

Shape object, 883

ShapeRange object, 888

SharedWorkspace object, 1055

SharedWorkspaceMember object, 1060

SmartTag object, 894

SmartTagAction object, 895

Style object, 902

TableStyle object, 905

ThemeFont object, 1074

TickLabels object, 914

Trendline object, 918

UpBars object, 921

UserAccess Collection, 921

VBcomponent object, 987

VBproject object, 991

Walls object, 926

Workbook object, 23, 75, 79, 940

WorkflowTask object, 1077

WorkflowTemplate object, 1078

Worksheet object, 956

XmlMap object, 968

XMLSchema object, 970

Name Table, 783NameAscii **property, Font2 object, 1036**NameComplex Script **property, Font2 object, 1036**NameFarEast **property, Font2 object, 1036**NameIsAuto **property, Trendline object, 918**NameLocal **property**

CommandBar object, 1002

Style object, 902

TableStyle object, 905

NameOther **property, Font2 object, 1036****names, 125–139**

hiding names, 131–132

naming ranges, 127–128

searching for, 133–139

special names, 128–129

storing arrays, 130–131

storing values in, 129

storing values in names, 129–130

working with named ranges, 132–133

Names **collection, 67, 126–129, 783–785****Names dialog box, 784****Names procedure, 58**Names **property**

Application object, 651

Workbook object, 940

Worksheet object, 956

Namespace Manager **property, CustomXMLPart object, 1021**NameSpace **object, 413**NameSpace **property, XMLSchema object, 970****namespaces, 243–245, 266, 267, 296, 308, 412, 415, 423, 968–970, 1017, 1018, 1020–1023, 1039****NameSpaces, XML, 243–245**NameSpaceURI **property**

CustomXMLNode object, 1017

CustomXMLPart object, 1021

CustomXMLPrefixMapping collection, 1023

CustomXMLSchema object, 1023

naming conventions, **variables, 44–45**NavigateArrow **method, Range object, 863****NavigKeys, 654**NegBinomDist, **965**Neither **method, 11**NETAPI32.DLL **file, 602****Network Library connection, 455****NetworkDays, 965**NetworkTemplatesPath **property, Application object, 651**

New button

New button, 126

New keyword, 415–417, 430, 643

New operator, 598

NewFile object, 1046

NewSeries method, 190, 192, 197

NewSession method, EncryptionProvider object, 1031

NewSheet event, Workbook object, 952

NewWindow method

Window object, 933

Workbook object, 946

NewWorkbook object event, Application object, 661

NewWorkbook property, Application object, 651

Next property

Chart object, 688

Range object, 855

Worksheet object, 956

Next statement, 53, 69, 87, 106, 134

NextLetter method, Application object, 660

NextRecordset method, 444–445, 461

NextSibling property, CustomXMLNode object, 1017

Node property, CustomXMLValidationError object, 1024

NodeAfterDelete event, CustomXMLPart object, 1022

NodeAfterInsert event, CustomXMLPart object, 1022

NodeAfterReplace event, CustomXMLPart object, 1022

Nodes property

Shape object, 883

ShapeRange object, 888

NodeType property, CustomXMLNode object, 1017

NodeValue property, CustomXMLNode object, 1017

Nominal, 965

NON EMPTY keyword, 516

non-contiguous range, 95, 113, 114, 852

non-standard data sources, using ADO with, 463–468

inserting and updating records in workbooks, 466–467

querying text files, 467–468

querying workbooks, 464–466

NormalizedHeight property, TextEffectFormat object, 907

Norwegian number format character, 558

Norwegian number formats, 556

Norwegian settings, 541, 542, 544

Norwegian-formatted date, 540

NOT operators, 443

Notepad document, 70

Notes property, SharedWorkspaceLink object, 1059

NoteText method, Range object, 863

Notification option, 496, 497

NotifySignatureAdded method, SignatureProvider object, 1065

Now function, 616

Number format, 25, 153, 169, 170, 420, 547, 548, 551, 558, 637, 683, 730, 732, 751, 816

Number format dialog box, 542, 547

Number format groups, 297

Number property

BulletFormat2 object, 996

TextColumn2 object, 1069

NumberFormat Local property, CellFormat object, 683

NumberFormat property, 15, 24, 153, 168, 548, 637, 902, 914, 915, 919

NumberFormatLinked property, TickLabels object, 914

NumberFormatLocal property

Range object, 855

Style object, 902

TickLabels object, 914

numbers, deleting, 107

NumIndices property, Property object, 983

O

Object Browser, 27–29, 62, 63, 186, 208, 214, 345, 412, 430, 539, 562, 580, 595

Object Browser button, 27

Object data type, 43

object model, Excel 2007

AboveAverage object, 636–638

Action object, 638–639

Actions collection, 638–639

Add-In object, 639

Addins collection, 639

Adjustments object, 640–641

AllowEditRange object, 642

AllowEditRanges collection, 641

Application object, 642–664

events, 661–663

example, 664

methods, 656–661

properties, 643–656

Areas collection, 664–665

AutoCorrect object, 665–667

AutoFilter object, 667–669

AutoRecover object, 669–670

Axis object and Axes collection, 670–674

AxisTitle object, 674–675

Border object and Borders collection, 676–677

CalculatedFields collection, 677

CalculatedItems collection, 678

CalculatedMember object, 678–679

CalculatedMembers collection, 679–680

CalloutFormat object, 680–682

CellFormat object, 682–684

Characters object, 684–685

Chart object

events, 694–695

example, 696

- methods, 690–694
- properties, 687–690
- ChartArea object, 696–697
- ChartColorFormat object, 698
- ChartFillFormat object, 698–700
- ChartFormat object, 700–701
- ChartGroup object, 701–704
- ChartObject object, 706–708
- ChartObjects collection, 704–706
- Charts collection, 686–687
- ChartTitle object, 708–710
- ChartView object, 710
- collection methods, 636
- collection properties, 635
- ColorFormat object, 710–711
- ColorScale object, 711–713
- ColorScaleCriteria collection, 713
- Comment object and Comments collection, 714–716
- Comment object and Comments collection, 714–716
- ConditionValue object, 716
- Connections object, 716–717
- ConnectorFormat object, 717–719
- ControlFormat object, 719–721
- CubeField object and CubeFields collection, 721–724
- CustomProperty object and CustomProperties collection, 724–726
- CustomView object and CustomViews collection, 726–727
- Databar object, 727–729
- DataLabel object and DataLabels collection, 729–734
- DataTable object, 734–735
- DefaultWebOptions object, 735–737
- Dialog object and Dialogs collection, 737–738
- DisplayUnitLabel object, 738–739
- DownBars object, 740
- DropLines object, 740–741
- Error object and Errors collection, 741–742
- ErrorBars object, 742–743
- ErrorCheckingOptions collection object, 743–744
- FillFormat object, 744–746
- Filter object and Filters collection, 746–747
- Floor object, 747–748
- Font object, 748–749
- FormatColor object, 749–750
- FormatCondition object and FormatConditions collection, 750–753
- FreeformBuilder object, 753–754
- Graphic object, 754–756
- Gridlines object, 756–757
- GroupShapes collection, 757
- HeaderFooter object, 757–758
- HiLoLines object, 758
- HPageBreak object and HPageBreaks collection, 758–759
- Hyperlink object and Hyperlinks collection, 759–761
- Icon object, 761–762
- IconCriteria and IconCriteria collection, 762
- IconSet and IconSets collection, 762–763
- IconSetCondition object, 763–765
- Interior object, 765–767
- IRtdServer object, 767–768
- IRTDUpdateEvent object, 768
- LeaderLines object, 769
- Legend object, 770–771
- LegendEntry object and LegendEntries collection, 771–773
- LegendKey object, 773–774
- LinearGradient object, 774
- LineFormat object, 775–776
- LinkFormat object, 776–777
- ListColumn and ListColumns collection, 777
- ListColumns Common Properties, 777–778
- ListDataFormat object, 778–779
- ListObject object and ListObjects collection, 779–781
- Mailer object, 782
- MultiThreadedCalculation object, 783
- Name object and Names collection, 783–785
- object properties, 636
- ODBCConnection object, 785–787
- ODBCError object and ODBCErrors collection, 787–788
- OLEDBConnection object, 788–790
- OLEDBError object and OLEDBErrors collection, 790–791
- OLEFormat object, 791–792
- OLEObject object and OLEObjects collection, 792–796
- Outline object, 796–797
- Page object and Pages collection, 797–798
- PageSetup object, 798–801
- Pane object and Panes collection, 802–803
- Parameter object and Parameters collection, 803–804
- Phonetic object and Phonetics collection, 804–805
- PictureFormat object, 806–807
- PivotAxis object, 807
- PivotCache object and PivotCaches collection, 807–811
- PivotCell object, 811–812
- PivotField object, PivotFields collection, and CalculatedFields collection, 812–819
- PivotFilter object and PivotFilters collection, 819–821
- PivotFormula object and PivotFormulas collection, 821–822
- PivotItem object, PivotItems collection, and CalculatedItems collection, 822–823
- PivotItemList object, 824
- PivotLayout object, 824–825

object model, Excel 2007 (continued)

object model, Excel 2007 (continued)

PivotLine object, PivotLines collection, and PivotLinesCells collection, 825

PivotTable object and PivotTables collection, 825–837

PlotArea object, 838–839

Point object and Points collection, 839–842

Protection object, 842–844

PublishObject object and PublishObjects collection, 844–846

QueryTable object and QueryTables collection, 846–852

Range object and Ranges collection object, 852–868

- methods, 858–868
- overview, 852
- properties, 852–857

Real-Time Data (RTD) object, 870

RecentFile object and RecentFiles collection, 868–869

RectangularGradient object, 869

RoutingSlip object, 869–870

Scenario object and Scenarios collection, 871–872

Series object and SeriesCollection collection, 872–877

SeriesLines object, 877

ServerViewableItems collection, 878

ShadowFormat object, 878–879

Shape object and Shapes collection, 880–885

ShapeNode object and ShapeNodes collection, 886–887

ShapeRange object collection, 887–891

Sheets collection, 891–893

- overview, 891
- Sheets common properties, 891
- Sheets methods, 892–893
- Sheets properties, 892

SheetViews object, 893

SmartTag object and SmartTags collection object, 893–894

SmartTagAction object and SmartTagActions collection object, 894–895

SmartTagOptions collection object, 895

SmartTagRecognizer object and SmartTagRecognizers collection object, 895–896

Sort object, 896–897

SortField object and SortFields collection, 897–898

SoundNote object, 898

Speech object, 898–899

SpellingOptions collection object, 899–901

Style object and Styles collection, 901–903

Tab object, 903–904

TableStyle object and TableStyles collection object, 904–906

TableStyleElement object and TableStyleElements collection object, 906

TextEffectFormat object, 907–908

TextFrame object, 908–909

TextFrame2 object, 909–911

ThreeDFormat object, 911–913

TickLabels object, 913–915

Top10 object, 915–917

TreeViewControl object, 917

Trendline object and Trendlines collection, 917–919

UniqueValues object, 919–920

UpBars object, 920–921

UsedObjects collection object, 921–922

UserAccessList collection object, 922

Validation object, 923–924

VPageBreak object and VPageBreaks collection, 924–925

Walls object, 925–926

Watch object and Watches collection object, 926–928

WebOptions object, 928–929

Window object and Windows collection, 929–935

Workbook object and Workbooks collection, 935–953

- events, 951–953
- methods, 935–937, 944–951
- overview, 935
- properties, 935–943

WorkbookConnection object, 953

Worksheet object, 953–963

- events, 962–963
- methods, 958–962
- properties, 955–958

WorksheetFunction object, 963–966

Worksheets collection, 954–955

WorksheetView object, 966–967

XmlDataBinding object, 967

XmlMap object and XMLMaps collection, 967–969

XmlNameSpace object and XMLNameSpaces collection, 969

XmlSchema object and XmlSchemas collection, 969–970

XPath object, 970

object model, Office 2007, 995–1078

BulletFormat2 object, 996–997

COMAddinObject and COMAddins collection object, 997–999

CommandBar object and CommandBars collection object, 999–1002

CommandBarButton object, 1003–1006

CommandBarComboBox object, 1006–1009

CommandBarControl object and CommandBarControls collection object, 1010–1013

CommandBarPopup object, 1013–1015

- common properties, 995–996

CustomTaskPane object, 1016

- CustomXMLNode object and CustomXMLNodes collection object, 1017–1020
- CustomXMLPart object and CustomXMLParts collection object, 1020–1022
- CustomXMLPrefixMapping object and CustomXMLPrefixMappings collection object, 1022–1023
- CustomXMLSchema object and CustomXMLSchemaCollection object, 1023–1024
- CustomXMLValidationError object and CustomXMLValidationErrors collection object, 1024–1025
- DocumentInspector object and DocumentInspectors collection object, 1025–1026
- DocumentLibraryVersion object and DocumentLibraryVersions collection object, 1027
- DocumentProperty object and DocumentProperties collection object, 1028–1030
- EncryptionProvider object, 1030–1031
- FileDialog object, 1031–1033
- FileDialogFilter object and FileDialogFilters collection object, 1033–1034
- FileDialogSelectedItems collection object, 1034
- FileTypes object, 1034–1035
- Font2 object, 1035–1037
- GlowFormat object, 1037
- GradientStop object and GradientStops collection object, 1037–1038
- IAssistance object, 1038–1039
- IBlogExtensibility and IBlogPictureExtensibility objects, 1039–1041
- ICTPFFactory object, 1041
- ICustomTaskPaneConsumer object, 1041–1042
- IDocumentInspector object, 1042
- IRibbonControl object, 1042
- IRibbonExtensibility object, 1043
- IRibbonUI object, 1043
- LanguageSettings object, 1043–1044
- MetaProperty object and MetaProperties collection object, 1044–1045
- MsoEnvelope object, 1045–1046
- NewFile object, 1046
- ODSOColumn object and ODSOColumns collection object, 1047
- ODSOFilter object and ODSOFilters collection object, 1047
- OfficeDataSourceObject object, 1047
- OfficeTheme object, 1047
- ParagraphFormat2 object, 1048–1049
- Permission object, 1049–1050
- PolicyItem object and ServerPolicy collection object, 1050–1051
- ReflectionFormat object, 1051
- Ruler2 object, 1051
- RulerLevel2 object and RulerLevels2 collection object, 1052
- ScopeFolder object and ScopeFolders collection object, 1052–1053
- SearchFolders collection object, 1053
- SearchScope object and SearchScopes collection object, 1054
- SharedWorkspace object, 1054–1055
- SharedWorkspaceFile object and SharedWorkspaceFiles collection object, 1055–1056
- SharedWorkspaceFolder object and SharedWorkspaceFolders collection object, 1057
- SharedWorkspaceLink object and SharedWorkspaceLinks collection object, 1058–1059
- SharedWorkspaceMember object and SharedWorkspaceMembers collection object, 1059–1060
- SharedWorkspaceTask object and SharedWorkspaceTasks collection object, 1060–1061
- Signature object and SignatureSet collection object, 1061–1063
- SignatureInfo object, 1063–1064
- SignatureProvider object, 1064–1066
- SignatureSetup object, 1066
- SmartDocument object, 1066–1067
- SoftEdgeFormat object, 1067
- Sync object, 1067–1068
- TabStop2 object and TabStops2 collection object, 1068–1069
- TextColumn2 object and TextColumns2 collection object, 1069
- TextRange2 object, 1070–1072
- ThemeColor object, 1072
- ThemeColorsScheme object, 1072–1073
- ThemeEffectScheme object, 1073
- ThemeFont object and ThemeFonts collection object, 1073–1074
- ThemeFontScheme object, 1074
- UserPermission object, 1074–1075
- WebPageFont object and WebPageFonts collection object, 1075–1076
- WorkflowTask object and WorkflowTasks collection object, 1076–1077
- WorkflowTemplate object and WorkflowTemplates collection object, 1077–1078
- object model, VBE, 971–994**
 - AddIn object and Add-Ins collection, 973–974
 - CodeModule object, 974–978
 - CodePane object and CodePanels collection, 978–980
 - CommandBarEvents object, 980–981
 - common properties and methods, 972–973
 - End FunctionReferencesEvents object, 986
 - Events object, 981–982
 - LinkedWindows collection, 982
 - links between Excel and, 971–972
 - overview, 971

object model, VBE (continued)

object model, VBE (continued)

- Property object and Properties collection, 982–983
- Reference object and References collection, 984–986
- VBComponent object and VBComponents collection, 986–989
- VBE object, 989–990
- VBProject object and VBProjects collection, 990–992
- Window object and Windows collection, 992–994
- object properties, 636**
- Object **property, 218, 401, 408, 437, 973, 983, 998**
- Object **statement, 91**
- Object type, 44, 45, 276, 306, 415, 639, 690**
- Object types, 90, 371, 413, 417, 430**
- object variables, 45–47**
- objectinput argument, 413**
- objects, 22–27**
 - collections, 22–23
 - events, 26–27
 - methods, 25
 - properties, 23–25
- Obscured **property, ShadowFormat object, 879**
- ODBC driver, 476**
- ODBC query, 786, 787**
- ODBCConnection methods, 787**
- ODBCConnection **object, 489, 785–787, 790**
- ODBCError **object, 787–788, 790**
- ODBCErrors **collection, 787–788**
- ODBCErrors **property, Application object, 651**
- ODBCTimeout **property, Application object, 651**
- ODC (Office Data Connect) files, 484–486**
- ODSColumn **object and ODSColumns collection, 1047**
- ODSFilter **object and ODSFilters collection, 1047**
- Office 2007**
 - Custom UI editor, 294–296, 305, 306
- Office 2007 language settings**
 - creating multilingual application, 562–564
 - identifying, 561–562
 - and regional settings, 537–538
 - rules for developing multilingual application, 565
 - where text comes from, 560–561
 - language version of Windows, 561
 - Office UI language settings, 561
 - Regional Settings location, 560–561
 - working in multilingual environment, 564–565
 - allowing extra space, 564
 - using Excel's objects, 564
 - using RibbonX, 565
 - using SendKeys, 565
- Office 2007 object model, 995–1078**
 - BulletFormat2 object, 996–997
 - COMAddinObject and COMAddins collection object, 997–999
 - CommandBar object and CommandBars collection object, 999–1002
 - CommandBarButton object, 1003–1006
 - CommandBarComboBox object, 1006–1009
 - CommandBarControl object and CommandBarControls collection object, 1010–1013
 - CommandBarPopup object, 1013–1015
 - common properties, 995–996
 - CustomTaskPane object, 1016
 - CustomXMLNode object and CustomXMLNodes collection object, 1017–1020
 - CustomXMLPart object and CustomXMLParts collection object, 1020–1022
 - CustomXMLPrefixMapping object and CustomXMLPrefixMappings collection object, 1022–1023
 - CustomXMLSchema object and CustomXMLSchemaCollection object, 1023–1024
 - CustomXMLValidationError object and CustomXMLValidationErrors CollectionObject, 1024–1025
 - DocumentInspector object and DocumentInspectors collection object, 1025–1026
 - DocumentLibraryVersion object and DocumentLibraryVersions collection object, 1027
 - DocumentProperty object and DocumentProperties collection object, 1028–1030
 - EncryptionProvider object, 1030–1031
 - FileDialog object, 1031–1033
 - FileDialogFilter object and FileDialogFilters collection object, 1033–1034
 - FileDialogSelectedItems collection object, 1034
 - FileTypes object, 1034–1035
 - Font2 object, 1035–1037
 - GlowFormat object, 1037
 - GradientStop object and GradientStops collection object, 1037–1038
 - IAssistance object, 1038–1039
 - IBlogExtensibility and IBlogPictureExtensibility objects, 1039–1041
 - ICTPFactory object, 1041
 - ICustomTaskPaneConsumer object, 1041–1042
 - IDocumentInspector object, 1042
 - IRibbonControl object, 1042
 - IRibbonExtensibility object, 1043
 - IRibbonUI object, 1043
 - LanguageSettings object, 1043–1044
 - MetaProperty object and MetaProperties collection object, 1044–1045
 - MsoEnvelope object, 1045–1046
 - NewFile object, 1046
 - ODSColumn object and ODSColumns collection object, 1047

- ODSOFilter object and ODSOFilters collection object, 1047
- OfficeDataSourceObject object, 1047
- OfficeTheme object, 1047
- ParagraphFormat2 object, 1048–1049
- Permission object, 1049–1050
- PolicyItem object and ServerPolicy collection object, 1050–1051
- ReflectionFormat object, 1051
- Ruler2 object, 1051
- RulerLevel2 object and RulerLevels2 collection object, 1052
- ScopeFolder object and ScopeFolders collection object, 1052–1053
- SearchFolders collection object, 1053
- SearchScope object and SearchScopes collection object, 1054
- SharedWorkspace object, 1054–1055
- SharedWorkspaceFile object and SharedWorkspaceFiles collection object, 1055–1056
- SharedWorkspaceFolder object and SharedWorkspaceFolders collection object, 1057
- SharedWorkspaceLink object and SharedWorkspaceLinks collection object, 1058–1059
- SharedWorkspaceMember object and SharedWorkspaceMembers collection object, 1059–1060
- SharedWorkspaceTask object and SharedWorkspaceTasks collection object, 1060–1061
- Signature object and SignatureSet collection object, 1061–1063
- SignatureInfo object, 1063–1064
- SignatureProvider object, 1064–1066
- SignatureSetup object, 1066
- SmartDocument object, 1066–1067
- SoftEdgeFormat object, 1067
- Sync object, 1067–1068
- TabStop2 object and TabStops2 collection object, 1068–1069
- TextColumn2 object and TextColumns2 collection object, 1069
- TextRange2 object, 1070–1072
- ThemeColor object, 1072
- ThemeColorsScheme object, 1072–1073
- ThemeEffectScheme object, 1073
- ThemeFont object and ThemeFonts collection object, 1073–1074
- ThemeFontScheme object, 1074
- UserPermission object, 1074–1075
- WebPageFont object and WebPageFonts collection object, 1075–1076
- WorkflowTask object and WorkflowTasks collection object, 1076–1077
- WorkflowTemplate object and WorkflowTemplates collection object, 1077–1078
- Office Data Connect (ODC) files, 484–486
- Office language pack, 545**
- Office language version, 542**
- Office library folder, 492**
- Office Menu, 294–296, 312, 527, 552**
- Office menu, customizing, 312**
- Office UI Language, 549, 557, 561–564, 568**
- Office XP setting, 648**
- OfficeDataSourceObject object, 1047
- OfficeTheme object, 1047
- Offline Cube file, 521, 717, 788, 808
- offline cubes, creating, 521–523**
 - manually, 521
 - using CreateCubeFile method, 521–522
- Offset parameters, 102
- Offset property, 102–103, 124, 855, 914
- OffsetX property, ShadowFormat object, 879
- OffsetY property, ShadowFormat object, 879
- OLAP cube, 509–512, 515, 519, 721**
- OLAP data sources, 507–523**
 - analyzing OLAP data via pivot tables, 508–512
 - browsing OLAP data source, 510–512
 - connecting to OLAP data source, 508–509
 - browsing without pivot tables, 517–520
 - creating inventory of dimensions, hierarchies, and levels, 519–520
 - using ADO MD to get cube schema information, 518–519
 - using ADO to return flattened recordsets, 517–518
 - creating offline cubes, 521–523
 - creating offline cube using ADO MD and VBA, 522–523
 - manually, 521
 - using CreateCubeFile method, 521–522
 - MDX behind OLAP-based pivot tables, 512–517
 - creating MDX log, 515–517
 - deciphering MDX queries, 515
- OLAP database, 507, 508, 513**
- OLAP server, 512, 515, 517–519, 789, 790, 808, 816**
- OLAP-based pivot tables, 512, 515, 517, 521, 522, 523**
- OLE DB providers, 436, 438–440, 448, 452, 455, 460, 464, 466, 476, 790, 791**
- OLE object, 880
- OLEDB connection, 651**
- OLEDBConnection object, 489, 788–790
- OLEDBError object, 790–791
- OLEDBErrors collection, 790–791
- OLEDBErrors property, Application object, 651
- OLEFormat object, 791–792
- OLEFormat property, Shape object, 883
- OLEMenuGroup property, CommandBarPopup object, 1014

OLEObject collection

OLEObject **collection**, 218, 792, 793
OLEObject **methods**, 796
OLEObject **object**, 217, 218, 219, 221, 676, 692, 754, 765, 776, 792–796
OLEObjects **collection**, 792–796
OLEObjects **method**
 Chart object, 692
 Worksheet object, 959
OLEUsage **property**
 CommandBarButton object, 1004
 CommandBarComboBox object, 1008
 CommandBarControl object, 1011
 CommandBarPopup object, 1014
OLTP database, 507
On Error Resume Next **statement**, 61–62
On Error statement, 59, 60, 62, 107, 170
OnAction callback, 304
OnAction macros, 332–333
OnAction **property**, 210, 216, 220, 322, 332, 333, 349, 398, 576, 577, 578, 579, 883, 999, 1004, 1008, 1012, 1014
OnAddInsUpdate **method**, 395
OnBeginShutdown **method**, 395
onChange callback, 304
OnConnection **method**, 388, 395, 397, 399
OnDisconnection **event**, 400
OnDisconnection **method**, 395, 397, 400
OneColorGradient **method**, ChartFillFormat **object**, 699
OnKey **method**, 72–73
onLoad callback, 309, 310
OnRepeat **method**, Application **object**, 660
OnStartupComplete **method**, 395
OnTime **method**, 71–72, 660, 767
OnUndo **method**, Application **object**, 660
OnUpdate **event**, CommandBars **collection**, 1001
OnWindow **property**
 Application object, 652
 Window object, 931
Open button, 235
Open dialog box, 233–236, 584, 658, 664, 755
Open **event**, 16, 150, 206, 338, 346, 351, 365, 577, 584, 952
Open **method**, 78, 80, 179, 417, 429, 439, 443, 450, 451, 462, 868, 936, 1027, 1040
Open **statement**, 223
Open XML files, using VBA to program, 265–272
OpenDatabase **method**, Workbook **object**, 936
opening text files, 223
OpenLinks **method**, Workbook **object**, 946
OpenText **method**, 482, 555–556, 560, 561, 937
OpenVersion **method**, Sync **object**, 1068

OpenXML **method**, 253, 937
OperatingSystem **property**, Application **object**, 652
Operator **property**, 751, 923
operator text, 752
Optimize memory checkbox, 512
option Base setting, 118, 190
option Base **statement**, 55, 56, 216
Option Button controls, 212–214
option buttons, 12, 210, 211, 213, 214, 221, 275, 276, 281, 590
Option Explicit **statement**, 39–40, 41, 278, 362, 575–577, 581, 583, 591, 599, 611, 616, 618, 620, 628
OptionButton **control**, 276
options Base **statement**, 56
options button, 489, 491, 499, 500, 648, 666
OR operator, 157, 440
Order **parameter**, 917
Order **property**
 SortFields collection, 897
 Trendline object, 918
OrganizationName **property**, Application **object**, 652
OrganizeInFolder **property**, WebOptions **object**, 928
Orientation **property**
 AxisTitle object, 675
 CellFormat object, 683
 Range object, 855
 Sort object, 896
 Style object, 902
 TextFrame object, 908
 TextFrame2 object, 910
 TickLabels object, 914
OUT parameters, 605
Outline object, 796–797
Outline **property**, Worksheet **object**, 956
OutlineLevel **property**, Range **object**, 855
Outlook, 21, 411–413, 415, 420–423, 659, 971, 997, 1039, 1045, 1054
Overwrite **parameter**, 715
overwriting, Workbook **object**, 81–82
OwnerDocument **property**, CustomXMLNode **object**, 1017
OwnerPart **property**, CustomXMLNode **object**, 1018

P

page Break view, 340, 341
page fields, 164, 166, 167, 813, 828, 829, 831, 832, 836
page **object**, 797
Page **object** and Pages **collection**, 797–798
Page Setup command, 754
PageBreak **property**, Range **object**, 855
pages **collection**, 797, 800
PageSetup **object**, 53, 549, 798–801

- PageSetup **property**
 - Chart object, 688
 - Worksheet object, 957
- Pane **object**, 802–803
- Panes **collection**, 802–803
- Panes **property**, Windows **object**, 932
- paper sizes**, 549
- Paragraph Format **property**, TextRange2 **object**, 1070
- ParagraphFormat2 **object**, 1048–1049
- Paragraphs **property**, TextRange2 **object**, 1070
- Parameter **data type**, 603
- Parameter **method**, 447
- Parameter **object**, 446, 447, 477, 478, 803
- Parameter **object and Parameters collection**, 803–804
- Parameter properties**, 332
- Parameter **property**, 332, 333, 345, 349, 399, 400, 1004, 1008, 1012, 1014
- parameter queries**, 476–479, 484
- parameter types, declaring**, 44
- Parameter value**, 333, 337
- ParameterDirectionEnum value**, 446
- parameters**
 - passing parameter values, 333–334
 - specified by name, 31–33
 - specified by position, 31
- Parameters argument**, 447
- Parameters **collection**, 446, 447, 457, 460, 477, 803
- Parameters **property**, QueryTable **object**, 847
- Parent **property**, 635, 636, 995, 996
- ParentGroup **property**
 - Shape object, 883
 - ShapeRange object, 888
- parentheses**, 34, 37, 38
- ParentNode **property**, CustomXMLNode **object**, 1018
- Parse **method**, Range **object**, 863
- PartAfterAdd **event**, CustomXMLPart **objects collection**, 1020
- PartAfterLoad **event**, CustomXMLPart **objects collection**, 1020
- PartBeforeDelete **event**, CustomXMLPart **objects collection**, 1020
- Password **property**, Workbook **object**, 940
- PasswordEncryptionAlgorithm **property**, Workbook **object**, 940
- PasswordEncryptionFileProperties **property**, Workbook **object**, 941
- PasswordEncryptionKeyLength **property**, Workbook **object**, 941
- PasswordEncryptionProvider **property**, Workbook **object**, 941
- Paste **method**
 - Chart object, 693
 - SeriesCollection object, 873, 876
 - TextRange2 object, 1071
 - Walls object, 926
 - Worksheet object, 959
- PasteFace **method**, CommandBarButton **object**, 1005
- PasteSpecial **method**
 - Range object, 863
 - TextRange2 object, 1071
 - Worksheet object, 959
- pasting text**, 557
- Path **property**
 - Add-In, 639
 - Application object, 652
 - AutoRecover object, 669
 - RecentFile object, 868
 - ScopeFolder object, 1053
 - Workbook object, 941
- Pathdoubled **property**, Workbook **object**, 941
- PathFormat **property**, TextFrame **object2**, 910
- paths, getting filename from**, 78–80
- PathSeparator **property**, Application **object**, 652
- Pattern **property**, ChartFillFormat **object**, 699
- Patterned **method**, ChartFillFormat **object**, 699
- Percent **property**, Top10 **object**, 915
- Percentage format**, 644
- Period **parameter**, 917
- Period **property**, Trendline **object**, 918
- Periods **parameter array**, 174
- Permission **object**, 1049–1050
- Permission **property**
 - UserPermission object, 1075
 - Workbook object, 941
- PermissionFromPolicy **property**, Permission **object**, 1049
- Personal Macro Workbook**, 5–6
- PersonalViewListSettings **property**, Workbook **object**, 941
- PersonalViewPrintSettings **property**, Workbook **object**, 941
- Personal.xlsb, 5, 6
- Perspective **property**
 - Chart object, 688
 - ThreeDFormat object, 912
- Phonetic**, 965
- Phonetic **object**, 804–805
- Phonetic **property**, Range **object**, 855
- PhoneticCharacters **property**, 685
- Phonetics **collection**, 804–805
- Phonetics **property**, Range **object**, 856
- PickSolution **method**, SmartDocument **object**, 1067
- PickUp **method**
 - Shape object, 885
 - ShapeRange object, 890
- Picture bulletformat2 **methods**, 997

Picture buttons

Picture buttons, 754

Picture file format, 700, 746

Picture format **object**, 700, 806

Picture **object**, 348, 758, 880, 1041

Picture **property**, `CommandBarButton` **object**, 1004

Picture type **property**, 774, 841

`PictureFormat` **object**, 806–807

`PictureFormat` **property**

`ChartFormat` **object**, 700

 Shape **object**, 883

`ShapeRange` **object**, 888

`PictureType` **property**

 Series **object**, 875

 Walls **object**, 926

`PictureUnit` **property**, Walls **object**, 926

`PictureUnit2` **property**, Series **object**, 875

Pivot cache, methods, 810

Pivot cache **object**, 164, 165, 487, 807, 809, 810

Pivot caches **collection**, 164, 165, 807

Pivot field **object**, 166, 167, 168, 170, 171, 174, 677, 812, 816, 819, 822

Pivot field **property**, 815

Pivot fields **collection**, 166, 170, 677, 723, 812, 819

Pivot item **object**, 678, 817, 822

Pivot itemList **object**, 824

Pivot items **collection**, 171, 176, 678, 822, 823

pivot tables

 analyzing OLAP data via, 508–512

 browsing OLAP data source, 510–512

 connecting to OLAP data source, 508–509

 browsing without, 517–520

 creating inventory of dimensions, hierarchies, and levels, 519–520

 using ADO MD to get cube schema information, 518–519

 using ADO to return flattened recordsets, 517–518

 OLAP-based, MDX behind, 512–517

 creating MDX log, 515–517

 deciphering MDX queries, 515

`PivotAxis` **object**, 807, 829

`PivotCache` **object**, 807–811

`PivotCaches` **collection**, 165, 807–811

`PivotCaches` **method**, `Workbook` **object**, 946

`PivotCell` **object**, 811–812, 824

`PivotCell` **property**, Range **object**, 856

`PivotCells` **collection**, 825

`PivotCharts`, 177–178

`PivotField` **object**, 166, 812–819

`PivotField` **property**, Range **object**, 856

`PivotFields`, 166–171

`PivotFields` **collection**, 677, 812–819

`PivotFilter` **methods**, 821

`PivotFilter` **object**, 819–821

`PivotFilter` **properties**, 820

`PivotFilters` **collection**, 818, 819–821, 832

`PivotFormula` **methods**, 822

`PivotFormula` **object**, 821–822

`PivotFormulas` **collection**, 821

`PivotItem` **object**, 822–823

`PivotItem` **property**, Range **object**, 856

`PivotItemList` **object**, 824

PivotItems, 171–176

`CalculatedItems` **collection**, 176

 grouping, 171–175

 Visible **property**, 175–176

`PivotItems` **collection**, 678, 822–823

`PivotLayout` **object**, 177, 180, 824–825

`PivotLayout` **property**, Chart **object**, 689

`PivotLine` **object**, 825

`PivotLines` **collection**, 825

`PivotLinesCells` **collection**, 825

PivotTable button, 162

`PivotTable` **cache**, 808, 823, 825, 826

`PivotTable` **calculated fields and items**, 557–558

`PivotTable` **chart**, 637, 712, 728, 752, 764

`PivotTable` **data**, 177

`PivotTable` **item**, 811, 813, 815–817, 819, 822

`PivotTable` **methods**, 832, 833, 835

`PivotTable` **object**, 166, 177, 512, 516, 721, 807, 811, 812, 821, 825–837

`PivotTable` **property**, Range **object**, 856

PivotTable Query, 808

`PivotTable` **report**, 162, 166, 509, 512, 521, 637, 651, 662, 663, 679, 721–723, 807, 809, 810, 812–814, 816, 821, 825–835, 837

 creating, 162

PivotTable Wizard, 827

`PivotTableCloseConnection` **event**, `Workbook` **object**, 952

`PivotTableOpenConnection` **event**, `Workbook` **object**, 952

PivotTables, 161–180

 creating reports, 162–166

`PivotCaches`, 165

`PivotTables` **collection**, 165–166

 external data sources, 178–180

`PivotCharts`, 177–178

`PivotFields`, 166–171

`PivotItems`, 171–176

`CalculatedItems` **collection**, 176

 grouping, 171–175

 Visible **property**, 175–176

`PivotTables` **collection**, 165–166, 825, 837

`PivotTables` **method**, `Worksheet` **object**, 960

`PivotTableSelection` **property**, `Application` **object**, 652

- PivotTableUpdate **event**, Worksheet **object**, 963
- PivotTableWizard **method**, Worksheet **object**, 960
- PixelsPerInch **property**, WebOptions **object**, 928
- Placement **property**, Shape **object**, 883
- plain text queries, retrieving data from Access using, 449–450**
- Platform SDK section of MSDN Library, 602**
- Play Macro button, 9**
- Play **method**, SoundNote, 898
- PlotArea **object**, 696, 838–839
- PlotArea **property**, Chart **object**, 689
- PlotBy **parameter**, 694
- PlotBy **property**, 184, 689
- PlotOrder **property**, Series **object**, 875
- PlotVisibleOnly **property**, Chart **object**, 689
- Ply command bar, 340**
- Point **object**, 196, 197, 839–842
- Points **collection**, 196, 197, 839–842
- Points **method**, Series **object**, 876
- Points **property**, ShapeNode **object**, 886
- PointsToPixelsX **method**, Windows **object**, 934
- PointsToPixelsY **method**, Windows **object**, 934
- PolicyDescription **property**, Permission **object**, 1049
- PolicyItem **object** and ServerPolicy **collection**, 1050–1051
- PolicyName **property**, Permission **object**, 1049
- popup command bars, 342–353**
- popup menus, 319, 338–341, 345, 354, 373, 382, 398**
- popups, 320–322**
- Portable Network Graphics, 306, 736**
- Position **parameter**, 343, 934
- Position **property**, 164
- AxisTitle **object**, 675
 - CommandBar **object**, 1002
 - GradientStop **object**, 1038
 - TabStop2 **object**, 1069
- POST field, 535**
- POST format, 535**
- POST mechanism, 535–536**
- Post **method**, Workbook **object**, 946
- POSTing data, 535**
- PostText **property**, QueryTable **object**, 847
- PowerPoint, 21, 395, 411, 971, 1017, 1039, 1061**
- Precedents **property**, Range **object**, 856
- PrecisionAsDisplayed **property**, Workbook **object**, 941
- Prefix **property**
- CustomXMLPrefixMapping **collection**, 1023
 - XmlNameSpace **object**, 969
- PrefixCharacter **property**, Range **object**, 856
- PresentInPane **property**, SmartTagAction **object**, 895
- Preserve keyword, 58**
- PreserveColumn Info **property**, QueryTable **object**, 848
- PreserveColumnFilter **property**, XmlMap **object**, 968
- PreserveFormatting **property**, QueryTable **object**, 848
- PreserveNumberFormatting **property**, XmlMap **object**, 968
- PresetCamera **property**, ThreeDFormat **object**, 912
- PresetDrop **method**, CalloutFormat **object**, 681
- PresetExtrusionDirection **property**, ThreeDFormat **object**, 912
- PresetGradient **method**, ChartFillFormat **object**, 699
- PresetGradientType **property**, ChartFillFormat **object**, 699
- PresetLighting **property**, ThreeDFormat **object**, 912
- PresetLightingDirection **property**, ThreeDFormat **object**, 912
- PresetLightingSoftness **property**, ThreeDFormat **object**, 912
- PresetMaterial **property**, ThreeDFormat **object**, 912
- PresetShape **property**, TextEffectFormat **object**, 907
- PresetTextEffect **property**, TextEffectFormat **object**, 907
- PresetTexture **method**, ChartFillFormat **object**, 699
- PresetTexture **property**, ChartFillFormat **object**, 699
- PresetThreeDFormat **property**, ThreeDFormat **object**, 912
- Pressing Tab, 93**
- Preview commands, 365**
- Previous **property**
- Chart **object**, 689
 - Range **object**, 856
 - Worksheet **object**, 957
- PreviousSelections **property**, Application **object**, 652
- PreviousSibling **property**, CustomXMLNode **object**, 1018
- Print dialog, 590–593**
- Print **method**, 590
- Print Preview **method**, Sheets, 893
- Print settings, 727**
- Print **statement, writing to text files using, 227–233**
- flexible separators and delimiters, 230–233
 - reading data strings, 229–230
- Print_Area, 8, 129, 759, 783, 925
- Print_Titles, 128, 129
- PrintDataList, 331, 332
- PrintOut **method**
- Chart **object**, 693
 - Charts **collection**, 686
 - Range **object**, 863
 - Sheets **collection**, 893
 - Window **object**, 934
 - Workbook **object**, 947
 - Worksheet **object**, 960
 - Worksheets **collection**, 954

PrintPreview method

PrintPreview method

- Chart object, 693
- Charts collection, 687
- Range object, 864
- Window object, 934
- Workbook object, 947
- Worksheet object, 961
- Worksheets collection, 955

PrintTo file parameters, 686

Priority buttons, 415

Priority **property**

- AboveAverage object, 637
- CommandBarButton object, 1004
- CommandBarComboBox object, 1008
- CommandBarControl object, 1012
- CommandBarPopup object, 1015
- SharedWorkspaceTask object, 1061
- SortFields collection, 897
- Top10 object, 915
- UniqueValues object, 919

Privacy Options category, 501

privacy options, Trust Center user interface, 501–503

Private **function, 632**

ProcBodyLine **property, CodeModule object, 976**

ProcCountLine **property, CodeModule object, 976**

ProcCountLines **method, 219**

ProcOfLine Line **property, CodeModule object, 976**

ProcOfLine **method, 593**

ProcStartLine **method, 219**

ProcStartLine **property, CodeModule object, 976**

Product field, 171

ProductCode **property, Application object, 652**

ProgId **property**

- Add-In, 639, 973
- COMAddin object, 998
- SmartTagRecognizer object, 896

programmatic name, 150, 215, 291, 376, 687

progress indicator, 288–291, 445

Progressinput **parameter, 290**

Project Explorer, Visual Basic Editor (VBE), 10–11

Project Explorer window, 9–11, 18, 150, 363, 374

Project option buttons, 590

Project options, 590

Project References dialog, 406

Project Window, 580, 582, 583

PromptForSummaryInfo **property, Application object, 652**

properties, 23–25

properties button, 185, 486

Properties **collection, 441, 445, 447, 574, 596, 982–983**

properties dialog box, 250, 251, 486, 588

properties group, 144

Properties **property, SmartTag object, 894**

Properties **property, VBComponent, 987**

Properties window, 9, 11, 150, 275, 284, 345, 375, 376, 384, 397, 407, 574, 575, 580, 982

Property Browser, 580

Property Get items **procedure, 361**

Property Get **procedure, 357, 358, 371**

Property Let **procedure, 357, 358, 371**

Property **object, 982–983**

property procedures, 357

Property Set **procedures, 357**

ProportionalFont **property, WebPageFont object, 1076**

ProportionalFontSize **property, WebPageFont object, 1076**

Protect **method**

- Chart object, 693
- Workbook object, 947
- Worksheet object, 961

ProtectContents **property**

- Chart object, 689
- Worksheet object, 957

ProtectData **property, Chart object, 689**

ProtectDrawingObjects **property**

- Chart object, 689
- Worksheet object, 957

ProtectFormatting **property, Chart object, 689**

ProtectGoalSeek **property, Chart object, 689**

Protection **object, 842–844**

Protection options, 842, 843

Protection **property**

- CommandBar object, 1002
- VBproject object, 991
- Worksheet object, 957

ProtectionMode **property**

- Chart object, 689
- Worksheet object, 957

ProtectScenarios **property, Worksheet object, 957**

ProtectSelection **property, Chart object, 689**

ProtectSharing **method, Workbook object, 947**

ProtectStructure **property, Workbook object, 941**

ProtectWindows **property, Workbook object, 941**

PrTo fileName **parameter, 686**

PTCondition **property**

- AboveAverage object, 637
- Top10 object, 916
- UniqueValues object, 920

Public **property, 397, 407**

public variable, 45, 281, 563

Publish method, 844, 845

PublishObject **collection, 844**

PublishObject **object, 844–846**

PublishObjects **collection, 844–846**

PublishObjects **property, Workbook object, 941**

PublishPost **method**, IBlogExtensibility
object, 1040

PurgeChangeHistoryNow **method**, Workbook **object**, 948

PutUpdate **method**, Sync **object**, 1068

Q

QAT, 297, 307, 313, 556

qualified IDs, 308

Query add-in, 639

Query operations, 787

Query Refresh operations, 790

QueryClose **event**, 281, 282, 594, 597, 633

querying

text files, 467–468

workbooks, 464–466

QueryTable **object**, 143, 470, 472, 473, 475, 476, 480,
483, 490, 780, 803, 846–852

QueryTable **property**, Range **object**, 856

QueryTables, 472–487

and parameter queries, 476–479

associated with ListObject, 475–476

creating and using connection files, 484–489

Office Data Connect (ODC) files, 484–486

Web Query (IQY) files, 486–487

from relational database, 472–475

from text file, 482–483

from web queries, 479–482

QueryTables **collection**, 476

QueryTables **property**, Worksheet **object**, 957

QueryType **property**, QueryTable **object**, 848

Quick Access Menu, 74, 158

Quick Access Toolbar, 11, 15–16, 158

Quick Layout Button, 182

QuickSort routine, 392

Quit **method**, Application **object**, 660

R

radio buttons, 275

RadioGroupSelection **property**, SmartTagAction
object, 895

Radius **property**, GlowFormat, 1037

Randomize **statement**, 390

RandUnique **function**, 390, 392, 394, 401

RandUnique Wizard, 399

Range **object**, 22, 23, 66, 76, 93, 95, 97, 101, 112, 113,
128, 144, 165, 195, 450, 659, 664, 684, 714, 805,
852–868, 901, 923

Range **property**

AllowEditRange object, 642

Application object, 652

AutoFilter object, 668

Cells property, 97–98, 99–101

cells used in range, 98

GroupShapes collection, 757

ListObject object, 147

overview, 95–99

Range object, 97, 856

ranges of inactive worksheets, 99

ranges on inactive worksheets, 96

Shapes collection, 880

shortcut Range object references, 96

single-parameter Range object reference, 101–102

SmartTag object, 894

Worksheet object, 957

RangeFromPoint **method**, Windows **object**, 934

ranges, 93–124

Activate and Select methods, 93–94

Columns and Rows properties, 112–114

CurrentRegion property, 108–110

direct reference to, 20–21

empty cells, 115–118

End property, 110–111

named, 132–133

naming, 127–128

Offset property, 102–103

Range property, 102–103

Cells property, 97–98, 99–101

cells used in range, 98

of Range object, 97

ranges of inactive worksheets, 99

ranges on inactive worksheets, 96

shortcut Range object references, 96

single-parameter Range object reference, 101–102

Resize property, 103–105

sorting, 142–144

SpecialCells method, 105–107

deleting numbers, 107

last cell, 105–107

summing, 111

transferring values between arrays and ranges, 118–123

Union and Intersect methods, 115

used in charts, determining, 194–195

RangeSelection **property**, Windows **object**, 932

Rank **property**, Top10 **object**, 916

Rate **property**, 355

ReadingOrder **property**

AxisTitle object, 675

Range object, 856

Style object, 902

TextFrame object, 909

TickLabels object, 914

ReadOnly **property**

SignatureInfo object, 1064

SignatureSetup object, 1066

Workbook object, 941

ReadOnlyRecommended property, Workbook object

- ReadOnlyRecommended **property**, Workbook **object**, **941**
- Ready **property**, Application **object**, **652**
- Real-Time Data (RTD) **object**, **870**
- RecentFile **object**, **868–869**
- RecentFiles **collection**, **868–869**
- RecentFiles **property**, Application **object**, **652**
- RecheckSmartTags **method**, Workbook **object**, **948**
- Recipients **property**, RoutingSlip **object**, **870**
- Recognize **property**, SmartTagOptions **collection**, **896**
- Record button**, **283**
- Record **method**, SoundNote, **898**
- Record **object**, **436**
- recording macros**. *See* **macro recorder**
- RecordMacro **method**, Application **object**, **660**
- RecordRelative **property**, Application **object**, **652**
- RecordsAffected** **argument**, **439**
- RecordsAffectedand **options**, **447**
- Recordset filter**, **463**
- Recordset **object**
 - collections, **445**
 - Fields collection, **445**
 - overview, **445**
 - Properties collection, **445**
 - Connection object, **462**
 - disconnected recordset, **461–463**
 - events, **445**
 - methods, **443–445**
 - close method, **444**
 - move methods, **444**
 - nextrecordset method, **444–445**
 - open method, **443**
 - overview, **443**
 - PivotCache, **809**
 - properties, **441–443**
 - ActiveConnection property, **441–442**
 - BOF and EOF properties, **442**
 - CursorLocation property, **442**
 - Filter property, **442–443**
 - overview, **441**
 - State property, **443**
- Recordset **property**, QueryTable **object**, **848**
- recordsets**
 - disconnected, **461–463**
 - multiple, **460–461**
- RectangleBottom **property**, RectangleGradient, **869**
- RectangleGradient **property**, RectangleGradient, **869**
- RectangleLeft **property**, RectangleGradient, **869**
- RectangleRight **property**, RectangleGradient, **869**
- RectangleTop **property**, RectangleGradient, **869**
- RectangularGradient **object**, **869**
- recursion**, **200, 201, 630, 632**
- recursive **function**, **327**
- ReDim **statement**, **58**
- Reference **object** and References **collection**, **984–986**
- Reference **parameter**, **659**
- references, **598–599**
- References **collection**, **984–986**
- References dialog box**, **259, 406, 448, 594, 598**
- References **property**, VBProject **object**, **991**
- ReferencesEvents **property**, Events **object**, **982**
- ReferenceStyleStyle **property**, Application **object**, **652**
- RefersTo **property**, **62, 135**
- Reflection **property**
 - Font2 object, **1036**
 - Shape object, **883**
 - ShapeRange object, **889**
- ReflectionFormat **object**, **1051**
- Refresh All button**, **472**
- Refresh button**, **472**
- Refresh Data button**, **247**
- Refresh **method**
 - Chart object, **693**
 - QueryTable object, **473–475, 851**
 - SharedWorkspace object, **1055**
 - XmlDataBinding, **967**
- Refresh **property**, QueryTable **object**, **848**
- RefreshAll **method**, Workbook **object**, **948**
- RefreshData **method**, **RDT, 870**
- Refreshing **property**, QueryTable **object**, **848**
- RefreshOnFileOpen **property**, QueryTable **object**, **848**
- RefreshPane **method**, SmartDocument **object**, **1067**
- RefreshPeriod **property**, QueryTable **object**, **848**
- Regional options**, **538**
- regional settings**
 - and Office 2007 UI language, **537–538**
 - and Windows language, **538–545**
 - identifying, **538–539**
 - VBA conversion functions, **539–545**
- Regional Settings applet**, **538, 560**
- Regional Settings location**, **560–561**
- RegionIndex **parameter**, **759**
- RegisteredFunctions **property**, Application **object**, **652**
- RegisterXLL **method**, Application **object**, **660**
- Regroup **method**, ShapeRange **object**, **890**
- RejectAllChanges **method**, Workbook **object**, **948**
- relational databases**, **419, 470, 472, 473, 482, 490**
- relative recording**, **7–8**
- RelativeSize **property**, BulletFormat2 **object**, **996**
- ReleaseFocus **method**, CommandBars **collection**, **1001**
- Reload **method**, CustomXMLSchema, **1024**
- ReloadAs **method**, Workbook **object**, **948**
- RelyOnCSS **property**, WebOptions **object**, **928**
- RelyOnVML **property**, WebOptions **object**, **928**
- Remove All button**, **501**

Remove button, 493Remove **method**

FileTypes collection, 1035
 LinkedWindows Collection, 982
 NewFile object, 1046
 Reference collection, 985
 SearchFolders collection, 1053
 UserPermission object, 1075
 VBcomponents collection, 988
 VBprojects collection, 992

RemoveAll **method**, Permission **object**, 1050

RemoveChild **method**, 265, 1618

RemoveCustomUI **method**, 317

RemoveDocument **method**, SharedWorkspace **object**, 1055

RemoveDocument **properties method**, 504

RemoveDocumentInformation **method**, 503–505, 506, 948

RemoveDuplicates **method**, Range **object**, 864

RemoveItem **method**, CommandBarComboBox **object**, 1009

RemoveMenus procedure, 348, 351

RemovePeriods **method**, TextRange2 **object**, 1071

RemovePersonalInformation **property**, Workbook **object**, 941

RemoveSubtotal **method**, Range **object**, 864

RemoveUser **method**, Workbook **object**, 948

Repeat **method**, Application **object**, 660

Repeating **property**, XPath **object**, 970

Replace format, 653, 682, 683, 684

Replace **method**, 649, 682, 864, 1072

Replace **parameter**, 88

ReplaceChildNode **method**, CustomXMLNode **object**, 1018

ReplaceChildSubtree **method**, CustomXMLNode **object**, 1019

ReplaceFormat **property**, Application **object**, 653

ReplaceHolders **function**, 564, 568

ReplaceLine **method**, 218, 977

Replacement **parameter**, 666

ReplacementList **property**, AutoCorrect **object**, 666

ReplaceText **property**, AutoCorrect **object**, 666

Reply **method**, Workbook **object**, 948

ReplyAll **method**, Workbook **object**, 948

ReplyWithChanges **method**, Workbook **object**, 948

RepublishPost **method**, IBlogExtensibility **object**, 1040

RequestPermissionURL **property**, Permission **object**, 1049

Requirements section, 606

RerouteConnections **method**

Shape **object**, 885

ShapeRange **object**, 890

Research **property**, Workbook **object**, 941

Reset method

CommandBar **object**, 1002

CommandBarButton **object**, 1006

CommandBarComboBox **object**, 1009

CommandBarControl **object**, 1013

CommandBarPopup **object**, 1015

RoutingSlip **object**, 870

ResetAllPageBreaks **method**, Worksheet **object**, 961

ResetColors **method**, Workbook **object**, 948

ResetPositionsSideBySide **method**, Windows **object**, 930

ResetRotation **method**, ThreeDFormat **object**, 913

ResetTimer **method**, QueryTable **object**, 851

resizable UserForms, 625

Resize **event**, 625–627, 695

Resize **property**, 103–105, 124, 133, 856

ResolveConflict **method**, Sync **object**, 1068

RestartServers **method**, RTD, 870

Restore **method**, DocumentLibraryVersion **object**, 1027

ResultRange **property**, QueryTable **object**, 848

Resume **statement**, 60

return values, 33–34

ReturnWhenDone **property**, RoutingSlip **object**, 870

ReversePlotOrder **property**, Axis **object**, 673

RevisionNumber **property**, Workbook **object**, 942

RGB **function**, 676, 677, 748, 750, 766, 840, 841

RGB **property**

ChartColorFormat **object**, 698

ThemeColor **object**, 1072

Ribbon buttons, 15, 62, 274

Ribbon controls, 312, 402–403

RibbonX, 293–318

adding customizations, 294–295

CommandBar extensions for, 316

control attributes, 301–303

control callbacks, 303–305

control types, 299–301

basic controls, 299–300

container controls, 300–301

controlling tabs, tab sets, and groups, 313–314

customizing Office menu, 312

customizing QAT, 313

in dictator applications, 312

dynamic controls, 314–316

comboBox control, 315

dropDown control, 315

dynamicMenu control, 315–316

gallery, 315

hooking built-in controls, 311–312

limitations, 317

managing control images, 305–307

overview, 293–294

RibbonX (continued)

RibbonX (continued)

- prerequisites, 294
- sharing controls among multiple workbooks, 308
- updating controls at run time, 309–311
 - and VBA, 298–299
 - XML structure, 295–298

RibbonX document, 296

RibbonX toggle button, 407

Right function, 79

RightAngleAxes property, 688, 689

RightClick event, 200

RightIndent property, ParagraphFormat2 object, 1048

Rng property, Sort object, 896

RobustConnect property, 786, 789, 848

RollZoom property, Application object, 653

root element, XML, 241–242

RootElementName property, XmlMap object, 968

RootElementNamespace property, XmlMap object, 968

RotateBounds method, TextRange2 object, 1072

RotatedChars property, TextEffectFormat object, 907

RotateWithShape property, ShadowFormat object, 879

Rotation property

- Chart object, 689

- Shape object, 883

- ShapeRange object, 889

RotationX property, ThreeDFormat object, 912

RotationY property, ThreeDFormat object, 912

RotationZ property, ThreeDFormat object, 912

Route method, Workbook object, 948

Routed property, Workbook object, 942

RoutingSlip object, 869–870

RoutingSlip property, Workbook object, 942

Row fields, 166, 167, 170, 179, 652, 830, 832

Row object, 112, 113

Row property, Range object, 856

RowCol settings, 727

RowDifferences method, Range object, 864

RowHeight property, 23, 856

RowIndex property, CommandBar object, 1002

RowItems properties, 811, 824

RowLevels parameter, 797

RowNumbers property, QueryTable object, 848

rows

- deleting, 121–123, 661

- visible

 - copying, 153–154

 - finding, 154–156

rows attribute, 302

Rows property

- Application object, 112–114, 653

- Range object, 112–114, 856

- Worksheet object, 112–114, 957

RowSetComplete event, Workbook object, 952

RowSource property, 276

RTD function, 767

RTD property, Application object, 653

RTD (Real-Time Data) object, 870

RTD server, 653, 767, 768

RtlRun method, TextRange2 object, 1072

Ruler2 object, 1051

RulerLevel2 object and RulerLevels2 collection, 1052

Run method

- Application object, 660

- Range object, 864

RunAutoMacros method, Workbook object, 949

Runs property, TextRange2 object, 1070

run-time error-handling, 59–62

S

Sales.xlsx workbook, 464, 467

SatelliteDllName value, 396

Save As dialog box, 5, 233, 235, 238, 295, 527, 625, 658, 663

Save button, 235

Save method

- EncryptionProvider object, 1031

- SharedWorkspaceLink object, 1059

- SharedWorkspaceTask object, 1061

- ThemeColorsScheme object, 1073

- ThemeFontScheme object, 1074

- Workbook object, 949

Save Query button, 486

Save tab, 669

Save Workspace dialog box, 486

SaveAs filename, 46, 47, 59, 61, 65, 77, 81, 82, 693

SaveAs method

- Chart object, 693

- VBProject object, 991

- Workbook object, 556, 949

- Workbooks collection, 77

- Worksheet object, 961

SaveAsODC method, QueryTable object, 851

SaveAsXMLData method, Workbook object, 949

SaveChartTemplate method, Chart object, 694

SaveCopyAs method, Workbook object, 949

Saved property

- VBComponent object, 987

- VBProject object, 991

- Workbook object, 942

SaveData property, QueryTable object, 848

SaveDataSourceDefinition property, XmlMap object, 968

SaveLinkValues property, Workbook object, 942

SavePassword property, QueryTable object, 848

SaveWorkspace method, Application object, 660

- ScaleHeight **method**
 - Shape object, 885
 - ShapeRange object, 890
- ScaleType **property**, Axis **object**, 673
- ScaleWidth **method**
 - Shape object, 885
 - ShapeRange object, 891
- Scenario **object**, 871–872
- Scenarios **collection**, 871–872
- Scenarios **method**, Worksheet **object**, 962
- Schema Collection **property**, CustomXMLPart **object**, 1021
- Schemas **property**, XmlMap **object**, 968
- SchemaXLM **property**, MetaProperties **collection**, 1044
- SchemeColor **property**, ChartColorFormat, 698
- ScopeFolder **object**, 1052–1053
- ScopeFolder **property**
 - ScopeFolder object, 1053
 - SearchScope, 1054
- ScopeFolders **collection**, 1052–1053
- ScopeType Condition Scope **property**, AboveAverage **object**, 637
- ScopeType **property**, UniqueValues **object**, 920
- screen updating**, 66
- ScreenRefresh **method**, 653
- ScreenSize **property**, WebOptions **object**, 929
- screentip attribute**, 302
- ScreenUpdating **property**, Application **object**, 653
- Script **property**, Shape **object**, 883
- Scroll **event**, 211
- Scroll **parameter**, 659
- ScrollArea **property**, 11, 957
- Scrollbar **control**, 211
- ScrollColumn **property**, Windows **object**, 932
- ScrollIntoView **method**, Windows **object**, 934
- ScrollRow **property**, Windows **object**, 932
- ScrollWorkbookTabs **method**, Windows **object**, 934
- sealed integer (Currency) data type**, 43
- Search Results window**, 27
- SearchFolders **collection**, 1053
- SearchFormat argument**, 682, 684, 843
- SearchHelp **method**, IAssistance **object**, 1038
- searching for names**, 133–139
- SearchScope **object** and SearchScopes **collection**, 1054
- security, macro**, 5
- Security settings**, 489–491, 493, 494, 499, 500
- SegmentType **property**, ShapeNode **object**, 886
- Select Case**, 49–50
- Select **event**, Chart **object**, 695
- Select Excel add-ins**, 379, 385
- Select **file**, 246
- Select **method**
 - Axis object, 673
 - AxisTitle object, 675
 - Chart object, 694
 - ChartArea object, 697
 - Charts collection, 687
 - Range object, 25, 93–94, 864
 - Series object, 876
 - SeriesLines object, 877
 - Shape object, 885
 - ShapeRange object, 891
 - Sheets collection, 893
 - TextRange2 object, 1072
 - TickLabels object, 914
 - Trendline object, 918
 - UpBars object, 921
 - Walls object, 926
 - Worksheet object, 88
 - Worksheets collection, 87, 955, 962
- SELECT statement**, 432–434, 441, 460
- SelectAll **method**, Shapes **collection**, 881
- SelectByID **method**, CustomXMLPart **objects collection**, 1020
- SelectByNamespace **method**, CustomXMLPart **objects collection**, 1020
- SelectCertificateDetailByThumbprint **method**, SignatureInfo **object**, 1064
- SelectedItems **property**, 235, 1032
- SelectedSheets **property**, Windows **object**, 932
- SelectedVbComponent **property**, VBE, 990
- Selection **property**
 - Application object, 653
 - Window object, 932
- SelectionChange **event**, 26, 963
- SelectNodes **method**
 - CustomXMLNode object, 1019
 - CustomXMLPart object, 1021
 - DOMDocument, 262, 263
- SelectSignatureCertificate **method**, SignatureInfo **object**, 1064
- SelectSingleNode **method**
 - CustomXMLNode object, 1019
 - CustomXMLPart object, 1021
- Send menu**, 312
- Send **method**, 423
- SendFaxOverInternet **method**, Workbook **object**, 949
- SendForReview **method**, Workbook **object**, 949
- SendKeys **method**, 70–71, 660
- SendMail **method**, Workbook **object**, 950
- SendMailer **method**, Workbook **object**, 950
- Sentences **property**, TextRange2 **object**, 1070
- <separator .../> **control type**, 300
- Sequence **function**, 385, 394, 401, 407

Sequence Wizard, 399, 401

Series **collection**, 188, 189, 192–194, 196, 203, 204, 220, 221, 366, 422, 694, 695, 703, 734, 742, 769, 839, 842, 876

Series formats, 691

SERIES **function**, 191, 193–195, 197

Series **object**, 187, 197, 694, 703, 729, 769, 840, 872–877, 917

SeriesChange **event**, Chart **object**, 695

SeriesCollection **collection**, 872–877

SeriesCollection **method**

Chart **object**, 694

SeriesCollection **object**, 873

SeriesCollection **object**, 187, 197, 742

SeriesLines **object**, 877

ServerActions **property**, Range **object**, 856

ServerPolicy **property**, Workbook **object**, 942

ServerViewableItems **collection**, 878

ServerViewableItems **property**, Workbook **object**, 942

Set **statement**, 20, 45, 69, 134, 356

SetBackgroundPicture **method**, Worksheet **object**, 962

SetBackgroundPictureString **method**, Chart **object**, 694

SetDefaultChartString **method**, Chart **object**, 694

SetDefaultContext **method**, IAssistance **object**, 1039

SetEditingType **method**, ShapeNode **objects collection**, 886

SetElement **method**, 186, 187, 694

SetExtrusionDirection **method**, ThreeDFormat **object**, 913

SetFirstPriority **method**

AboveAverage **object**, 638

Top10 **object**, 916

UniqueValues **object**, 920

SetFocus **method**

CommandBarButton **object**, 1006

CommandBarComboBox **object**, 1009

CommandBarControl **object**, 1013

CommandBarPopup **object**, 1015

Window **object**, 993

SetIcon **method**, SortFields **object**, 898

SetLastPriority **method**

AboveAverage **object**, 638

Top10 **object**, 916

UniqueValues **object**, 920

SetLinkOnData **method**, Workbook **object**, 950

SetParam **method**, 479

SetPasswordEncryptionOptions **method**, Workbook **object**, 950

SetPhonetic **method**, Range **object**, 864

SetPosition **method**, ShapeNode **objects collection**, 886

SetPresetCamera **method**, ThreeDFormat **object**, 913

SetRange **method**, 143, 146, 897

SetSegmentType **method**, ShapeNode **objects collection**, 886

SetSelection **method**, CodePane **object**, 979

SetShapesDefaultProperties **method**

Shape **object**, 885

ShapeRange **object**, 891

SetSourceData **method**, 177, 184, 187

SetSourceDataRange **method**, Chart **object**, 694

SetThreeDFormat **method**, ThreeDFormat **object**, 913

Setup **buttons**, 687, 693

Setup **property**, Signature **object**, 1062

SetupBlogAccount **method**, IBlogExtensibility **object**, 1040

SetUpMenus **procedure**, 351

SetValue **method**, XPath **object**, 970

SetWindowLong **function**, 623

SFI **controls**, 496

sFormatDate **function**, 567

Shadow **format**, 700, 710

Shadow **property**

AxisTitle **object**, 675

ChartArea **object**, 697

Font2 **object**, 1036

Series **object**, 875

ShapeRange **object**, 889

Shadow **property**, Shape **object**, 883

ShadowFormat **object**, 878–879

Shape **object**, 65, 177, 178, 184, 186, 187, 640, 680, 707, 715, 717, 719, 744, 753, 757, 775, 776, 791, 793, 795, 806, 878, 880–885, 907, 911

ShapeNode **object**, 886–887

ShapeNodes **collection**, 886–887

ShapeRange **object collection**, 887–891

ShapeRange **objects**, 908

Shapes **collection**, 177, 181, 192, 196, 757, 880–885

Shapes **property**

Chart **object**, 689

Worksheet **object**, 957

ShapeStyle **property**

Shape **object**, 883

ShapeRange **object**, 889

sharedStrings **XML file**, 268–269

SharedWorkspace **object**, 1054–1055

SharedWorkspace **property**, Workbook **object**, 942

SharedWorkspaceFile **object**, 1055–1056

SharedWorkspaceFiles **collection**, 1055–1056

SharedWorkspaceFolder **object**, 1057

SharedWorkspaceFolders **collection**, 1057

SharedWorkspaceLink **object**, 1058–1059

SharedWorkspaceLinks **collection**, 1058–1059

SharedWorkspaceMember **object**, 1059–1060

SharedWorkspaceMembers **collection**, 1059–1060

SharedWorkspaceTask **object**, 1060–1061

- SharedWorkspaceTasks **collection**, 1060–1061
- Sheet **object**, 77, 90
- Sheet **property**
 - PublishObject object, 845
 - WorksheetView object, 967
- SheetActivate event**
 - Application object, 661
 - Workbook object, 952
- SheetBeforeDoubleClick **event**
 - Application object, 661
 - Workbook object, 952
- SheetBeforeRightClick **event**
 - Application object, 661
 - Workbook object, 952
- SheetCalculate **event**
 - Application object, 661
 - Workbook object, 952
- SheetChange **event**, 206, 661
- SheetChangeRange **event**, Workbook **object**, 952
- SheetDeactivate **event**
 - Application object, 662
 - Workbook object, 952
- SheetFollowHyperlink **event**
 - Application object, 662
 - Workbook object, 953
- SheetPivotTableUpdate **event**
 - Application object, 662
 - Workbook object, 953
- Sheets **collection**
 - Copy and Move methods, 85–87
 - grouping worksheets, 87–88
 - overview, 891
 - Sheets common properties, 891
 - Sheets methods, 892–893
 - Sheets properties, 892
 - Worksheets collection, 83–85
- Sheets **property**
 - Application object, 653
 - Workbook object, 942
- SheetSelectionChange **event**
 - Application object, 662
 - Workbook object, 953
- SheetsInNewWorkbook **property**, Application **object**, 653
- SheetViews **object**, 893
- SheetViews **property**, Windows **object**, 932
- SHELL32.DLL **file**, 602
- Shift Button **parameter**, 695
- Shift **parameter**, 695
- ShipperID **field**, 458
- shortcut keys, for macros**, 6–7
- Shortcut menus**, 319, 338, 340, 352
- shortcut Range object references**, 96
- ShortcutText **property**, CommandBarButton **object**, 1004
- ShortDate format**, 539, 541, 549
- Show Dependents **method**, Range **object**, 865
- Show **method**
 - CodePane object, 979
 - DialogSelected items object, 235
 - FileDialog object, 1033
 - UserForm, 273
 - WorkflowTask object, 1077
 - WorkflowTemplate object, 1078
- Show pages command**, 512
- Show **parameter**, 690
- ShowAllData **method**
 - AutoFilter object, 668
 - Worksheet object, 962
- ShowAsAvailable **property**, TableStyle **object**, 905
- ShowAsAvailableTableStyle **property**, TableStyle **object**, 905
- ShowAutoFilter **property**, 148
- ShowChartTipNames **property**, Application **object**, 653
- ShowChartTipValues **property**, Application **object**, 653
- ShowConflictHistory **property**, Workbook **object**, 942
- ShowDataForm **method**, 128, 159, 556, 962
- ShowDataLabelsOverMaximum **property**, Chart **object**, 689
- ShowDetail **property**, 828, 856
- ShowDetails **method**, Signature **object**, 1063
- ShowDevTools **property**, Application **object**, 653
- ShowError **property**, Validation **object**, 923
- ShowErrors **method**, Range **object**, 865
- ShowHelp **method**, IAssistance **object**, 1039
- showImage **attribute**, 302
- ShowImportExportValidationErrors **property**, XmlMap **object**, 968
- ShowInput **property**, Validation **object**, 923
- showItemImage **attribute**, 302, 305
- showItemLabel **attribute**, 302
- showLabel **attribute**, 303
- ShowMenuFloaties **property**, Application **object**, 653
- ShowPivotChartActiveFields **property**, Workbook **object**, 942
- ShowPivotTableFieldList **property**, Workbook **object**, 942
- ShowPopup **method**, 342, 1002
- ShowPrecedents **method**, Range **object**, 865
- ShowSelectionFloaties **property**, Application **object**, 653
- ShowSettings **method**, EncryptionProvider **object**, 1031

ShowSignatureAdded method, SignatureProvider object

- ShowSignatureAdded **method**, SignatureProvider **object**, 1065
- ShowSignatureCertificate **method**, SignatureInfo **object**, 1064
- ShowSignatureDetails **method**, SignatureProvider **object**, 1065
- ShowSignatureSetup **method**, SignatureProvider **object**, 1065
- ShowSignaturesPane **property**, SignatureSet **collection**, 1062
- ShowSignDate **property**, SignatureSetup **object**, 1066
- ShowSigningCeremony **method**, SignatureProvider **object**, 1065
- ShowStartupDialog **property**, Application **object**, 653
- ShowToolTips **property**, Application **object**, 653
- ShowWindow **function**, 623
- ShowWindowsInTaskbar **property**, Application **object**, 653
- ShrinkToFit **property**
 - CellFormat object, 683
 - Range object, 857
 - Style object, 902
- Sht As **object**, 89, 90, 324
- Sht **parameter**, 91
- SideWall **property**, Chart **object**, 690
- Sign **method**, Signature **object**, 1063
- Signature **object**, 1061–1063
- SignatureComment **property**, SignatureInfo **object**, 1064
- SignatureImage **property**, SignatureInfo **object**, 1064
- SignatureInfo **object**, 1063–1064
- SignatureLineShape **property**, Signature **object**, 1063
- SignatureProvider **object**, 1064–1066
- SignatureProvider **property**
 - SignatureInfo object, 1064
 - SignatureSetup object, 1066
- Signatures **property**, Workbook **object**, 942
- SignatureSet **collection**, 1061–1063
- SignatureSetup **object**, 1066
- SignatureText **property**, SignatureInfo **object**, 1064
- SigningInstructions **property**, SignatureSetup **object**, 1066
- SignXmlDsig **method**, SignatureProvider **object**, 1065
- Simple **class**, 384, 385, 401, 408
- single-parameter Range **object reference**, 101–102
- size **attribute**, 303
- Size **parameter**, 691, 708
- Size **property**
 - Font2 object, 1036
 - ShadowFormat object, 879
- Sizeable **property**, 625
- sizeString **attribute**, 303
- Smallcaps **property**, Font2 **object**, 1036
- SmallScroll **method**, Windows **object**, 935
- SmartDocument **object**, 1066–1067
- SmartDocument **property**, Workbook **object**, 942
- SmartTag **object**, 893–894
- SmartTagAction **object**, 894–895
- SmartTagActions **collection**, 894–895
- SmartTagActions **property**, SmartTag **object**, 894
- SmartTagOptions **collection**, 895
- SmartTagOptions **property**, Workbook **object**, 942
- SmartTagRecognizersRecognizers **property**, Application **object**, 653
- SmartTagReconizer **object** and SmartTagRecognizers **collection**, 895–896
- SmartTags **collection**, 893–894
- SmartTags **property**
 - Range object, 857
 - Worksheet object, 957
- Smooth **property**, Series **object**, 875
- sNumToUS **function**, 542–543
- SoftEdge Format **property**, Font2 **object**, 1036
- SoftEdge **property**
 - Shape object, 883
 - ShapeRange object, 889
- SoftEdgeFormat **object**, 1067
- Solid **method**, ChartFillFormat **object**, 699
- SolutionID **property**, SmartDocument **object**, 1067
- SolutionURL **property**, SmartDocument **object**, 1067
- SolveOrder **property**, CalculatedMember **object**, 679
- Sort fields **collection**, 143
- Sort **method**, Range **object**, 144, 865
- Sort **object**, 143, 144, 146, 896–897
- Sort **property**
 - AutoFilter object, 668
 - Worksheet object, 957
- SortField **object**, 143, 146, 897–898
- SortFields **collection**, 897–898
- SortFields **property**, Sort **object**, 896
- SortHint **property**, Signature **object**, 1063
- sorting
 - ranges, 142–144
 - tables, 145–146
- SortMethod **property**, Sort **object**, 896
- SortOn **property**, SortFields **object**, 897
- SortOnValue **property**, SortFields **object**, 897
- Sort **property**, QueryTable **object**, 848
- SortSpecial **method**, Range **object**, 866
- SoundNote **object**, 898
- SoundNote **property**, Range **object**, 857
- Source **argument**, 443
- Source **file**, 270, 271

- Source **parameter**, 694
- Source **property**
 - PublishObject object, 845
 - Watch object, 927
- Source type **parameter**, 180
- SourceConnection File **property**, QueryTable **object**, 848
- SourceDataFile **property**, QueryTable **object**, 848
- SourceName **property**, CalculatedMember **object**, 679
- SourceType **property**, PublishObject **object**, 845
- SourceURL **property**, 967, 1055
- SpaceAfter **property**, ParagraphFormat2 **object**, 1048
- SpaceBefore **property**, ParagraphFormat2 **object**, 1048
- SpaceWithin **property**, ParagraphFormat2 **object**, 1048
- Spacing **property**
 - Font2 object, 1036
 - TextColumn2 object, 1069
- Speak **method**
 - Range object, 866
 - Speech object, 899
- SpeakCellOnEnter **property**, Speech **object**, 898
- special names**, 128–129
- SpecialCells **method**, 87, 105–107, 113, 123, 124
- SpecialCells **method**, Range **object**, 866
- Speech **object**, 898–899
- Speech **property**, Application **object**, 654
- SpellingOptions **collection**, 899–901
- SpellingOptions **property**, Application **object**, 654
- Spin Button **control**, 211–212
- SpinDown **event**, 211, 212
- SpinUp **event**, 211, 212
- Split **function**, 230
- Split **property**, Windows **object**, 932
- SplitColumn **property**, Windows **object**, 932
- SplitHorizontal **property**, Windows **object**, 932
- SplitRow **property**, Windows **object**, 932
- SplitVertical **property**, Windows **object**, 932
- SQL (Structured Query Language) overview**
 - DELETE statement, 435
 - INSERT statement, 434
 - UPDATE statement, 434–435
- SQL command**, 440, 808
- SQL property**, 785, 788
- SQL Server, using ADO with**, 454–463
 - connecting to SQL Server, 455–456
 - disconnected recordsets, 461–463
 - multiple recordsets, 460–461
 - stored procedures, 456–460
- SSO database**, 786, 790
- Standard toolbar**, 9, 27, 320, 321, 580
- StandardFont **property**, Application **object**, 654
- StandardFontSize **property**, Application **object**, 654
- StandardHeight **property**, Worksheet **object**, 957
- StandardWidth **property**, Worksheet **object**, 957
- Start **parameter**, 174, 684, 715, 805
- Start **property**, TextRange2 **object**, 1070
- Start Recording button**, 4
- StartupPath **property**, Application **object**, 654
- StartValue **property**, BulletFormat2 **object**, 996
- State **property**
 - CommandBarButton object, 1004
 - Connection object, 438
 - Recordset object, 443
- Statement **property**, ServerPolicy **object**, 1050
- Static **statement**, 41
- Status **property**
 - RoutingSlip object, 870
 - SharedWorkspaceTask object, 1061
 - Sync object, 1068
- StatusBar **property**, 70, 654
- Step option**, 53
- Stop Recording button**, 4
- StopifTrue **property**
 - AboveAverage object, 637
 - Top10 object, 916
 - UniqueValues object, 920
- stored procedures**, 456–460
- StoreLicenses **property**, Permission **object**, 1049
- Str **function**, 542
- Stream **object**, 436
- Strike **property**, Font2 **object**, 1036
- StrikeThrough **property**, Font2 **object**, 1036
- String (fixed-length) **data type**, 43
- String (variable-length) **data type**, 43
- StripeSize **property**, TableStyleElement **object**, 906
- Structured Query Language (SQL) overview**
 - DELETE statement, 435
 - INSERT statement, 434
 - UPDATE statement, 434–435
- structures**, 606–609
- structuring data**, 141–142
- Style **method**, Style **object**, 903
- Style **object**, 676, 901–903
- Style **property**
 - BulletFormat2 object, 997
 - CommandBarButton object, 1005
 - CommandBarComboBox object, 1008
 - Range object, 857
 - ShadowFormat object, 879
- Styles **collection**, 901–903
- Styles **property**, Workbook object **collection**, 942
- sub procedures, calling**, 35–36
- Subject **property**, RoutingSlip **object**, 870
- SubscribeTo **method**, Range **object**, 866

Subscript **property**, Font2 **object**, 1037
Subset **property**, SignatureSet **collection**, 1062
Subtotal **method**, Range **object**, 867
SuggestedSigner **property**, SignatureSetup **object**, 1066
SuggestedSignerEmail **property**, SignatureSetup **object**, 1066
SuggestedSignerLine2 **property**, SignatureSetup **object**, 1066
SuggestMainOnly **property**, SpellingOptions **collection**, 900
SUM **function**, 111
Sum Wizard, 639
Summary **property**, Range **object**, 857
supertip, 303, 1001
supertip **attribute**, 303
Switch Row/Column button, 182
Sync **object**, 1067–1068
Sync **property**, Workbook **object**, 943
SyncEvent **event**, Workbook **object**, 953
synchronizing worksheets, 90–91
SynchScrollingSideBySide **property**, Windows **object**, 929

T

Tab **object**, 690, 903–904
Tab **property**
 Chart **object**, 690
 Worksheet **object**, 957
tab sets, controlling using RibbonX, 313–314
Table button, 144
Table feature, 129
Table **method**, Range **object**, 867
table-driven command bar creation, 344–353
TableName **parameter**, 165
tables
 creating, 144–145, 417
 sorting, 145–146
Tables group, 144, 162
TableStyle **element**, 748, 765
TableStyle **object**, 904–906
TableStyleElement **object**, 906
TableStyleElements **collection**, 906
TableStyleElements **property**, TableStyle **object**, 905
TableStyles **collection**, 904–906
TableStyles **property**, Workbook **object**, 943
TabRatio **property**, Windows **object**, 932
tabs, controlling using RibbonX, 313–314
TabStop2 **object**, 1068–1069
TabStops **property**
 ParagraphFormat2 **object**, 1048
 Ruler2 **object**, 1051
TabStops2 **collection**, 1068–1069
tag **attribute**, 303
Tag **property**
 and Click events, 398, 577
 CommandBarButton **object**, 1005
 CommandBarComboBox **object**, 1008
 CommandBarControl **object**, 1012
 CommandBarPopup **object**, 1015
 and Parameter **property**, 333
Tag **property**, IRibbonControl **object**, 1042
TakeFocusOnClick **property**, 15
TargetBrowser **property**, WebOptions **object**, 929
TargetFile **variable**, 266
task panes, showing UserForms as, 405–406
Tasks **property**, SharedWorkspace **object**, 1055
Tax **function**, 47
template letter, 425, 427, 429
TemplateRemoveExtData **property**, Workbook **object**, 943
TemplatesPath **property**, Application **object**, 654
Temporary **parameter**, 400
Terminate **code**, 614
Terminate **event**, 577, 615, 619
= TEXT() worksheet **function**, 559
 Application.ConvertFormula **function**, 560
 Application.Evaluate **function**, 560
 Application.ExecuteExcel4Macro **function**, 560
 Range.AdvancedFilter **method**, 559–560
 Range.Formula **property**, 559
 Range.FormulaArray **property**, 559
 Range.Value **property**, 559
Text Box **control**, 276
text files
 importing, 223, 849, 850
 opening, 223
 querying, 467–468
 reading, 226–227
 writing to, 224–233, 227
Text for Boolean True setting, 538
Text Import Wizard, 555, 560, 561
Text **object**, 758
Text **parameter**, 658, 715, 720
Text **property**
 AxisTitle **object**, 675
 Characters **object**, 685
 CommandBarComboBox **object**, 1008
 CustomXMLNode **object**, 1018
 CustomXMLValidationError, 1024
 Range **object**, 857

- TextEffectFormat, 907
- TextRange2 object, 1070
- TextBox **control**, 371
- TextBox **object**, 369
- TextboxText **property**, SmartTagAction **object**, 895
- TextCode page**, 693
- TextColumn2 **object and** TextColumns2 **collection**, 1069
- TextDirection **property**
 - ParagraphFormat2 object, 1049
 - TextColumn2 object, 1069
- TextEffect **property**
 - Shape object, 883
 - ShapeRange object, 889
- TextEffectFormat **object**, 907–908
- TextFileColumnDataTypes **property**, QueryTable **object**, 849
- TextFileCommaDelimiter **property**, QueryTable **object**, 849
- TextFileConsecutiveDelimiter **property**, QueryTable **object**, 849
- TextFileDecimalSeparator **property**, QueryTable **object**, 849
- TextFileFixedColumnWidths **property**, QueryTable **object**, 849
- TextFileOtherDelimiter **property**, QueryTable **object**, 849
- TextFileParseType **property**, QueryTable **object**, 849
- TextFilePlatform **property**, QueryTable **object**, 849
- TextFilePromptOnRefresh **property**, QueryTable **object**, 849
- TextFileSemicolonDelimiter **property**, QueryTable **object**, 849
- TextFileSpaceDelimiter **property**, QueryTable **object**, 849
- TextFileStartRow **property**, QueryTable **object**, 849
- TextFileTabDelimiter **property**, QueryTable **object**, 850
- TextFileTextQualifier **property**, QueryTable **object**, 850
- TextFileThousandsSeparator **property**, QueryTable **object**, 850
- TextFileTrailingMinusNumbers **property**, QueryTable **object**, 850
- TextFileVisualLayout **property**, QueryTable **object**, 850
- TextFrame **object**, 908–909
- TextFrame **property**
 - Shape object, 883
 - ShapeRange object, 889
- TextFrame2 **object**, 909–911
- TextFrame2 **property**, ShapeRange **object**, 889
- TextRange **property**, TextRange2 **object**, 910
- TextRange2 **object**, 1070–1072
- TextToColumns **method**, Range **object**, 867
- TextureName **property**, ChartFillFormat **object**, 699
- TextureType **property**, ChartFillFormat **object**, 699
- Theme **property**, Workbook **object**, 943
- ThemeColor **object**, 1072
- ThemeColor **property**, 676, 677, 749, 750, 766, 904
- ThemeColorScheme **property**, OfficeTheme **object**, 1047
- ThemeColorSchemeIndex **property**, ThemeColor, 1072
- ThemeColorsScheme **object**, 1072–1073
- ThemeEffectScheme **object**, 1073
- ThemeEffectScheme **property**, OfficeTheme **object**, 1047
- ThemeFont **object and** ThemeFonts **collection**, 1073–1074
- ThemeFontScheme **object**, 1074
- ThemeFontScheme **property**, OfficeTheme **object**, 1047
- Thickness **property**, Walls **object**, 926
- ThisCell **property**, Application **object**, 654
- ThisWorkbook **property**, Application **object**, 654
- Thousand Separator setting**, 538
- ThousandsSeparator **property**, Application **object**, 654
- ThreeD format **object**, 701, 710
- ThreeD **property**
 - Shape object, 883
 - ShapeRange object, 889
 - TextFrame2 object, 910
- ThreeDFormat **object**, 911–913
- ThrottleInterval **property**, RTD, 870
- ThunderDFrame **class**, 608
- ThunderRT6DFrame **class**, 608
- ThunderXFrame **class**, 608
- TickLabelPosition **property**, Axis **object**, 673
- TickLabels **object**, 913–915
- TickLabels **property**, Axis **object**, 673
- TickLabelSpacing **property**, Axis **object**, 673
- TickLabelSpacingIsAuto **property**, Axis **object**, 673
- TickMarkSpacing **property**, Axis **object**, 673
- Time **parameter**, 661
- Time **property**, AutoRecover **object**, 669
- Timer **function**, 616
- TimeSerial **function**, 71, 413
- TintAndShade **property**
 - Border object, 677
 - Borders collection, 676
- TintAndShade **property**, Tab **object**, 904
- title **attribute**, 303
- Title button**, 186
- Title **property**
 - Add-In, 639
 - AllowEditRange object, 642
 - CustomTaskPane object, 1016
 - FileDialog object, 1032
 - SharedWorkspaceTask object, 1061

TODAY function

TODAY function, 73

<toggleButton .../> **control type, 299**

ToggleFormsDesign **method**, Workbook **object, 950**

ToggleVerticalText **method**, TextEffectFormat **object, 907**

toolbars, 320–322, 335–338

toolkit add-in, 571

Tools menu, 5

ToolTipText **property**

CommandBarButton object, 1005

CommandBarComboBox object, 1008

CommandBarControl object, 1012

CommandBarPopup object, 1015

Top **property**

Application object, 654

Axis object, 673

AxisTitle object, 675

ChartArea object, 697

CommandBar object, 1002

CommandBarButton object, 1005

CommandBarComboBox object, 1008

CommandBarControl object, 1012

CommandBarPopup object, 1015

Range object, 857

Shape object, 883

ShapeRange object, 889

Window object, 932, 993

Top10 **object, 915–917**

TopBottom **property**, Top10 **object, 916**

TopLeftCell **property**, Shape **object, 884**

TopLine **property**, CodePane **object, 978**

ToReferenceStyle **parameter, 657**

Tracking **property**, TextEffectFormat **object, 907**

TrackStatus **property**, RoutingSlip **object, 870**

Training **method, 355**

TransitionExpEval **property**, Worksheet **object, 958**

TransitionFormEntry **property**, Worksheet **object, 958**

TransitionMenuKey **property**, Application **object, 654**

TransitionMenuKeyAction **property**, Application **object, 654**

TransitionNavigKey **property**, Application **object, 654**

Transparency **property**

GradientStop object, 1038

ShadowFormat object, 879

TrapApplication **events procedure, 365**

TreeViewControl **object, 917**

Trendline **object, 917–919**

Trendlines **collection, 917–919**

Trendlines **method**, Series **object, 876**

TrimText **method**, TextRange2 **object, 1072**

Trust Center dialog box, 5, 489, 491

Trust Center settings, 489, 491, 571

Trust Center user interface, 491–503

ActiveX settings, 495–496

Add-ins category, 494

External Content category, 499–500

macro settings, 497–498

Message Bar, 498–499

privacy options, 501–503

trusted locations, 492–493

trusted publishers, 492

Trusted Locations category, 492

Trusted Locations settings, 496

TwoColorGradient **method**, ChartFillFormat **object, 699**

TwoInitialCapitals **property**, AutoCorrect **object, 666**

type libraries, 411, 412, 414, 430

Type **parameter, 68, 670, 690, 693, 841, 917**

Type **property**

Action object, 639

Axis object, 673

BulletFormat2 object, 997

CalculatedMember object, 679

CalloutFormat object, 680

ChartColorFormat object, 698

ChartFillFormat object, 699

CommandBar object, 1002

CommandBarButton object, 1005

CommandBarComboBox object, 1008

CommandBarControl object, 1012

CommandBarPopup object, 1015

CustomXMLValidationError, 1025

DocumentProperty object, 1030

MetaProperty object, 1045

Reference object, 984

ReflectionFormat, 1051, 1067

SearchScope, 1054

Series object, 875

ShadowFormat object, 879

Shape object, 884

ShapeRange object, 889

SmartTagAction object, 895

TabStop2 object, 1069

Top10 object, 916

Trendline object, 918

UniqueValues object, 920

Validation object, 923

VBcomponent object, 987

VBproject object, 991

VPageBreak object, 925

Window object, 932, 993

Worksheet object, 958

TypeConditionType **property**, AboveAverage **object, 637**

U

- UBound **function**, **57, 109, 190**
- UDFs (user-defined **functions**), **1, 18–21, 44, 333, 373, 384**
 - creating, 18–21
 - limitations, 21
- UFI **controls**, **496**
- UINT C **data type**, **605**
- UINT FAR * C **data type**, **605**
- UK settings**, **148, 540**
- Underline Color **property**, Font2 **object**, **1037**
- Underline Style **property**, Font2 **object**, **1037**
- Undo **method**, Application **object**, **660**
- Ungroup **method**
 - Range **object**, 868
 - Shape **object**, 885
 - ShapeRange **object**, 891
- Unhide button**, **6**
- Union **method**, **115, 661**
- UniqueValues **object**, **919–920**
- Unload **statement**, **273, 277, 282**
- Unmerge **method**, Range **object**, **868**
- Unprotect **method**
 - AllowEditRange **object**, 642
 - Chart **object**, 694
 - Workbook **object**, 950
 - Worksheet **object**, 962
- UnprotectSharing **method**, Workbook **object**, **950**
- Unsuspend **method**, Sync **object**, **1068**
- Until **statement**, **52**
- unzipping Excel containers**, **266–267**
- UpBars **object**, **920–921**
- Update **method**
 - Add-ins **collection**, 974
 - COMAddins **collection**, 997
- UPDATE **statement**, **434–435, 453**
- UpdateFromFile **method**, Workbook **object**, **950**
- UpdateLink **method**, **71, 950**
- UpdateLinks **property**, Workbook **object**, **943**
- UpdateRemoteReferences **property**, Workbook **object**, **943**
- Uri **property**, XMLNameSpace, **969**
- URL encoding**, **535**
- URL **property**
 - SharedWorkspace **object**, 1055
 - SharedWorkspaceFile **object**, 1056
 - SharedWorkspaceLink **object**, 1059
- U.S. settings**, **148, 546, 549, 829**
- UsableHeight **property**
 - Application **object**, 654
 - Window **object**, 932
- UsableWidth **property**
 - Application **object**, 655
 - Window **object**, 932
- Use Advanced Filter**, **156, 160**
- UseCommandObject procedure**, **459**
- UseDefaultFolderSuffix **method**, WebOptions **object**, **929**
- UsedObjects **collection**, **921–922**
- UsedObjects **property**, Application **object**, **655**
- UsedRange **property**, Worksheet **object**, **958**
- UseLegacyKeyboardShortcuts **property**, Application **object**, **655**
- UseLongFileNames **property**, WebOptions **object**, **929**
- User key**, **396**
- User Table**, **844**
- User_Form **resize event**, **630**
- USER32 .EXE **file**, **602**
- UserAccessList **collection**, **922**
- UserAccessList **object**, **921**
- UserControl **property**, Application **object**, **655**
- User-defined (using Type) **data type**, **43**
- user-defined **functions (UDFs)**, **1, 18–21, 44, 333, 373, 384**
 - creating, 18–21
 - limitations, 21
- User-defined menus**, **319**
- UserDict **property**, SpellingOptions **collection**, **900**
- UserForm **control**, **298, 368**
- UserForm **module**, **289**
- UserForm **object**, **26, 273, 277, 406, 594, 597, 622, 623, 624, 625, 628, 629, 633**
- UserForm styles**, **622**
- UserForm text boxes**, **370**
- UserForm window**, **618**
- UserForm_Initialize **event**, **284, 369, 629, 633**
- UserForm_Resize **event**, **626, 627, 633**
- UserForms
 - creating, 275–277
 - directly accessing controls, 277–281
 - displaying, 273–274
 - freezing, 618
 - maintaining data lists, 282–288
 - modeless, 288–291
 - overview, 594–598
 - resizable, 625–634
 - absolute changes, 626–627
 - CFormResizer class, 628–634
 - relative changes, 627–628
 - showing as task panes, 405–406
 - stopping Close button, 281–282

UserForms (continued)

UserForms (continued)

styles, modifying, 622–625
 CFormChanger class, 624–625
 Window styles, 623–624
variable UserForm name, 291
UserId **property**, UserPermission **object**, 1075
UserLibraryPath **property**, Application **object**, 655
UserName **property**, Application **object**, 655
UserPermission **object**, 1074–1075
UserPicture **method**, ChartFillFormat **object**, 700
Users **property**, AllowEditRange **object**, 642
UserStatus **property**, Workbook **object**, 943
UserTextured **method**, ChartFillFormat **object**, 700
UseStandardFormula **parameters**, 812, 821, 822
UseStandardHeight **property**, Range **object**, 857
UseStandardWidth **property**, Range **object**, 857
UseSystemSeparators **property**, Application **object**, 655
UseTextColor **property**, BulletFormat2 **object**, 997
UseTextFont **property**, BulletFormat2 **object**, 997

V

Val **function**, 543–544
Val(“6 My St.”) **expression**, 544
Validate **method**, CustomXMLSchemaCollection **object**, 1023
Validate **property**
 MetaProperties collection, 1044
 MetaProperty object, 1045
Validation **object**, 923–924
Validation **property**, Range **object**, 857
Val(myDate) **expression**, 544
Val(myDbf) **expression**, 544
Val(“SomeText”) **expression**, 544
Val(True) **expression**, 544
Value **property**
 ActiveCell object, 29
 Application object, 655
 Borders collection, 676
 ComboBox object, 216
 control, 279
 DocumentProperty object, 1030
 Error object, 741
 MetaProperty object, 1045
 Property object, 983
 Range object, 24, 67, 559, 857
 scbNavigator, 284–287
 Style object, 903
 Validation object, 923
 XPath object, 970
Value2 **property**, Range **object**, 857

values

returning, 33–34
storing in names, 129–130
Values **property**, 190, 192, 197, 871, 875
variable types, 42–45
 constants, 44
 declaring, 43–44
 declaring function and parameter types, 44
 variable naming conventions, 44–45

variables

declaring, 38–40
naming conventions, 44–45
object variables, 45–47
scope and lifetime of, 40–42
Variant (with characters) **data type**, 43
Variant (with numbers) **data type**, 43

Vb **method**, 383

VBA Extensibility library, 598

VBA language

arrays, 55–58
 dynamic arrays, 58
 multi-dimensional arrays, 57
basic input and output, 30–35
 constants, 33
 InputBox, 34–35
 parameters specified by name, 31–33
 parameters specified by position, 31
 return values, 33–34
 returning values, 33–34
calling functions and sub procedures, 35–36
creating offline cube using, 522–523
looping, 50–55
 Do...loop, 50–53
 For Each...Next loop, 54–55
 For...Next loop, 53–54
making decisions, 47–50
 block If structure, 48–49
 If statements, 47–48
 Select Case, 49–50
object variables, 45–47
 With...End With structure, 46–47
parentheses and argument lists, 37–38
 with Call statement, 38
 without Call statement, 37
run-time error-handling, 59–62
scope and lifetime of variables, 40–42
variable declaration, 38–40
variable type, 42–45
 constants, 44
 declaring, 43–44
 declaring function and parameter types, 44
 variable naming conventions, 44–45
VBA MsgBox **function**, 561

- vbAbortRetryIgnore **constant, 32**
- vbApplicationModal **constant, 32**
- vbAppTaskManager **constant, 282**
- vbAppWindows **constant, 282**
- VBASigned **property, Workbook object, 943**
- VBComponent **object, 573–574, 596, 986–989**
- VBComponent.Properties("<Other Properties>")
 - VBE **property, 972**
- VBComponent.Properties("Name") VBE **property, 972**
- VBComponents **collection, 986–989**
- VBComponents **property, VBProject object, 991**
- vbCritical **constant, 32**
- vbDefaultButton1 **constant, 32**
- vbDefaultButton2 **constant, 32**
- vbDefaultButton3 **constant, 32**
- vbDefaultButton4 **constant, 32**
- VBE (Visual Basic Editor), 8–11**
 - adding menu items to, 576–580
 - code modules, 9–10
 - COM Add-ins, 599–600
 - identifying VBE objects in code, 572–575
 - CodeModule object, 574
 - CodePane object, 574
 - Designer object, 574–575
 - VBComponent object, 573–574
 - VBE object, 572
 - VBProject object, 572–573
 - procedures, 10
 - Project Explorer, 10–11
 - Properties window, 11
 - starting up, 575–576
 - UserForms, 594–598
 - workbooks, 580–589
 - working with code, 589–593
 - working with references, 598–599
- VBE command bars, 319, 326, 350**
- VBE **object model, 971–994**
 - AddIn object and Add-Ins collection, 973–974
 - CodeModule object, 974–978
 - CodePane object and CodePanels collection, 978–980
 - CommandBarEvents object, 980–981
 - common properties and methods, 972–973
 - End FunctionReferencesEvents object, 986
 - Events object, 981–982
 - LinkedWindows collection, 982
 - links between Excel and, 971–972
 - overview, 971
 - Property object and Properties collection, 982–983
 - Reference object and References collection, 984–986
 - VBComponent object and VBComponents collection, 986–989
 - VBE object, 989–990
 - VBProject object and VBProjects collection, 990–992
 - Window object and Windows collection, 992–994
- VBE Print dialog box, 591**
- VBE **property, Application object, 655**
- VBE toolbar, 579, 581**
- VBE Toolkit add-in, 575**
- vbExclamation **constant, 32**
- vbFormCode **constant, 282**
- vbFormControlMenu **constant, 282**
- vbInformation **constant, 32**
- vbMsgBoxHelpButton **constant, 32**
- vbMsgBoxRight **constant, 32**
- vbMsgBoxRtlReading **constant, 32**
- vbMsgBoxSetForeground **constant, 32**
- vbOKCancel **constant, 32**
- vbOKOnly **constant, 32**
- VBProject **object, 571, 572–573, 585, 589, 971, 972, 990–992**
- VBProject **property, Workbook object, 943**
- VBProject.FileName VBE **property, 972**
- VBProjects **collection, 572, 990–992**
- VBProjects **property, VBE, 990**
- vbQuestion **constant, 32**
- vbRetryCancel **constant, 32**
- vbSystemModal **constant, 32**
- vbYesNo **constant, 32**
- vbYesNoCancel **constant, 32**
- Vector Markup Language, 736**
- VerifyXmlDsig **method, SignatureProvider object, 1066**
- Version **property, Application object, 655**
- Version **property, VBE, 990**
- VerticalAlignment **property**
 - AxisTitle object, 675
 - CellFormat object, 683
 - Range object, 857
 - Style object, 903
 - TextFrame object, 909
- VerticalAnchor **property, TextFrame2 object, 910**
- VerticalFlip **property**
 - Shape object, 884
 - ShapeRange object, 889
- Vertices **property**
 - Shape object, 884
 - ShapeRange object, 889
- View **property, Windows object, 932**
- View tab, 5, 6, 402, 403**
- ViewName **parameter, 727**
- visible **attribute, 303**
- Visible **property**
 - Application object, 655
 - BulletFormat2 object, 997

Visible property (continued)

Visible property (continued)

ChartFillFormat object, 699
Charts collection, 686
and ClearManualFilter method, 724, 818
CommandBar object, 1002
CommandBarButton object, 1005
CommandBarComboBox object, 1008
CommandBarControl object, 1012
CommandBarPopup object, 1015
CustomTaskPane object, 1016
ensuring Excel window is active with, 218
hiding names with, 131, 175–176
ShadowFormat object, 879
Shape object, 884
ShapeRange object, 889
Sheets collection, 892
ThreeDFormat object, 912
and toolbar creation, 337
Window object, 932, 993
Worksheets collection, 954, 958

visible rows
copying, 153–154
finding, 154–156

VisibleRange **property**, Windows **object**, 933
VisibleStateChange customtaskpane **events**, 1016
VisibleVisibility **property**, Chart **object**, 690

Visual Basic 6, 384, 402

Visual Basic button, 8

Visual Basic Editor. *See* VBE (Visual Basic Editor)

Visual Basic Editor (VBE), 8–11
adding menu items to, 576–580
code modules, 9–10
COM Add-ins, 599–600
identifying VBE objects in code, 572–575
CodeModule object, 574
CodePane object, 574
Designer object, 574–575
VbComponent object, 573–574
VBE object, 572
VbProject object, 572–573
procedures, 10
Project Explorer, 10–11
Properties window, 11
starting up, 575–576
UserForms, 594–598
workbooks, 580–589
working with code, 589–593
working with references, 598–599

Visual Basic for Applications Extensibility, 571, 572, 600, 971

Visual Basic language, 1, 73

VK_ **control**, 583

VLookup **function**, 19, 563
Volatile **method**, Application **object**, 661
Volatile **parameter**, 661
VPageBreak **object**, 924–925
VPageBreaks **collection**, 924–925

VPageBreaks property
Charts collection, 686
Sheets collection, 892
Worksheet object, 958
Worksheets collection, 954

W

Wait **method**, Application **object**, 661
Walls **object**, 687, 690, 765, 925–926
Walls **property**, Chart **object**, 690
WarnOnFunctionNameConflict **property**, Application **object**, 655
WarpFormat **property**, TextFrame2 **object**, 910
Watch **object**, 926–928
Watch Window, 580
Watches **collection**, 926–928
Watches **object**, 655
Watches **property**, Application **object**, 655
Web Browser **control**, 404, 405, 525
Web formatting argument, 487
web pages. *See* Internet
Web Queries, 470, 472, 479, 481, 484, 525, 528, 530, 536, 554, 558, 569
Web Query (IQY) files, 486–487
WebConsecutiveDelimitersAsOne **property**, QueryTable **object**, 850
WebDisableDateRecognition **option**, 558
WebDisableDateRecognition **property**, QueryTable **object**, 850
WebDisableRedirections **property**, QueryTable **object**, 850
WebFormatting **property**, QueryTable **object**, 850
WebOptions **object**, 928–929
WebOptions **property**, Workbook **object**, 943
WebPageFont **object**, 1075–1076
WebPageFonts **collection**, 1075–1076
WebPagePreview **method**, Workbook **object**, 951
WebPreFormattedTextToColumns **property**, QueryTable **object**, 850
WebSelectionType **property**, 480, 850
WebSingleBlockTextImport **property**, QueryTable **object**, 850
WebTables **property**, 480, 482, 851
Weekday **function**, 116
WeeklyPay **function**, 358
Weight **property**, Border **object**, 677

WeightWeight **property**, Borders **collection**, 676

What **parameter**, 666

Where **parameter**, 692

Width **property**

Application object, 656

Axis object, 673

ChartArea object, 697

CommandBar object, 1002

CommandBarButton object, 1005

CommandBarComboBox object, 1008

CommandBarControl object, 1012

CommandBarPopup object, 1015

CustomTaskPane object, 1016

Range object, 857

Shape object, 884

ShapeRange object, 889

Window object, 933, 993

Window list, 314

Window menu, 314

Window **object**, 77, 89–91, 92, 802, 929–935

Window **property**

CodePane object, 978

CustomTaskPane object, 1016

WindowActivate **event**

Application object, 662

Workbook object, 953

WindowDeactivate **event**

Application object, 662

Workbook object, 953

WindowNumber **property**, Windows **object**, 933

WindowResize **event**

Application object, 662

Workbook object, 953

Windows API, 601–634

anatomy of API call, 602–603

constants, structures, handles, and classes, 606–609

example classes, 616–622

CFreezeForm class, 618–619

class module CHighResTimer, 616–618

high-resolution timer class, 616

System Info class, 619–622

if something goes wrong, 609–611

interpreting C-style declarations, 603–606

modifying UserForm styles, 622–625

CFormChanger class, 624–625

Window styles, 623–624

resizable UserForms, 625–634

absolute changes, 626–627

CFormResizer class, 628–634

relative changes, 627–628

wrapping API calls in class modules, 611–616

Windows applications, 591, 608

Windows **collection**, 929–935

Windows **Control Panel**, 159, 538, 655

Windows File Manager, 373

Windows language

and regional settings, 538–545

identifying, 538–539

VBA conversion functions, 539–545

version of, 561

Windows **property**

Application object, 656

VBE, 990

Workbook object, 943

Windows Regional settings, 528, 537–539, 544, 546, 549, 552–556, 558, 559, 561, 565–568, 601

Windows Scripting Runtime, 601

Windows settings, 541, 545

Windows Sharepoint Server, 781

Windows Task Manager, 282

Windows TEMPdirectory, 602, 612

Windows temporary path, 612

Windows XP, 266, 611

WindowsForPens **property**, Application **object**, 656

WindowState **property**

Window object, 933, 993

WindowStateXlWindow **property**, Application **object**, 656

WININET.DLL **file**, 602

WINMM.DLL **file**, 602

WINSPOOL.DRV **file**, 602

With **statement**, 190, 281

With...End With structure, 46–47

WithEvents keyword, 363, 398

WithEvents object, 399, 406

Wizard dialog box, 397

Wizard form, 401

Word

accessing active document, 417

creating new document, 418

opening documents in, 416–417

WORD C **data type**, 605

Word **object**, 411

Word window, 418

WordArt Format **property**, Font2 **object**, 1037

WordArt **objects**, 907

WordArtFormat **property**, TextFrame **object2**, 910

Words **property**, TextRange2 **object**, 1070

WordWrap **property**

ParagraphFormat2 object, 1049

TextFrame2 object, 910

Workbook **object and Workbooks collection**, 935–953

events, 951–953

methods, 935–937, 944–951

overview, 935

properties, 935–943

Workbook object Connections dialog

- Workbook **object Connections dialog**, **471, 486**
- Workbook **object events**, **92, 199, 205–207, 208, 363, 573, 951**
- Workbook **object level**, **126**
- Workbook **object Links**, **500**
- Workbook **object tabs command bar**, **340**
- Workbook_BeforeClose **event**, **206, 338, 351, 377, 576**
- Workbook_Open **event**, **16, 150, 338, 346, 351, 365, 577**
- Workbook_Open procedure**, **577, 581**
- Workbook_Open routine**, **589**
- Workbook_SheetActivate **event**, **91**
- Workbook_SheetDeactivate **event**, **91**
- WorkbookActivate **event**, Application **object**, **662**
- WorkbookAddinInstall **event**, Application **object**, **662**
- WorkbookAddinUninstall **event**, Application **object**, **662**
- WorkbookAfterXMLExport **event**, Application **object**, **662**
- WorkbookAfterXMLImport **event**, Application **object**, **662**
- WorkbookBefore XMLExport **event**, Application **object**, **663**
- WorkbookBefore XMLImport **event**, Application **object**, **663**
- WorkbookBeforeClose **event**, Application **object**, **663**
- WorkbookBeforePrint **event**, **364, 371**
- WorkbookBeforePrint **event**, Application **object**, **663**
- WorkbookBeforeSave **event**, Application **object**, **663**
- Workbook.CodeName Excel **property**, **972**
- WorkbookConnection **object**, **469, 487–489, 490, 717, 953**
- WorkbookConnection **property**, QueryTable **object**, **851**
- WorkbookConnection **property**, XmlMap **object**, **968**
- WorkbookDeactivate **event**, Application **object**, **663**
- WorkbookNewSheet **event**, Application **object**, **663**
- WorkbookOpen **event**, Application **object**, **663**
- WorkbookPivotTableCloseConnection **event**, Application **object**, **663**
- WorkbookPivotTableOpenConnection **event**, Application **object**, **663**
- WorkbookRowsetComplete **event**, Application **object**, **663**
- workbooks and worksheets**
 - inserting and updating records in, **466–467**
 - opening web pages as, **528**
 - querying, **464–466**
 - Sheets collection, **83–88**
 - Copy and Move methods, **85–87**
 - grouping worksheets, **87–88**
 - Worksheets collection, **83–85**
 - synchronizing worksheets, **90–91**
 - using Internet for storing, **526–527**
 - Window object, **89–91**
 - Workbooks collection, **77–83**
 - files in same directory, **81**
 - getting filename from a path, **78–80**
 - overwriting existing workbook, **81–82**
 - saving changes, **82–83**
 - Workbooks **collection**
 - files in same directory, **81**
 - getting filename from a path, **78–80**
 - overwriting workbooks, **81–82**
 - saving changes, **82–83**
 - Workbooks **object**, **46, 380, 589**
 - Workbooks **property**, Application **object**, **656**
 - Workbook.VBProject Excel **property**, **972**
 - WorkflowID **property**, WorkflowTask **object**, **1077**
 - WorkflowTask **object**, **1076–1077**
 - WorkflowTasks **collection**, **1076–1077**
 - WorkflowTemplate **object**, **1077–1078**
 - WorkflowTemplates **collection**, **1077–1078**
 - Worksheet Change **event**, **200**
 - Worksheet DATE **function**, **87**
 - worksheet **events**, **199–202, 962**
 - Worksheet **function**, **19, 20, 56, 57, 74, 76, 96, 103, 121, 150, 192, 211, 212, 304, 324, 358**
 - Worksheet Index **property**, **85**
 - Worksheet Max **function**, **212**
 - Worksheet menu bar**, **320–322, 330, 332, 334, 341, 347, 351, 999**
 - Worksheet **object**
 - events, **962–963**
 - methods, **958–962**
 - properties, **955–958**
 - Worksheet **property**, Range **object**, **857**
 - Worksheet Range **object**, **23**
 - Worksheet Transpose **function**, **121, 150**
 - Worksheet_Activate **event**, **199**
 - Worksheet_BeforeRightClick **event**, **200**
 - Worksheet_Calculate **event**, **200, 201–202, 208, 958**
 - Worksheet_Deactivate **event**, **199, 200**
 - Worksheet_SelectionChange **event**, **26, 115, 199, 200**
 - Worksheet.CodeName Excel **property**, **972**
 - WorksheetFunction **object**, **963–966**
 - WorksheetFunction **property**, Application **object**, **656**
 - WorksheetName **function**, **75**
- worksheets. See workbooks and worksheets**
- Worksheets **collection**, **23, 53, 83–85, 87, 355, 635, 954–955**
- Worksheets **object**, **23, 46**
- Worksheets **property**
 - Application **object**, **656**
 - Workbook **object**, **943**

WorksheetView **object**, 966–967
 WorkspaceLastChangedBy **property**,
 Sync **object**, 1068
WPARAM C data type, 605
 WrapText **property**
 Range object, 857
 Style object, 903
 WrapText **property**, CellFormat **object**, 683
 Write **statement**, 224, 227
 WritePassword **property**, Workbook **object**, 943
 WriteReserved **property**, Workbook **object**, 943
 WriteReservedBy **property**, Workbook **object**, 943
WRS language, 539, 541, 542, 551
WRS number formats, 557

X

X values **property**, 190, 193
 xlBuiltinDialog **constants**, 737
XIChart item, 202, 692, 694, 695
XIClipboard format, 646
XICountry setting, 538, 561
 xlCredentials **method**, 786, 789
 XLDESK **class**, 608
XLL extension, 639
XLM format, 5
 XLMAIN **class**, 608
XIMouse chart, 695
XIMouse mouse button, 695
XIPicture clipboard, 691
 xlPicture **type constants**, 773
 xlPivotTableSource **type constants**, 807
XML (eXtensible Markup Language), 239–272
 attributes, 242–243
 comments, 241
 consuming XML data directly, 246–249
 elements and root element, 241–242
 namespaces, 243–245
 processing instructions, 241
 using VBA to program Open XML files, 265–272
 programmatically unzipping Excel containers, 266–267
 programmatically zipping Excel containers, 267–272
 using VBA to program XML processes, 253–265
 Document Object Model (DOM), 258–265
 programming XML maps, 253–258
 XPath, 262–264
 viewing and editing XML documents, 245
 XML declaration, 240–241

XML maps, 249–253
 creating, 251–253
 creating XML schema description, 249–251

XML Maps button, 251

XML property

CustomXMLNode object, 1018
 CustomXMLPart object, 1021
 SmartTag object, 894
 XMLSchema object, 970

XML Workbook object file, 402

XML-based format, 736

XmlDataBinding object, 967

XmlDataQuery method, Worksheet object, 962

XmlImport method, Workbook object, 951

XmlImportXml method, Workbook object, 951

XmlMap object and XMLMaps collection, 967–969

XmlMapQuery method, Worksheet object, 962

XmlMaps collection, 251, 254, 256, 257

XMLMaps property, Workbook object, 943

XmlNameSpace object and XMLNameSpaces collection, 969

XMLNamespaces property, Workbook object, 943

XMLSample files folder, 246, 260

XmlSchema object and XmlSchemas collection, 969–970

XPath, traversing and modifying XML files with, 262–265

XPath object, 257, 777

XPath property

CustomXMLNode object, 1018
 Range object, 857

XSD files, 247, 249, 251, 255

XValues property, Series object, 875

xxxLocal functions, 554

Z

Z method, ThreeDFormat object, 912

zero-length string, 35, 36, 40, 51, 52, 58, 68, 82, 117, 118, 138, 369, 417, 418, 427

zipping Excel containers, 267–272

 editing sharedStrings XML file to implement mass updates to text, 268–269
 unprotecting worksheet via Open XML manipulation, 269–270
 updating connection strings, 270–272

Zoom property, Windows object, 933

ZOrder method

Shape object, 885
 ShapeRange object, 891

ZOrderPosition property, 884, 889



Programmer to Programmer™

[BROWSE BOOKS](#)

[P2P FORUM](#)

[FREE NEWSLETTER](#)

[ABOUT WROX](#)

Get more Wrox at **Wrox.com!**

Special Deals

Take advantage of special offers every month

Unlimited Access. . .

. . . to over 70 of our books in the Wrox Reference Library (see more details online)

Meet Wrox Authors!

Read running commentaries from authors on their programming experiences and whatever else they want to talk about

Free Chapter Excerpts

Be the first to preview chapters from the latest Wrox publications

Forums, Forums, Forums

Take an active role in online discussions with fellow programmers

Browse Books

.NET
SQL Server
Java

XML
Visual Basic
C# / C++

Join the community!

Sign-up for our free monthly newsletter at
newsletter.wrox.com

powered by

books24x7

Programmer to Programmer™



Take your library wherever you go.

Now you can access more than 70 complete Wrox books online, wherever you happen to be! Every diagram, description, screen capture, and code sample is available with your subscription to the **Wrox Reference Library**. For answers when and where you need them, go to wrox.books24x7.com and subscribe today!

Find books on

- ASP.NET
- C#/C++
- Database
- General
- Java
- Mac
- Microsoft Office
- .NET
- Open Source
- PHP/MySQL
- SQL Server
- Visual Basic
- Web
- XML



www.wrox.com