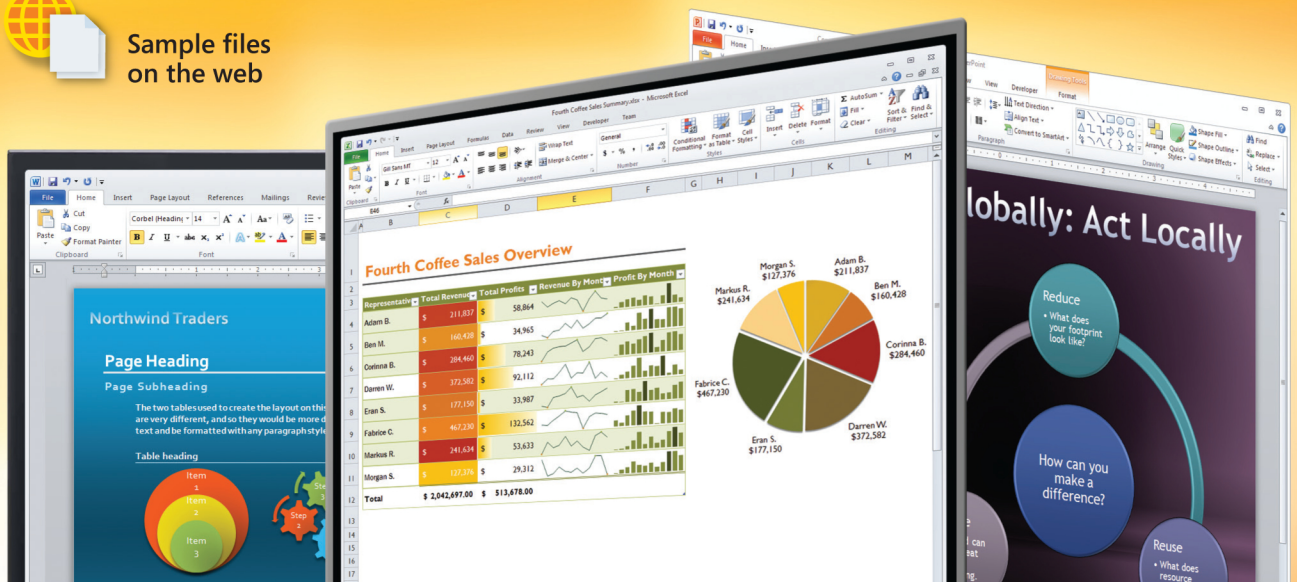
Microsoft

Documents, Presentations, and Workbooks

Using Microsoft® Office to create content that gets noticed

Stephanie Krieger

Sample files on the web



Microsoft®

Documents, Presentations, and Workbooks: Using Microsoft® Office to Create Content That Gets Noticed

Stephanie Krieger

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 M 6 5 4 3 2 1

Microsoft Press titles may be purchased for educational, business or sales promotional use. Online editions are also available for most titles (http://my.safaribooksonline.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or send comments to mspinput@microsoft.com.

Acquisitions and Development Editor: Kenyon Brown
Production Editor: Kristen Borg
Production Services: Octal Publishing, Inc.
Technical Reviewer: Kurt Schmucker
Indexing: Lucie Haskins
Cover: Karen Montgomery
Illustrator: Robert Romano

978-0-735-65199-9

For Shauna Kelly—a talented and generous document expert, and a woman of great strength and grace. I wish I had gotten to know you earlier and better, but am honored to know you at all—and I'm just one of a great many people who are better for having met you.

Contents at a Glance

Table of Contents

Introduction

Welcome to the PCs and the Macs. I am delighted to be able to bring you a resource for experienced Microsoft Office users on both platforms. Whether you work only with Microsoft Office 2010 or only with Microsoft Office for Mac 2011, or you work across platforms as I do, this book is for you.

As a document consultant and trainer, I've often been frustrated by the lack of available resources that move beyond 'click here' or 'point there' to explain *why* things work the way they do, which best practices can make a real difference to your work, and what tools you may be missing that could simplify your work and expand your possibilities. So when I began writing books about Microsoft Office, it was exactly what I wanted to provide. For Microsoft Office 2007, I had the opportunity to do that with my book *Advanced Microsoft Office Documents 2007 Edition Inside Out*. Now, a few years later, I've updated and expanded on that book to bring you *Documents, Presentations, and Workbooks* for both Office 2010 and Office for Mac 2011.

Throughout this book, you'll learn about new features in both Office 2010 and Office 2011. You'll also learn about new technologies and related tools available to users on both platforms, such as Microsoft SharePoint 2010, Windows Live SkyDrive, and Microsoft Office Web Apps. But this book is *not* about new features.

What this book *is* about is helping you to put what you already know together with both new and existing methods and concepts to work the way the experts do; helping you create the kind of documents and templates you've always wanted; and giving you the tools to take full advantage of the capabilities in these programs to find the simple solutions you've often wondered about. In short, it's about doing less work, getting better results, and expanding your possibilities.

Who Will Benefit Most from This Book

You're an experienced Microsoft Office user and you don't need to start from scratch. This book takes you at your word, so the basics you already know are not repeated here. Though a few chapters that are specific to advanced tasks (such as Microsoft Excel PivotTables) do start from the beginning and move at an advanced pace, you'll find far more lists of key tips, hands-on concepts, and advanced timesaving or troubleshooting methods in most chapters than step-by-step instructions for using the basics of a feature. Following are a few examples of what you'll find here:

- In Part I of this book, "Document Essentials," you'll find guidance for planning effective documents, presentations, and workbooks; sharing content electronically; choosing the best program for the task; and understanding features that make a difference to the content you create in multiple Microsoft Office programs.

  For example, in Chapter 5, "Doing More with Less Work: Key Cross-Program Features," you'll learn about a few features, including the incredibly valuable formatting functionality known as themes. But it's not about how to click to apply a built-in theme to your document. In that

chapter, you'll find a thorough introduction to themes that explains not just what they are, but why they are important and how they integrate with features across Microsoft Word, PowerPoint, and Excel. You'll also learn how to customize and create your own themes to more easily implement your own formatting requirements (such as corporate branding) across your documents, presentations, and workbooks.

- In Part II, "Word," you'll learn about the six features that can enable you to create any document you need and how to put the pieces together to create better documents more easily.

  For example, the Word chapter on styles (Chapter 8, "Styles") does not walk you through steps for how to use the New Style dialog box. Instead, the chapter addresses the way that styles are structured, how to create effective style sets, and how to manage styles in documents and templates. It also provides guidance for more advanced tasks, such as how to simplify your work with the often overcomplicated lists (bullets, numbering, and outline numbering).

- In Part III, "PowerPoint," learn how to craft great presentations that are as easy to edit as they are powerful to share, discover how you can do more with Office Art graphics, and get help for taking your presentations to the next level.

  For example, the PowerPoint chapter on creating presentations (Chapter 13, "Creating Presentations: From Theme to Master to Slide") does not step you through the basics of applying a layout or explain the difference between adding your logo to a master or an individual slide. Instead, this chapter explains the relationship between themes, masters, layouts to help you create presentations and templates that look and behave the way you want. It provides best practices for creating and customizing layouts, working with various content types (such as charts or embedded Word tables) in your presentation, and tips for managing and troubleshooting presentations.

- In Part IV, "Excel," explore the documents known as Excel workbooks and the powerful functionality you can include in them, from formatting worksheets and working with data to using charts, tables, data visualizations tools, and PivotTables.

  For example, the chapter on charts (Chapter 20, "Charts") does not step you through creating a basic chart or explain basics such as what an axis is. Instead, the chapter gives you the direction you need to create and customize charts efficiently; tips for creating more effective charts; help for more advanced tasks such as managing data, combining chart types, and working with secondary axes; and step by step instructions and troubleshooting for creating complex chart types such as price/volume charts and bubble charts.

- In Part V, "Templates, Automation, and Customization," get ready to take it to the next level. Here you can learn about creating and sharing templates for Word, PowerPoint, and Excel, but you can also discover new ways to both simplify and expand on your use of Microsoft Office.

In Chapter 23, "VBA Primer," and Chapter 24, "Office Open XML Essentials," you get thorough, detailed introductions to extending Microsoft Office using Microsoft Visual Basic for Applications (VBA) macros and the technology that underlies the current Word, PowerPoint, and Excel file formats, Office Open XML.

I strongly believe that the programming capabilities that are built into Microsoft Office can greatly simplify your work and save you time. You absolutely don't need to be a developer to make use of this powerful functionality. That said, an understanding of core features and experience working with Office 2010 or Office 2011 is essential to being able to capitalize on the available programmability and customization options in these versions of Word, PowerPoint, and Excel. So, those who have mastered the essentials of complex document production that are covered in Parts I through IV of this book will get the most from Part V.

Additional Resources for Reviewing the Basics

At the beginning of most feature-specific chapters throughout this book, you see a reference to this introduction as the place to find recommendations of additional resources for those who want more basic information on a given topic. The *Step by Step* book series is a good place to start for core basics with Office 2010:

- *Microsoft Word 2010 Step by Step*, by Joyce Cox and Joan Lambert III
- *Microsoft PowerPoint 2010 Step by Step*, by Joyce Cox and Joan Lambert III
- *Microsoft Excel 2010 Step by Step*, by Curtis D. Frye

For Office for Mac 2011, one good resource that starts at the beginning is *Office 2011 for Macintosh: The Missing Manual*, by Chris Grover.

Additionally, though Microsoft Outlook is not an application covered in this book, Microsoft Outlook for Mac 2011 is a completely new program that warrants a mention. So, if you're looking for help getting started with Outlook for Mac, you might want to check out *Microsoft Outlook for Mac 2011 Step by Step*, by Maria Langer.

For additional resources at all levels, including links to blogs and other websites from members of the Microsoft Office product teams for both Windows and Mac as well as several Microsoft MVPs, see the online companion content for this book, described later in this introduction.

What You Can Expect from This Book

This book is a comprehensive guide to advanced document and template production, trouble-shooting, and customization using Word, PowerPoint, and Excel.

- Approximately 30 percent of this book's content covers Word topics.
- Approximately 20 percent each is devoted to PowerPoint and Excel topics.
- About 15 percent each focuses on big-picture concepts that cross multiple programs (such as managing electronic documents and planning documents) and programmability topics (VBA and Office Open XML).

The most important distinction I want to make for those venturing into this book is that it's not a general guide to Office 2010 or Office 2011. For example, you will learn how to lay out complex pages, create professional presentation graphics, and troubleshoot documents more easily, but you won't learn how to configure email settings or create a database here.

So, if you're ready to take your work with Microsoft Office documents to the next level, read on, and welcome to Office 2010 and Office 2011.

Companion Content

A number of sample files are available for working with the tasks discussed in several chapters throughout this book. Additionally, the companion content for this book includes a selection of bonus content on topics not specifically addressed in the book itself. You'll see references to the sample files and bonus content where applicable throughout the book, and a complete list follows.

To access and download the companion content, visit: *http://aka.ms/651999/files*

Chapter or Topic	Content
Chapter 3	■ Hidden Data Checklist.xltx
Chapter 6	■ Fearless.docx ■ Fearless.vsd
Chapter 9	■ Table exercise.docx ■ Completed table.docx
Chapter 10	■ Adjacency Report.dotx
Chapter 12	■ Nested field sample.docx
Chapter 19	■ Data Visualization Examples.xlsx
Chapter 20	■ Price-Volume.xlsx
Chapter 21	■ Pivot Data.xlsx ■ PivotTables.xlsx
Chapter 23	■ Sample macros.docm ■ PrimerSamples.bas
Chapter 24	■ Copy XML.txt ■ Text editing.docx
Bonus Content	■ "Securing Access to Your Documents—Document Protection Tools and Options" ■ "Should I Consider Microsoft Publisher for My Document?" ■ "Reference Tables and Tools" ■ "Visualizing Data with Excel and Visio" ■ "Managing VBA Errors" ■ "Creating VBA UserForms"

Support for This Book

Every effort has been made to ensure the accuracy of this book and the companion content. Microsoft Press provides support for books and companion content at the following website:

http://www.microsoftpressstore.com/about/support

You can also look for updates and a list of errata at the following website:

http://aka.ms/651999/files.

Questions and Comments

If you have comments, questions, or ideas regarding the book or the companion content, or questions that are not answered by visiting the sites above, please send them to Microsoft Press via email to mspinput@microsoft.com.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback is our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress.*

Acknowledgments

Writing a book such as this is anything but a sole effort. Many people contributed to this project in many ways. I don't think it's possible to thank everyone I should. But, here is a start:

- To Kenyon Brown, acquisitions editor, developmental editor, and dedicated shepherd of this book. Thank you for believing in this book and in me, and for your (tremendous) patience.
- To Kristen Borg, production editor. Thank you for your meticulous care of this book, your patience, and your kindness.
- To Kurt Schmucker, this book's technical reviewer and one of the best people on this planet to work with. I am so lucky that you were willing to be a part of this project.
- To Dennis Cohen, who stepped in and completed the technical review of several chapters.
- To all of the many people at O'Reilly and Microsoft Press who worked hard to bring this project to completion and helped to make it a better book, especially copyeditor Rachel Monaghan, indexer Lucie Haskins, illustrator Robert Romano, and designer Mark Paglietti—as well as desktop publishers Bob and Dianne Russell at Octal Publishing, Inc.
- To Beth Melton, for jumping in to help when you didn't have the time.
- To the many generous and wonderful people on the Office 2010 and Office for Mac 2011 product teams who answer my (incessant) questions and contributed many invaluable tips to this book. In particular, thank you Marcus Aiu, Roger Baerwolf, Bill Barry, Adam Callens, Tristan Davis, Jodie Draper, Amani Ahmed Dye, Shawn Larson, Doug Mahugh, Chris Maloney, Blair Neumann, Rick Schaut, Derek Snook, Stuart Stuple, and Scott Walker.
- To the many wonderful people I have the pleasure to work with, especially those of you who have been so patient in the face of book deadlines these last several months, including Nathalie Alfred, Chris Bryant, Nancy Crowell, Erika Ehrli, Sabrina Goebel, Stephanie Hancock, Tess Kander, Courtney Keppler, Tal Krzypow, Sharon Meramore, Steven Michalove, Andrew Mole, Michael Pierce, Kurt Schmucker, Karen Shogren, Helena Snowden, Kendall West, Jennifer Winters, and Ayça Yuksel.
- To Echo Swinford, Adrienne Sam, and Stacey Barone, for making it possible for me get everything else done, too.
- To the Microsoft MVP program that I am so privileged to be a part of, and the many smart and generous experts within that community who make me better at what I do.

- To my family—Mom, Dad, Elise, Michael, Ari, and Jared—for always believing in me despite rarely seeing me.
- To the dear friends who are always there for me with unwavering support and text-message therapy, especially Alex Jennings, David Rubin, and Stuart Stuple.
- To Gayle Gibbons Madeira, the lazy operator herself.
- And to the two small, furry creatures who provide the daily joy that is better for stress relief than all the lavender and chamomile tea in the world, Janie and Murphy.

Chapter 5

Doing More with Less Work: Key Cross-Program Features

In this chapter, you will:

- Uncover the power of document themes
- Discover how themes integrate with features across Microsoft Office
- Learn to create your own themes
- Explore advanced picture editing tools
- Understand how to easily share content across programs
- Get tips for leveraging tools across Microsoft Office
- Learn about recovering your unsaved content in Office 2010

Are you ready? This chapter just might change your life (well, the part of your life that you spend creating documents, anyway). Here we explore how you can use the same tools and even the same content in multiple programs.

One of the best developments in recent versions of Microsoft Office is the proliferation of features that exist across the suite. Why is that such a good thing? The obvious answer is that it's easier to get up to speed with a new version when features you learn in one program work the same in others. But, in the long term, it also means you can reuse more content (and formatting) not just between the documents you create in one program but across multiple programs. So you can work more quickly, do less work, and get more consistent results in the process.

In a chapter on this subject, there is only one place to begin—document themes. Themes were first introduced in Office 2007 and Office 2008 for Mac, and they are among the best things to ever happen to Microsoft Office documents. Like nested tables in Microsoft Word, customizable layouts in Microsoft PowerPoint, and tables in Microsoft Excel, themes dramatically increase your options for formatting content. Even better, themes help you create higher-quality content more easily in all three programs.

In this chapter, you'll learn why themes are important and what they can do for your documents across Word, PowerPoint, and Excel. In addition, we'll look at new cross-program functionality for formatting pictures and explore the types of content and formatting that you can easily share across programs.

And before you leave this chapter, Office 2010 users can learn about a new feature available in Word, PowerPoint, and Excel that just might make your day (or save it). You can now often recover document content you believe you've lost, even if you never saved the file.

See Also To learn how to use nested tables in Word to simplify complex page layouts, see Chapter 9, "Tables." To learn about customizing PowerPoint slide layouts, see Chapter 13, "Creating Presentations: From Theme to Master to Slide." To learn about the power of Excel tables, see Chapter 17, "Data-Based Documents: Formatting and Managing Worksheets," and Chapter 18, "Working with Data."

Note Office 2010 users should note that themes are also available in some types of Microsoft Outlook content. For example, when editing the body of an email message, find the Themes group on the Options tab and theme fonts and colors on the Format Text tab, under Change Styles. You can also find the Format Text tab options in the body of a calendar appointment or the notes area of a contact.

Because this book is about content you create in Word, PowerPoint, and Excel, Outlook is not referenced throughout this chapter. However, note that Outlook rich content is based on Word capabilities. So, for example, just as you can save a custom theme or apply different themes or theme elements from the Themes group on the Ribbon in Word, you can do the same from the Themes group on the Options tab in Outlook 2010.

Themes are not available in Outlook for Mac 2011. It's too bad, but—considering that the Office for Mac team built Outlook from the ground up for this release and it's far better in its first version than Microsoft Entourage ever was—let's give them a pass on this omission.

Introducing Document Themes

Themes were born in PowerPoint. Fundamentally, they are the evolution of PowerPoint design templates. And they are one of many powerful tools enabled by the Office Open XML Formats.

- In Word, PowerPoint, and Excel, themes provide sets of colors, fonts, and graphic formatting effects that you can apply to an entire document, presentation, or workbook with just a click.
- In PowerPoint, the same themes also provide the slide master, slide layouts, and slide background gallery styles for your presentation.

  See Also In Office 2010, you can take advantage of those same slide background gallery styles for shape fills in Word, PowerPoint, and Excel. To learn more about using this feature (known as Other Theme Fills) to further coordinate your Word documents and PowerPoint presentations, see Chapter 10, "Managing Graphics," and Chapter 14, "Creating Professional Presentation Graphics."

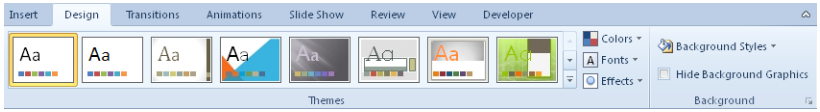
There are 40 themes built in to Office 2010 and 57 themes built in to Office for Mac 2011. Office 2010 also provides additional themes from Office.com in the Themes gallery.

Note In Office 2010, you see an Office.com category in a few places, such as the Themes gallery and the SmartArt graphics dialog box. The Office.com categories are known as *connected galleries* because they periodically update automatically as new content becomes available online. These galleries don't display all available content on Office.com, but rather a selection of recommended content.

Some of the Office.com themes that you see in the connected galleries in Office 2010 are among the built-in themes that Mac users see in Office 2011. If you are working in Office 2010 and you don't see this category in the Themes gallery, at the bottom of the gallery, click Enable Updates from Office.com. (If the gallery doesn't appear and the option is unavailable, it may have been disabled by your system administrator.)

You can apply a built-in theme as-is, customize a built-in theme, or create your own completely custom theme (to help implement your company's branding across all of your documents, for example). One quick way to quickly customize a theme is to mix and match the elements of built-in themes. As you see in Figure 5-1, you can apply an entire theme from the Themes gallery or apply just the colors, fonts, or (in Office 2010) graphic effects.

PowerPoint 2010



PowerPoint 2011

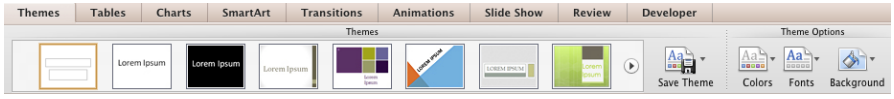


FIGURE 5-1 The Themes gallery and theme element galleries, shown in PowerPoint 2010 and PowerPoint 2011.

As shown in Figure 5-1, the Themes gallery is available on the Design tab in PowerPoint 2010 and the Themes tab in PowerPoint for Mac 2011. In Word 2010 and Excel 2010, this gallery (shown in Figure 5-2) is available on the Page Layout tab. In Word for Mac 2011 and Excel for Mac 2011, find this gallery on the Home tab.

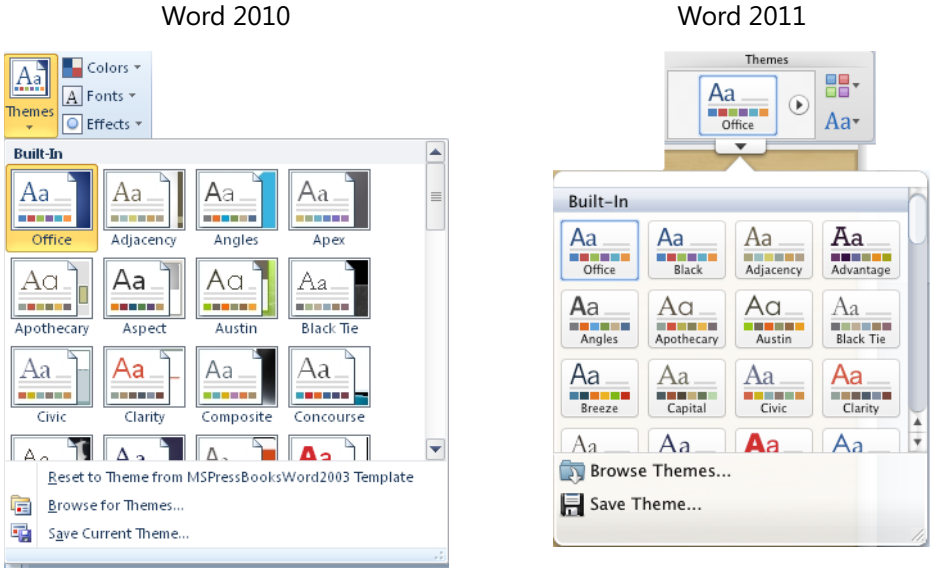


FIGURE 5-2 The Themes group, shown in Word 2010 (left) and Word 2011 (right).

Note As you see in Figures 5-1 and 5-2, you get different information about the available themes in these galleries depending on the program you're using. In PowerPoint 2010 and PowerPoint 2011, the Themes gallery displays thumbnails of the title slide layout for each theme; in PowerPoint 2010, those thumbnails also show a preview of the fonts and colors from the theme (though they're admittedly tough to see in a black-and-white image). In Word and Excel on both platforms, the theme thumbnails in the gallery display just font and color previews for the theme.

For Mac Users

In Word 2011 Print Layout view, on the Home tab, you see the Themes gallery but do not see the Theme Color and Theme Font galleries by default. However, you can easily add the full Themes group (including the Theme Color and Font galleries) to the Layout tab. Find this option in the Word Preferences dialog box, on the Ribbon tab.

Note also that just because you don't have a separate gallery for theme effects like Office for Windows users get, it doesn't mean you can't easily mix and match themes with different graphic effects.

See Also Adding the Themes group to the Layout tab in Word 2011 is one of several options that you can configure on the Ribbon in the Office 2011 programs. For more information about the Office 2011 Ribbon, see Chapter 1, "Welcome to Office 2010 and Office for Mac 2011." Learn about how Mac users can mix-and-match theme effects, and how advanced Microsoft Office users on either platform can create their own custom theme effects, later in this chapter.

Understanding the Importance of Themes

Every Office 2010 and Office 2011 document contains a theme, whether you make use of it or not. And themes integrate with a great deal of the available formatting tools across Word, PowerPoint, and Excel.

- In both Word and Excel, theme fonts and colors integrate with all style types that you can customize. In Word, this includes paragraph, character, list, and table styles. In Excel, it includes cell, table, and PivotTable styles.
- In Word, PowerPoint, and Excel, theme fonts, colors, and graphic effects integrate with the Quick Style galleries (that is, the formatting galleries) for SmartArt graphics, charts, and shapes. Additionally, fonts and colors integrate with the WordArt gallery options in all three programs (a feature also known in Word as Text Effects).
- In PowerPoint, virtually everything integrates with themes because a theme controls not only colors, fonts, and effects, but also the masters, layouts, and slide background options.

Essentially, themes integrate with almost any feature that uses color, fonts, or graphic formatting effects. For example, in Word, many types of building block entries (called *document elements* in Word 2011)—such as cover pages, headers, and footers—coordinate with the active theme because they include content that is formatted with font and color (and sometimes graphic effects). Similarly, in Excel, themes can integrate with features such as conditional formatting and sparklines.

Simply put: you can't take advantage of most Office 2010 or Office 2011 formatting tools without using themes. And since themes make it easier to format your documents, why would you not want to use them?

What if you don't use color or graphics in your content and always want your font to remain the same? As mentioned earlier, the document contains a theme whether you make use of it or not. So it's just as important to know how themes work and how they are implemented in your documents whether or not you take advantage of them.

Exploring Theme Elements

See Also This section addresses cross-program elements of themes—colors, fonts, and graphic effects. For a detailed look at PowerPoint-specific theme elements (including the slide background gallery styles that are available in Office 2010 as other theme fills for shapes), see Chapter 13.

As mentioned earlier, the cross-program elements of a theme (those that are available in all three programs) are colors, fonts, and graphic formatting effects. In this section, we'll take a brief look at what that means in more detail.

Explore Theme Colors

A theme color set includes 12 colors. Their breakdown is easy to see in the Create New Theme Colors dialog box (known as the Create Theme Colors dialog box in Office 2011), shown in Figure 5-3 and described in the list following it.

Note This dialog box is available in Office 2010 at the bottom of the Theme Colors gallery in Word, PowerPoint, and Excel. In Office 2011, this dialog box is available only in PowerPoint, from the bottom of the Theme Colors gallery.

See Also Learn about customizing themes in the section "Customizing Themes," later in this chapter.

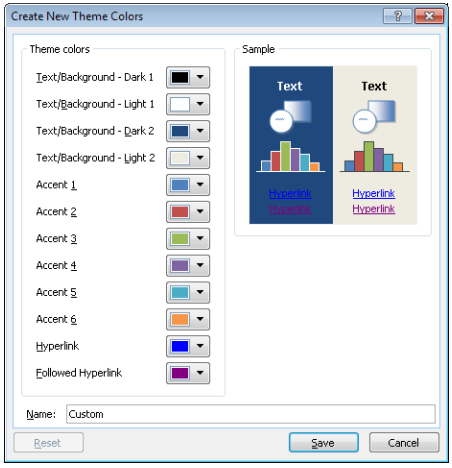


FIGURE 5-3 The Create New Theme Colors dialog box, shown in Office 2010.

- Dark 1 and Light 1 (the primary dark and light text and background colors). Note that these are typically black and white but are not required to be.
- Dark 2 and Light 2. These are also designed for use as text and background colors (predominantly for the purposes of PowerPoint slides), so they should be substantially dark and light colors, rather than the mid-range colors you are likely to use for content such as graphic fills.

- Six accent colors. These are used automatically to format Office Art graphics such as Excel charts (the six accent colors correspond to the first six data series in a chart), SmartArt diagrams, and shapes. It is recommended that these colors be mid-range (that is, not extremely dark or extremely light) so that they are visible on either light or dark backgrounds.

  See Also When you choose a colorful range of accent fills for a SmartArt layout, many SmartArt layouts begin with Accent 2 by default rather than Accent 1. There is a reason for this, but it is also possible for advanced users to create their own SmartArt color styles that begin with Accent 1. To learn more about SmartArt styles, see "Explore Theme Effects," later in this section, and Chapter 14.

  Note that creating custom SmartArt color styles requires the use of Office Open XML. To learn about the basics of Office Open XML, see Chapter 24, "Office Open XML Essentials." Those experienced with Office Open XML can check out the MSDN Office Developer Center article "Creating Custom SmartArt Graphics" to learn how to create your own SmartArt color styles and download samples that include a custom SmartArt color style that you can install and try out for yourself. Find this article at http://msdn.microsoft.com/en-us/library/gg583880.aspx.

- Hyperlink and followed hyperlink colors. These colors do not appear in any color palette but are automatically used when you create hyperlinks in a Word, PowerPoint, or Excel document.

The first 10 theme colors (that is, theme colors other than hyperlink and followed hyperlink) appear as the top row of the color palette that you see across Word, PowerPoint, and Excel. Notice, as shown in Figure 5-4, the heading Theme Colors at the top of the palette. The subsequent five rows beneath the primary theme colors are tints (lighter variations) and shades (darker variations) of the theme colors that Microsoft Office generates automatically.

FIGURE 5-4 The Theme Colors palette, shown in Office 2010.

Can You Control the Tint and Shade Percentages in the Theme Colors Palette?

The short answer to this question is no: you can't control the tints and shades that appear in the Theme Colors palette. From top to bottom, following are the tints and shades you'll see:

- Mid-range colors get tints of 80%, 60%, and 40% lighter than the original and shades of 25% and 50% darker than the original.
- Extremely light colors get shades of 10%, 25%, 50%, 75%, and 90% darker than the original.
- Extremely dark colors get tints of 90%, 75%, 50%, 25%, and 10% lighter than the original.
- Pure white gets shades of 5%, 15%, 25%, 35%, and 50% darker.
- Pure black gets tints of 50%, 35%, 25%, 15%, and 5% lighter.

So, you can't change how colors look in the palette, but that doesn't mean that you are limited to those tints and shades when creating custom content that you want to behave like part of the theme (that is, to update automatically if a different theme is applied). You can't make this type of change in the Microsoft Office programs, but this kind of customization is one of the benefits of learning how to work with Microsoft Office extensibility using Visual Basic for Applications (VBA) or the Office Open XML Formats. You can use a macro to apply a custom tint or shade while still adhering to the theme, or you can do the same by adding a custom tint or shade value in the XML under the hood of your document.

See Also To learn about the basics of VBA, see Chapter 23, "VBA Primer." To learn about the basics of Office Open XML, see Chapter 24.

Explore Theme Fonts

At its simplest, a theme font set specifies two fonts—one for headings and one for body text. If you (and anyone you might create a theme for) use only one editing language, that's most of what you need to know.

However, if you or those you work with use more than one editing language, you might want to know that a theme font set can actually specify up to three pairs of heading and body fonts, as follows:

- **Latin** This font set is for Latin script languages such as English and French, and also includes some other scripts, such as Cyrillic.
- **East Asian** This font set is for Asian languages such as Japanese and Chinese.

- **Complex Script** This font set is for complex script languages such as Hebrew, Arabic, Thai, or Hindi.

In addition to these font pairings, a theme can specify unique heading and body fonts to use for specific language scripts.

It's important to understand that a theme specifies fonts; it does not actually store font files in the document. So, if you share documents electronically, fonts appear correctly for recipients only if they have the fonts you're using, whether or not you have specified those fonts in the theme.

Additionally, keep in mind that heading and body fonts are applied by default to certain aspects of your documents in Word, PowerPoint, and Excel. However, you can choose to use your heading font in body text or assign your body font to a heading as needed. Some key default settings for heading and body fonts are as follows:

- In Word, the theme body font is the primary font (that is, the font used for the document defaults and Normal template) in a new Word 2010 or Word 2011 document. Additionally, the theme heading font is applied by default to several of the built-in heading styles.
- In PowerPoint, the theme heading font is applied by default to title text placeholders on the slide master and layouts; the theme body font is applied by default to body text placeholders and other objects, such as Office Art objects including charts, SmartArt graphics, and text boxes within shapes.
- In Excel, the theme body font is the default font for worksheet and chart text. It's also used for most theme-aware built-in cell styles (other than the Title style, which uses the theme heading font), and is used by default in other built-in style types, even for content such as heading row formatting in table styles.

  In Excel 2010, the worksheet row and column headings also take on the active theme body font by default.

In all of the preceding examples, you can customize these default settings. For example, you can customize Word styles or PowerPoint placeholders to use whatever font you choose (for example, you can apply the theme heading font where the theme body font is the default).

In Excel, you can do the same. Notably, if you set the default font for new workbooks to be something other than the theme body font, whatever font you select (theme-ready or otherwise) will then be the automatic font anywhere the theme body font is the default (such as worksheet text and most cell styles).

See Also You can create your own theme fonts from within the Office 2010 programs Word, PowerPoint, and Excel. You cannot do this from within Office 2011 programs, but Mac users still have an easy way to create custom theme fonts. To learn more, see the section "Customizing Themes," later in this chapter.

To learn about how to select fonts to simplify document sharing, see Chapter 3, "Managing Electronic Documents."

For help with Word styles, see Chapter 8, "Styles." To learn about customizing layouts and masters in PowerPoint, see Chapter 13. For information about customizing worksheet formatting in Excel, see Chapter 17.

Explore Theme Effects

Theme effects are all about Office Art. That is, they are the settings that populate the Quick Styles galleries for shapes, charts, and SmartArt graphics. (In PowerPoint, the top two rows of table styles, ambiguously labeled Best Match For Document, also coordinate with theme effects.)

If you're never going to create your own theme effects, that's really all you need to know about this feature. Just apply different themes and then check out the Shape Styles, SmartArt Styles, and Chart Styles galleries (or check out the changes to the Office Art content in your document) to see if you like the effects. Or, for Office 2010 users, you can also point to different options in the Theme Effects gallery to see a preview of how they will affect your Office Art content.

However, if you want to understand how theme effects work (that is, how those galleries are populated), and even potentially create or customize your own theme effects, there's quite a bit more to understand.

Note You cannot create custom theme effects from within any Office 2010 or Office 2011 program. You can, however, customize theme effects through Office Open XML or—for Windows users—by using a free download called the Theme Builder.

See Also Learn more about creating custom themes, including creating theme effects and how to get the Theme Builder tool, in the section "Customizing Themes," later in this chapter.

A theme effect set consists of three fill formats, three line formats, and three effect formats. Within each group (fill, line, and effect), the three formats are categorized as subtle, moderate, and intense. That is, there are subtle, moderate, and intense format settings for fill, line, and effect.

- Fill formats can be solid, gradient, or image.
  - Solid and gradient fills can use tints, shades, and transparencies (as well as HSL variations). Gradients can include up to 10 stops.
  - Image fills can be static (so that the image always looks the same), or they can use duotone recoloring to take on the hues of applicable theme colors.
- Line formats can also be solid or gradient, and they can specify settings such as line style and end and join types.
- Effect formats can include multiple effects, including inner and outer shadows, glow, soft edges, reflections, and 3-D format settings.

The idea that there are three sets of formats seems straightforward, but when you look at the various Office Art Quick Style galleries, you'll see that it's not quite that simple. Nowhere in any of these galleries will you actually see a subtle, moderate, or intense style that corresponds to the complete subtle, moderate, and intense formatting sets in the active theme effects. Figures 5-5 through 5-7 illustrate which elements of a theme effects set are used to populate the Quick Style galleries for shapes, SmartArt graphics, and charts.

Note Figure 5-6 shows the five SmartArt styles that appear in the Best Match For Document category at the top of the gallery in Office 2010. In Office 2011, there are no defined categories in this gallery, but these are the first five styles in the gallery (starting from top-left).

These five styles coordinate with the effects in the active theme. The additional built-in SmartArt style options (not shown here) are in the 3-D category in that gallery. Some of these use elements from theme effects as well, but some elements of each 3-D SmartArt style remain static regardless of the theme applied.

FIGURE 5-5 The Shape Styles gallery, with an explanation of included theme effects.

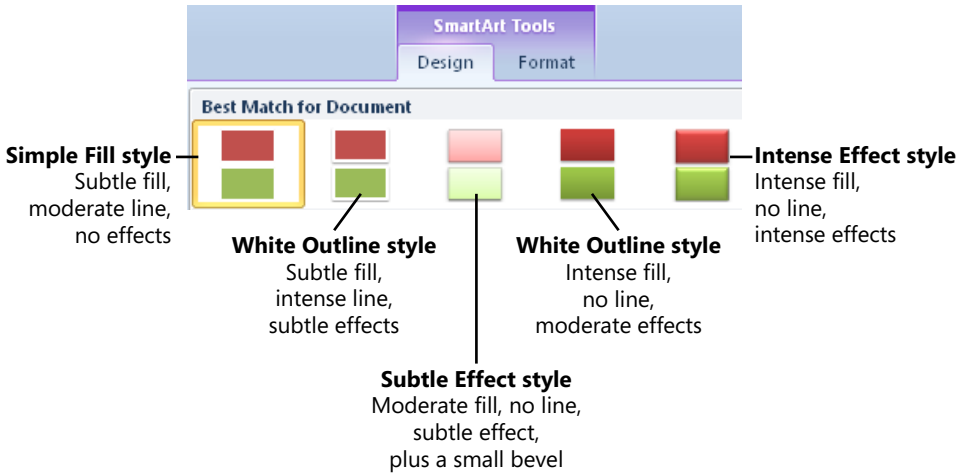


FIGURE 5-6 The first five styles in the SmartArt Styles gallery, with an explanation of included theme effects.

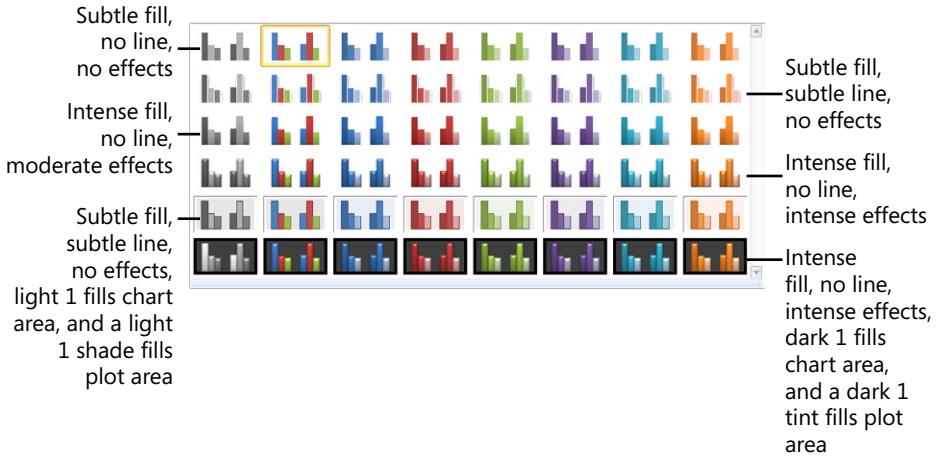


FIGURE 5-7 The Chart Styles gallery, with an explanation of included theme effects.

Following are a few key points that address common questions or areas of confusion about working with theme effects in these Office Art galleries:

- As you see in Figure 5-6, SmartArt styles are named (the names are visible in a ScreenTip when you point to the style). However, shape styles and chart styles are not named. If you point to a style in the Shape Styles or Chart Styles galleries, you see only a style number.
- You cannot change what theme effect elements are included in a given gallery entry because they are automatically populated from the theme effect settings in your active

theme. However, when you create your own theme effects, you can customize the subtle, moderate, and intense fill, line, and effect definitions that make up the majority of settings in these galleries—so you can tremendously affect the appearance of Office Art styles by customizing your theme effects.

- Although this book's screenshots are in grayscale, note that the Office Art galleries all provide options that span the six accent colors and also include your Dark 1 theme color (for SmartArt, color options are in a separate gallery; for shapes and charts, they're displayed across the galleries shown in Figures 5-5 and 5-7).

Understanding How Themes Work

If you've tried to apply a theme to a document in the past and nothing appeared to change, the document was not formatted using *theme-aware* (also referred to as *theme-ready*) formatting. For example, look at the paragraph style definition for Heading 1 in a new Word 2010 or Word 2011 document, as shown in Figure 5-8.

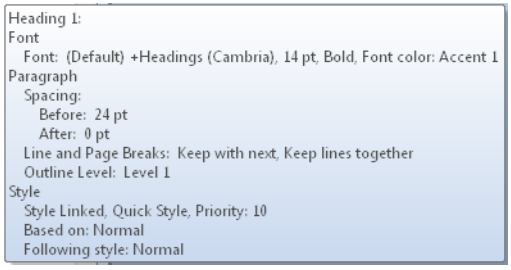


FIGURE 5-8 ScreenTip displaying the built-in Heading 1 style definition, shown in Word 2010.

- Notice that the font definition in Figure 5-8 uses the term +Headings. This indicates that it is the heading font from the active document theme. So, if the theme were changed, the font in that style would automatically change as well to the heading font from the new theme. If you only saw the font name (Cambria, in this case) and did not see either +Body or +Heading, then the font specified is not theme-ready (in other words, it will not update when the theme changes).
- Similarly, notice that the font color in the Heading 1 definition shown in Figure 5-8 is Accent 1. That means that when the theme changes, the style will update automatically to whatever color is set as Accent 1 in the active theme. If you see an RGB value in the style definition instead, that color is not theme-ready.

Note Remember that a style definition in Word includes only those features that are set for the specific style. That is, if a style uses the same font as the style it's based upon, you will not see any font or color definition in that style.

It's also worth noting that the way theme colors are specified in style descriptions is not always complete. When a style uses a tint or shade of the theme color (from the Theme Colors palette, as shown in Figure 5-10), the definition still indicates only the theme color. For example, if you apply the 40% lighter tint of Accent 1, the style definition would still list only Accent 1 as the color.

See Also To learn about base styles in Word, see Chapter 8.

Essentially, to apply theme-aware formatting, you have to apply content from theme-specific entry points. For example, consider the following:

- If the theme heading font in your active theme is Calibri, to apply that as *theme-aware* formatting (that is, formatting that updates automatically when the theme changes to use the heading font in the active theme), you must select Calibri from the Theme Fonts portion of any font list, as shown in Word in Figure 5-9. If you instead select Calibri from the All Fonts portion of the list, the font will not update when the theme is changed.

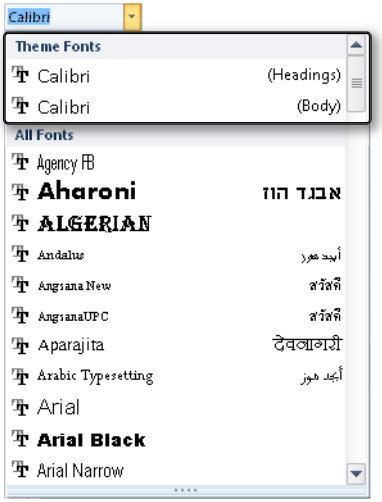


FIGURE 5-9 Use the Theme Fonts portion of a font list to apply theme-aware formatting.

- Similarly, if you want to apply the Accent 3 color from your active theme, you must do so from the Theme Colors portion of the gallery, as shown in Figure 5-10. If you apply the same color value, but do so by specifying the RGB value, for example, that color will not be theme-aware where you apply it (that is, it won't update to take on the new Accent 3 color when the theme is changed).

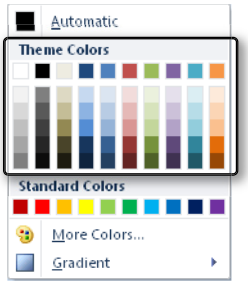


FIGURE 5-10 Use the Theme Colors portion of the color palette to apply theme-aware formatting.

- The Office Art galleries shown earlier in Figures 5-5 through 5-7 are the locations from which to apply graphic effect formatting to make that formatting to be theme-aware. Although you can apply the same effects—such as shadows, bevels, reflections, and fills—from the individual fill, line, or effect galleries on various Format tabs and dialog boxes, only the Quick Styles are theme-aware—not the individual effect options.

The difference in behavior between using and not using theme-aware formatting is similar in concept to whether you use styles or direct formatting in Word. When you apply direct character or paragraph formatting to text in Word, updating styles in the document will not affect the portions of the text that you directly formatted. Similarly, any content to which you apply direct formatting for font, color, or graphic effects—rather than the theme-aware alternatives in Word, PowerPoint, or Excel—will be unaffected when you change the active document theme.

How Themes and the Microsoft Office File Formats Are Connected

Themes are available only in the Office Open XML file formats. So, if you're working with a document that was created in a version earlier than Office 2007 on Windows or Office 2008 on Mac, you need to convert the file to the latest format before you can begin to apply theme-ready formatting.

In Office 2010 programs, find the option to convert a document in Backstage view, on the Info tab. In Office 2011, find this option on the File menu. Note that these options appear if the document was created in any version earlier than Office 2010 or Office 2011, because some capabilities have been added to the Office Open XML file formats in these versions.

It's a good idea to convert the document when this option appears, regardless of the file format. For example, in Word, the latest Office Art graphics—such as the new text effects (WordArt) and shape formatting—are not available in Word 2007 or Word 2008 for Mac documents. But if the document was created in Office 2007 or Office 2008 and is an Office Open XML Format document (that is, a document that uses the current four-character file extensions for Microsoft Office documents), you don't have to convert it to work with themes. All Office Open XML Format documents contain a theme and may already contain theme-ready formatting.

Note also that, because themes are the evolution of PowerPoint design templates, applying a theme in a presentation that uses the legacy file formats will affect the presentation. At a glance, it will appear that the theme has been applied. But there's not a one-to-one mapping between legacy design templates and themes, and correctly updating a legacy PowerPoint presentation to use themes will take a few more steps.

See Also To learn about the Office Open XML file format extensions and get an introduction to working with these file formats in Office 2010 or Office 2011, see Chapter 1. For information on how legacy design templates relate to themes in PowerPoint, see Chapter 13.

For Mac Users

Office 2011 introduces the template galleries in Word, PowerPoint, and Excel, from which you can access built-in templates, tens of thousands of templates that are hosted on Office.com, your own custom templates, and your recently opened documents. In Word and PowerPoint, these galleries also integrate with themes to give you the ability to customize a template (or, in PowerPoint, a theme) before you even create a document. (You can customize both built-in templates and those that you save to the My Templates and My Themes categories in the template galleries.)

It's often a great idea to start with an existing template even if it's not exactly what you need, because you can save time by starting with the elements that fit your requirements and customizing the rest. Office 2011 just makes that even easier.

To access the template gallery in the applicable program, on the File menu, click New From Template. (And note that these galleries appear by default each time you open the program, unless you've told them not to.) The galleries are named for the program in which they reside, including the Word Document Gallery, PowerPoint Presentation Gallery, and Excel Workbook Gallery.

In all three galleries, you can use the options on the right side to browse previews of all pages of the template. Or, just drag your mouse pointer across the thumbnail in the gallery to see multipage previews. However, in Word and PowerPoint, the righthand pane is a customization pane that enables you to apply different theme colors and theme fonts (and, in PowerPoint, to change the presentation aspect ratio), and preview those changes directly in that pane before you create the document.

In Word, you can customize built-in Print Layout view and Publishing Layout view templates before creating your document. In PowerPoint (as shown in Figure 5-11), you can customize built-in templates and themes before creating your presentation.

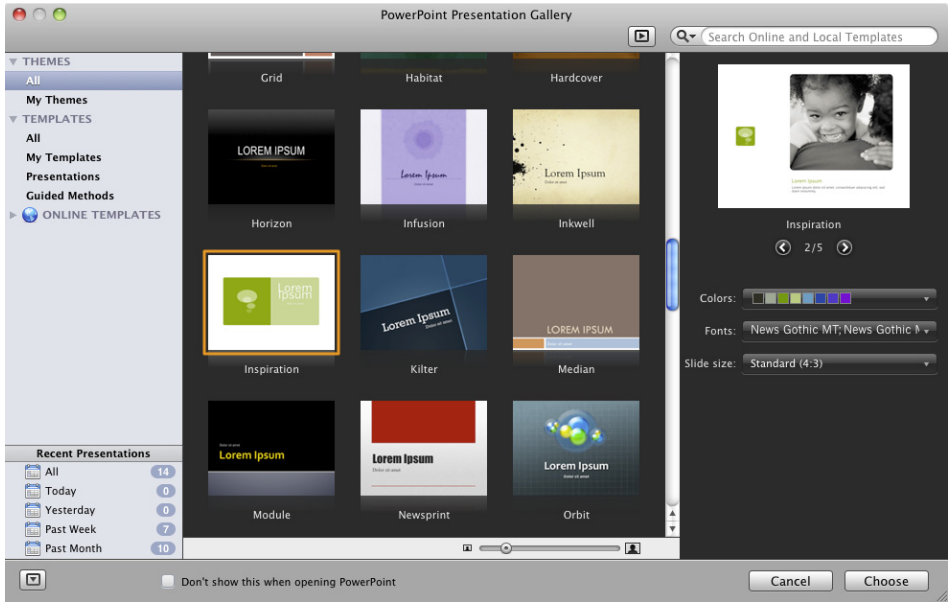


FIGURE 5-11 The PowerPoint Presentation Gallery dialog box.

Customizing Themes

Whether your goal is to create branded content across Microsoft Office (for example, to implement company branding or your own personal or professional branding), or just to create polished, professional content, the ability to customize themes is one of your most powerful tools.

Note *Branding* refers in this case to visual brand identity—that is, a set of components (such as logo, colors, fonts, layouts, and style) that define the way content created for the company (or brand within the company, or individual) should look. Typically, the purpose of a visual brand identity is to provide consistent, professional content that expresses the personality of the company or product. Just as overall branding is about distinguishing the company or brand from the competition (such as when a logo or tagline becomes instantly recognizable to the public), visual brand identity is about conveying that unique identity in all of the content you create.

That said, you don't have to be part of a large company to create a visual brand identity. You might be an independent contractor or a student, for example, and still want to create a consistent, professional look that expresses your personality in all of the content you create. So, regardless of whether you have a professional design team at your disposal to help develop your brand identity, or you're entirely on your own, you can use themes to make the job substantially easier.

Mixing and Matching to Create Your Own Theme

As we discussed earlier in this chapter, you can mix and match elements of a theme with just a few clicks to quickly create your own custom theme. Use the following tips to help you do this with as little work as possible:

- If you use PowerPoint, it's a good idea to customize your theme from within that program; in PowerPoint presentations, the theme controls many features beyond the fonts, colors, and graphic effects that it controls in Word and Excel.

  In other words, you may want to select the overall theme that you like best in PowerPoint, since that theme includes the slide master, slide layouts, and slide background style options for your presentations. Then, you can select different theme colors and fonts (and, in Office 2010, effects) to mix and match for your custom theme.

  See Also For help getting this done and more information about PowerPoint-specific theme elements, see Chapter 13.

- If you're using Office 2010, remember that you have Live Preview functionality available, so you can just point to options in the Themes, Theme Colors, Theme Fonts, and Theme Effects galleries to see how they'll appear in your content before selecting them.

- If you're using Office 2011, remember that you don't have a separate gallery for theme effects. So, even if you use only Word or Excel, start by finding a complete theme containing graphic formatting effects that you like. Then, you can select different theme colors and fonts to mix and match for your custom theme.

After you apply the theme, theme colors, theme fonts, and theme effects that you want in your document, it takes just a couple of quick steps to save that combination as a custom theme. Once it's saved, your new custom theme is automatically available to all of the documents you create in Word, PowerPoint, or Excel.

Note In Office 2010, you see custom themes and theme elements in the galleries in Word, PowerPoint, and Excel as soon as you save them in any of these programs. In Office 2011, new themes and theme elements appear the next time you start the program.

To save a custom theme:

1. At the bottom of the Themes gallery in Word, PowerPoint, or Excel, click Save Current Theme (Save Theme in Office 2011).
2. Name your new theme and then click Save.

Important Do not change the file path when saving your custom theme. Custom themes must appear in the default location to appear in the Themes gallery.

Apply the Theme from One Document to Any Other

You can apply the theme contained in any Office 2010 or Office 2011 document to any other. For example, apply the theme in a PowerPoint presentation to your Word document or Excel workbook. You can even apply a theme from an Office 2010 document to an Office 2011 document (or vice versa).

To do this, find the Browse For Themes feature (called Browse Themes in Office 2011) that appears at the bottom of all Themes galleries. From there, you can select any Office Open XML document (so it includes your Office 2007 and Office for Mac 2008 files as well) to apply the theme stored in that document to any other.

Keep in mind that Word and Excel documents don't store the slide master and slide layouts that are part of the theme in PowerPoint. So, to get all theme elements when you use the browse feature to apply a theme in PowerPoint, start from another PowerPoint presentation.

Creating a Complete Custom Theme

Just as you can apply different built-in options from the Themes and Theme Elements galleries and then save the combination as a custom theme, you can create your own theme elements to include a custom theme or even your own entirely custom theme.

- In Office 2010, you can create custom theme colors from Word, PowerPoint, and Excel.

  In Office 2011, you can do this from PowerPoint.

- In Office 2010, you can create custom theme fonts from Word, PowerPoint, and Excel.

  In Office 2011, you cannot do this from within the applications, but you can still do it easily using the TextEdit Mac OS utility.

  In both Office 2010 and Office 2011, to specify a font for a specific language script (such as a font that is specific to Japanese text rather than to any Asian language), you must use Office Open XML; you cannot do it from within the Microsoft Office programs.

- You cannot customize theme effects from within any Microsoft Office program. However, as with any other aspect of themes that can't be customized from within the programs, you can do it using Office Open XML.

- In PowerPoint 2010 or PowerPoint 2011, you can customize the masters and layouts for your theme from within the program. However, the slide background gallery options cannot be customized from within PowerPoint.

  See Also To learn about PowerPoint-specific theme elements as well as customizing slide masters and layouts, see Chapter 13.

Because themes are shared across Word, PowerPoint, and Excel, when you customize a theme element from one program, it becomes available in all three.

Note Some other Office 2010 programs include a feature named Themes, but even if you see some of the same theme names (in Microsoft Publisher 2010, for example), the features are not the same. Custom themes you create in Word, PowerPoint, or Excel are available only in those three programs.

To create a completely custom theme, including your own custom theme effects and slide background gallery options, you must use Office Open XML. As mentioned earlier in this chapter, Windows users can download a free tool—the Theme Builder—to help create the theme effects and slide background gallery options (and you can use that tool for creating theme colors and fonts as well) without having to read or write Office Open XML.

See Also For help using Office Open XML to create complete custom themes, and for additional information about creating complete custom themes using either Office Open XML or the Theme Builder tool, see the following resources:

To learn about the basics of Office Open XML, see Chapter 24.

To download the Theme Builder, visit http://connect.microsoft.com/themebuilder. *This tool is in a public beta at the time of this writing, and is expected to be available at this link indefinitely. Once you install the Theme Builder, on the Help menu, you can find a detailed theme creation guide and theme SDK documentation, both of which provide in-depth recommendations and best practices for creating themes. These documents were written for Office 2007 but are still applicable to creating themes in Office 2010 (and Office 2011). The theme creation guide was written by members of the PowerPoint team, and I am proud to have been engaged by Microsoft to write the Office 2007 version of the themes SDK document.*

Additionally, Office 2010 and Office 2011 users who want to venture into Office Open XML for creating complete custom themes can see the article "Creating Custom Themes with the Office Open XML Formats," available on MSDN at http://msdn.microsoft.com/en-us/library/cc964302.aspx. *I wrote this paper for Office 2007 and Office 2008 for Mac, but it is still applicable to the themes that you create in Office 2010 and Office 2011.*

Companion Content *If you'd like to check out the themes SDK document without downloading the Theme Builder tool, it's accessible to everyone. See the links list available in the Bonus Content folder online at* http://aka.ms/651999/files *for a link to this document. At the time of this writing, the document was written for Office 2007. However, the PowerPoint team expects to update the document for Office 2010 (which will also apply to Office 2011) and the link will remain the same.*

Create Custom Theme Colors

You can create theme colors from the Office 2010 programs Word, PowerPoint, and Excel, or from PowerPoint 2011. To do this, follow these steps:

1. If an existing set of theme colors is similar to the one you want, apply it to the active document.

   When you start to create a custom theme color set from within the Microsoft Office programs, the dialog box always starts with the theme colors in the active document. So if there is an existing set that contains some colors you want (or even colors similar to those you want), you can save time by applying that color set before you begin.

2. At the bottom of the Theme Colors gallery, click Create New Theme Colors (Create Theme Colors in PowerPoint 2011).

   Find this gallery on the Page Layout tab in Word 2010 or Excel 2010, the Design tab in PowerPoint 2010, and the Themes tab in PowerPoint 2011.

3. Set the colors as desired for each of the 12 colors included in the theme.

4. Type a name for your custom theme colors and then click OK.

   If you are creating a complete custom theme, it's customary (for ease of use by others) to use the same name for your theme elements as you do for the complete theme.

In Office 2010, after you have created a custom theme color set, you can right-click that entry in the Theme Colors gallery at any time to edit those theme colors. In PowerPoint 2011, to edit those theme colors, apply them in a document and then create a new set of custom theme colors based on them. You can then delete the original from the folder where the theme color XML file is stored, if you no longer need it.

See Also For theme file locations, see Chapter 22, "The Many Faces of Microsoft Office Templates."

Create Custom Theme Fonts

You can create custom theme fonts from within the Office 2010 programs Word, PowerPoint, or Excel. To do this, follow these steps:

1. At the bottom of the Theme Fonts gallery in Word, PowerPoint, or Excel, click Create New Theme Fonts.

   By default, when just one editing language is enabled for Microsoft Office, the dialog box that opens will show space for just one heading and one body font. For example, when English is your editing language, you can change the heading and body fonts for Latin text languages.

   To set theme fonts for East Asian languages or Complex Script languages as well, enable a language that uses those options and then restart the applicable Microsoft Office program. To do this:

   1. Click the File tab and then click Options.
   2. On the Language tab, under Choose Editing Languages, select additional languages as needed and then click Add.
   3. Click OK and then restart the program.

2. In the dialog box (shown in Figure 5-12 with theme fonts enabled for Latin, East Asian, and Complex Script languages), select the fonts that you want to include in your theme font set.

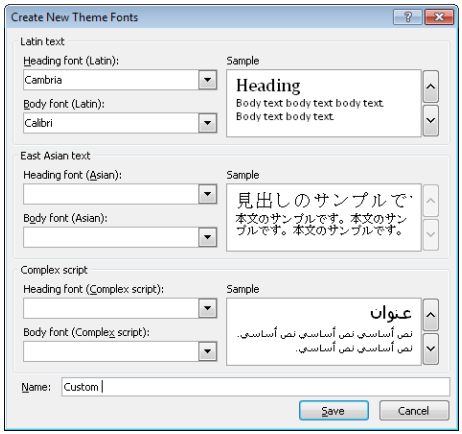


FIGURE 5-12 The Create New Theme Fonts dialog box with Latin, East Asian, and Complex Script fonts enabled.

Also notice in Figure 5-12 how the arrows to the right of the previews for Latin text and Complex Script text are enabled, but the arrows to the right of the East Asian text preview are not. This is because multiple Latin text and Complex Script languages are currently enabled, but only one East Asian language. Where you have multiple languages enabled, you can scroll through previews to see your selected fonts previewed in each language.

The drop-down list under each section provides only fonts that support the specified type of scripts. For example, the East Asian font list will include fonts such as Meiryo and MS Mincho that are designed for use with Asian fonts (Japanese, in the case of both examples).

Note that you don't have to have a font installed to save a theme font set that utilizes that font (for example, if you are creating a theme for others to use and you do not have a license for their proprietary font). However, if the font is not installed on your system, Microsoft Office will substitute another font when you apply that theme, and the results might not be desirable.

3. Name your theme fonts and then click OK.

See Also *If you want to customize scripts for specific languages, keep in mind that you can only do this in Office Open XML (or using the Theme Builder tool). However, you can copy a built-in theme font set that includes some script definitions and use a text editor to do this much more easily than you might expect. To learn how, see Chapter 24.*

For Mac Users

All of the built-in theme font sets are installed on your computer. So, even though you can't create your own theme fonts from within the Office 2011 programs, you can easily edit an existing theme font file using any text editor—such as the Mac OS utility TextEdit. Sound scary and complicated? Don't worry. You might be amazed at just how easy it is.

See Also For a tip that walks you through customizing theme fonts, see Chapter 24.

Share Custom Themes

You already know that you can apply the theme from any Office Open XML document to any other document using the Browse For Themes feature. However, you can also share the custom themes and theme elements that you create with other people or with your other computers by sharing the actual theme files.

Themes use the file extension .thmx. You can find the built-in themes for Office 2010 or Office 2011, and your own custom themes, in .thmx files that are saved on your computer. When you create custom theme colors or custom theme fonts, they are stored in separate shareable files as well (.xml files). And if you use Office Open XML and create your own custom theme effects, you can also save those as a separate file for use in Office 2010 (a theme effects file uses the .eftx file extension).

See Also To learn about where to find the files for built-in and custom themes, and how to share them, see Chapter 22.

Exploring the Advanced Picture Formatting Tools

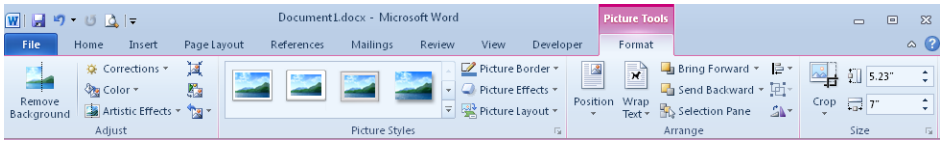
Okay, so picture formatting can be useful in many types of documents. But when it comes to complex document creation, the importance of picture formatting is hardly on par with that of a pervasive set of tools like themes. So why include a section on picture formatting here? Well, picture formatting is much improved in both Office 2010 and Office 2011. But it's also one of the best examples of consistency in tools across the Microsoft Office programs and even across platforms.

While it's true that a picture can be worth 10,000 words, it's only useful to include that picture (or any graphic) in your document if it helps you express the right 10,000 words. Using graphics just for the sake of it detracts from your points rather than enhancing them. And unless you are a photographer, it's unlikely that the picture itself is the point of your document.

However, when you have the right image that helps convey your message, the ability to customize that image can potentially add to both its relevance and its impact. So, let's take a look through the greatly improved tools for customizing images in Office 2010 and Office 2011.

To find picture editing tools, just select an image in your document. In Office 2010, this enables the Picture Tools Format tab. In Office 2011, it enables the Format Picture tab. Figure 5-13 displays both. As with any contextual tabs, they appear only when you select applicable content and provide easy access to most of the available feature-specific tools. However, we'll also venture into the Format Picture dialog box a few times in this section for a look at several advanced options.

Office 2010



Office 2011

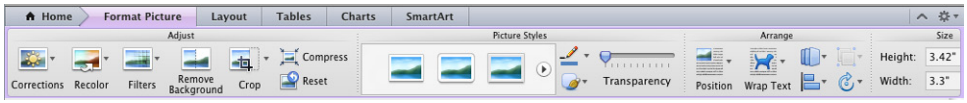


FIGURE 5-13 The Picture Tools Format tab (Office 2010) and the Format Picture tab (Office 2011) contain nearly identical options across Word, PowerPoint, and Excel.

Adjusting Images

By using a mix of new and improved tools, you can do much more than just tweak brightness and contrast. All of the tools addressed in the list that follows are available on the Picture Tools Format tab (Format Picture tab in Office 2011) in the Adjust group:

- Click Corrections for Sharpen, Soften, Brightness, and Contrast options.

  Sharpen and Soften are new; Brightness and Contrast are not. For even more options, you can click Picture Correction Options at the bottom of the gallery to open the Format Picture dialog box, where you can set precise Sharpen, Soften, Brightness, and Contrast values—giving you far more control over brightness and contrast than you've had in previous versions.

  Note that, when you open the dialog box in Office 2011, it takes you to the Adjust Picture tab, which includes both correction and color settings. This tab also contains a Mac-exclusive setting—the ability to set a specific level of transparency for the entire

image. This setting is supported in Office 2010 (which means if it's set for an image in a document you open, you can see the transparency), but it cannot be set from within the Office 2010 programs.

Tip Office 2010 users can insert a shape and then fill the shape with the desired picture to apply transparency to the picture. To do this, right-click the shape, click Format Shape, and then, on the Fill tab, click Picture Or Texture Fill to browse for your image. After you select your image, you can find the transparency setting on the same Fill tab.

Note that inserting an image into a shape doesn't constrain the image proportions, so if the shape has different proportions than the image, the picture will be distorted. If this occurs, on the Picture Tools Format tab, in the Size group, click to expand the Crop option, and then click Fill. This action turns on the crop tool and fills the shape with your image proportionally. You can then drag or resize the image within the shape for the desired result. Note that doing this will not, however, turn on Lock Aspect Ratio. To enable this setting, on the Size tab of the Format Shape dialog box (or the Layout dialog box in Word 2010), select Lock Aspect Ratio.

See Also Do you Office 2010 users feel adventurous and want a simple approach to adding transparency on your images without inserting them into shapes? Use Office Open XML—it just takes one setting. To learn how, see the sidebar "For Windows Users: Set Picture Transparency" in Chapter 24. For more information on using the improved cropping tools, see the "Cropping Images" section of this chapter.

- Click Color for new Color Saturation and Color Tone tools, as well as Recolor options.

  You can use the range of presets in the gallery or, at the bottom of the gallery, click Picture Color Options for the Format Picture dialog box, where you can set specific tone and saturation values.

  At the bottom of this gallery, also find the Set Transparent Color option, which is not new but is one of the most useful picture editing tools in Microsoft Office. With this tool, you can set one color in the active picture (and only one—don't get greedy!) to transparent (that is, remove the color from the image entirely). It's particularly handy for bitmap images (such as a logo in JPG or TIF format) where you have an object on a solid white or single-color background and want to remove the background so that your document content shows through. Just select Set Transparent Color and then click anywhere in the image on the color that you want to remove.

  Recolor isn't new either, but it offers an improved range of variations that can be very useful in some circumstances. Essentially, it enables you to provide a monochrome wash in any of your active theme colors (or a color that you specify) on a selected image. A color wash might not seem like the most powerful tool you could ask for, but it might be handier than it seems at first glance. See the sidebar "Use the Recolor Tool to Make Custom Graphics Theme-Aware," at the end of this section, to learn how to use this tool to coordinate custom graphics with your theme.

- The new Remove Background tool sounds fantastic—just drag to remove unwanted elements from a picture and leave only those you want.

  With simple images that have a clearly defined, clean background, this tool can be as great as it sounds. Unfortunately, with most images, it's likely to be less friendly than you might hope. So, if you try to use this feature and find that results are not what you expected, it's probably not something that you're doing. The feature just has its limits.

  When you click Remove Background, it guesses at what you might want removed. Again, if the image has a very clearly defined background, you might be in luck. Otherwise, it's probably not accurate, but it does get you started. You can then drag the mouse pointer across parts of the image that were removed or left behind to add or remove them from your image.

  Remove Background works a bit differently in Office 2010 and Office 2011:

  - In Office 2010, you get a new contextual tab on the Ribbon from which you can select options to add or remove elements from the background, or remove existing marks for additions or deletions. After you select add or remove, you then drag your mouse pointer across the image where you want to execute that action.
  - In Office 2011, the tool is a bit more intuitive because it just knows what to add or remove based on what you're dragging over (that is, it can tell whether what you're dragging over is currently part of the visible image). There is no contextual tab, so you just click or drag to execute the actions you want. For example, to delete an existing mark, just click the center icon on that mark where you see it on the image.

  In either version, definitely give it a try. You might find it much more effective for your particular image than what is described here. But if you don't get the results you're looking for right away, you probably shouldn't spin your wheels with this one.

- Artistic Effects (called Filters in Office 2011) are very cool. You're not likely to use them in all of your documents, or even in many. But it's good to know what they are and what they can do, because they might just provide the edge you need for one particular image to make the exact visual impact you want in your document.

  The Artistic Effects (Filters) gallery provides a broad range of effects from subtle to dramatic, as shown in the examples in Figure 5-14. There are 22 effects in all. You can also click Artistic Effect Options (Artistic Filter Options in Office 2011) at the bottom of the gallery to open the corresponding tab of the Format Picture dialog box, where you can access adjustment settings for any effect. The adjustment settings vary by effect, and you can make them more subtle or intense as desired. (Note that there is a transparency setting for effect options, but it refers to the transparency of the effect, not the image.)

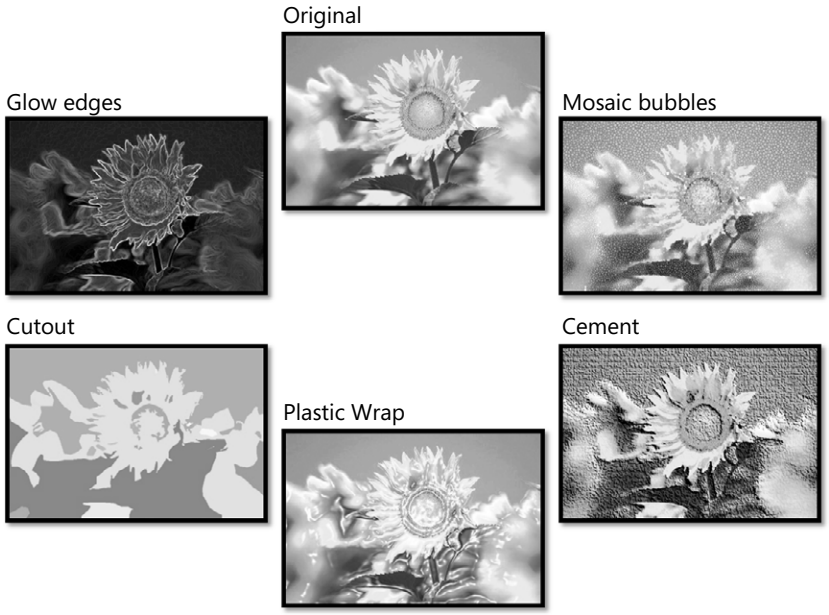


FIGURE 5-14 An image shown with several different artistic effects applied.

Use the Recolor Tool to Make Custom Graphics Theme-Aware

If you insert a photo and apply a color wash using the Recolor gallery, you might not think the results are that impressive. But a talented graphic designer recently gave me a tip about how to make better use of this tool, and I think it's one worth sharing.

If you create custom graphics in another application (such as Adobe Illustrator), you might have previously tried exporting those graphics as a Windows enhanced metafile image that (in Office for Windows) you can convert to Office Art shapes and then recolor to match your theme. In many cases, this is a good approach. But if the graphic is very complex, converting the image to shapes might be overwhelming for the document and diminish performance (of either the graphic or the document overall).

Instead, if the graphic can easily be exported as one or a few PNG images (one PNG image for each element that uses a given color), you can recolor each PNG in Office 2010 or Office 2011, layer the recolored images to reconstruct the appearance of your original graphic, and group the images together.

Of course, this solution is not ideal if you need a dozen or more colors because layering tons of images is not likely to make things easy to work with. But when your graphic

requires just one or even a few colors, this can be a nice way to save file size, preserve document performance, and still have your graphics coordinate with your theme. If you'd like to give it a try, note the following tips when preparing your graphics for export as PNG images:

- Do not export graphic objects with the color applied that you want to use in Microsoft Office, because the Recolor tool results will be affected by the original color of the object.
- When you use the Recolor tool, solid (no transparency) medium to light gray fill for your PNGs is likely to give you the best results.
- Filling the original graphic with solid black before exporting will cause only the light variations in Recolor to be available. Filling the graphic with pure white before exporting is likely to make any Recolor option appear washed-out.
- The results with this approach tend to work best with mid-range colors. Extremely bright, dense colors (such as fluorescents) are difficult, if not impossible, to match.

Be sure to export your graphic as a PNG with a transparent background, so that only the elements of the object that should get fill are recolored in Microsoft Office. For graphics with large solid areas and broad strokes, exporting your PNG at a medium resolution is likely sufficient. For graphics with fine strokes and delicate drawing detail, you'll get the best results with a high resolution.

See Also To learn about converting metafile images to shapes, see the section "Getting Your Vector Graphics into Microsoft Office" in Chapter 14.

Cropping Images

The cropping tools in Office 2010 and Office 2011 are probably the best improvements in picture editing (and maybe one of the best improvements overall in these Microsoft Office releases). Yes, picture cropping is generally pretty simple, but it can cause a lot of stress when it's cumbersome to do.

In Office 2010 and Office 2011, you can now see the entire image in shadow while you crop, and you can drag and resize the image within the crop area. Here are a couple of prime examples of where these improvements can help simplify your work with Microsoft Office:

- Have you ever inserted a picture into a picture placeholder on a PowerPoint slide and it didn't automatically crop the way you wanted? If you've ever inserted a picture into a picture placeholder, there's a good chance that your answer is yes. And in previous versions, the solution usually meant cropping the image and then manually resizing and repositioning the placeholder (often defeating the purpose of the placeholder and the layout).

- If you've used Publishing Layout view in Word 2008 for Mac to lay out documents, or you've created complex layout pages with floating images in Word 2007 or earlier for Windows, you know that cropping an image can require a lot of fussing and fighting to resize and reposition the object after it's cropped. This is easier in Publishing Layout view than anywhere else in Word, but it's never before been actually easy.

Regardless of whether you use Office 2010 or Office 2011 and regardless of whether you need to work with images in PowerPoint, Word, or Excel, you no longer have to mess with your layout at all to get the perfect crop every time. Just turn on the cropping tool, shown in Figure 5-15, and then resize or drag the image as needed within the crop area until you get exactly the results you need. Turn off the cropping tool when you're done, and the results are perfect—no trial, error, or torn hair follicles required.

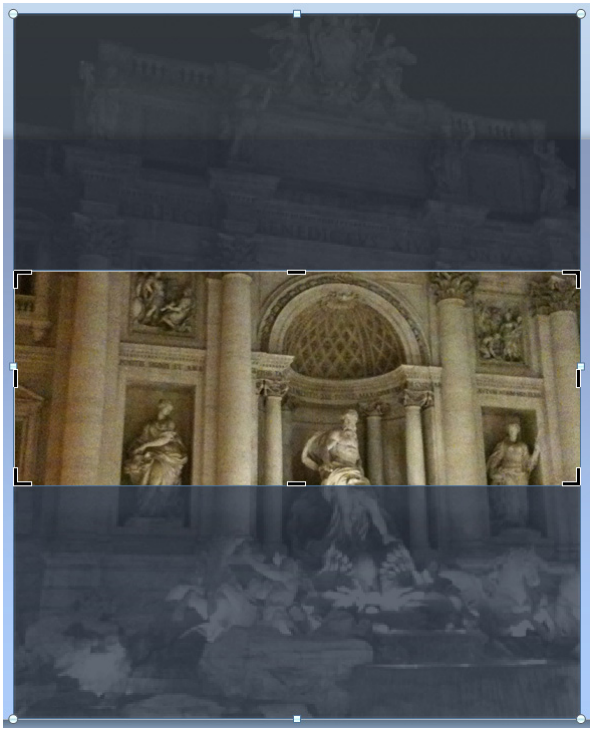


FIGURE 5-15 When the improved Crop tool is active, you can drag or resize an image within the crop boundaries without affecting your layout.

Notice in Figure 5-15 that the crop handles are available around the visible part of the image. But you also see the entire image in shadow and note that the round image sizing handles are also available. This means that if I want to show a different part of the Trevi Fountain in this image, I can just leave the crop handles alone (thus leaving my page layout unaffected) and drag the image (up or down, in this case) to change which part is displayed. I can also drag to resize the image within the crop area—for example, if I wanted to zoom right in on the figure of Oceanus in the center of the fountain.

In addition to this generally much-improved crop functionality, when you click the arrow beneath the Crop command on the Picture Tools Format tab (Format Picture tab in Office 2011), you get some very nice options, including Crop To Shape (essentially a renaming of the Picture Shape feature from earlier versions), crop to a specified aspect ratio, or shortcuts to fit or fill the image within the crop area.

Note This improved crop functionality is also available in Microsoft Publisher 2010.

For Mac Users

When you insert a picture into a picture placeholder or an object placeholder in PowerPoint 2011, you automatically see shortcuts below the placeholder to turn on the crop tool or to apply the fit and fill crop shortcuts.

Using Picture Styles and Effects

Picture styles and effects are not new to Office 2010 and Office 2011. The current Office Art effects that you also see for shapes, SmartArt graphics, and charts were introduced in Office 2007 and Office 2008 for Mac. However, there are a few things that are important to point out about these features:

- Unlike shape styles, SmartArt styles, and chart styles, picture styles do not coordinate with theme effects. Regardless of the active document theme, you get the same set of available picture styles.

  However, picture styles can be very helpful, particularly because a few of them include formatting you can't recreate from within the Office 2010 or Office 2011 applications. For example, the picture style applied in Figure 5-16 uses a shadow that requires settings not available from within the applications. If you were to copy all of the shadow settings from the dialog box and input them for another image, the results would not be the same. This is because additional properties are set under the hood, in the XML markup for the image formatting.

FIGURE 5-16 An image using the Relaxed Perspective, White picture style.

- If you like some aspects of a picture style, apply the style before applying individual formatting or effects (such as the Crop To Shape feature or a gradient border) because applying the style will remove direct formatting and effects.
- If your image contains some transparent areas and you want to use a color fill on the image, you see that the Picture Styles group on the Picture Tools Format (Format Picture) tab doesn't include a fill option. However, you can apply a color fill in the Format Picture dialog box, on the Fill tab.
- If you apply several different types of formatting or effects on an image and then later want to use the same formatting on other images, remember that you can copy formatting for images and any Office Art object just as you can copy text. To do so, just select the image containing the formatting to copy and then press Ctrl+Shift+C (Command+Shift+C in Office 2011). Then select the object to which you want to apply the formatting and press Ctrl+Shift+V (Command+Shift+V). Notice that these shortcuts are just the standard copy and paste shortcuts with the Shift key added.

Tip You can also use this method to copy and paste Office Art formatting effects between some different types of Office Art objects. For example, copy the formatting of a picture and apply it to a shape. This is particularly handy when you're customizing picture placeholders on PowerPoint slide layouts, since the Picture Styles are not available to shapes such as picture placeholders.

See Also In Office 2010, you see a new Picture Layout option on the Picture Tools Format tab, in the Picture Styles group. This option converts selected images to SmartArt graphics. To learn about working with SmartArt graphics, as well as to learn about using the various Office Art formatting effects available to images (such as shadows, reflections, and bevels), see Chapter 14.

Replacing and Managing Images

In the Adjust group on the Picture Tools Format (Format Picture) tab, you also see options to compress pictures and reset the selected picture. Additionally, in Office 2010, you see the option to change the selected picture.

- The compress feature can be invaluable in documents containing many pictures, or when you have very high-resolution images that tremendously increase the document file size. In both Office 2010 and Office 2011 (new to Office 2011, and a very welcome addition), this feature enables you to reduce picture resolution to a level that will still provide good quality for your purposes but might substantially reduce file size. It also gives you the option to permanently delete cropped areas of images to reduce file size.

- If you experiment with the various Adjust formatting tools and then decide you don't like what you've done, just click Reset to return your image to its original state (that is, its state when you inserted it). (Note that this tool does not reset cropping actions.)
- Although the Change Picture command is available on the Ribbon in Office 2010, you can also access this feature in both Office 2010 and Office 2011 when you right-click an image. With Change Picture (introduced in the previous versions of Office for Windows and Office for Mac), you can swap out the image and retain all of your formatting. It can be an incredible timesaver.

See Also For more on working with images, see the following:

For best practices when working with graphics in Word, see Chapter 10.

To learn about Word 2010 picture placeholder content controls, see Chapter 12, "Dynamic Content."

For working with all Office Art objects in PowerPoint and across the suite, see Chapter 14.

For a tip about using pictures as chart data series or data point fills, see Chapter 20, "Charts."

Sharing Content Across Programs

Because there are so many of the same tools for creating and formatting objects across the Microsoft Office programs today, reusing content across Word, PowerPoint, and Excel is much easier than ever before. For example, create a SmartArt graphic or an Excel chart in one program and then paste it into another and continue to edit it just as easily in the destination program.

In previous versions, I often recommended creating presentation graphics in PowerPoint and charts in Excel, regardless of the program in which your document lived, for ease of editing. But in many cases, this is no longer necessary. As long as you are aware of the editable content you're sharing in your documents (such as the source data that travels with a live Excel chart)—and as long as you don't mind giving recipients who have electronic access to your documents the option to edit that content—creating or editing graphics in the program in which your document lives can save you time and make it easier to keep all content visually coordinated (by using themes, for example).

Additionally, remember that when you share your documents online using Microsoft SharePoint 2010 or Windows Live SkyDrive, you and those you share with can edit those documents using Office Web Apps. So you can even update content such as SmartArt graphics in a PowerPoint presentation or charts in an Excel workbook right online.

When copying content between programs, keep the following points in mind to keep things simple in your documents:

- Remember that you have the Paste Special command—as well as Paste With Live Preview in Office 2010 and Paste SmartTags in Office 2011—which gives you choices about how your content will look (and in some cases, about what its behavior will be, such as the difference between pasting an Office Art drawing object, a picture, or an embedded object).
- Some types of content provide different options depending on whether they're pasted within the source application or in another Microsoft Office application. For example, consider the following:
  - If you copy an image that resides in a picture placeholder in PowerPoint, when you paste that picture on another PowerPoint slide, it will not include the placeholder formatting (such as the picture style). However, if you paste that same image in Word or Excel, it will carry formatting from the placeholder, such as picture styles and effects.
  - Similarly, if you copy a chart that you create in Excel, you'll get different options in Excel for how to paste the data than if you paste it into Word or PowerPoint (when pasting into Word or PowerPoint, the data will be linked to the source workbook by default).
  - If you paste a Word 2010 table into PowerPoint 2010, it pastes by default as a PowerPoint table (regardless of whether you paste it into a content placeholder or directly on a slide), and will apply the default formatting. However, when you paste the same table into Excel, it becomes Excel cell content by default and carries along some of the Word table formatting (such as borders and shading) by default.

    This is one example where behavior differs by platform. In Office 2011, if you copy a Word table and paste it into PowerPoint, it pastes as a linked document object by default. However, when you paste that table into Excel, the behavior is the same as in Office 2010—the table pastes as cell content.

When you need to share content across programs, it's important to be aware of your options so that if the result is not what you expect, you know which tools to use to help you easily get the results you wanted.

See Also For more information about Paste With Live Preview and other paste options, see Chapter 1. For more on what you can do with Office Web Apps, see Chapter 2, "Collaborating and Sharing When and Where You Choose." To learn about linking and embedding objects in Word, see Chapter 10. To learn about formatting layouts and placeholders in PowerPoint, see Chapter 13. To learn about working with Excel charts, see Chapter 20.

Using Microsoft Office As Your Toolbox

The examples in the previous section raise an important point: some programs handle certain actions better than others and have features that the others lack (even when they share overall functionality). So, whichever program you're working in, remember that you've always got the full suite at your disposal. Consider the following examples:

- If you copy a table from the web, depending on its configuration, you might not get desired results when you paste it into Word or Excel, regardless of the paste options you use. But chances are that you can get it into a table format by pasting it into one of those programs. So, for example, if the table will only paste into Word as text, try pasting it into Excel and see if it takes on a table structure (that is, see if its content fits into the cells of a worksheet as needed to create a table). If so, you can then just copy it from Excel and paste it into Word as a table.
- If you create a SmartArt graphic in Word and later realize that it can't quite do what you need, and you really need to create a diagram using shapes, you can't convert that graphic to shapes in Word. But you can do so from PowerPoint. Just copy the SmartArt graphic into PowerPoint, convert it to shapes, and then copy the shapes back into Word. (Or, in that particular example, because a graphic made of shapes might be easier to edit with the tools available in PowerPoint, finish your graphic in PowerPoint and then paste it back into Word.)
- In the example from the preceding section for Office 2011, which noted that Word tables paste as objects by default into PowerPoint 2011, you might actually find it quite difficult to get that Word table into a PowerPoint table—but not if you use Excel as a go-between. Paste the table into Excel and then copy and paste it into PowerPoint, where it will paste as a PowerPoint table by default.

Some features, such as themes, work across the suite to help keep your formatting easy to apply and to manage. Other features are specific to each program because each program has its strengths. But when you use those strengths together, you can often end up with better results for less work.

Recovering Your Important Content in Office 2010

Before leaving this chapter, we have to touch on one incredible new cross-program feature that is exclusive to Office 2010. Actually, it's two related features—AutoSave Versions and Recover Unsaved Documents. This is not like the AutoRecover capabilities that you know from previous versions. You can now recover earlier versions of documents as you work on them. You can even recover documents that you close without saving.

In Office 2010, Word, PowerPoint, and Excel save up to five AutoSave versions for a previously-saved active document and make them accessible as you work, as shown in Figure 5-17.

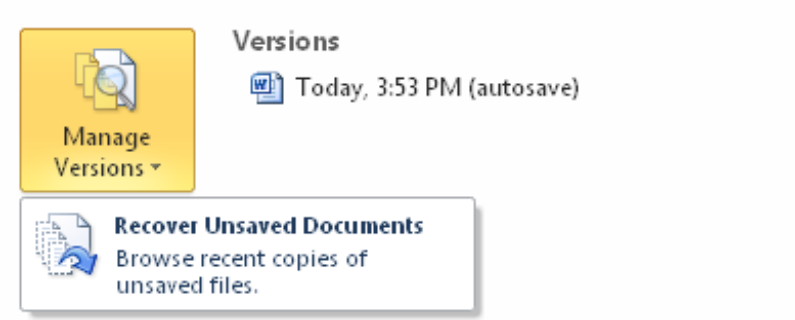


FIGURE 5-17 The Versions panel on the Info tab in Backstage view, shown in Word 2010.

If you close the document without saving, the program automatically saves the last AutoSave version, and you can recover it using the Recover Unsaved Documents command that you see in Figure 5-17. (Note that this option is called Recover Unsaved Presentations in PowerPoint and Recover Unsaved Workbooks in Excel.)

To enable the recover unsaved documents functionality, in the <Program> Options dialog box, on the Save tab, select the option Keep The Last AutoSaved Version If I Close Without Saving.

Accessible unsaved versions are automatically deleted after four days. You can, however, delete them manually at any time. To do this, start in a new (unsaved) document in Word, PowerPoint, or Excel. Click File and then, on the Info tab, under the Versions heading, click Manage Versions. In an unsaved document, the pop-up menu shown in Figure 5-17 includes the option to delete unsaved versions.

See Also For more detail about these features, see the Power User Tips sections in the Office 2010 product guides for Word, PowerPoint, and Excel. You can download the Office 2010 product guides from the Microsoft Download Center at http://www.microsoft.com/downloads/en/details.aspx?FamilyID=e690baf0-9b9a-4c47-88da-3a84f3e9b247.

Chapter 23

VBA Primer

In this chapter, you will:

- Explore the benefits of working with VBA code
- Learn how to read and understand VBA code
- Discover the core basics of how to write your own macros
- Get tips and guidance on ways to apply VBA basics for accomplishing a wide range of tasks

If you have any concerns about venturing into this chapter, take a deep breath and relax. You'll be perfectly comfortable here. This thorough primer on Microsoft Visual Basic for Applications (VBA) is written for advanced Microsoft Office users, not for programmers.

I'm not a programmer, so I won't treat you like one. The fact is that you don't have to be a programmer to make effective use of VBA (or Office Open XML, as discussed in Chapter 24). Yes, I use VBA and Office Open XML to develop solutions for clients, but that just means I'm taking advantage of all the tools that Microsoft Word, PowerPoint, and Excel have to offer for creating documents. If you can learn to format a table, create styles, or create fields in Word; to write formulas or generate charts in Excel; or to customize masters in PowerPoint, you can learn VBA.

After years of avoiding VBA because it seemed technical and scary, I fell head over heels one day after I had no choice but to venture into the Visual Basic Editor for a client. I discovered how easy it is and how much you can do with VBA even with just a basic level of knowledge. But the most important discovery was how much of the VBA language I already knew just from being an advanced Microsoft Office user. Nearly all elements of VBA that are specific to a program are the names of features and tasks you already know from using that program. Keep in mind that VBA is just an additional way to work with, and expand the capabilities of, the programs you already know.

Beyond the program-specific feature and task names, most VBA language and structure is virtually identical across Word, PowerPoint, and Excel. So, the majority of what you'll learn in this primer will apply to macros you may want to write in any of these programs. However, because this chapter assumes that this is your first introduction to writing VBA (or writing any programming language, for that matter), it uses one program for most examples, to avoid the confusion of trying to cover too much too fast. Because Word is the primary document production program for Microsoft Office, most examples throughout this primer use Word VBA. Once you're comfortable with Word VBA, you can apply all of the basics you learn to VBA tasks in PowerPoint and Excel as well.

Understanding When and Why to Use VBA

One of my favorite examples of when and why to use VBA if you're not a programmer came up one evening at dinner with a friend. She had been up until 3 A.M. the night before cleaning up tables for a report that was due that day. It was a Word document containing 50 tables copied from Excel that needed to be cleaned up and reformatted. The task took her, a power user, about six hours. At just over seven minutes per table, that isn't bad, but she wanted to know if there was a way she could have done it more quickly. She had created a few table styles and even recorded a macro for some of the formatting, but she still had click into each table to apply them and then manually take care of any unique elements for each table.

In reply to her question, I asked if she knew any VBA, and she looked at me as if I were insane. But then I told her that if she had known some basic VBA (just part of what you'll learn in this primer, by the way), she could have accounted for most of the differences among her tables in one macro and then formatted all of them at once. The task would have taken about six minutes instead of six hours. As you can imagine, learning VBA no longer seemed like a crazy idea to her.

Of course, this timesaving example is just one of several types of situations where you can benefit from VBA. As you saw in a couple of simple examples in the Excel chapters of this book, you can often use a single line of code to save substantial time or even do things you can't do through the features in the user interface. Or, to take things further, you might also use VBA to create customizations or automation for your documents and templates, such as custom dialog boxes that can help users complete form documents.

In general, the answer to the question of when to use VBA is the same as when to use any feature in the Microsoft Office programs—use it when it's the simplest solution for the task at hand. In the case of VBA, however, you may also be able to use it when there doesn't appear to be a solution for the task at all. VBA expands the capabilities of Word, PowerPoint, and Excel, so you might find yourself with easy answers to tasks that you didn't even know were possible.

In Office 2010 and Office for Mac 2011, however, it's important to ask yourself if VBA is still the simplest solution before you embark on a complex project. With the Office Open XML Formats, you can do some things in today's Microsoft Office more easily using Office Open XML—such as automatically populating document content with data from other sources. Also, some functionality that would have required automation in the past can now be done with built-in features, such as using a content control in Word 2010 to display a custom building block gallery when you need a selection of boilerplate text options that can't be deleted. However, VBA macros are still almost exclusively the way to go when you want to use automation to save time on repetitive or cumbersome tasks.

Introducing the VBA Language and Code Structure

The easiest way to begin learning VBA is to record macros and then look at what you've recorded in the Visual Basic Editor. In the subsections that follow, we'll use this method to help you become acquainted with how to read VBA code.

Note Macros can no longer be recorded in PowerPoint, but you can still write VBA macros in PowerPoint. Macros can be recorded and written in Word and Excel.

So, what is a macro? A *macro* is simply a set of commands that can be executed together, similar to a paragraph style. However, whereas a style is a collection of settings that you can apply at once, a macro is a collection of actions.

Recording Macros

When you record a macro, every step you take is recorded, including moving your insertion point up or down or making a selection.

Note Experienced VBA users continue to find macro recording useful for learning how to accomplish new tasks in VBA. One thing we all run into at some point, however, is the fact that there are a few commands that can't be recorded. For example, if you record a macro while adding items to the Quick Access Toolbar in Office 2010 or using the new Search In box above the Ribbon in Office 2011, your steps won't be recorded. In some cases, a macro that can't be recorded means that you can't accomplish the task through VBA, but it doesn't always. You can do a great many things when writing VBA that can't be done by recording macros, such as applying a document theme (or, in the preceding example from Office 2011, automating Find and Replace tasks). Learn more about this later in this chapter, as well as how to get help for finding commands that can't be recorded.

To begin recording a macro, on the Developer tab, in the Code group, click Record Macro. You can also access the Macro Recorder from the Status bar in Office 2010 or the Tools menu in Office 2011.

Once you click Record Macro, the Developer tab (or Status bar) icon changes to indicate that recording is in progress. The appearance of the button differs by program and where you access it. Click Stop Recording (accessible from the same location where you accessed the Record Macro feature) when you've finished recording the actions you need.

Note The ability to pause macro recording also becomes available on the Developer tab when macro recording is in progress.

Let's try one together as an example. Say that you're starting a new, long presentation document. Each page of the document needs to begin with Headings 1, 2, and 3, consecutively, followed by a paragraph of Normal text. The first several pages of that document will each begin with the text *Company Overview—* (including the em dash) in the Heading 1 paragraph, followed by different text on each page.

To save a bit of time, let's record a macro for setting up these pages.

Important In the interest of using the simplest method for any task, set up your document as follows before recording the macro:

- Set Style For Following Paragraph for Headings 1, 2, and 3 to the style that follows each heading at the top of every page. (Heading styles are followed by Normal style by default, so no change is needed for Heading 3.)
- Add Page Break Before formatting to the Heading 1 style so that new pages start automatically when you apply Heading 1.

Even after taking these steps, you can still save time by setting up these pages using a macro.

See Also For help with the Style For The Following Paragraph feature used in this setup, see Chapter 8, "Styles." For help with line and page break options such as Page Break Before, see Chapter 7, "Working with Text."

With your insertion point at the top of the empty document, click Record Macro and then do the following:

1. In the Record Macro dialog box, type a name for your new macro. Macro names must start with a letter and can include letters, numbers, and the underscore character, but can't include spaces or most special characters.

   Notice, in the Record Macro dialog box, that recorded macros in Word are stored by default in the global template Normal.dotm. (In Excel, recorded macros are stored by default in the active workbook.)

   In the Store Macro In list, you have the option to select any open document or template, including currently loaded global document templates in Word. For now, leave the default storage location and click OK to begin recording.

2. Apply Heading 1 style to the active paragraph.

3. Type **Company Overview—**. (To add the em dash, you can use the keyboard shortcut Ctrl+Alt+Hyphen in Word 2010 or Command+Alt+Hyphen in Word for Mac 2011. Note that you can only use the hyphen on the number keypad for this shortcut.)

4. Press Enter (Return) four times.

   Because Style For Following Paragraph is set as needed for the first three heading styles, these four hard returns add paragraphs with the styles Heading 2, Heading 3, and Normal, consecutively, followed by an additional Normal paragraph. That

additional Normal paragraph is where your insertion point will be when the macro starts to run again, so it will become Heading 1 style in the first step of the macro.

5. Click Stop Recording.

To run that macro, on the Developer tab, click Macros, select the macro you just recorded, and then click Run.

You could run this particular macro each time you need to set up a page, or run it as many times as you'll need identical pages. Or, you could edit it to add even more functionality, such as automatically adding the number of pages you need. But, for the moment, let's just look at this macro as an example to demonstrate how to read VBA code.

Reading VBA Code

To view the macro you just recorded, on the Developer tab, click Macros. Then, select your macro from the Macro Name list and click Edit. This will open the Visual Basic Editor with your macro open on screen. Your screen should look something like Figure 23-1.

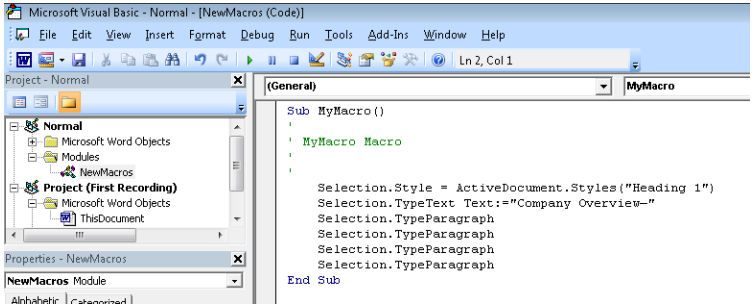


FIGURE 23-1 The Record Macro dialog box, shown in Word 2010.

For now, focus on the macro itself—we'll look at the different elements of the Visual Basic Editor shortly.

- Sub stands for subroutine, which is basically just another term for macro. Every macro begins with Sub and ends with End Sub, as you see in the preceding example.
- The first few lines below Sub in Figure 23-1 have an apostrophe at the beginning of the line. These are comments. An apostrophe at the beginning of a line of VBA code tells the macro to skip this line. When you record macros, VBA automatically adds some comment lines, one of which includes the name of the macro, as you can see in Figure 23-1.

  You can delete any line that begins with an apostrophe without damaging the macro. Be sure, however, not to delete the apostrophe and leave other text on the line that you don't want to run as a VBA command. The apostrophe is what tells VBA to skip the line when the macro runs.

- After the comment text, you see the commands that make up the steps of this macro. If you tried this for yourself and you see more lines of code in your macro than in this sample, ask yourself if you took other steps. If, for example, you made a typo in the *Company Overview* text and went back to correct it, that could have been recorded as a collection of several steps. Remember that when a macro is recorded, every keystroke is recorded. So, each time you use a different arrow key to move your insertion point, for example, you'll get another line of code. Take a look again at the commands from the preceding macro.

```
Selection.Style = ActiveDocument.Styles("Heading 1")
Selection.TypeText Text:="Company Overview–"
Selection.TypeParagraph
Selection.TypeParagraph
Selection.TypeParagraph
Selection.TypeParagraph
```

Notice that this code doesn't include any unfamiliar terms, even if you've never seen a line of VBA code before. Selection, style, active document, type text, and type paragraph all refer to extremely basic Word tasks. The majority of program-specific terms in VBA will be similarly familiar, just from your experience with the program.

As you progress through this primer, you'll come to understand how to construct the preceding lines of code and how you can write your own macros that are even simpler than recorded macros for accomplishing the same tasks.

Why Does My Recorded Macro Have So Many Lines of Code, When I Did Only One Thing?

As mentioned earlier, when you record a macro, every keystroke is recorded. So, you often end up with much more code for a simple action than you would if you wrote the macro yourself.

In particular, if you use a dialog box to execute an action while recording a macro, you're likely to get far more code than you may expect. When you click OK to accept the settings in a dialog box, you're accepting all settings in that dialog box. VBA doesn't record your keystrokes while you're in most dialog boxes, so it must record every setting you accepted when you clicked OK.

For example, if one step in my macro was to bold a selected word, and I used the bold icon in the Font group on the Home tab, the code for that command would look like this:

```
Selection.Font.Bold = wdToggle
```

If, on the other hand, I opened the Font dialog box to apply bold and then clicked OK to close the dialog box, the code for that command would include all of this:

```
With Selection.Font
.Name = "+Body"
.Size = 11
.Bold = True
.Italic = False
.Underline = wdUnderlineNone
.UnderlineColor = wdColorAutomatic
.StrikeThrough = False
.DoubleStrikeThrough = False
.Outline = False
.Emboss = False
.Shadow = False
.Hidden = False
.SmallCaps = False
.AllCaps = False
.Color = wdColorAutomatic
.Engrave = False
.Superscript = False
.Subscript = False
.Spacing = 0
.Scaling = 100
.Position = 0
.Kerning = 0
.Animation = wdAnimationNone
End With
```

Notice that, because of the limitations related to recording macros with dialog box commands, VBA recorded a setting for every option in the Font dialog box.

If you write a macro, or edit your recorded macro, you don't need to specify any setting unless you want the macro to execute that setting. In this example, if you were to delete everything between the lines that begin `With` and `End With`, except the `Bold` setting, you'd still get the result you need.

See Also Learn about the `With`...`End With` syntax in the section "Grouping Statements," later in this primer.

Understanding Statements, Procedures, Modules, and Projects

To begin to work in the Visual Basic Editor, you need to understand how files work in VBA—that is, how macros are organized and stored. The following common items are the principal components you need to know:

- A **statement** is a single command or action in a macro—that is, it's a line of code. For example, `Selection.Font.Bold = wdToggle` is a statement. As you'll see in the section "Writing, Editing, and Sharing Simple Macros," later in this chapter, when you think of VBA as a language, think of a statement as a sentence.
- A **procedure** is essentially another way of referring to a macro, although there are other types of procedures as well, such as functions. A **function** is a procedure that returns a result.
- A **module** is a collection of code. Think of a module as a code document. A module can contain several procedures. And, like documents, modules can be saved as files, copied, and shared.
- A **project** is the collection of all modules and related VBA objects in your document, template, or add-in. A project might have one or several modules, as well as other elements such as UserForms (dialog boxes).

Companion Content *All of the VBA elements discussed in this list are covered in this chapter, with the exception of UserForms. You can learn about creating and working with UserForms in the online companion content provided for this book. Once you master the VBA basics covered in this chapter, find information about the online companion content—and other resources for taking your work with VBA further—in the chapter conclusion.*

Using the Visual Basic Editor

Before you start working with VBA code, take a few minutes to settle in to your surroundings. To help you work more comfortably, the subsections that follow describe a bit about each component of the Visual Basic Editor that is identified in Figures 23-2 (Word 2010) and 23-3 (Word 2011).

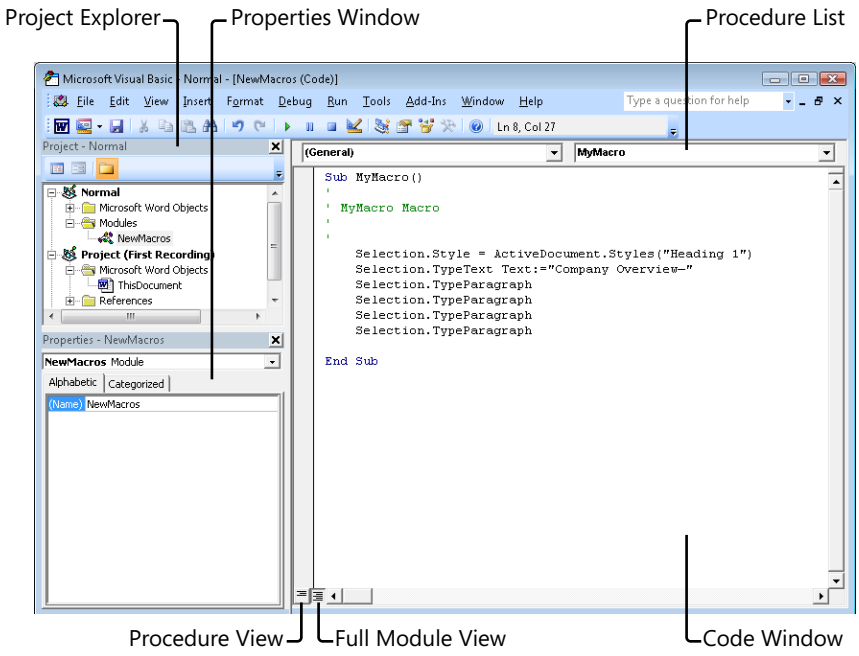


FIGURE 23-2 The Visual Basic Editor in Word 2010.

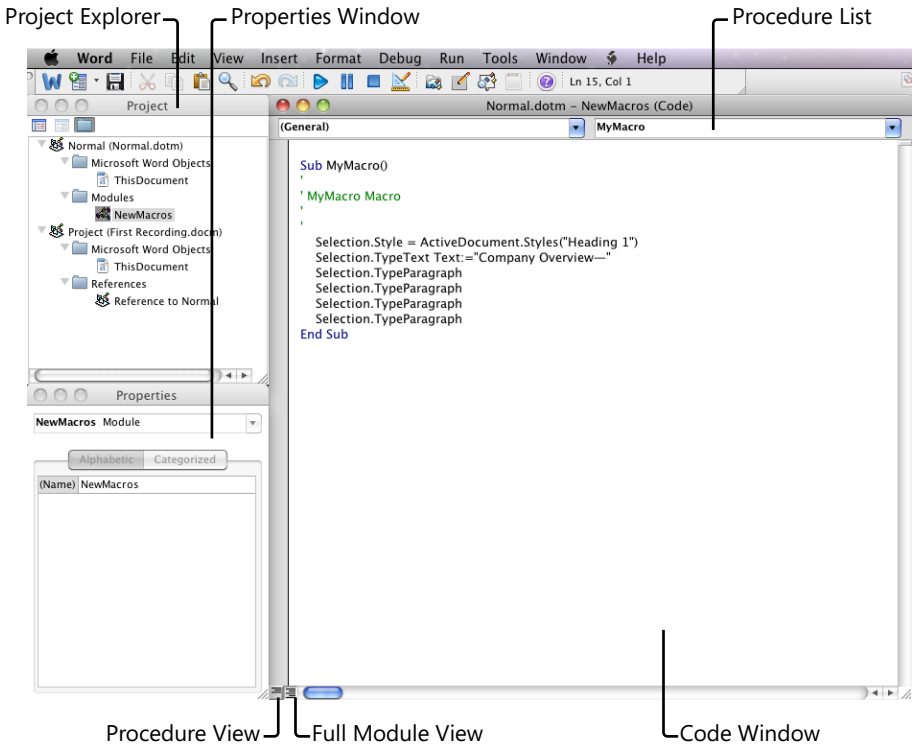


FIGURE 23-3 The Visual Basic Editor in Word 2011.

Introducing the Code Window

The code window is where your procedures appear. This is where you type macros when writing code and where you find the macros you've recorded. Notice that the procedure list is at the top-right of the code window. From this list, you can quickly move to any procedure in the active module.

Also notice the view options at the bottom of the screen. When you have several macros in a module, it can be helpful to view them one at a time. Full Module view is the default, but you can change this setting and many others through the Options dialog box (Preferences in Office 2011).

See Also For more about setting preferences for the Visual Basic Editor, see the section "Setting Up Your Workspace," later in this chapter.

Introducing Project Explorer

Project Explorer is where you see the list of all VBA projects that are currently open or loaded. All open documents, as well as open or loaded document templates, appear here, whether or not they contain macros. You can collapse or expand a project to view the modules and objects that it contains.

Caution Documents appear in this list whether or not they're macro-enabled file formats. This is important to keep in mind because, if you add code to a document using an Open XML Format that ends with the letter *x*, you won't be able to save the document with its code. Save the document with the equivalent file format that ends in the letter *m* to make sure your code will be saved along with the document or template.

A project has a Modules or Forms folder only if it contains code modules or UserForms. However, in Word and Excel, every project contains an Objects folder, such as the Microsoft Word Objects folder you see under each of the projects visible in Figures 23-2 and 23-3.

In Word, the Objects folder contains a document object referred to as ThisDocument. In Excel, it contains both a ThisWorkbook object and a sheet object for each existing sheet in the workbook. Some types of code (such as a type of procedure known as a document-level event) are added directly in the code window for the document object rather than in a module. However, you will often have projects that have no code added to the document objects.

See Also Learn more about using the document objects in the section "Introduction to Using Events," later in this chapter.

Introducing the Properties Window

The Properties window shown in Figures 23-2 and 23-3 doesn't look like much, but don't be fooled. For modules, the Properties window is generally used only to edit the module name. However, for some object types (such as UserForms), the Properties window becomes extremely important because it's populated with many settings that you can edit directly within it, ranging from the height and width of a UserForm to the value to display on a form control (such as a text box or an option button).

To edit the name of a module in the Properties window:

1. Click into the name where it appears on either the Alphabetic or Categorized tabs.
2. Edit it as you would document text.

   Module naming rules are the same as macro naming rules—no spaces or special characters, and the name must begin with a letter.
3. Press Enter (Return) to set it.

Note All names in VBA subscribe to a similar set of rules as the module name. Names must always start with a letter and can't include spaces or most special characters. Most names are limited to 255 characters. However, module names can't exceed 31 characters, and macro names added in the Record Macro dialog box are limited to 80 characters.

Note that, when you record macros, they're always added to a module named NewMacros. You can rename that module if you like, but the next time you record a macro, a new module will be created with the name NewMacros.

Setting Up Your Workspace

You'll find many settings that can be customized in the Options dialog box, available on the Tools menu in the Visual Basic Editor. (In Office 2011, this is the Preferences dialog box, available from the application name menu, such as Word.) I don't recommend spending much time in this dialog box just yet, because you might not be familiar with many of the settings. However, it's good to know that it's there, because you are likely to need it. This primer will point out when settings can be customized in this dialog box.

Possible settings in the Options (or Preferences) dialog box include default behavior for a number of programming actions (such as the way you're notified about errors in your code), the formatting for each type of text or notification you see in the code window (such as comment text or errors), and the way the window itself is arranged.

In addition to settings in the Options dialog box, notice that you can drag to resize panes in the Visual Basic Editor window (such as the Project Explorer or Properties window), and can close those you don't need. In the Visual Basic Editors for Office 2010, you can also drag to dock or float panes.

Use the View menu to access any windows you've closed. If you're unable to dock any window in an Office 2010 Visual Basic Editor, you can change the setting for that window on the Docking tab of the Options dialog box.

For Mac Users

Although the Preferences dialog box is available in the Office 2011 Visual Basic Editors, at the time of this writing, this dialog box does not hold custom settings beyond the active session. By the time you read this primer, this may no longer be the case when you try to add custom settings here. But if it is, don't be alarmed. This glitch isn't likely to cause you too much inconvenience, because most of the defaults are just fine—especially for folks who are new to VBA.

See Also If this is still the case when you venture into Office 2011 VBA, see the section "Declaring Variables," later in this chapter, to learn about the Option Explicit statement, which I strongly recommend adding when you create a new module.

Writing, Editing, and Sharing Simple Macros

Companion Content *All code samples shown throughout this section are available in procedures in a module named PrimerSamples.bas, available in the Chapter23 sample files folder online at* http://aka.ms/651999/files.

See Also For help importing a module into your Visual Basic Editor, see the section "Saving and Sharing Macros," later in this chapter.

One of the most important differences between macros you record and macros you write is that, when you record a macro, you need to select an object to act on it. But when you write macros, you can usually identify items to act on instead of selecting them. That apparently simple difference gives you tremendous power and flexibility. For example, you can write a macro to act on all tables in your document automatically, rather than recording a macro that you run from each table.

We've now reached the core of this primer. From creating a macro to reading and understanding essential VBA language constructs, the sections that follow progress in a logical order to help you learn in such a way that you can immediately put your knowledge into

practice. Review the content under each heading and try the examples for yourself in the Visual Basic Editor. Be sure that you understand the content covered under each heading before progressing, and you'll be using VBA comfortably before you know it.

Note Most of the features you'll learn about in the following sections are programming basics. They're written here specifically for VBA. However, should you ever want to learn another programming language, it's useful to know that many of the concepts and terms used here are fairly standard across common programming languages.

Creating Modules and Starting Procedures

To create a module:

1. Select the project (in Project Explorer) to which you want to add the module.

   You can click any element contained in the project to select the project, such as the project name or the Modules folder (if one exists).

2. On the Insert menu, click Module.

   You can also insert a module from the Insert icon on the Standard toolbar. Notice that this icon defaults to what you last inserted (such as a module or a UserForm). Click the arrow beside the icon to select a different item from the available options, as you see in Figure 23-4.

FIGURE 23-4 Quickly insert a module from the Standard toolbar in any Visual Basic Editor.

3. To rename the module, click into the name field in the Properties window, as mentioned earlier. Type the new module name and then press Enter.

Once you have a module in which to create your macros, you can just click in the code window and begin typing to create a macro. As you saw in the sample recorded macro, every macro begins with the term Sub, followed by the name of the macro, and then followed by a pair of parentheses. Those parentheses can be used to hold instructions for the macro or information about references in the macro, but it's rarely necessary to type anything between them for basic document production macros. Even if you type nothing between the parentheses, however, note that they are required.

Notice as well that every macro ends with the line `End Sub`. Many types of instructions you'll learn throughout this section are paired (such as `With` and `End With`, demonstrated under the upcoming heading "Grouping Statements.").

When you type **Sub** plus a procedure name and then press Enter, VBA automatically adds the parentheses at the end of the first line and adds the `End Sub` line. However, with most paired terms, the end term isn't added for you. It's good practice to always type both ends of a paired structure at the same time so that you don't forget to do so later. When macros become longer or more complex, finding the missing end portion of a paired structure can be a frustrating use of your time.

So, to start a macro in your new module, type the following:

```
Sub MacroName
```

After you press Enter, the procedure will look like this:

```
Sub MacroName()
End Sub
```

The statements that your macro comprises will go between these two lines.

Note The next several headings provide code samples that show only the relevant code for the particular topic. To run that code in the Visual Basic Editor, remember that it has to appear within a procedure, so you need to add the surrounding `Sub` and `End Sub` statements discussed here.

Learning the Language of Objects, Properties, and Methods

Just as the languages you speak comprise nouns, verbs, adjectives, and other parts of speech, VBA comprises objects, properties, and methods. Think of objects as nouns, properties as adjectives, and methods as verbs.

- An object is just that—it's a thing that can be acted on.
- A property is a characteristic of an object—something that describes the object, such as its size or style.
- A method is an action you can perform on an object. For example, `Save` and `Close` are both available methods for the `ActiveDocument` object.

The only difference between the sentence structure in a spoken language and in VBA is that, though a sentence always requires a noun and a verb, a VBA statement requires an object and *either* a property or a method. Let's take a look at a few examples.

- In the following statement, `ActiveDocument` is an object and `Save` is a method.

  ```
  ActiveDocument.Save
  ```

- In the following statement, `Selection` is the object (referring to the location of the insertion point—the actively selected content) and `Style` is a property of that selection. `Heading 1`, in this case, is the value (the setting) for the indicated property.

  ```
  Selection.Style = "Heading 1"
  ```

- Objects are often used as both objects and as properties of other objects, depending on where they're placed in a statement. In the following statement, `Tables(1)` refers to the first table in the active document. Though a table is an object, it's also used here as a property of the active document. `Style`, in this statement, is a property of the specified table.

  ```
  ActiveDocument.Tables(1).Style = "Table Normal"
  ```

  Even though `Tables(1)` in this case is a property of `ActiveDocument`, it's still an object. Notice that the style being set is a property of the specified table.

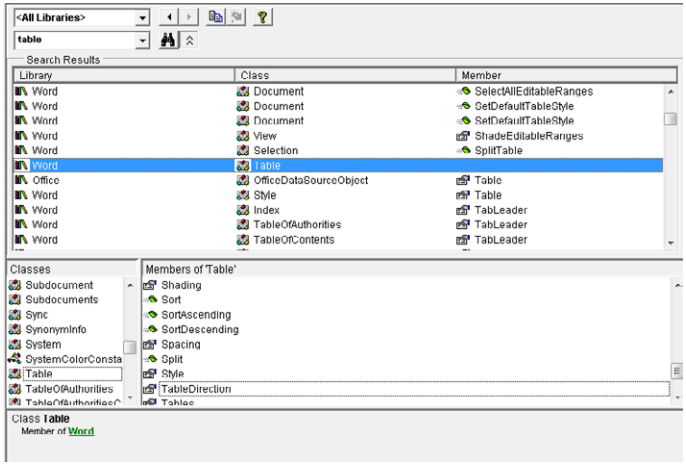
  You don't typically need to think about whether an object is being used as an object or a property, similar to distinguishing whether an *-ing* word (such as *creating*, *editing*, or *dancing*) is being used in a given sentence as a noun or a verb. What's important to note is that many objects, such as a table, require a higher-level object to make the reference specific enough for VBA to understand. For example, you can't write simply `Tables(1).Style` to indicate the style of the first table, because VBA needs to know what range you're referring to when you tell it to act on the first table. Otherwise, you might be referring to the first table in the document, the first table in the selection, or a number of other possible ranges. Just keep in mind that many objects can also be used as properties of other objects, because this will come in handy when you reach the "Getting Help" section later in this chapter.

Looking at the preceding list of examples, you might be wondering how you're supposed to memorize every possible object, property, and method name in each program for which you need to learn VBA. Well, relax. You hardly need to memorize anything at all when it comes to program-specific terms. When you understand the concept of using objects, properties, and methods to create statements, and you remember what you already know (the features of the program you're automating), you'll learn the names of the particular objects, properties, and methods the same way you learn vocabulary in a spoken language—simply by using it.

Introducing Object Models

The set of VBA vocabulary that's specific to a given program is known as the program's *object model*. The Visual Basic Editor in each program also contains a "dictionary" of sorts for that object model, known as the Object Browser. You can use the Object Browser (available from the View menu) to search for the correct terminology to use for a given feature, or to see what properties or methods are available to a given object. For example, Figure 23-5 shows the range of results you get when you use the Object Browser in the Word Visual Basic Editor to search for the term *table*.

Office 2010



Office 2011

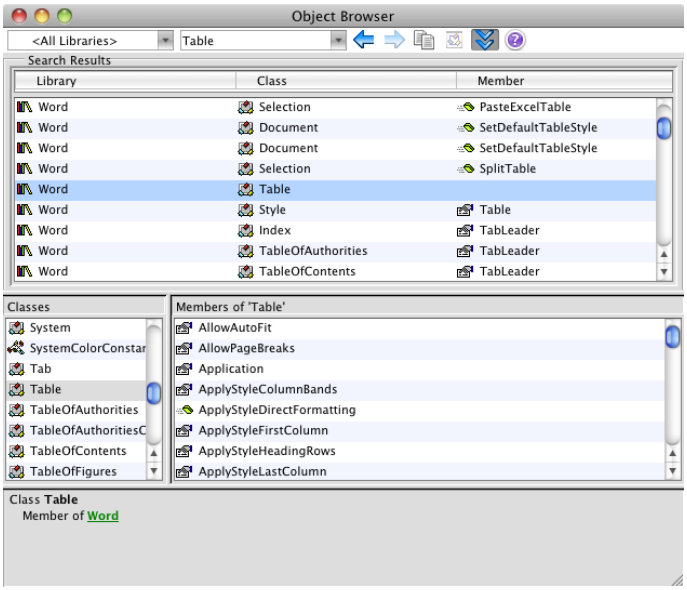


FIGURE 23-5 Despite different organization, the Object Browsers in Office 2010 and Office 2011 look and function very much the same.

Notice in Figure 23-5 that the selected item in the search results is the table object. The heading *Classes* refers to items in an object model that can have an available set of members—such as objects or modules. Properties and methods are members of a specified class. Notice the headings *Classes* and *Member Of 'Table'* in the bottom panes of the Object Browser.

Note In both Office 2010 and Office 2011, the Object Browser dialog boxes are adjustable. If you don't see all of the panes shown in Figure 23-5, you can just drag to expand them.

Navigating the Object Browser

When searching for terms in the Object Browser, remember that multiple-word terms don't get spaces in VBA. Separate words in a single term are instead denoted by initial capital letters, such as the `ActiveDocument` object or the `PageSetup` property. Searching in the Object Browser isn't case-sensitive, but the Object Browser won't recognize multiple words with spaces as a single term. For example, searching for *page setup* in the Object Browser will return no results, but searching for *pagesetup* will return several.

Note that the Object Browser is also available from the Standard toolbar. Or, to access the Object Browser by keyboard shortcut, use F2 in Office 2010 and Ctrl+Command+B in Office 2011.

In the following list, also notice the icons used in the Object Browser to denote objects, properties, methods, or library. These will also be displayed while you're writing code, as explained under the next heading. (All of these icons are the same in Office 2010 and Office 2011.)

- Object

- Property

- Method

- Library

  (An object model is a type of library. For example, results shown in the Object Browser in Figure 23-5 are members of the Word library, which is the same as saying the Word object model.)

Why Do I Get an Error When I Try to Set Some Properties?

The key to this question is to remember that you sometimes need to use VBA statements to get information about the document as well as to apply settings or execute actions. Many properties are read-only, meaning that you can use them only to return information, not to apply a setting.

For example, `ActiveDocument.Name` is a read-only property that tells you the name of the active document. You can't set the name using this property, but that doesn't mean you can't name a document using VBA. For example, to change the name of the document, you'd use the `SaveAs` method (that is, `ActiveDocument.SaveAs`). With this method, you can specify several settings for how you want the document saved, including its name.

To learn whether a property is read-only, select that property in the Object Browser. At the bottom of the Object Browser is a pane where you see the hierarchy for the selected item (what class and library it belongs to). This pane also indicates when a property is read-only, as you see in Figure 23-6.

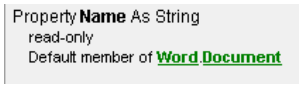


FIGURE 23-6 Definition for a member of the Word object model, shown in the Object Browser.

In the example shown in Figure 23-6, Word is the library and Document is the object to which the read-only property `Name` belongs. You'll learn more about ways to use read-only properties as this primer progresses.

Using Auto Lists

One of the main reasons you don't have to memorize the object model for the program you're automating is that the Visual Basic Editor often gives you the available options as you write. When you type an object, for example, followed by a period, you automatically see a list of properties and methods available to that object, as shown in Figure 23-7.

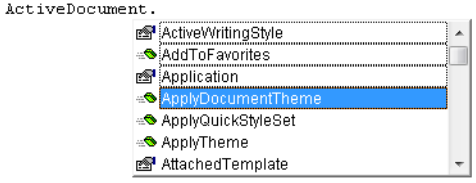


FIGURE 23-7 An Auto List in the Word 2010 Visual Basic Editor.

Notice the icons, shown earlier, that appear in this Auto List to indicate properties or methods. All the members of a given object (that is, all properties and methods available to that object) appear in the list.

To scroll through an Auto List, you can use the up or down arrows as well as the Page Up and Page Down keys. You can also begin to type the item you need, if you know at least the first characters, to move to that position in the list. For example, if you type **t** immediately after the period that follows `ActiveDocument`, the list would move to the image you see in Figure 23-8.

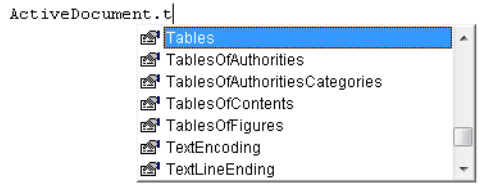


FIGURE 23-8 Scroll quickly through Auto Lists in Office 2010 or Office 2011 VBA.

When you select the item you need in the Auto List, press the Tab key to add the item to your statement. (You can also press the Spacebar instead of using the Tab key. However, doing so will add a space in your code after the selected item.) Note that, if you press Enter once an item is selected in an Auto List, you'll get an error unless the selected item was the last required term in the statement.

Understanding Variables

In addition to objects, properties, and methods, most macros use other types of terms as well, including variables and constants (the latter of which is discussed in the section "Using Constants," later in this chapter).

Variables are types of data that represent objects, statements, or other elements required in your code. They're often used to save time and make code more efficient, such as using a single term in place of a statement that you have to reference several times. They are also handy when you need to refer to any instance of a given object type, rather than specifying an instance of an object. Consider the following examples.

- If you need to refer to the full name (the `FullName` property includes the file path) of the active document in a few places within your macro, you might want to declare a variable to represent it, as shown in the following statement:

  ```
  myName = ActiveDocument.FullName
  ```

  The name of the variable in this case is `myName`. Once you've typed this statement in your macro, you can use the term `myName` in place of `ActiveDocument.FullName` wherever you need to use the full name of the document.

- When you use loops (discussed in the section "Looping Code," later in this chapter) to execute a command for several instances of an object, you might use a variable as

a counter to help you accomplish that. For example, say you want to apply a specific table style to all tables in the document, as shown in the following code.

```
Dim myI as Integer
For myInt = 1 To ActiveDocument.Tables.Count
   ActiveDocument.Tables(myI).Style = "Table Contemporary"
Next
```

The preceding code uses a `For...Next` loop, explained in the section "Using For Each... Next and For...Next Loops," later in this chapter. However, notice how the variable `myI` is used here.

- First, you declare the variable as an integer. (Declaring variable data types is discussed in the upcoming section "Declaring Variables.")
- Then, the start of the loop (the line that begins with the word `For`) tells the code to begin executing with the variable equal to the number 1 and run until the variable equals the number of tables in the document. Each time the loop executes, the number is automatically increased by 1.
- Next, notice that the variable is used to denote the table number in the statement that applies the style to the table.

Using variables in place of a complete statement, or as counters, is a common, useful tool. Other uses of variables are demonstrated under applicable headings later in this chapter, including "Using Conditional Structures" as well as "Looping Code."

Note For code that's easier to read, follow, and edit, use intuitive variable names. Variable names can't contain spaces and can't be VBA terms used for any other purpose (such as the name of an object, property, or method). Keeping those requirements in mind, make your variable names as short as possible to save yourself work.

Introducing Variable Data Types

As you saw in the preceding examples, variables can be used to represent different types of information, such as numbers, text strings, or objects. Several variable data types are available, and you can even create your own. However, to help you keep things simple as you begin using variables, Table 23-1 lists commonly used variable data types.

Note For a complete list of data types supported in VBA and their definitions, search the topic "Data Type Summary" in Visual Basic Help, available from the menu bar in any Visual Basic Editor.

TABLE 23-1 Commonly used variable data types

Data type	Possible values
Boolean	True or False
Integer	An integer ranging from –32,768 to 32,767
Long	A long integer ranging from –2,147,483,648 to 2,147,483,647
Currency	A scaled integer ranging from –922,337,203,685,477.5808 to 922,337,203,685,477.5807
String	A text string, such as a VBA statement (text strings are relatively unlimited—they can reach up to approximately two billion characters in length)
Variant	A number or a text string (if you don't specify the data type for a variable, it is a variant by default)

You can also declare variables as specific types of objects (such as a table, a style, or a document). Variables declared as a specific object type are called *object variables*, and they offer additional benefits, discussed next.

Declaring Variables

When you specify a variable type, which is called *declaring* the variable, you can save time and reduce errors. For more complex macros, declaring variables is also important because undeclared variables default to the variant data type, which uses more storage space than other data types and thus creates more work for the program running your macro.

Additionally, when you require that variables be declared in your modules, VBA lets you know while you're still working on your code if variables contain spelling errors that could cause an error when users run your macro.

See Also For more on this subject, see the section "Running Macros and Compiling Projects," later in this chapter.

When you declare an object variable—that is, a variable declared as a specific type of object—VBA recognizes the object so that you get Auto Lists for completing statements that include the variable.

Caution When you declare a variable as a particular data type, you must use it as that data type. For example, if you declare `myI` as a string, VBA won't understand if you use it in a statement as if it were a number (such as `For myI = 1 to ActiveDocument.Tables.Count`, as demonstrated earlier). Variables you want to use as numbers must be declared with an appropriate numeric data type (see the preceding table for the possible values available to different numeric data types). Similarly, to use a variable as a text string, you must set the value of that variable (the information after the equal sign) as either a VBA statement or a text string enclosed in quotation marks.

To declare a variable, use a `Dim` statement. For example:

```
Dim myI as Integer
Dim myName as String
```

Once you type the word **as** in a `Dim` statement, you get an Auto List of available options to help you complete the statement, as shown in Figure 23-9.

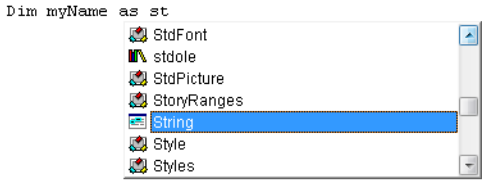


FIGURE 23-9 The Auto List shown here provides options for specifying a variable data type.

Declare Multiple Variables in One Statement

You can declare multiple variables on the same line; just be sure that you specify a data type for each. For example, the following statement does *not* declare all three variables as strings:

```
Dim myName, myPath, myStyle as String
```

The preceding code will seem to work, and it won't generate any errors, as long as `myStyle` is used as a string data type. That's because `myName` and `myPath` are declared as variants—no data type is specified for them. The correct statement to declare all three variables as strings would read as follows:

```
Dim myName as String, myPath as String, myStyle as String
```

To require variable declaration in a module, click in the very top of the module, type the words **Option Explicit**, and then press Enter. This statement is one of several that you can place at the top of a module to apply to all procedures in your module. Notice that, when you press Enter after typing this statement, a line appears beneath it, just as a line automatically appears between macros. This part of the module is known as the General Declarations section.

Note You can set the Visual Basic Editor to require variable declaration automatically whenever you create a new module, through the Options dialog box in Office 2010 or the Preferences dialog box in Office 2011. On the Editor tab of that dialog box, check Require Variable Declaration.

For Mac users, as mentioned earlier, the Preferences dialog box did not hold these kinds of customizations at the time of publication. If you experience this, just get into the habit of typing **Option Explicit** each time you create a module.

Sharing Variables Throughout a Project

If you have multiple macros that need to refer to the same variables, you can declare them publicly for the entire project so that you don't need to type out the declarations in each applicable macro.

To do this, type your variable declarations in the General Declarations section of any module in the project, and use the word `Public` instead of the word `Dim` to begin the statement. For example, the following statement makes `myName` a string variable, and `myI` an integer variable, available to all procedures in the project:

```
Public myName as String, myI as Integer
```

Note, however, that you must be in a procedure to assign a value to a variable. For example, you can declare `myI` as an integer variable for use throughout the project, but the statement `myI = 1` must appear inside a procedure. To use one set of variable values for multiple macros across all modules in your project, put all value assignments for public variables in one macro, and then access that macro from any procedure where you need to use those values.

See Also To learn how to do this, see the section "Running One Macro from Another," later in this chapter.

Note You can also use the General Declarations area at the top of a module to declare variables so that they're available to all macros in the same module, but not other modules. To do this, use `Private` or `Dim` instead of `Public` to start the variable declaration.

Never Write the Same Code Twice

One of the best pieces of advice I received when I first started learning VBA was this: if you have to type the same statement twice, ask yourself if there's a faster way. For example, consider the steps discussed previously for declaring public variables to use the same set of declarations in all procedures throughout your project. Using grouping structures and loops (both discussed later in this chapter) is another way to avoid doing the same work twice.

Keep in mind that writing efficient code isn't just about typing less. As with documents, the less work you do, the better your results will be every time—and the easier job you'll have when that content needs editing. What's more, efficient code also makes it easier for the program to run your macros, so you get macros that are easier to write, easier to edit, and easier to run.

What Do I Do When the Variable Type Doesn't Work?

If you don't know which variable type you need, you can't find the variable type you think you need in the Auto List that appears in your `Dim` statement, or you get a "Type Mismatch" error (which means the variable type declared doesn't match the way you've used the variable), there is an easy way out.

Though it's not good practice to do this regularly, particularly in long or complex code, you can simply type `Dim <variable name>` and not specify it as a particular type. When you do this, VBA classifies the variable as a variant data type, so you won't get Auto Lists when using the variable in statements. However, even if you have `Option Explicit` set for the module, as I hope you do, you can declare a variable in this way and continue on. (Alternatively, to avoid other editors of the code thinking that you've accidentally omitted the data type, you can declare the variable as a variant.)

It's a bit of a sloppy workaround—what programmers refer to as a *hack*—but if you're writing simple macros just for your own use, there's really no harm in doing it occasionally, and it can save you time while you're still learning about variable types. Just try not to make it a habit.

Understanding Document Variables and Data Storage Options

In addition to the variables that you use in your macros, there is an object type named `Variable` in the Word object model. These are known as document variables, because you use them to store information in the document that's collected or created by your macros, rather than as a place to store data just while the macro is running—such as when you need the document to remember information from one use of a given macro to the next.

For example, in template automation projects I do for clients, I sometimes add document variables to store user preferences that are specific to the individual document, such as which of a selection of design choices the user wants for the active document. The document needs to store that information after the macro runs so that the user's preferences are remembered the next time the design macros are used.

In Word, this type of information can be stored using either a document variable or a custom document property (which you're most likely familiar with from the Document Properties dialog box). However, Excel and PowerPoint don't offer a document variable object, so custom document properties are the way to go for storing document-level data in your workbooks and presentations.

In addition to document-level data, there are several ways to store data on the system level—that is, so that data can be accessed by your macros for use by more than an individual document. One of the easiest and most common methods is storing data in the Windows Registry (or, for Mac Users, in Library>Preferences).

As you can imagine, there are many uses for storing data in variables, document properties, or system-level resources such as the Registry. To explore this topic, use the Object Browser in your Visual Basic Editor to look up the `Variable` object, the property named `CustomDocumentProperties`, and the `GetSetting` and `SaveSetting` functions (the last two are functions used for storing data on the system level).

Storing Data on the System

The `SaveSetting` function warrants an additional mention here, particularly for Mac users, because this information isn't easily accessible in Help on the Mac. When you use the `SaveSetting` function on either platform, you essentially create a named location on the system where the value you indicate is stored. You can then access that value using the `GetSetting` function from VBA in any Microsoft Office program.

- On Windows, the locations you create are added as keys in the Windows Registry under the key: HKEY_CURRENT_USER\Software\VB and VBA Program Settings\.
- On Mac, the locations you create are added as files (with the file extension .plist) in Library>Preferences (located in your Home folder). Note that earlier versions created these files in the Microsoft subfolder within Preferences, but that is no longer the case.

The structure of this function is as follows:

```
SaveSetting "AppName", "Section", "Key", "Setting"
```

Each of the four arguments is a value you define. For example, if you are automating the creation of documents for a company with multiple brands, you might create an `AppName` for the company name, such as `NorthwindTraders`, and then define a section for storing the user's brand preferences, as follows:

```
SaveSetting "NorthwindTraders", "BrandPrefs", "Department", "Marketing"
SaveSetting "NorthwindTraders", "BrandPrefs", "Title", "Marketing Manager"
SaveSetting "NorthwindTraders", "BrandPrefs", "Location", "London"
SaveSetting "NorthwindTraders", "BrandPrefs", "Paper Size", "A4"
SaveSetting "NorthwindTraders", "BrandPrefs", "LetterStyle", "Personal Letterhead"
```

Typically, you would collect this information from the user via a UserForm (dialog box) or from another data source. If you ran the preceding `SaveSetting` functions, the data you collected would be stored as follows:

- In the Windows Registry, you'd then see a key named `NorthwindTraders` with a subkey `BrandPrefs`. Within `BrandPrefs`, you would see each of the five values specified along with their data.

- In your Mac Home folder Library>Preferences, you'd see one file named NorthwindTraders.plist that includes all of the data for any sections and values within those sections.

The most important difference between using this function on Windows and on Mac is that you can edit the Windows Registry values you create directly in the Registry if desired. On Mac, you can view the .plist file in Text Edit, but trying to edit that file directly in Text Edit will destroy your data. There are tools available for editing .plist files on Mac, or you can just replace stored values by using the `SaveSetting` function again to resave the value you need. On both platforms, you can delete the setting from the Registry or Preferences folder if they are no longer needed without affecting any other data or functionality on the system.

See Also *Pref Setter is a user-friendly third-party tool for editing .plist files on the Mac. Find it at* www.nightproductions.net.

Working with Object Model Member Arguments

In addition to the variables that you can declare for use in your procedures, many items in the VBA object models include elements that use the same data types as variables to specify settings for that item. The elements, known as *arguments* (similar to arguments in an Excel formula), can be required or optional.

An argument might be as simple as the index number of an object to specify it within the collection, such as the third table in the active document, written as `ActiveDocument.Tables(3)`. Or, it might be a series of parameters that define how an action is to be executed, as is commonly used for VBA methods. Take a look at a few examples.

- When you use the `FollowHyperlink` method of the `Document` object in a statement, you get the options shown in Figure 23-10 in the Quick Info ScreenTip that appears after you type the open parenthesis following `FollowHyperlink`.

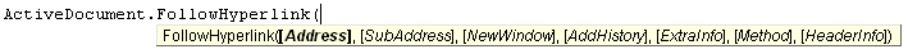


FIGURE 23-10 The FollowHyperlink method is available in Word, PowerPoint, and Excel.

Most of the arguments shown in Figure 23-10 are optional. Typically, optional arguments appear in brackets, but as you see here, that's not always the case. You can't follow a hyperlink without an address. This example is an exception.

In most cases, if an argument appears in parentheses but seems to be key information, a default value is used when the parameter is omitted. For example, `Selection.Move`

has parameters to define the `unit` and `count` by which to move the active selection. If those parameters are omitted, the default for `unit` is a character and for `count` is 1. So the insertion point is moved one character forward.

- When you use the Add method for a `Table` object, you get the arguments shown in Figure 23-11.

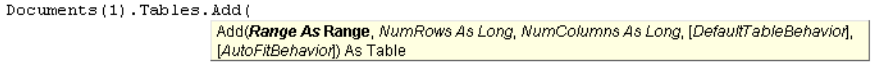


FIGURE 23-11 Parameters for adding a table to a Word document.

The Add method is used for many objects in Word, PowerPoint, and Excel. It has different arguments, of course, for each, depending on the type of object being added. For the `Table` object, the range argument (that is, the location where you want the new table to appear), number of rows, and number of columns are required.

Notice that the required parameters here (those not inside brackets) specify particular data types. The range is an object variable (referring to the `Range` object), and the number of rows and columns both use the `long` data type (as noted in the Quick Info). Note that the optional AutoFit behavior setting is a variant (default) data type, but it requires a value from an available set of constants. Learn about constants in the upcoming section "Using Constants."

- The `HomeKey` method, shown in Figure 23-12, is used with the `Selection` object. It's the VBA equivalent of using the Home key on your keyboard.

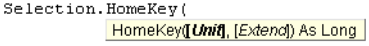


FIGURE 23-12 The HomeKey method displaying optional parameters.

The two available arguments used here—both of which are optional and use the variant data type—determine how far your insertion point moves (`Unit`) and whether the selection is extended (equivalent to holding the Shift key when you press the Home key) or your insertion point is simply moved to the new location. Both arguments require selections from a set of available constants, as we'll discuss shortly in the "Using Constants" section.

There are two ways to specify most arguments in statements such as those in the preceding list of examples. The first approach is to type the values for the arguments between parentheses immediately following the method (as you saw in the Quick Info ScreenTips for the three sample methods shown in Figures 23-10 through 23-12). When you use this approach, type a comma after each value you add. You'll see that the active argument (the one for which you can add a value at your insertion point) is shown as bold in the ScreenTip. If you don't intend to include a value for each argument, type consecutive commas until the argument you want to specify is bolded. If you place an argument in the wrong position between

parentheses, the method won't work correctly. Notice, however, that this approach can be confusing and difficult to read when you need to edit or troubleshoot a macro.

Note Some types of arguments can be specified simply in quotation marks after the statement name, as with the `SaveSetting` function demonstrated previously in the sidebar "Storing Data on the System."

Instead, for methods that take more than a single argument, specify each by typing the argument name, followed by a colon and an equal sign, followed by the value you want to assign. Separate each argument you specify with a single comma, and note that argument order doesn't matter when you use this approach. Take a look at the following two examples:

```
ActiveDocument.FollowHyperlink Address:="http://office.com", NewWindow:=True
Selection.HomeKey Unit:=wdStory, Extend:=wdExtend
```

Using the explicit approach shown here helps to keep your code easy to read, edit, and troubleshoot.

How Much Do You Really Need to Know About Arguments?

It's important to know the syntax for specifying arguments or parameters, but in most cases, you don't need to worry about the data type or whether the argument is required.

Of course, you typically see the data type for required arguments in the Quick Info. But you'll often know the type for optional arguments as well, simply from using the program. Because you're an experienced user of the program you're automating, you're more likely to actively look for an argument that you know should be available for a given method (such as specifying a file type or setting a password for the file when you save a new document) than to need help understanding the ones you happen to find.

In the case of the `SaveAs` method to save a file with a new name or file type (`SaveAs2` in Word 2010), not even the file name is a required argument because the Save As dialog box, as you've surely seen, always provides a default name. Word and PowerPoint default either to the first phrase in the document or, for blank documents, to the document or presentation number assigned when a new document was generated. In Excel, the Save As dialog box always defaults to the book number assigned when the workbook was generated.

Remember that one of the most important tools you have for working in VBA is your knowledge of the program you're automating. As an advanced user, you'll likely find that VBA is easier for you to learn than it is for a professional developer who doesn't

use the programs. Think of it this way: if you're already a pretty good skier, you'll probably learn how to snowboard much faster than someone who designs snowboards for a living but has never set foot on a mountain.

However, there will still be times when you'll need help determining what data type to use for an argument's value or knowing when an argument is required. You can find a description of each argument for most applicable object model members, along with its data type and whether it's optional or required, in the help topic for that item.

See Also Learn how to find the help you need in the section "Getting Help," later in this chapter.

Using Constants

As mentioned previously, many items in VBA require the use of another data type, known as a constant. Unlike variables that can change as needed, constants are used when a defined set of options exists for the feature. Most constants in VBA are either specific to the individual program object model or are available in VBA for any Microsoft Office program.

Note It's possible to define your own constants in VBA as well. However, the discussion of constants in this chapter is limited to built-in constants, generally referred to as *intrinsic* constants.

Constants specific to the Word object model start with the letters *wd*; those specific to the Excel object model start with the letters *xl*; those specific to PowerPoint start with *pp*; and those for use across the Microsoft Office programs start with *mso*. There are also sets of constants that are specific to the Visual Basic language and available to VBA in all of the Microsoft Office programs—these constants begin with the letters *vb*.

Because constants are defined members of an object model, you can search for them in the Object Browser. For the purposes of searching the Object Browser, note that a set of constants is considered an enumeration class, and the constants within that enumeration are the members of that class. Sets of available constants for a given argument are also usually easy to find through VBA help. Additionally, Auto Lists are available for many constant sets, particularly object and property constants. Take a look at a few examples.

- The `Type` property of the `Field` object is available as a set of constants, provided in an Auto List when you type a valid statement for using this property. The example shown in Figure 23-13 is the beginning of a conditional statement.

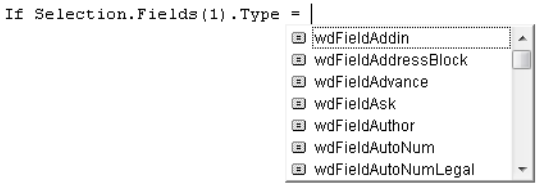


FIGURE 23-13 Field type constants, shown in the Word 2010 Visual Basic Editor.

See Also Learn about conditional statements in the section "Using Conditional Structures," later in this chapter.

Note that available constants might differ by platform or application. For example, the field type list in Figure 23-13 shows Word 2010 field types. Most, but not all, field types are available in both Word 2010 and Word 2011.

- Because different header or footer types are available in each section, the `Header` and `Footer` objects have a set of constants from which to select when you use those objects, as you see in Figure 23-14.

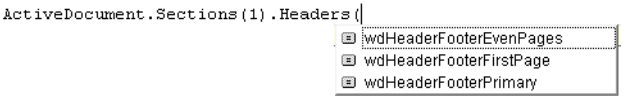


FIGURE 23-14 Available header and footer constants are the same in Word 2010 and Word 2011.

- The first macro you saw in this primer (in the section "Recording Macros") recorded four consecutive statements for adding four paragraphs to the document. If you had written that macro instead, you could have used the constant `vbCr`, which is the VBA constant to indicate a carriage return. In that case, that first macro could have been written with the following code, in just two statements instead of six:

```
Selection.Style = ActiveDocument.Styles("Heading 1")
Selection.TypeText("Company Overview—" & vbCr & vbCr & vbCr & vbCr)
```

Note The ampersand (&) is used to combine the text and constant portions of the text string, just as you can do to combine text, functions, and cell references into a text string in Excel. Learn more about using operators in VBA in the section "Using Operators," later in this chapter.

- Many arguments for different methods use the same sets of constants, which often are not available in Auto Lists, but are still easy enough to find. For example, the `HomeKey` method shown earlier uses constants for both of its arguments. The `Unit` argument uses the `wdUnits` set of constants; the `Extend` argument uses the `wdMovementType` set of constants.

The easiest way to learn which constant set you need is to search VBA help for the applicable method. This is because, in some cases, not all members of a constant set are available to all methods that use those constants. For example, `wdUnits` includes 16 constants, but only 4 are available when used with the `HomeKey` method. (The four available in this case are `wdLine` [the default if you don't specify the argument], `wdStory`, `wdRow`, and `wdColumn`—the last two of which apply only when your selection is in a table.) If you searched for the `HomeKey` method in VBA help, you'd see information about the available constants for both arguments.

See Also Note that the upcoming "Getting Help" section shows you how to use the Object Browser and VBA help reference together to save time. Comprehensive VBA help might not appear to be available in Office 2011, but Mac users, take heart—the "Getting Help" section provides an easy solution.

VBA by the Numbers: Using Numeric Values in Place of Constants

It doesn't take a mathematical genius to know that a term comprising several alpha characters, such as `wdLine` or `msoThemeColorAccent2`, is not a number, right? Actually, it is.

A set of constants is referred to as an enumeration because each constant represents a numeric value. When writing your code, if you know the numeric value that corresponds to the constant you need, you can use that value instead. If you select a constant in the Object Browser, the definition you see in the browser (below the search results window) indicates the numeric equivalent. Similarly, when you use a constant in your macro, you can right-click the constant and then click Quick Info to see its value, as shown in Figure 23-15.

```
Selection.MoveDown Unit:=wdLine, Count:=2
                           wdLine = 5
```

FIGURE 23-15 The numeric value of the wdLine constant.

Understanding Collection Objects

Objects for which there can be many instances of the object type within a given scope are available as both an object and a collection object. A collection comprises all instances of a given object type within the specified scope. This distinction is important because the object and its collection object can have very different members (that is, a very different set of available properties and methods). For example, compare the two statements that follow:

```
Documents(1).Tables.Count
Documents(1).Tables(1).AllowAutoFit = True
```

The first of the two preceding statements uses the `Tables` collection object. The second uses the `Table` object, specifying the first table in the collection. Both statements also use the `Document` object, specifying the first document in the `Documents` collection. (Note that the `Documents` collection in the Word object model refers to all currently open documents. The first document in the collection refers to the most recently opened document.)

The `Table` object has a very broad set of members, as you see in Figure 23-16. It's used whenever a single object is being referenced from the collection. Notice that only a fraction of this object's member list is visible in a single screen.

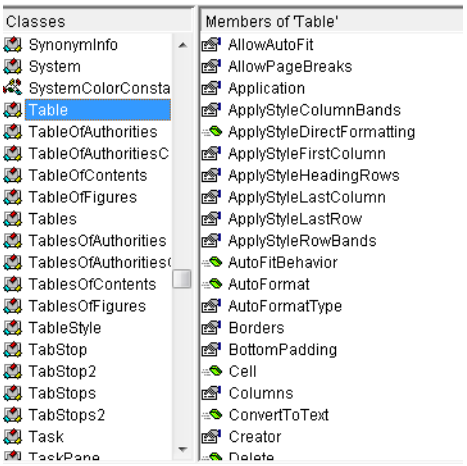


FIGURE 23-16 Members of the Table object, shown in the Object Browser.

In contrast, the `Tables` collection object has very few members (shown in Figure 23-17), including only those items that can apply to the entire collection at once.

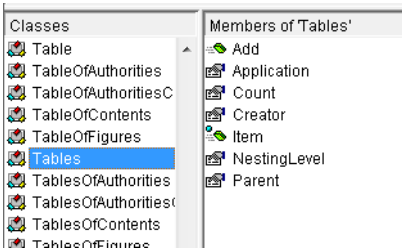


FIGURE 23-17 Members of the Tables collection object, shown in the Object Browser.

To Toggle or Not to Toggle

When you record some formatting options, such as applying bold or italics, you see the constant `wdToggle` set as the property's value. That's because, as you've likely noticed when using some types of font formatting, these settings toggle on and off—so you could, for example, apply a style that contains bold formatting to unbold text that was previously bolded.

However, when you write VBA macros, you can set toggle commands to absolute values where you need them, rather than using the `wdToggle` constant. To do that, simply use either `True` or `False` as the value instead of `wdToggle`.

Writing the macro `Selection.Bold = True`, for example, will apply bold text regardless of whether the selection was bolded before you ran the macro.

I Can't Find Properties or Methods That Should Clearly Be Available to My Object

If you can't find a member of an object that you just *know* has to be there, the problem probably isn't a limitation in VBA; it's syntax. More often than not, the `Range` object is the solution.

For example, if you want to act on all cells in a specific table, you might be looking for the following:

```
Documents(1).Tables(1).Cells…
```

However, when you get the Auto List after you type the **C** for the word `Cells`, you see that `Cell` is an option, but the `Cells` collection is not. Does this mean that VBA can't act on all cells in a table at once? Of course not. What VBA is looking for is the following:

```
Documents(1).Tables(1).Range.Cells…
```

The `Range` object is often used to specify that you're identifying the preceding object as the active scope of your statement. So, before you decide that you can't do what you need to do with a given object, try using `Range` in your statement, as shown in the preceding example, and see what options you get in the Auto List that follows that range.

The Range object is also important to know because it can give you added flexibility. People new to VBA often use the Selection object too frequently because they don't know how to identify objects in the document without first selecting them, but that slows down your code and often limits what you can do. You can use the Range object to identify any element in your document, which lets you take action without having to first select that item in the document. So, it's less code for you to write and less code for the program to execute.

For example, you could use the following statement to identify the page number on which the first table in the document ends, without having to first select that table:

```
ActiveDocument.Tables(1).Range.Information(wdActiveEndPageNumber)
```

Grouping Statements

Say that you're in a restaurant and you need three things from the waiter. If you ask for some ketchup, then ask for a glass of wine when the waiter drops off your ketchup, and then ask for a glass for your friend when the waiter returns with your wine, that's a lot of work for the waiter (not to mention, he might be tempted to sneeze in your soup).

Instead, if you say to the waiter, "I need some ketchup, please. I'd also like another glass of wine, and my friend will have one as well," you've given the waiter three tasks that he can execute together. That is, you've just grouped a set of statements (and saved yourself from a possible cold).

Though VBA won't sneeze in your soup, macros do run more slowly when you force the program to execute several related tasks independently. Grouping related statements together helps make your code more efficient (and saves you time writing code, because you'll be writing less).

Statements can be grouped using the With...End With structure, as you saw in the recorded macro example in the earlier sidebar, "Why Does My Recorded Macro Have So Many Lines of Code, When I Did Only One Thing?" You can use With...End With anywhere that two or more statements apply to the same object, or the same combination of objects, properties, and methods. For example, the very first macro we looked at in this chapter contains six statements, all of which apply to the Selection object. So, if you had written that macro instead of recording it, you could have typed the following:

```
With Selection
   .Style = "Heading 1"
   .TypeText "Company Overview—" & vbCr & vbCr & vbCr & vbCr
End With
```

Note Recorded macros will sometimes include lengthier statements than are necessary when you write the macro. This is typically because a recorded macro often explicitly states defaults that are assumed if omitted. For example, in the recorded version of this macro, the style was applied with the statement `Selection.Style = ActiveDocument.Styles("Heading 1")`. However, a style is applied using the style definition in the active document by default. So when written, as you see in the preceding code, that information can be omitted.

Though you might not be saving much by grouping statements when the macro is just two lines long, imagine something a bit lengthier. For example, say that you wanted to do several things to the first table in the document. Instead of starting each line with `ActiveDocument.Tables(1)`, you can group the statements using a `With`...`End With` structure, as follows:

```
With ActiveDocument.Tables(1)
   .Style = "Table Contemporary"
   .Range.Style = "Table text"
   .Columns(4).Shading.ForegroundPatternColor = wdColorLavender
   .Rows(1).Range.Style = "Table heading"
   .Rows(1).HeadingFormat = True
End With
```

In fact, you can take that grouping a step further. Notice that the first row of the table is referred to more than once. You can add a nested `With`...`End With` structure for those rows as follows:

```
With ActiveDocument.Tables(1)
   .Style = "Table Contemporary"
   .Range.Style = "Table text"
   .Columns(4).Shading.ForegroundPatternColor = wdColorLavender
   With .Rows(1)
         .Range.Style = "Table heading"
         .HeadingFormat = True
   End With
End With
```

With grouping structures, just remember that all items in the `With` statement must apply to all statements between `With` and `End With`, if the statement starts with a period (which indicates that it uses the object referred to in the `With` statement). For example, you can do some things directly to the `Row` object that you can't do directly to the `Column` object, such as applying a style. In that case, you might want to first select the column for which you need to apply a paragraph style, as you see here:

```
With ActiveDocument.Tables(1)
   .Style = "Table Contemporary"
   .Range.Style = "Table text"
   With .Columns(4)
         .Shading.ForegroundPatternColor = wdColorLavender
         .Select
   End With
```

```
   Selection.Style = "Table Subheading"
   With .Rows(1)
         .Range.Style = "Table heading"
         .HeadingFormat = True
   End With
End With
```

In the preceding code, `Selection.Style` doesn't have to refer to the object in the `With` statement, because it isn't using that object.

Caution As mentioned earlier in this chapter, remember that `With`...`End With` structures (as well as the code structures described in the upcoming section "Looping Code") require a pair of statements. For ease of editing and to reduce errors, whenever you type the first part of the structure (the `With` statement, in this case), type its paired closing statement (`End With`) as well, so that you don't forget to do so later.

How Can I Apply Theme Fonts and Colors with Word VBA?

In the preceding examples of grouping structures, you may have noticed that an intrinsic constant was used to apply a shading color. The set of constants you get when you apply colors in Word VBA code are from the standard Microsoft Office color palette that was available in versions prior to Office 2007 and Office 2008 for Mac. These are not theme colors.

However, the `wdThemeColor`... constants are available only to Office Art objects in Word 2010 and Word 2011, such as shapes. (Note that in Excel and PowerPoint VBA, the `xlThemeColor`... and `msoThemeColor` sets of constants, respectively, are available to most content types.) So, how do you apply colors that will update if the document theme changes?

If you record a macro in Word to apply a color from the Theme Colors palette (such as to a paragraph border), you'll see a numeric value in the resulting code that appears to represent the color you selected. That value actually represents the position you selected in the Theme Colors palette. So, using that value in your own macros will consistently apply the theme color at that palette position.

For example, in the preceding grouping structure code sample, if you wanted to apply the Theme Color palette position for the Accent 1 color instead of the standard color lavender, the line of code to apply the color would read as follows:

```
.Shading.ForegroundPatternColor = -738131969
```

But you don't have to take the time to discover the 60 values that comprise the 10 Theme colors in the palette and their variations. I've done it for you.

Companion Content *Find a macro named ThemeColorReferenceTable in the file named Sample macros.dotm that's available in the Chapter23 sample files folder online at* http://aka.ms/651999/files*. That macro will generate a new document with a table representing the Theme Colors palette. The color and associated value number for each position in the palette are applied to the table, so you can save that table and use it as a reference tool whenever you need it.*

Applying theme fonts in Word VBA is much simpler. Wherever you would specify the font name, you simply specify that name as "+Headings" or "+Body" to apply the theme heading or body font, rather than using the name of a specific font. For example, the first line of code in the following example applies the font "Cambria." The second line applies the active theme's heading font.

```
ActiveDocument.Paragraphs(1).Range.Font.Name = "Cambria"
ActiveDocument.Paragraphs(1).Range.Font.Name = "+Headings"
```

Insider Tip: Get More Flexibility for Theme Colors When You Use VBA

The preceding sidebar provided a workaround for situations where you can't use the theme color constants in Word. But in cases where you *can* use those constants in Word, PowerPoint, and Excel, you can also use VBA to give you more design flexibility than you get from within the program—without sacrificing the capability for colors to change automatically when you apply new themes.

Use the `TintAndShade` property with an `ObjectThemeColor` property to specify any percentage for a tint or shade of the applied theme color. For example, the following code applies the Accent 3 color to a shape and then sets it to be 35 percent lighter than the applied theme color (a tint that is not available in the color palette for mid-range colors):

```
With Documents(1).Shapes(1).Fill.ForeColor
   .ObjectThemeColor = wdThemeColorAccent3
....TintAndShade = 0.35
End With
```

When using the `TintAndShade` property, use positive percentage values for tints (to lighten the color) and negative percentage values for shades (to darken the color).

See Also You can also get more flexibility for customizing tints and shades of theme colors for all Microsoft Office content to which theme colors can be applied when you use Office Open XML. Learn more about using Office Open XML to customize document content in Chapter 24, "Office Open XML Essentials."

Looping Code

If I had to pick one feature of VBA that's the most useful on a daily basis for document production and document troubleshooting, it would be loops. Loops enable you to act on several instances of a given object within one macro. Fortunately, as much as loops can do for you, they're also extremely easy to use.

In this primer, we'll look at variations on two of the most common types of loops, `For` loops and `Do` loops.

Using For Each...Next and For...Next Loops

A `For Each...Next` loop enables you to act on all instances of a given object within a specified range. For example, you might use this type of loop to format all tables in your document at once or to change the fill color of all text boxes in your document to a particular theme color. Similarly, a `For...Next` loop enables you to specify a range of instances of the given object on which you want to act. For example, say that all tables in your document other than the first five need to have the same formatting. You can use a `For...Next` loop to specify that the formatting should apply to only those tables you want.

To use a `For Each...Next` loop, start by declaring a variable of the object type upon which to act and then use that variable in your loop. Take a look at the code for the two examples given in the preceding paragraph.

- Apply the style Table Contemporary to all tables in your document.

```
Dim atb as Table
For Each atb in ActiveDocument.Tables
   atb.Style = "Table Contemporary"
Next atb
```

  The use of `atb` as the variable name for the table object is just a personal choice. As mentioned earlier in this chapter, you can use any name for a variable that meets VBA naming requirements (no spaces and a letter for the first character) and isn't the name of any member of an available object model.

- Remove any user-defined styles from the active document.

```
Dim ast as Style
For Each ast in ActiveDocument.Styles
   If ast.BuiltIn = False Then
         ast.Delete
   End If
Next ast
```

  Specifying the variable in the `Next` statement, as shown in both preceding examples, is optional. However, it's good practice to do this to avoid confusing the statements you need to keep or alter when you edit a macro, particularly when you use multiple loops in the same procedure.

To use a `For...Next` loop, start by declaring a numeric variable data type to use for counting the instances upon which you want to act. Following is the code for the example given earlier (formatting all but the first five tables in the document).

```
Dim myI as Integer
For myI = 6 to ActiveDocument.Tables.Count
   ActiveDocument.Tables(myI).Style = "Table Contemporary"
Next myI
```

Notice that I could have used a `With...End With` structure instead of retyping `ActiveDocument` each time I needed it. Of course, that would be more helpful if I were doing more than just applying a table style, as you see in the following example:

```
Dim myI as Integer
With ActiveDocument
   For myI = 6 to .Tables.Count
      With .Tables(myI)
         .Style = "Table Contemporary"
         .AutoFitBehavior (wdAutoFitWindow)
      End With
   Next myI
End With
```

In the preceding code, notice that I use the `For...Next` loop with nested `With...End With` structures to make this macro as efficient as possible to write, and as efficient as possible for Word to execute.

Using Do Loops

A `Do` loop, aside from being fun to say, can be another useful way of creating a loop for specified instances of an object. (Note that this type of loop is usually referred to as a `Do...Loop` structure, which helps to clarify the fact that, like `For...Next` loops or `With...End With` structures, a `Do...Loop` actually requires a pair of statements.)

Do...Loop structures can either be executed while a qualification is true or until a qualification becomes true. Similar to For...Next loops, a Do While...Loop is usually used with a numeric variable. A Do Until...Loop may be used with a numeric variable or until a given condition is true. Take a look at a couple of examples.

- Say that you're troubleshooting a document. Using Open And Repair in Word 2010, you find that a floating object is causing the unstable document behavior. However, you don't see any floating objects in the document (this would happen if floating objects were off the page, or hidden behind opaque document elements because of the Behind Text wrapping style). Using a Do...Loop, you can delete all floating objects in the body of the document, as follows:

```
With ActiveDocument
   Do Until .Shapes.Count = 0
         .Shapes(1).Delete
   Loop
End With
```

  In the preceding code, notice that ActiveDocument.Shapes(1) refers to the first shape in the document. In this case, you wouldn't use a For...Next loop with a counter, because each time a shape is deleted, the shape object reference .Shapes(myI) would refer to a different object. Instead, if you continually delete the first shape until there are no more shapes, you don't need to be concerned with the way VBA counts the shapes in the document as their number is being reduced.

  In the case of deleting all shapes in a document, you may wonder why a For Each...Next loop wasn't used, since we want to act on all instances of shapes in the document. For Each...Next loops are an easy solution in most cases that require acting on all instances of an object type. However, there are two reasons why the Do...Loop was the better choice here. First, there's less code with a Do...Loop in this case because you don't need to declare the object variable before executing the loop. Second, there's an anomaly when you use a For Each...Next loop specifically to delete floating graphics (that is, members of the Shapes collection object), and one or more shapes may be left behind. Using the Do...Loop structure instead ensures that all shapes are deleted.

- The following code uses a Do While...Loop instead of a For...Next loop for formatting all tables other than the first five with the Table Contemporary style and AutoFit To Window behavior.

```
Dim myI as Integer
myI = 6
With ActiveDocument
   Do While myI <=.Tables.Count
      With .Tables(myI)
         .Style = "Table Contemporary"
         .AutoFitBehavior (wdAutoFitWindow)
      End With
      myI = myI + 1
   Loop
End With
```

Notice in the preceding code that the integer variable was set to start counting at six, so the first five tables in the document would be ignored. The `Do While` statement says to execute the code in the loop while the integer value is less than or equal to the number of tables in the active document. Then, at the bottom of the commands that fall within the loop, you see a counter for the integer variable to increase the number by one with each iteration of the loop.

In the first of the two preceding examples, a `Do...Loop` structure is a better choice than a `For...Next` loop (as explained in the text that follows that sample code). However, in the second example, a `For...Next` loop would have been the more efficient choice. Notice that, in the second example, if you use a `For...Next` loop, you don't need a separate statement for the counter—the `For` statement is a built-in counter.

So, how do you decide whether to use a `For...Next` loop or a `Do...Loop` structure? You just need to ask yourself a few simple questions, as follows (and as summarized in Figure 23-18).

Note I wish I had conceived the questions that follow, but I can't take the credit. Many thanks to Beth Melton, who was the technical reviewer for the first version of this primer and for sharing her clear and concise approach to this topic (and others).

- Do you know the number of repetitions you need in the loop?

  As demonstrated by the preceding code samples in this section, if the answer is yes, use a `For...Next` loop. If the answer is no, use a `Do...Loop`.

- If you're using a `Do...Loop` structure, is the condition initially true?

  If the condition is initially true, you need a `Do While` statement to begin your loop. If, on the other hand, the loop needs to execute until the condition *becomes* true, start your loop with a `Do Until` statement.

There's one more factor to consider when deciding on the loop type you need. You can evaluate the condition specified in a `Do...Loop` structure either at the top of the loop (as shown in the earlier example of a `Do While...Loop` structure) or at the bottom of the loop (with a `Do...Loop Until` or `Do...Loop While` structure).

A top evaluation loop is structured as follows:

```
Do While <condition>
   <statements>
Loop
```

A bottom evaluation loop, on the other hand, looks like this:

```
Do
   <statements>
Loop While <condition>
```

(Remember, in the preceding structures, to substitute `Until` for `While` if you need to execute the code *until* the condition becomes true.)

So, to determine whether you need a top or bottom evaluation loop, ask the following question: must the code execute at least once?

If the code must run at least once for your macro to do what you need, use a bottom evaluation loop so that the condition isn't evaluated until after the first time the code runs. If the code doesn't have to run at least once, use a top evaluation loop so that the condition is evaluated before the first time the code runs. For example, in the sample `Do...Loop` structure shown earlier—in which the loop is used to delete all shapes from the active document—a top evaluation loop is appropriate, because the code doesn't need to run if the document contains no shapes from the outset.

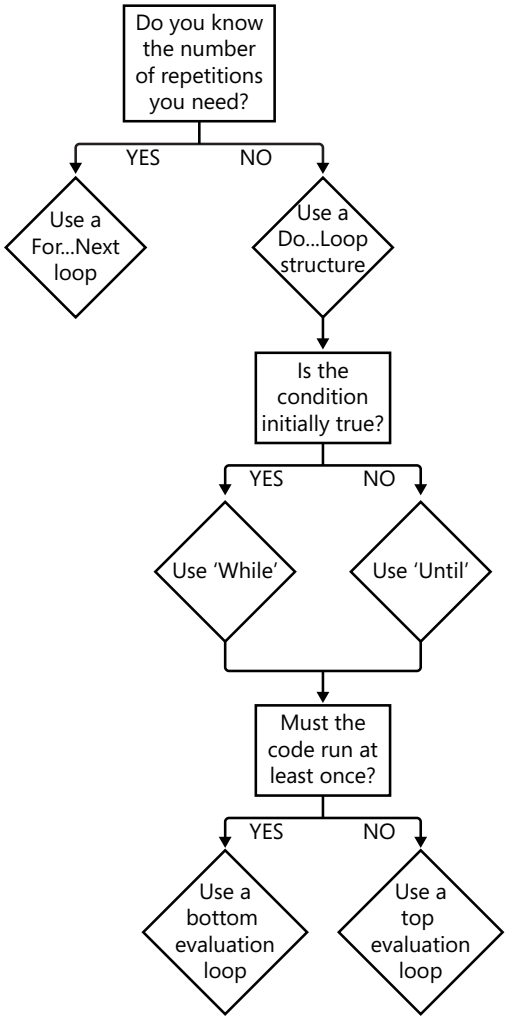


FIGURE 23-18 A summary of the decision process for selecting the best type of loop for your macro.

Using a Loop to Delete Objects from a Word Document Leaves Behind Objects in the Header and Footer

If you press Ctrl+A to select the entire Word document and then press Ctrl+Shift+N to apply Normal paragraph style to everything selected, content in headers, footers, footnotes, endnotes, comments, or floating text boxes remains unaffected. This is because Select All really means selecting everything in the active Word story.

See Also For an explanation of Word stories, see Chapter 6, "Building Easy-to-Manage, Robust Documents."

Similarly, when you work in VBA, there are some commands that you must execute them separately for each story in which you want to take action. For example, deleting all floating objects from the active document, as discussed in the preceding section, "Using Do Loops," won't delete those objects in the header or footer stories. To do that, you need to access the particular story you need. To delete floating objects in the main header of section 1, for example, instead of `ActiveDocument.Shapes`, you would specify `ActiveDocument.Sections(1).Headers(wdHeaderFooterPrimary).Shapes`.

Keep in mind that you can nest multiple loops inside one another. For example, say that you want to apply a paragraph style to every header in the document. You need to loop through each header in each section, which requires two loops, as follows:

```
Dim asc As Section
Dim hft As HeaderFooter
For Each asc In ActiveDocument.Sections
   For Each hft In asc.Headers
         hft.Range.Style = "Heading 1"
   Next hft
Next asc
```

Note that some actions require that you access the story in which you want to act before executing actions, rather than just identifying it. Also, before you try to loop through all story ranges in a document, be sure that the type of object upon which you're acting is accessible to all story ranges, or you'll get an error. For example, floating objects aren't allowed in footnotes, endnotes, comments, or other floating objects, so including those ranges in your loop to delete shapes will throw an error.

Also note that, although a Word document is defined by stories, programmatically accessing content in PowerPoint takes a similar approach. For example, taking action on all slides in a presentation will not affect content on slide masters and slide layouts. If you wanted to delete all shapes in the presentation, you would need to include code to delete shapes from the slides, masters, and layouts (and separately include notes pages and notes and handouts masters, if applicable). This is less applicable in Excel, since most content resides directly on worksheets. However, note that it's often necessary to programmatically access different types of sheets or objects (such as worksheets and chart sheets) separately.

Using Conditional Structures

As demonstrated with `For...Next` and `Do...Loop` structures, there are several ways to apply conditions to the commands you want to execute with VBA. Frequently, however, the condition you need may be something other than the instances of an object. Conditional structures in VBA, other than loops, are formed using either the paired `If` and `End If` statement or the `Select Case...End Select` statement.

Creating If Statements

Much like the IF function in Excel and the IF field in Word, `If...End If` structures in VBA are used for executing actions when specified criteria are met. Take a look at the following examples:

- Say that you're creating automation to format new business presentation documents. Your branding specifies that any presentation of longer than three pages should use landscape orientation. If the user clicks the button to use your formatting macro, you may want the macro to first check the length of the document and then set the orientation to landscape if the document exceeds three pages.

```
With ActiveDocument
  If .RangeInformation(wdActiveEndPageNumber) > 3 Then
     .PageSetup.Orientation = wdOrientLandscape
  End If
End With
```

- Say that you're applying a template to a document that uses only built-in Word styles, such as Normal and Headings 1–9. Once you've reformatted the document content as needed, you may want to clean up the document styles to help ensure that the document continues to be formatted with the styles you want. The following code removes any styles from the document that are not built in:

```
Dim ast As Style
For Each ast In ActiveDocument.Styles
   If ast.BuiltIn = False Then
         ast.Delete
   End If
Next ast
```

`If...End If` structures are often used with multiple conditions, such as when you want to set one value if the condition is true and another if it's false, as you see in the following example:

```
With ActiveDocument
  If .Range.Information(wdActiveEndPageNumber) > 3 Then
     .PageSetup.Orientation = wdOrientLandscape
  Else
     .PageSetup.Orientation = wdOrientPortrait
  End If
End With
```

The preceding example adds an extra qualifier to the similar code shown earlier, so that if the document is three pages or shorter, your macro ensures that the document uses portrait orientation.

`If` statements can also contain multiple conditions by including `ElseIf` statements. For example, say that you have many tables in your document with different layouts, but all financial tables have either four or six columns. Those financial tables with four columns should use the custom table style named Table Financial 4, those with six columns should use the style named Table Financial 6, and all other tables in the document should be formatted using Table Normal style.

```
Dim atb As Table
For Each atb In ActiveDocument.Tables
   With atb
         If .Columns.Count = 4 Then
               .Style = "Table Financial 4"
         ElseIf .Columns.Count = 6 Then
               .Style = "Table Financial 6"
         Else
               .Style = "Table Normal"
         End If
   End With
Next atb
```

Notice that both `If` and `ElseIf` statements require `Then` at the end of the line. Also notice that, regardless of the number of conditions in an `If` statement, `End If` is still required at the end of the complete structure.

Note For simple `If` structures where there is a single condition, you can get an exception to the paired statement requirement. You can often add just a single statement on one line and leave off the `End If` statement. For example, the `If`...`End If` statements in the first two examples given earlier could have been written on a single line.

```
Dim ast As Style
For Each ast In ActiveDocument.Styles
   If ast.BuiltIn = False Then ast.Delete
Next ast
```

Creating Select Case Statements

Although `If` structures are the most common conditional structure used in VBA, `Select Case` can be an extremely efficient alternative in some situations, so it is definitely worth a look.

Here, `Select Case` is used for one of the previous `If` statement examples:

```
Dim atb As Table
For Each atb In ActiveDocument.Tables
   With atb
         Select Case .Columns.Count
            Case 4
               .Style = "Table Financial 4"
            Case 6
               .Style = "Table Financial 6"
            Case Else
               .Style = "Table Normal"
         End Select
   End With
Next atb
```

For this code, the `If` structure and `Select Case` structure are very similar and essentially equally good choices. Where `Select Case` can be more useful is when several options meet a given condition.

For example, say that a long report document has been through a lot of hands and had content copied and pasted from several sources, and now you want to go through it and quickly clean up the styles. Perhaps you want to replace the style for paragraphs formatted with a number of different body text styles with Normal style, and replace the use of a few custom heading styles with the Heading 1 style. Notice in the code that follows that multiple options for a given case are separated by commas.

```
Dim myI As Integer
myI = 1
With ActiveDocument
    For myI = 1 To .paragraphs.Count
        With .paragraphs(myI)
            Select Case .Style
                Case "Body Text", "Body Text 2", "Body Text 3"
                    .Style = "Normal"
                Case "Body Heading", "Document Heading", "Page Heading"
                    .Style = "Heading 1"
            End Select
        End With
    Next
End With
```

If you used an `If` statement instead of `Select Case` here, the code would be as follows:

```
Dim myI As Integer

myI = 1
With ActiveDocument
    For myI = 1 To .paragraphs.Count
        With .paragraphs(myI)
```

```
            If .Style = "Body Text" Or .Style = "Body Text 2" Or .Style = "Body Text 3" Then
                    .Style = "Normal"
             ElseIf .Style = "Body Heading" Or .Style = "Document Heading" Or _
                .Style = "Page Heading" Then
                    .Style = "Heading 1"
            End If
        End With
    Next
End With
```

The savings with `Select Case` is the ability to separate multiple options with just a comma rather than repeating the entire condition for each option. As you gain experience writing your own macros, you will run into many situations where `If` or `Select Case` is the better choice.

The Value of Indenting Code

As you can see throughout the code samples in this chapter, code is indented to indicate statements within a group, loop, or condition. Where multiple structures are nested, code is indented a bit further for each level of nesting.

Though VBA doesn't require indenting code, it's fairly standard practice to do so, because it makes the code and the logic of the macro's progression much easier to read.

For example, consider that when structures are nested, it's essential that the end statements of paired structures fall in the correct order. So, for example, if you nest an `If`…`End If` structure inside a `For`…`Next` loop, the `End If` statement needs to appear above `Next` (that is, if the `If` structure starts inside the loop, it has to end inside the loop). Indenting phrases can make it much easier to diagnose this type of hierarchy issue.

To indent a line of code, press the Tab key at the beginning of the line. Your indent will remain when you press Enter. Press Backspace or Shift+Tab to remove the indent. Note that you can customize the indent width on the Editor tab of the Options (or, for Office 2011, Preferences) dialog box. Tab width is set to four characters by default.

To indent several lines of existing text, select those lines and then press Tab. Similar to how outline numbered lists behave, using Tab and Shift+Tab with one or more paragraphs selected will demote or promote the text rather than deleting it. However, if only one statement is selected, you must be sure to select the entire statement, or pressing a keystroke will replace the existing text.

Using Operators

VBA uses both symbols (such as &, <, >, =, +, -, /, *) and terms (such as And, Or, and To) for operators, depending on the usage. In all cases, however, operators follow standard mathematical syntax rules. Take a look at a few examples.

- When I finish writing a chapter of this book, I need to copy all of the Heading 1 and Heading 2 paragraphs to update the table of contents. To do that, I make a copy of the document, from which I delete any paragraphs that don't have those two styles applied.

```
Dim apr as Paragraph
For each apr in ActiveDocument.Paragraphs
   If apr.Style <> "Heading 1" And apr.Style <> "Heading 2" Then
         apr.Range.Delete
   End If
Next apr
```

  The less than and greater than operators are used together here to mean "is not equal to." Note that I could also have written the `If` portion of that statement as follows:

```
   If Not apr.Style = "Heading 1" And Not apr.Style = "Heading 2" Then
```

  Notice that the use of `Not` to mean "anything other than" is repeated for each option meeting the condition.

- If, instead, I wanted to delete all paragraphs that match either of those criteria, I would have written the following code:

```
Dim apr as Paragraph
For each apr in ActiveDocument.Paragraphs
   If apr.Style = "Heading 1" Or apr.Style = "Heading 2" Then
         apr.Range.Delete
   End If
Next apr
```

- What if I wanted to delete all paragraphs that use Heading 1 or Heading 2 style, but only if they don't appear in a table?

```
Dim apr as Paragraph
For each apr in ActiveDocument.Paragraphs
   If (apr.Style = "Heading 1" Or apr.Style = "Heading 2") And _
   apr.Range.Information(wdWithinTable) = False Then
         apr.Range.Delete
   End If
Next apr
```

  In the first line of the `If` structure, the space followed by an underscore at the end of the line breaks a single statement of code to a second line. Breaking the line is not required, but is used when the line of code is too wide to read in a single screen.

Notice in the preceding code that the conditions that use the logical operator `Or` are grouped in parentheses, with the `And` operator outside the parentheses. Just as in a mathematical equation, that phrasing ensures that the condition within the parentheses is evaluated first.

As you've seen in examples throughout the primer to this point, an ampersand combines arguments into a text string, and typical arithmetic operators can be used on numeric values as they are in Excel formulas, including +, - ,*, and /. The plus sign can be used in some cases to combine text strings, but when you want to mix different types of variables in a text string, the plus sign can cause a "Type Mismatch" error, because it tries to calculate a result rather than combine the strings. So, using the ampersand to combine arguments into a string is always a good practice.

Notice also throughout these examples that comparison operators can be used either individually or together, such as < to indicate "less than" or <= to mean "less than or equal to."

See Also Although the operators mentioned in this section are likely to be all that you need, they're not an exhaustive list of every operator available in VBA. To learn about others, search for the topic Operator Keyword Summary in VBA help.

Note Although it's not included in the aforementioned Operator Keyword Summary, the term *To* can be used to represent a range of values (such as a more efficient alternative to using a greater than and less than range combined), as you've already seen in some code examples in this chapter. Consider the following examples, which are individual lines of a code from a `For…Next` loop and a `Select Case` statement.

```
For myI = 1 to ActiveDocument.Shapes.Count
Case 2, 3 To 10, 14
```

Introducing Message Boxes and Input Boxes

When creating macros for others to use, you'll likely need to either give the user information or have the user specify information. Use message boxes to share information and input boxes to collect it.

Using Message Boxes

A message box might simply provide information, or it might require a response, such as Yes, No, Cancel, Abort, Retry, or Ignore.

The `MsgBox` command is one of several in VBA that can be used both as a statement and as a function. Use a `MsgBox` statement to provide information; use `MsgBox` as a function when you need a response from the user.

- To create a message box statement, type `MsgBox` with the string of text you want the user to see. For example, take a look at the following message box statement and the message box it produces when run in Word, shown in Figure 23-19.

```
MsgBox "You're an unstoppable VBA genius!"
```

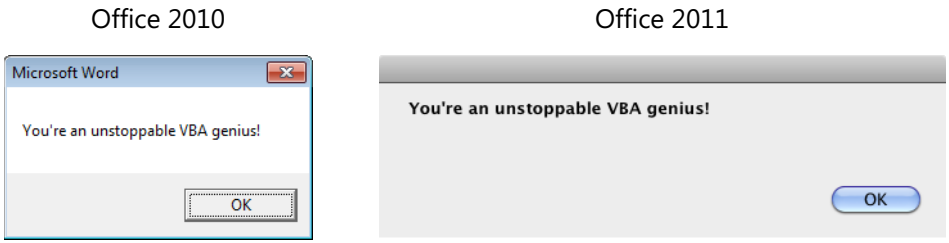


FIGURE 23-19 Notice that message boxes and input boxes automatically coordinate with the visual aesthetic of Microsoft Office on their respective platforms.

Even if your message box doesn't require a reply, however, you might want to get a bit more creative with it. The `MsgBox` command includes optional arguments that let you customize the title bar and add an information icon, as shown in Figure 23-20.

```
MsgBox "You're an unstoppable VBA genius!",vbInformation,"My Message Box"
```

FIGURE 23-20 A message box with custom title bar and information icon, shown in Office 2010 (left) and Office 2011 (right).

The intrinsic constant `vbInformation` is one of a set of options in the buttons argument that enables you to add both an icon (as you see here) and response buttons. The third argument customizes the title of the message box.

Notice that including a title in Office 2011 reduces the size of the message box body text and places the title within the body area of the message. Also notice that the custom icons are not provided in Office 2011. Instead, the information icon is read as the application icon (Word, in this case).

- To use `MsgBox` as a function (that is, to require a response from the user), first declare an integer variable for your message box so that you can use the response in the macro, as you see in the following example:

```
Dim myRes As Integer
myRes = Msgbox("Are you an unstoppable VBA genius?", vbQuestion _
+ vbYesNo, "My Message Box")
```

```
If myRes = vbYes Then
   Msgbox "I knew it!", vbExclamation, "You're a genius!"
Else
   Msgbox "Hang in there.", vbCritical, "It will get easier!"
End If
```

The first message box in the preceding code is shown in Figure 23-21.

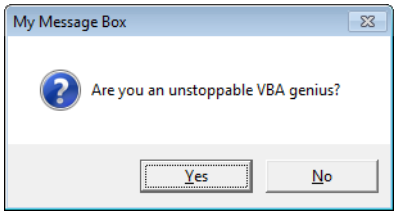


FIGURE 23-21 A message box providing options to the user, shown in Office 2010.

Depending upon the user's response, one of the two message boxes shown in Figure 23-22 is returned.

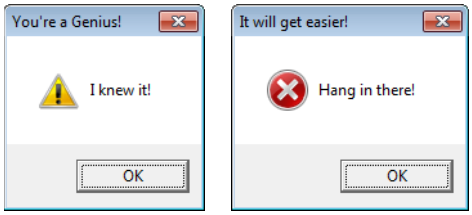


FIGURE 23-22 Message boxes customized in reply to a user response.

See Also Both message box and input box functions also include optional arguments for adding context-sensitive help files to those boxes. For additional resources where you can find information on VBA tasks that are not covered in this primer, such as creating custom help files for your VBA projects, see the "Getting Help" section, later in this chapter.

Using Input Boxes

Input boxes are similar to messages boxes, except that they're always used as a function because they always require a response. Take a look at the following example:

```
Dim myInp As String
myInp = InputBox("How would you score on a basic VBA exam?", _
"My Input Box", "Perfect")
Msgbox myInp & " is pretty good!", vbExclamation, "My Input Box"
```

The input box from this code sample is shown in Figure 23-23. Notice that the Office 2010 and Office 2011 input boxes are more similar than message boxes. For example, note that the title bar is utilized in Office 2011 message boxes.

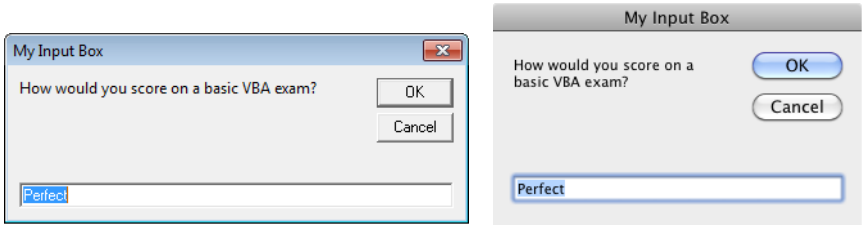


FIGURE 23-23 An input box shown in Office 2010 (left) and Office 2011 (right).

The text of the message box shown in Figure 23-23 is referred to as the prompt, the title bar text is the title argument (as in a message box), and the value you see in this image is the default value of "Perfect" specified in the third argument. Note that input boxes also include optional arguments for vertical and horizontal position on the screen (not shown here) for cases where you don't want the box to automatically appear in the center of the screen.

Because the input box was declared as a string variable, notice that the response is used as part of a text string in a message box, as shown in the preceding code sample and in Figure 23-24.

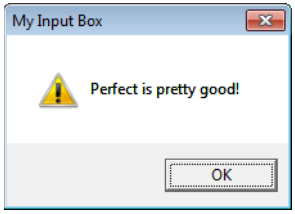


FIGURE 23-24 A message box constructed using a combination of a string variable and text string.

If, instead, you need to use a response as a numeric value, declare the variable accordingly. In the following example, the input box asks for the number of columns to include in a new table being created by the macro. The variable defined as the input box reply is declared as an integer. (Notice that the input box in this case has only a prompt and a title bar—no default value is set, so the text box within the input box appears blank to the user.)

```
Dim myInp As Integer
myInp = InputBox("How many columns would you like?", "My Input Box")
With Selection
.Tables.Add Range:=.Range, NumRows:=5, NumColumns:=myInp
End With
```

Caution There is a possible problem, however, with the preceding code sample. If the response is not an integer (including if the user cancels the input box without adding a reply), the macro will end in an error. You can, however, add what's known as an *error handler* to correct for any error that may occur. Error handlers are an important part of writing macros effectively.

Companion Content *To learn how to work with code errors and create error handlers, see the article "Managing VBA Errors," available in the Bonus Content folder as part of the online companion content for this book, at* http://aka.ms/651999/files*. You'll find an example in that article of an error handler created specifically for the preceding macro.*

Running One Macro from Another

When you create a solution, such as developing a set of document production macros for yourself or creating a set of macros to help users format a template, you're likely to have some of the same commands repeat in multiple macros. When those duplicated commands run to more than a few lines of code, it can be helpful to put the duplicated code into its own macro and run it as part of each macro that needs it. That way, you don't have to write that code out in every macro where you need it.

Running one macro from another is also commonly done when several macros use the same variable definitions. For example, say that you declare the following public variables in the General Declarations section of the module:

```
Public myName as String, myComp as String, myIn as Integer
```

If several macros need to use the same values for that information, create a procedure just to store the values of those variables. That entire macro might look something like this:

```
Public Sub VarDefs()
myName = Application.UserName
myComp = ActiveDocument.BuiltinDocumentProperties("Company").Value
myIn = 1
End Sub
```

To then use these variable definitions in any macro in the project, simply call the macro that includes the definitions. The statement to call a macro is just the word `Call` plus the macro name. If the macro exists in a different module from the macro where you're calling it, also specify the module name.

For example, to call the preceding macro from a macro in the same module, type the following statement:

```
Call VarDefs
```

If the macro from which you want to call `VarDefs` is in a different module, the statement would look like the following (assuming that `VarDefs` is in a module named `myMod`):

```
Call myMod.VarDefs
```

Note that, as long as the variables are declared as public, you don't actually have to specify `Public` in the `Sub` statement of the preceding macro to make the contents of that procedure available to other macros in the project. However, if you want to allow the contents of that procedure to be shared only by other macros in the same module (such as in cases where macros in a different module might need to share a different set of values for the same variables), use `Private Sub <procedurename>()` to start the macro. Keep in mind that private procedures don't appear in the Macros dialog box available from the Developer tab, so identifying a procedure as private is also a good way to keep it hidden from the user.

Caution When you call one macro from another for the purpose of using variable definitions, make sure the call to the source macro appears prior to where you use those variables in the destination macro.

Note Your macros might share many types of variables in a given project. The preceding variable example is just intended to point out one place where the Microsoft Office programs may offer you a simpler solution than VBA, depending on your particular needs. For adding document property information to your documents and templates, you can use the built-in Document Property Quick Parts.

See Also For more information about Document Property Quick Parts that have information already bound to content controls, as well as for resources where you can learn about creating custom bound controls, see Chapter 12, "Dynamic Content."

Setting Macros to Conditionally Stop Executing Commands

You can add a statement to end the macro under specified conditions or to exit just a part of the macro.

To end execution of a macro, use the term `Exit Sub`. For example, say that you want to stop a macro from running if no document is open. That code would look like this:

```
If Documents.Count = 0 Then
Exit Sub
End If
```

`Exit Sub`, however, exits the active procedure. If you have one procedure running from another, you might need to end code execution entirely instead of exiting the individual subroutine. In that case, just use the term `End`.

```
If Documents.Count = 0 Then
End
End If
```

To exit a loop when a condition is met, use an `Exit` statement specifically for the loop type, such as `Exit For` or `Exit Do`. Following is an example of an `Exit For` statement:

```
Dim ast as Style
For each ast in ActiveDocument.Styles
   If ast.NameLocal = "Sample" Then
      ast.Delete
      Exit For
   End If
Next
```

Running Macros and Compiling Projects

You can run a macro directly from the Visual Basic Editor or from the Macros dialog box available in the Microsoft Office programs, or you can customize the user interface to add the macro to either the Quick Access Toolbar or the Ribbon (or, in Office 2011, to a toolbar or menu). In Word 2010 or Word 2011, you can also assign a keyboard shortcut to a macro.

- To add a macro to the Quick Access Toolbar or the Ribbon in an Office 2010 program, using the Customize Quick Access Toolbar and Customize Ribbon tabs of the Options dialog box in the applicable program.
- To add a macro to a toolbar or menu in an Office 2011 program, on the View menu, point to Toolbars, and then click Customize Toolbars And Menus.
- To assign a keyboard shortcut to a macro in Word 2010, on the Customize Ribbon tab of the Word Options dialog box, under the heading Keyboard Shortcuts, click Customize. To do this in Word 2011, on the Tools menu, click Customize Keyboard.

You can save keyboard shortcut assignments, Quick Access Toolbar customizations, and Office 2011 menu and toolbar customizations either for the application as a whole or in the individual document or template. When you customize the Ribbon through the options dialog box, you can customize it only for the application as a whole. To customize the Ribbon for an individual document, template, or add-in in Office 2010, you have to use Office Open XML.

See Also For more on Office Open XML, see Chapter 24. Additionally, see the article "Using VBA to Create Add-Ins," available on the MSDN Office Developer Center at http://msdn.microsoft.com/en-us/library/gg597509.aspx. *(This article applies to both Office 2010 and Office 2011. Office 2010 UI customization examples are presented in the article but the downloadable samples for the article also include an Office 2011 UI customization example.)*

Do More Than You Might Imagine with One Line of Code

Depending on what you need to do with VBA, you might not need to write or run a macro at all. The Immediate Window in the Visual Basic Editor enables you to run just one line of code at a time. It's simple to use and can be an incredibly powerful tool for troubleshooting document issues or executing actions that you can't do from within the Microsoft Office programs.

Several chapters in this book include tips about using the Immediate Window and send you here for more information. Those tips include such things as resetting the used range in an Excel workbook, setting precise positioning for content on PowerPoint slide layouts and masters, or getting information about problem content that you don't see in a Word document.

To use the Immediate Window, start in the Visual Basic Editor and then:

1. On the View menu, click Immediate Window. Or Press Ctrl+G (Ctrl+Command+G in Office 2011).

   The Immediate Window opens at the bottom of the Visual Basic Editor by default.

2. With your insertion point in the Immediate Window, do one of the following:

- To execute an action, just type the line of code that you need and then press Enter (Return). For example, as shown in Chapter 18, "Working with Data," to reset the used range on the active worksheet, type the following:

  ```
  ActiveSheet.UsedRange
  ```

- To get information, type a question mark, immediately followed by the line of code you need (no spaces) and then press Enter (Return). For example, to count the number of tables in the active Word document, type the following:

  ```
  ?ActiveDocument.Tables.Count
  ```

For an article that offers more detail about using this tool, including many examples, see the additional resources provided in the last section of this chapter, "Working with VBA: Next Steps."

Compiling Projects

As you're writing lengthy macros, or when you're ready to use your macros, compiling the project is an important step. Compilers are actually used to translate source code to executable code, which isn't strictly necessary for typical VBA macros. But using the compiler in the Visual Basic Editor is an essential way to help ensure that your code works properly.

To compile a project, just select a module or click into a procedure in the project and then, on the Debug menu, click Compile <Project>. Depending on the size of the VBA project, compiling might be instantaneous or it might take a few moments. If VBA recognizes errors in your code, it will select the code containing the error and display a message box telling you the type of error.

Companion Content *Learn about recognizing, fixing, and managing errors in the article "Managing VBA Errors," available in the Bonus Content folder online at* http://aka.ms/651999/files.

Getting Help

In Office 2010 VBA, you can easily search for help using the Type A Question For Help box that appears on the right side of the Visual Basic Editor menu bar. In Office 2011, VBA help takes you to an online index of VBA help topics.

But there are also often faster ways to get to exactly what you need:

- In the case of error messages, the Help button in those message boxes takes you directly to a help article on that specific error message. If, however, you need information on an error message any time other than right when it occurs, search for the help topic Trappable Errors. You can then use the Find feature in the help or browser window (Ctrl+F on Windows and Command+F on Mac) to quickly locate the name or number of the particular error you need. The Trappable Errors article lists each error with a hyperlink to its article.
- In the case of any object model member, right-click the name of the item where it appears in the code and then click Definition. This opens the Object Browser to the selected item—which might be enough information if you just need, for example, to see the available members of a selected object.

  However, in the Object Browser in Office 2010, you can right-click any item and then click Help to open the help topic on that article. Note that some items, such as individual constants, might not have help articles—but articles are available for most members of the active object model.

For Mac Users

If you use the Help menu in the Visual Basic Editor, you get program help—not VBA help. If you click the Help button on the Standard toolbar in the Visual Basic Editor, you get a link to Visual Basic Editor help. This is an index of several useful help topics broken down by application. For example, you can see a handy list of what is different from the Office 2010 version of the object model for that Office 2011 application. But it's not searchable and not nearly as comprehensive as Office 2010 VBA help. So, are you out of luck?

By now, you know when I ask a question like that, there's usually an easy solution. The fact is that you have access to almost the exact VBA help that is available in Office 2010, because the help for Office 2010 VBA is hosted on Office.com.

To get to the developer home page for Word, go to the following link (yes, it's long and messy but you only have to type it once and then bookmark it):

http://office.microsoft.com/client/helphome14.aspx?lcid=1033&NS=WINWORD%2EDEV&Version=14

To get to the developer home pages for PowerPoint and Excel, just substitute the text **WINWORD** in the preceding URL with **POWERPNT** or **EXCEL**. (Note that characters are missing from PowerPoint in the preceding text; this is intentional.)

Then, to get to the help you need, either search or (as you might find more effective in some cases) click the application Object Model Reference link and then drill down in the object model links for what you need.

When searching the Object Model help to get to what you need, keep in mind that many types of VBA elements (such as properties and methods) are members of an object, so you can find help for them under the applicable object. And if you're not sure of the object to which an item belongs, remember that the Object Browser in the Visual Basic Editor gives you a definition for the item that includes its membership information.

Saving and Sharing Macros

You can export a module of code (as well as some other types of project elements), which is the equivalent of saving a copy of the file, by right-clicking the module in the Project Explorer pane and then clicking Export. Note that the file name you choose for the export doesn't need to match the module name. Notice also that VBA modules have the file extension .bas.

To import a module of code, such as the samples available in the online bonus content for this book, right-click the project in Project Explorer and then click Import.

Caution If you export or import modules, remember that some modules refer to code outside the module itself, such as when the project contains a UserForm (dialog box) or when one macro calls another from a different module in the project. Be sure that you're aware of the project components that need to work together, so that you export or make note of everything you'll need when you or someone else imports that content later.

See Also For additional resources, such as where you can learn to create and work with UserForms, see the section "Working with VBA: Next Steps" at the end of this chapter.

Because you can share an entire VBA project by sharing the Word, Excel, or PowerPoint file in which the project is stored, exporting is more often used as backup. This is usually a good idea, because if you lose a document or template, you of course lose any code it contained.

In particular, if you store a module of document production macros, for example, in Normal.dotm, exporting that module periodically for backup is an important safety measure. This is because you might solve some Word performance issues by deleting Normal.dotm and allowing Word to regenerate a new default template, in which case your macros would be lost.

Sharing Projects

To share an entire project, just compile the project, save the file, and share it as you would any file. Keep in mind that some networks block files that contain macros, so you might want to use a different method for safely sharing the content (such as saving a document or template containing a VBA project to a Windows Live SkyDrive folder or a Microsoft SharePoint library).

Some macro projects need to be saved as particular file types, such as for Excel and PowerPoint add-ins. Also, adding a digital signature to projects can help to avoid systems or programs blocking your macros.

See Also Learn about using VBA to create add-ins and signing your code in the MSDN Office Developer Center article "Using VBA to Create Add-Ins," at http://msdn.microsoft.com/en-us/library/gg597509.aspx.

You can also protect your code when sharing projects—such as when you want others to be able to use the macros, but not to be able to see or copy your source code. To do this, select the project in Project Explorer. Then, on the Tools menu, click <Project> Properties.

In the <Project> Properties dialog box, you can rename the project (following VBA naming conventions), which does not affect the file name of the document, template, or add-in where the project resides. You can also click the Protection tab to require a password to view

the code. For this feature to work, you must enable the Lock Project For Viewing option and provide a password. When you do, double-clicking the project in Project Explorer will display a box where you can type the password. Without the correct password, the macros can still be run from the user interface, but their code can't be viewed. Note that, once you add a password, the password protection starts the next time the project is opened.

Caution Be sure to keep a record of the password you choose. Lost passwords might render your code permanently locked.

Working with VBA: Next Steps

Once you've mastered the basics in this primer, you're likely to find more complex VBA to be quite easy. And there are many online resources to help you progress.

- The MSDN Office Developer Center (*http://msdn.microsoft.com/office*) is a fantastic resource for all things related to automating Microsoft Office. The majority of content on this site is for managed code (Microsoft Visual Studio) development (the platform often used by professional Microsoft Office developers), but far from all of it. You can find a growing number of resources on VBA as well as a volume of information on Office Open XML (which is addressed in Chapter 24).
- The Office Developer Center also has a VBA-specific site that's a fantastic online home base for learning about VBA. And it has one of my all-time favorite friendly URLs: *http://iheartmacros.com*.
- In addition, when you don't find the information you need in VBA help, search the MSDN library. The library's resources are pretty amazing—I can't tell you how much I've learned there: *http://msdn.microsoft.com/library*.

But in addition to general resources, there's also more VBA topic-specific content that I'd like to point out. The online bonus content for this book includes a couple of VBA articles, and there are also a few articles available on the Office Developer Center that might be particularly handy for next steps with VBA.

Note The list of four articles that follows was written for Office 2007 but also applies to Office 2010 and Office 2011. In the additional resources at the end of each article, find a link to a related video walk-through of tasks covered in it.

- The Immediate window in the Visual Basic Editor (discussed in the earlier sidebar "Do More Than You Might Imagine with One Line of Code" is one of my favorite tools, and by far one of the most valuable tools for using VBA to troubleshoot document formatting. Learn more about the Immediate window here: *http://msdn.microsoft.com/en-us/library/dd535470.aspx*.
- Get tips on using VBA specifically to troubleshoot documents here: *http://msdn.microsoft.com/en-us/library/dd630112.aspx*.
- Learn about controlling built-in Microsoft Office commands using VBA (including an introduction to using events) here: *http://msdn.microsoft.com/en-us/library/dd627337.aspx*.
- See examples of how to use some of what you learned in this primer to format long documents more quickly and easily here: *http://msdn.microsoft.com/en-us/library/dd554917.aspx*.

Companion Content *Find articles on how to create and use UserForms (dialog boxes) in VBA and how to manage VBA errors, as well as a list of all links provided in this chapter and others, in the online Bonus Content folder available online as part of the companion content for this book, at http://aka.ms/651999/files.*

Before you go, however, the next (and final) chapter in this book provides a similar primer on the basics of using the Office Open XML Formats to edit documents and create custom content.

Index

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z