Join the discussion @ p2p.wrox.com

wrox

Wrox Programmer to Programmer™

WPF
Programmer's Reference

Windows® Presentation Foundation with C# 2010 and .NET 4

Rod Stephens

WPF PROGRAMMER'S REFERENCE

Continues

WPF

PROGRAMMER'S REFERENCE

WPF

PROGRAMMER'S REFERENCE

WINDOWS PRESENTATION FOUNDATION WITH C# 2010 AND .NET 4

Rod Stephens

WILEY

Wiley Publishing, Inc.

WPF Programmer's Reference: Windows Presentation Foundation with C# 2010 and .NET 4

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

Published simultaneously in Canada

ISBN: 978-0-470-47722-9

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

Library of Congress Control Number: 2009942828

ABOUT THE AUTHOR

ROD STEPHENS started out as a mathematician, but while studying at MIT, discovered the joys of programming and has been programming professionally ever since. During his career, he has worked on an eclectic assortment of applications in such fields as telephone switching, billing, repair dispatching, tax processing, wastewater treatment, concert ticket sales, cartography, and training for professional football players.

Rod is a Microsoft Visual Basic Most Valuable Professional (MVP) and ITT adjunct instructor. He has written more than 20 books that have been translated into languages from all over the world, and more than 250 magazine articles covering Visual Basic, C#, Visual Basic for Applications, Delphi, and Java. He is currently a regular contributor to DevX (`www.DevX.com`).

Rod's popular *VB Helper* web site `www.vb-helper.com` receives several million hits per month and contains thousands of pages of tips, tricks, and example code for Visual Basic programmers, as well as example code for this book.

CREDITS

EXECUTIVE EDITOR
Bob Elliott

SENIOR PROJECT EDITOR
Adaobi Obi Tulton

TECHNICAL EDITOR
John Mueller

SENIOR PRODUCTION EDITOR
Debra Banninger

COPY EDITOR
Cate Caffery

EDITORIAL DIRECTOR
Robyn B. Siesky

EDITORIAL MANAGER
Mary Beth Wakefield

MARKETING MANAGER
Ashley Zurcher

PRODUCTION MANAGER
Tim Tate

VICE PRESIDENT AND EXECUTIVE GROUP PUBLISHER
Richard Swadley

VICE PRESIDENT AND EXECUTIVE PUBLISHER
Barry Pruett

ASSOCIATE PUBLISHER
Jim Minatel

PROJECT COORDINATOR, COVER
Lynsey Stanford

COMPOSITOR
James D. Kramer, Happenstance Type-O-Rama

PROOFREADER
Nancy Carrasco

INDEXER
J & J Indexing

COVER DESIGNER
Michael E. Trent

COVER IMAGE
© Ben Blankenburg / istockphoto

ACKNOWLEDGMENTS

THANKS TO Bob Elliott, Adaobi Obi Tulton, Kristin Vorce, Cate Caffrey, and all of the others who worked so hard to make this book possible.

Thanks also to John Mueller for giving me another perspective and the benefit of his extensive expertise. Visit `www.mwt.net/~jmueller` to learn about John's books and to sign up for his free newsletter *.NET Tips, Trends & Technology eXTRA*.

CONTENTS

INTRODUCTION

WINDOWS PRESENTATION FOUNDATION (WPF) is Microsoft's next evolutionary step in user interface (UI) development. While WPF lets you drop controls on forms just as developers have been doing for years, WPF provides a quantum leap beyond what is possible using Windows Forms. WPF lets you use a consistent development model to build applications that run in more environments, on more hardware, using more graphical tools, and providing a more engaging visual experience than is normally possible with Windows Forms.

WPF lets you build stand-alone desktop applications that run as executable on a Windows system. WPF can also build simple web pages, compiled applications that run within a web browser, or Silverlight applications that run in a browser with enhanced security. By using these browser techniques, you can build applications that run just about anywhere, even on UNIX or Macintosh systems!

WPF allows you to build engaging interfaces that are responsive, interactive, and aesthetically pleasing. WPF interfaces can include static documents or documents that rearrange their content as needed, two- and three-dimensional graphics, high-resolution vector graphics that draw lines and curves instead of using bitmaps, animation, audio, and video.

All of the examples shown in this book are available for download in C# and Visual Basic versions on the book's web pages. See the section, "Source Code," later in this chapter for details. The names of the programs are shown in their title bars so it's easy to tell which figures show which programs.

In fact, WPF makes it almost embarrassingly easy to:

- Draw normal controls and simple graphics, as shown in Figure 0-1.

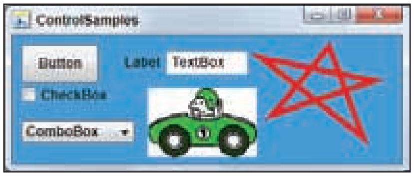


FIGURE 0-1

- Play audio and video files, as shown in Figure 0-2.

FIGURE 0-2

- Add eye-catching graphical effects such as drop shadows and color gradients, as shown in Figure 0-3.

FIGURE 0-3

- Use shared styles to give control similar appearances and skins, as shown in Figure 0-4.

FIGURE 0-4

- Transform objects including shapes, controls, and even video, as shown in Figure 0-5.

FIGURE 0-5

- Display simple animations similar to those provided by Adobe Flash, as shown in Figure 0-6. (OK, I admit I faked this one. Figure 0-6 shows three steps in a WPF animation. Although WPF allows you to show videos easily, this printed book does not.)

FIGURE 0-6

- Create and even animate intricate three-dimensional (3D) graphics, as shown in Figure 0-7.

FIGURE 0-7

- Draw vector graphics that scale without jagged aliasing. The pictures at the top of Figure 0-8 were drawn with vector graphics so they scale smoothly. In contrast, the images at the bottom of Figure 0-8 are scaled views of a bitmap image so they become jagged as they are enlarged.

FIGURE 0-8

- Create and display sophisticated documents that rearrange their contents to make the best use of available space, as shown in Figure 0-9.

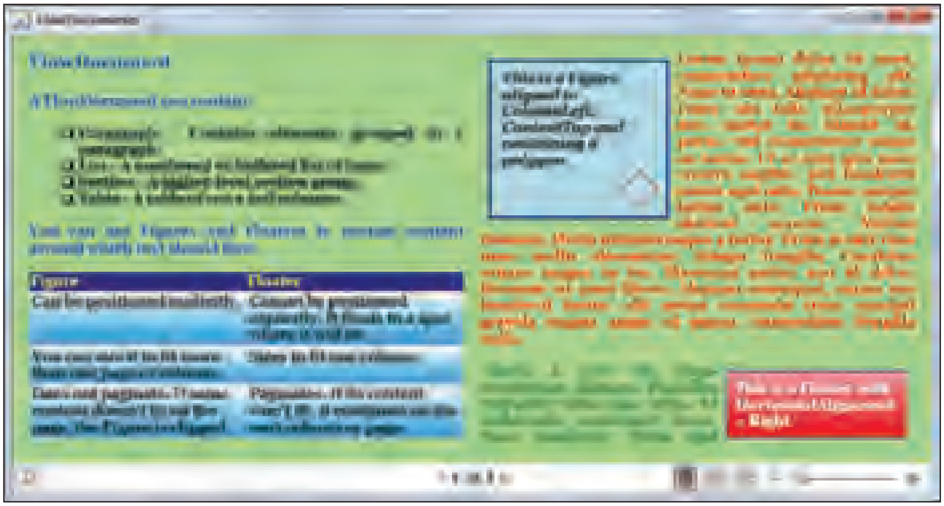


FIGURE 0-9

> *In this introductory chapter, don't worry about how the examples work. For now, focus on the cool and amazing things they can do. You'll see how they work in later chapters.*

Unfortunately, to use WPF, you must overcome a rather steep learning curve. Many of the fundamental concepts in modern Windows UI design are different from those used by WPF. Concepts as basic as how events are handled and how program code is attached to the user interface are different in WPF.

Many of these new concepts are unified, elegant, and simple. Ideas such as declaratively building an interface in Extensible Markup Language (XAML — pronounced *zammel*), property value inheritance, and allowing controls to contain any type of content make a simple yet powerful programming paradigm.

Unfortunately, shortcuts, exceptions, and inconsistencies built into WPF make it much harder to understand and use than you might hope from its elegant underlying philosophy. Depending on how properties are used, developers must use several different XAML notations, property value inheritance is trumped by performance issues in some cases, and some controls can only contain certain other kinds of controls.

This book provides an introduction to WPF development. It explains fundamental WPF concepts so you can start building applications quickly and easily. As it progresses, the book covers more complex topics, explaining how to handle the exceptions and shortcuts built into WPF.

The book finishes with a series of appendixes summarizing WPF concepts and syntax for easy reference. You can use these appendixes to refresh your memory of WPF's intricate and sometimes counterintuitive syntax.

Of course, many future applications will be written without WPF. Many will be written using clunky old technologies such as command-line interfaces and pure HTML. Others will be written with competing technologies like Java and Flash.

Finally, some developers will continue using good old familiar Windows Forms in C# or Visual Basic. There's a lot to be said for sticking with what you know, but the future of development in the Windows environment is WPF. Soon the beauty, grace, and level of responsiveness provided by WPF will become de rigueur, and if you're still using Windows Forms, you'll be left behind.

WHO THIS BOOK IS FOR

This book is for anyone who wants to use or better understand WPF. In particular, it is intended for:

- Specialized UI designers who build user interfaces but don't write program code
- Programmers who write the code behind the user interface
- Jack-of-all-trades developers who create user interfaces and write the code behind them
- Web developers who want to learn how to use WPF in loose web pages, browser applications, and Silverlight applications
- Project managers who want a better understanding of what WPF is and what kinds of features it can provide

For decades, good developers have separated UI construction from the code behind the user interface. Keeping the two separate makes it easier to distribute work among different developers and makes it easier to build each piece separately. WPF continues this philosophy by making the separation between the user interface and the code behind it more distinct than ever before.

In fact, in Microsoft's original vision, specialized graphic designers built the user interface, and programmers added the code behind it completely separately.

While many development projects cannot afford separate graphic designers and programmers, it's still worthwhile to keep these two tasks separate. This book squarely addresses those who perform either of those tasks.

This book provides an introduction to WPF and does not require that you have any experience with it. In fact, it doesn't require that you have any previous programming or UI design experience.

> *I don't want to receive a bunch of flaming e-mails complaining that some of the material is too basic, so I'm warning you right now! If you're mortally offended by introductory material, you're welcome to skim the first few chapters and move on to the more advanced material.*

Although this book does not require previous programming experience, it covers a lot of material and does get into some rather advanced topics. By the time you finish reading it, you should have learned a lot no matter how experienced you are at the start.

WHAT THIS BOOK COVERS (AND WHAT IT DOESN'T)

This book explains WPF development. It explains how to build user interfaces by using Microsoft's Expression Blend tool, Visual Studio, and the XAML programming language. It also explains how to use Visual Studio to attach code to the user interface.

WPF is a very flexible tool, and you can use it to make amazing user interfaces. Unfortunately, it is also often complicated, occasionally confusing, and sometimes downright intractable. You can use it to perform remarkable feats of UI sleight-of-hand, but doing so can be a heroic adventure in experimentation and web browsing.

Such deeds of development heroism fly in the face of Microsoft's intent that graphic designers build user interfaces that programmers then attach to code. Perhaps I'm hanging out with the wrong crowd, but the graphic designers that I've met did not have the skills or interest to spend their time constructing elaborate UI animations. Instead, they wanted to focus on the interface's appearance and usability.

This book's philosophy is that the user interface is a front end to the application, not the application itself. It should not take six years of experience and a PhD in WPF to build a data entry form.

If it takes a huge assortment of sneaky tricks to make a program perform some esoteric stunt, this book doesn't cover it. For more complex situations, the book will freely jump between the user interface and the code behind it. For example, if a particular animation is hard to control with pure WPF but easy to control using code behind the scenes, I'll opt for option two every time.

This book also doesn't cover programming the code behind the interface. It demonstrates some of that code so you can learn how to write your own code, but it doesn't cover C#, Visual Basic, or any other programming language in detail.

HOW THIS BOOK IS STRUCTURED

The chapters in this book are generally arranged from the most basic in the beginning to the more advanced at the end. They start with fundamentals such as adding controls to windows and selecting the kinds of controls to use. Later chapters cover more advanced topics such as animation, transformations, and 3D graphics. The appendixes provide a handy reference for controls and other objects, and XAML syntax.

The book will probably make the most sense if you read the chapters in order, but you can skip around a bit if you need information on a particular topic. For example, after you read the first few chapters and know how to build simple WPF applications, you might want to skip ahead and read a bit more about styles or transformations.

If you have previous development experience, particularly with Expression Blend or Visual Studio, you may want to skim the earliest chapters.

If you know that you will not be using Expression Blend or Visual Studio, you may want to skip the corresponding chapters entirely. For example, if you know that you will be using Visual Studio and not Expression Blend, then you may want to skip Chapter 3.

- **Chapter 1: WPF Overview —** Chapter 1 covers basic WPF concepts. It explains WPF's advantages, how WPF is layered on top of DirectX, and how WPF separates UI design from the code behind it. It also describes the different kinds of WPF projects (stand-alone, XBAP, library) and explains how Page, Frame, and PageFunction projects work in general terms.
- **Chapter 2: WPF in Visual Studio —** Chapter 2 explains how to build WPF projects with Visual Studio. It tells how to build a simple user interface and how to connect interface elements with the code behind them. This chapter explains how to set control properties and how to edit XAML code in Visual Studio. It does not explain WPF controls in great depth because they are covered in later chapters.
- **Chapter 3: Expression Blend —** Chapter 3 explains how to build WPF projects with Expression Blend. It tells how to edit XAML code in Expression Blend and how to link to Visual Studio to add code behind the user interface.
- **Chapter 4: Common Properties —** Chapter 4 describes some properties that are common to many WPF controls. These properties determine basic control features such as color, size, and position.
- **Chapter 5: Content Controls —** Chapter 5 describes WPF's controls that are intended to display content (as opposed to the controls described in the following chapters). These include

such controls as `Label`, `GroupBox`, `ListBox`, and `Image`. This chapter describes the purpose of each control and summarizes its most important properties and behaviors.

- **Chapter 6: Layout Controls** — Chapter 6 describes WPF's controls that are intended to arrange other controls. These include such controls as `Grid`, `DockPanel`, `StackPanel`, and `WrapPanel`. This chapter describes the purpose of each control and summarizes its most important properties and behaviors.
- **Chapter 7: User Interaction Controls** — Chapter 7 describes WPF's controls that are intended to allow the user to control the application. These include such controls as `Button`, `RadioButton`, `TextBox`, and `Slider`. This chapter describes the purpose of each control and summarizes its most important properties and behaviors.
- **Chapter 8: Two-Dimensional Drawing Controls** — Chapter 8 describes WPF objects that perform two-dimensional (2D) drawing. These include `Line`, `Ellipse`, `Rectangle`, `Polygon`, `Polyline`, and `Path`. This chapter also explains the `Path` mini-language and geometries, which can contain multiple drawing objects.
- **Chapter 9: Properties** — Chapter 9 explains WPF properties in detail. Whereas the earlier chapters use properties to provide simple examples, this chapter describes properties in greater depth. It explains basic properties entered as simple text, properties that can be entered as multiple text values, properties that are objects, dependency properties, and attached properties.
- **Chapter 10: Pens and Brushes** — Chapter 10 describes the `pen` and `brush` objects that you can use to determine the graphical appearance of WPF objects. In addition to simple single-color pens and brushes, this chapter describes more complex objects such as dashed pens, gradient brushes, and image brushes.
- **Chapter 11: Events and Code-Behind** — Chapter 11 explains routed events, tunneling (preview) events, bubbling events, and attached events. These different kinds of events allow you to attach a user interface that was created with WPF to the code behind the scenes.
- **Chapter 12: Resources** — Chapter 12 explains WPF resources. It tells how to use static and dynamic resources in XAML code.
- **Chapter 13: Styles and Property Triggers** — Chapter 13 explains styles and property triggers. It tells how to use styles, usually stored as resources, to give objects a consistent appearance. (For an example, see Figure 0-4.) It also explains property triggers, which are often defined in styles, to change a control's appearance when a property value changes.
- **Chapter 14: Event Triggers and Animation** — Chapter 14 explains event triggers and the animations they can run. It explains storyboards and timelines that let WPF applications perform animations with surprisingly little effort.
- **Chapter 15: Templates** — Chapter 15 describes control templates. It explains how you can use templates to change the appearance and behavior of predefined controls. It also tells how to use `ItemsPresenter` and `ContentPresenter` objects to change the way lists and menus work.

- **Chapter 16: Themes and Skins —** Chapter 16 explains how to use resource dictionaries to provide application themes and skins. By changing a single resource dictionary, you can make a WPF application change the appearance of some or all of its graphical components.
- **Chapter 17: Printing —** Chapter 17 explains how a WPF application can display print previews and how it can print documents.
- **Chapter 18: Data Binding —** Chapter 18 explains how to bind control properties to data. It explains basic data binding and also shows how to use `DataTemplate` objects to provide more complicated data display.
- **Chapter 19: Commanding —** Chapter 19 explains commanding, a tool that lets you associate controls to command objects that represent the actions they should perform. For standard operations such as copy, cut, and paste, these objects make providing consistent features much easier.
- **Chapter 20: Transformations and Effects —** Chapter 20 explains rotation, scaling, and other transformations that you can use to rotate, stretch, and otherwise change the appearance of WPF objects. It also describes special graphical effects such as blur, drop shadow, and glow. (For examples of drop shadow, see Figure 0-3.)
- **Chapter 21: Documents —** Chapter 21 explains the document objects provided by WPF. It explains fixed documents, which display items in the precise positions where you place them, and flow documents, which can rearrange objects much as a web browser does to take advantage of the available space. (For an example, see Figure 0-9.)
- **Chapter 22: Navigation-Based Applications —** Chapter 22 describes programs that use special navigation controls to manage how the user moves through the application. It explains how to build Page, Frame, and PageFunction projects.
- **Chapter 23: Three-Dimensional Drawing —** Chapter 23 explains how to display and control 3D drawings in WPF. Although it is possible to build these objects in XAML code, it is often easier to generate 3D scenes programmatically, so this chapter provides both XAML examples and examples that use C# code to build scenes.
- **Chapter 24: Silverlight —** Chapter 24 briefly introduces Silverlight, WPF's web-oriented cousin. Although Silverlight has some restrictions that WPF doesn't, it lets you build applications that can run in a web browser on any operating system.
- **Appendix A: Common Properties —** Appendix A summarizes properties that are shared by many WPF controls.
- **Appendix B: Content Controls —** Appendix B summarizes the most useful properties and behaviors of WPF controls that are intended to display content such as `Label`, `ListBox`, and `Image`.
- **Appendix C: Layout Controls —** Appendix C summarizes the most useful properties and behaviors of WPF controls that are intended to contain and arrange other controls such as `Grid`, `StackPanel`, and `WrapPanel`.

- **Appendix D: User Interaction Controls —** Appendix D summarizes the most useful properties and behaviors of WPF controls that let the user control the application such as `Button`, `RadioButton`, and `TextBox`.
- **Appendix E: MediaElement Control —** Appendix E summarizes the `MediaElement` control.
- **Appendix F: Pens —** Appendix F summarizes `Pen` classes and properties that an application can use to determine the graphical appearance of line features.
- **Appendix G: Brushes —** Appendix G summarizes `Brush` classes and properties that an application can use to determine the graphical appearance of filled areas.
- **Appendix H: Path Mini-Language —** Appendix H summarizes the `Path` mini-language that you can use to draw shapes with the `Path` object. Complicated paths are much easier to build with the `Path` mini-language rather than using objects contained inside a Path.
- **Appendix I: XPath —** Appendix I summarizes the XPath expressions that you can use to bind XML data to WPF controls.
- **Appendix J: Data Binding —** Appendix J summarizes data binding techniques you can use to bind property values to values provided by different objects such as other WPF controls or objects created by code-behind.
- **Appendix K: Commanding Classes —** Appendix K summarizes the most useful predefined commanding classes.
- **Appendix L: BitmapEffects —** Appendix L provides an example demonstrating the different `BitmapEffect` classes.
- **Appendix M: Styles —** Appendix M summarizes the syntax for creating named and unnamed styles.
- **Appendix N: Templates —** Appendix N provides example templates for the `Label`, `CheckBox`, `RadioButton`, `ProgressBar`, `ScrollBar`, and `Button` controls.
- **Appendix O: Triggers and Animation —** Appendix O summarizes the syntax for creating event triggers and the animations that they control.
- **Appendix P: Index of Example Programs —** Appendix P lists this book's more than 250 example programs, all of which are available for download on the book's web site. It gives a brief description of each program, tells where it is shown in a figure (if it is), and tells which page has more information. You can use this list to find examples that may help with specific problems.

WHAT YOU NEED TO USE THIS BOOK

There are several ways you can build and view WPF applications and XAML files, and each has different requirements.

If you're a devout minimalist, all you really need to install is the latest version of the .NET Framework and a WPF-enabled browser such as one of the newer versions of Internet Explorer or Firefox.

At least some XAML files should work with .NET Framework 3.0, Internet Explorer 6, and Firefox 2, but I recommend installing the latest versions. Currently, that's .NET Framework 3.5 with Service Pack 2, Internet Explorer 8, and Firefox 3. Don't forget to look for other Service Packs for all three products!

> *The current release of Expression Blend doesn't understand the .NET Framework version 4.0 so, for now at least, you may want to stick with version 3.5 if you plan to use Expression Blend.*

Note that Windows Vista comes with the .NET Framework 3.0 preinstalled, so, if you're running Vista, you may be all set. You can install the .NET Framework version 3 and later in Windows XP, although not in earlier versions of Windows. As far as I know, you cannot run WPF in UNIX or Macintosh operating systems, although in theory that could change some day.

In this bare-bones Framework-and-browser environment, you can create XAML files in a text editor and then look at them in your browser.

If you want to attach program code to your WPF user interfaces, or if you want to build compiled WPF applications or XAML Browser Applications (XBAP — pronounced *ex-bap*), you'll need a programming environment that can write that code. The easiest solution is to install Visual Studio and write application code in C# or Visual Basic.

> *The programming code examples shown in this book are written in C# and Visual Basic; versions of the programs are available for download on the book's web site.*

Visual Studio makes attaching code to the WPF user interface practically trivial. The interactive Window Designer is missing a lot of functionality, so you often need to write XAML code to get the job done, but the Visual Studio Express Editions come at the unbeatable price of $0. Download the C# or Visual Basic Express Editions at `www.microsoft.com/express`.

If you want a more graphic designer–oriented development tool, you can install Expression Blend. It won't help you build code to attach to the user interface, but it does have some nice features that are missing from Visual Studio. Its support for interactively manipulating different WPF objects is more complete than that provided by Visual Studio, and it provides fairly simple editors that let you build simple triggers and animations interactively.

Unfortunately, Expression Blend is far from free. At the time of writing, it's priced at $499, although you can get a 30-day free trial. You can learn more about Expression Blend at `www.microsoft.com/expression/products/Overview.aspx?key=blend`.

Of course, the best configuration for building WPF applications includes both Visual Studio and Expression Blend. Neither of these tools is perfect, but they each cover some of the other's shortcomings.

I also often find that one gives a mysterious error message, while the other is easy to understand — so switching back and forth sometimes helps with debugging.

To summarize, the best WPF development environment includes the latest .NET Framework and a WPF-enabled web browser, together with the latest versions of Visual Studio and Expression Blend, all installed in Windows Vista or Windows XP. If you want to save some money, you can do without Expression Blend, but then you'll do a lot more XAML coding by hand.

CONVENTIONS

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

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

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

As for styles in the text:

- We *highlight* new terms and important words when we introduce them.
- We show keyboard strokes like this: [Ctrl]+A.
- We show URLs and code within the text in monofont type like so: `persistence.properties`.
- We present code like this:

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

The Code Editors in Visual Studio and Expression Blend provide a rich color scheme to indicate various parts of code syntax. That's a great tool to help you learn language features in the editor and to help prevent mistakes as you code.

To take advantage of the editors' colors, the code listings in this book are colorized using colors similar to those you would see on screen in Visual Studio or Expression Blend. In order to optimize print clarity, some colors have a slightly different hue in print than what you see on screen. But all of the colors for the code in this book should be close enough to the default Visual Studio colors to give you an accurate representation of the colors.

SOURCE CODE

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. Many of the examples show only the code that is relevant to the current topic and may be missing some of the extra details that you need to make the example work properly.

All of the source code used in this book is available for download at `www.wrox.com`. Once at the site, simply locate the book's title (either by using the Search box or by using one of the title lists) and click on the "Download Code" link on the book's detail page to obtain all the source code for the book.

> *Because many books have similar titles, you may find it easiest to search by ISBN; this book's ISBN is 978-0-470-47722-9.*

Once you download the code, just decompress it with your favorite compression tool. Alternatively, you can go to the main Wrox code download page at `www.wrox.com/dynamic/books/download.aspx` to see the code available for this book and all other Wrox books.

You can also download the book's source code from its web page on my VB Helper web site, `www.vb-helper.com/wpf.htm`. That page allows you to download all of the book's code in one big chunk, the C# or Visual Basic versions separately, or the code for individual chapters.

ERRATA

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

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

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

P2P.WROX.COM

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

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

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

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

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

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

Using the P2P forums allows other readers to benefit from your questions and any answers they generate. I monitor my book's forums and respond whenever I can help.

If you have other comments, suggestions, or questions that you don't want to post to the forums, feel free to e-mail me at `RodStephens@vb-helper.com` with your comments, suggestion, or questions. I can't promise to solve every problem but I'll try to help you out if I can.

1

WPF Overview

This chapter explains fundamental Windows Presentation Foundation (WPF) concepts. Normally, it's the glaringly obvious chapter that you skip to get to the good stuff. If this were a cookbook, this would be where I explain *food* and tell you why it's important ("so you don't starve").

In this case, however, I encourage you to at least skim this chapter before plunging ahead. Many parts of WPF are confusing and seemingly inconsistent. This chapter gives some useful background on what WPF is (that question has caused more confusion than you might imagine), WPF's goals, and the underlying architecture used by WPF.

These tidbits of information will give you some useful perspective for understanding WPF's quirks and idiosyncrasies. For example, this information will let you say, "Oh, WPF does it that way because Direct3D does it that way" or "I'll bet this weird behavior was provided to save me a few keystrokes of typing."

In addition to this background, this chapter describes the basic types of WPF projects.

Finally, this chapter can help you understand what's contained in the later chapters. This chapter briefly defines resources, styles, control templates, and other terms that are described more completely in later chapters. A quick introduction to those terms now will help you know which chapters to read later.

WPF IN A NUTSHELL

WPF has been around for quite a while now, but there are still plenty of people out there who don't really know what it is. I've heard people claim it's everything from a set of controls to a "Vista thing" to XAML.

In fact, there's a kernel of truth in each of these attitudes. WPF does include a new set of controls that largely replace the Windows Forms controls. The libraries you need to run WPF applications are installed by default in Vista and Windows 7, so it is sort of a Vista thing, although you can also run WPF applications in Windows XP and certainly in future versions of Windows (and perhaps even UNIX some day). WPF applications can use XAML to build interfaces, and XAML is all you really need to write loose web pages; but there's a lot more to WPF than just XAML.

As far as WPF's importance and usefulness go, opinions range the gamut from "I don't have time for jiggling buttons and spinning labels" to "It's the wave of the future, and every new application will be written in WPF by the end of the year" (although that was last year, so perhaps this latter attitude isn't quite correct).

Again, the truth lies somewhere between these two extremes. You certainly can abuse WPF to build completely unusable interfaces full of bouncing buttons, skewed video, stretched labels, garish colors, and rotating three-dimensional (3D) graphics. You can add animation to the controls until the interface behaves more like a video game than a business application.

Figure 1-1 shows the Clutter example program displaying a (faked) series of rotated images as an invoice spins into view. This program demonstrates some interesting techniques but goes way overboard with gratuitous animation, displaying a spinning invoice area, animated buttons, and sound effects. If you think it's ugly in this book, you should see how annoying it is when you run it!

FOCUS ON WHAT, NOT HOW

In this overview chapter, don't worry about how the examples work. For now, focus on the cool and amazing things they can do. You'll see how they work in later chapters.

FIGURE 1-1

AMPLE EXAMPLES

All of the example programs that are available for download on the book's web site have titles that match their names. For example, Figure 1-1 shows the Clutter program and its title is "Clutter."

If you see a picture in the book and you want to find the corresponding example program, download the programs for that chapter and look for a program with a name matching the title shown in the picture. For information about downloading the examples, see the section "Source Code" in the Introduction.

If you use restraint and good design principles, you can use WPF to make user interfaces that are more visually appealing and inviting. You can use animation to hide and display information to reduce clutter while giving the user hints about where the data has gone so it's easy to find later.

ANIMATION OVERLOAD

Before you go overboard with animation, ask yourself, "Does this animation serve a purpose?" If the purpose is to hide a picture while showing where it is going so the user can find it again later — great. If the purpose is to be cool and make an invoice fly out of a file cabinet icon while growing, spinning, and changing opacity — think again. The first time, the user might think this is cool; but by the second or third time, the user will find it annoying; and by the end of the day, the user will be seasick.

It may not be true that all new applications will use WPF by the end of the year, but you should consider using WPF for new development. While getting the most out of WPF takes a lot of study and practice, it's easy enough to use WPF controls instead of the corresponding Windows Forms controls in most cases. You may not stretch WPF to its limits, but you can take advantage of some of WPF's new features without a lot of work.

Of course, some applications will probably never need WPF. Some programs run most naturally as automatic services or from the command line and don't need graphical user interfaces at all.

What Is WPF?

So, what exactly is WPF? I've heard it described as a library, framework, subsystem, set of controls, language, and programming model.

Probably the easiest way to understand WPF is to think of it as an assortment of objects that make it easier to build cool user interfaces. Those objects include a new set of controls, some replacing your favorite Windows Forms controls (such as `Label`, `TextBox`, `Button`, `Slider`) and others providing new features (such as `Expander`, `FlowDocument`, and `ViewBox`).

WPF also includes an abundance of new objects to manage animation, resources, events, styles, templates, and other new WPF features.

Your application uses some combination of these objects to build a user interface.

What Is XAML?

XAML (pronounced *zammel*) stands for "eXtensible Application Markup Language." It is an extension of *XML* (eXtensible Markup Language). Microsoft invented XAML to represent WPF user interfaces in a static language much as HTML represents the contents of a web page. It defines special tokens to represent windows, controls, resources, styles, and other WPF objects.

A program can use a file containing XAML code to load a user interface. For example, a web browser can load a file containing XAML code and display the user interface (UI) it defines. If you use Expression Blend or Visual Studio to create a WPF application, the application automatically loads the project's XAML for you so you don't need to add code to do that yourself.

NO XAML REQUIRED

Note that XAML is not required to make a WPF application. A program could build all of the objects that it needs by using code. For example, instead of placing a `Label` object in a XAML file, the program could create an instance of the `Label` class and use that instead. XAML is just there for your convenience. It makes it easier to build and store interfaces.

All of the usual XML rules apply to XAML files. In particular, XAML files must have a single root element that contains all of the other elements in the file. What element you use as the root element depends on the type of project you are building.

For example, in a compiled application, the root element is a `Window` that represents the window displayed on the desktop. In contrast, a loose XAML page is displayed in a web browser, so the browser plays the role of the window. In that case, the root element is typically some container control such as a `Grid` or `StackPanel` that can hold all of the other elements.

CONTROL SNEAK PEEK

Later chapters describe these controls in detail, but for now, know that a `Grid` arranges controls in rows and columns, and a `StackPanel` arranges controls in a single row either vertically or horizontally.

Each opening element must have a corresponding closing element with the same name but beginning with a slash. For example, the following code snippet defines a `StackPanel`:

```
<StackPanel>
</StackPanel>
```

If an element doesn't need to contain any other elements, you can use a special shorthand and end the opening element with a slash instead of a separate closing element. The following snippet shows an `Image` object. It doesn't contain any other items, so it uses the shorthand notation.

```
<Image Margin="10" Width="75" Height="75" Source="Volleyball.jpg"/>
```

The preceding snippet also demonstrates attributes. A XAML *attribute* is a value contained inside an item's opening tag. In this snippet, the `Image` object has attributes `Margin`, `Width`, `Height`, and `Source` with values `10`, `75`, `75`, and `Volleyball.jpg`.

XAML elements must be properly nested to show which WPF objects contain other objects. The following XAML code shows a `Window` that contains a horizontal `StackPanel` that holds several other vertical `StackPanel` objects, each holding an `Image` and a `Label`.

Available for download on Wrox.com

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="Window1"
    x:Name="Window"
    Title="SimpleCritters"
    Width="557" Height="156"
    FontSize="22" FontWeight="Bold" FontFamily="Comic Sans MS"
    >
    <StackPanel Orientation="Horizontal" Margin="5">
        <StackPanel>
            <Image Margin="5" Height="50" Source="Frog.jpg"/>
            <Label Margin="5" Content="Frog"/>
        </StackPanel>
        <StackPanel>
            <Image Margin="5" Height="50" Source="Butterfly.jpg"/>
            <Label Margin="5" Content="Butterfly"/>
        </StackPanel>
        <StackPanel>
            <Image Margin="5" Height="50" Source="Shark.jpg"/>
            <Label Margin="5" Content="Shark"/>
        </StackPanel>
        <StackPanel>
            <Image Margin="5" Height="50" Source="Tiger.jpg"/>
            <Label Margin="5" Content="Tiger"/>
        </StackPanel>
        <StackPanel>
            <Image Margin="5" Height="50" Source="Platypus.jpg"/>
            <Label Margin="5" Content="Platypus"/>
        </StackPanel>
    </StackPanel>
</Window>
```

SimpleCritters

Figure 1-2 shows the result.

FIGURE 1-2

Figure 1-3 shows the program with its `StackPanels` highlighted so they are easy to see. In this figure, you can see how the outer `StackPanel` arranges the inner `StackPanels` horizontally and how the inner `StackPanels` arrange their `Images` and `Labels` vertically.

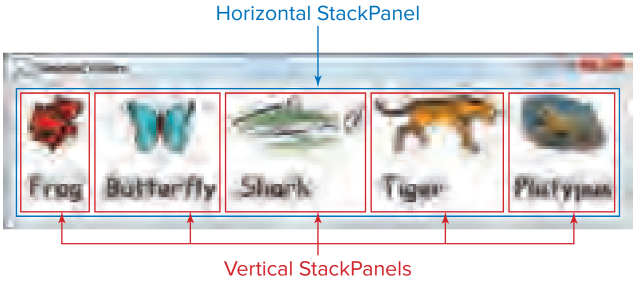


FIGURE 1-3

Object Trees

The controls that make up a user interface such as the one shown in Figure 1-2 form a natural hierarchy with some controls containing others, which may then contain others. Figure 1-4 shows this program's control hierarchy graphically.

WPF has two concepts of trees that represent structure similar to the one shown in Figure 1-4. Normally, you don't need to worry explicitly about these, but knowing what they are can make it a bit easier to understand some of the online documentation.

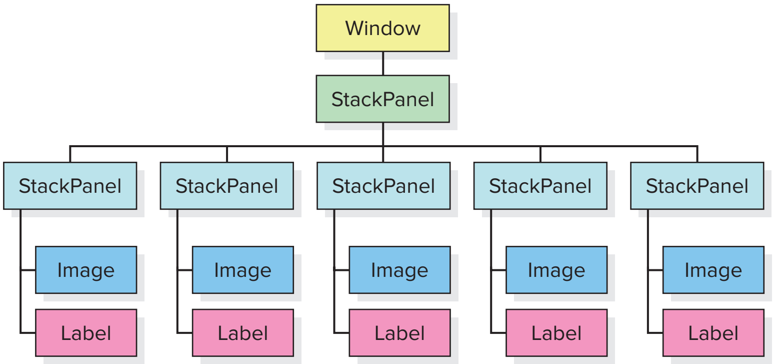


FIGURE 1-4

Logical Tree

The *logical tree* is defined by the content relationships among the objects in the interface. That includes controls contained within other controls (such as a series of `Image` controls contained within a `StackPanel`) and simple content (such as the `string` contained in a `Label`).

It also includes some objects that you may not think of as separate entities. For example, if you add items to a `ListBox`, those items are automatically added as `ListBoxItem` objects inside the `ListBox`. If you use the Expression Blend or Visual Studio editors to add the items, you may not think of them as separate objects, but they are, as far as the logical tree is concerned.

Visual Tree

The second WPF tree is the visual tree. The *visual tree* represents the structure of visual objects including the components that define them. For example, a scrollbar includes a draggable thumb, two arrow buttons at the ends, and two clickable areas between the thumb and the arrows. Each of those pieces is a separate object that is wrapped up inside a scrollbar, and each of the pieces is part of the visual tree.

Why should you care about the logical and visual trees? First, controls tend to inherit property values according to their positions in the logical tree. For example, the preceding XAML code used the following attributes to set the main `Window`'s `FontSize`, `FontWeight`, and `FontFamily` properties.

```
FontSize="22" FontWeight="Bold" FontFamily="Comic Sans MS"
```

These properties are inherited throughout the logical tree so the `Labels` at the bottom of the tree all use these font values.

The second reason you should care about these trees is that events tend to follow the visual tree. For example, when you click on an arrow in a scrollbar, you don't really want to have to deal with that arrow's events. Instead you can let the event propagate up through the visual tree to the scrollbar, and you can catch it there.

Non-Treelike Structure

Because the controls that make up a user interface such as this one naturally form a containment hierarchy, they map fairly reasonably into XML. Unfortunately some of the things that make up an interface don't fit quite as neatly in a tree structure.

For example, Figure 1-5 shows a more complicated version of the previous user interface.

FIGURE 1-5

In this version, all of the `Image` controls have drop shadows. You could give each `Image` its own drop shadow, but that would mean duplicating code, which would make maintaining the program harder. For example, if you later decided to remove the drop shadows or use some other bitmap effect (perhaps to put a glow around the images), you would have to update each control separately.

Rather than repeating the drop shadow code, you can define a `Style` and apply it to each `Image`. Now to change every `Image`, you only need to change the `Style`.

Similarly, this example defines a style for its `Label` controls that gives them drop shadows, makes them yellow, and centers their text.

Unfortunately, each of these styles applies to several different controls in the logical hierarchy, and that messes up the nice neat tree structure shown in Figure 1-4.

XAML still uses a hierarchical XML structure anyway, even though some scenarios such as this one require objects to refer to others in a non-hierarchical way. It works, but, as you'll see in later chapters, it does complicate the XAML syntax considerably.

What Is Silverlight?

Silverlight (formerly known as *WPF/e*, where the *e* stands for "everywhere") is a restricted version of WPF designed to run safely in a browser while still providing a rich user interface. It runs on most major browsers including Mozilla Firefox, Microsoft Internet Explorer, and Apple Safari.

To minimize library size and to work safely in the browser, Silverlight does not provide all of the features that are included in WPF. Some features are missing, while others are provided in a restricted way. While there are some differences between WPF and Silverlight, the basics are the same; so much of what you learn about WPF applies to Silverlight as well.

There are several differences between WPF and Silverlight, but so far Microsoft has not published an authoritative list. Here are some of the restrictions in Silverlight:

> **DEFERRED UNDERSTANDING**
>
> Some of these features may not make much sense to you now. But they will become clearer, I hope, as you read through the book. For now, just be aware that Silverlight doesn't do everything that WPF does.

- Once assigned, you cannot change a control's style.
- You must explicitly assign each control's style. You cannot create a style that applies to every control of a given type (e.g., buttons).
- One style cannot inherit from another.
- You cannot put triggers in styles and templates.
- You cannot use dynamic resources. All resources are static.
- Silverlight has more restrictive data binding.
- Access to Windows API functions is limited.
- You cannot use commands in Silverlight.
- Silverlight doesn't support 3D graphics and graphics hardware.
- Silverlight doesn't include preview (tunneling) events.

This list will change over time (it may even be outdated by the time you read this). Microsoft is trying to include as many WPF features as possible in Silverlight, while still keeping the necessary libraries as small as possible.

This isn't a Silverlight book, so it doesn't cover Silverlight in any detail. You can learn more about Silverlight on these web pages:

- *Silverlight Overview:* `msdn.microsoft.com/bb404700(VS.95).aspx`
- *Silverlight Home Page:* `www.microsoft.com/Silverlight`
- *Silverlight FAQ:* `www.microsoft.com/silverlight/resources/faq/default.aspx`
- *Differences between WPF and Silverlight:* `msdn.microsoft.com/cc903925(VS.95).aspx`

In general, you should use WPF if you plan to build a high-powered desktop system that needs extensive access to the host computer. You should use Silverlight if you don't need as much access to the host computer and you want your application to run easily in different web browsers on different operating systems.

PROJECT TYPES

WPF lets you build three main kinds of applications: stand-alone, XAML Browser Applications, and loose XAML pages.

A *stand-alone application* is compiled and runs locally on the user's computer much as any stand-alone application does. This type of application runs with full trust and has full access to the computer. It can read and write files, modify the System Registry, and do just about anything else that you can do from a C# or Visual Basic program.

A XAML Browser Application (*XBAP* — pronounced *ex-bap*) is a compiled application that runs within a web browser. For security purposes, it runs within the Internet Zone so it doesn't have full trust and cannot access all of the parts of the computer that are available to a stand-alone application. XBAPs can only run in browsers that support them (currently Internet Explorer and Firefox) and require that the .NET Framework version 3 or later be installed on the user's computer.

Loose XAML pages are simply XAML files displayed in a web browser. They can be viewed by any web browser that understands XAML. Loose XAML pages do not require the .NET Framework to be installed on the user's computer, so they can run on operating systems that cannot install the .NET Framework, such as Macintosh and UNIX systems. Loose XAML pages cannot use script, C#, or Visual Basic code, however. They can display interesting graphics and let the user manipulate the display through XAML animations, but they don't have the power of the other kinds of applications.

In addition to these different types of applications, WPF provides several different navigation models for stand-alone applications and XBAPs. First, they can provide navigational tools similar to those used by Windows Forms applications. Buttons, links, and other code-based mechanisms can display other pages and windows.

WPF also provides browser-style navigation that lets the user go forward and backward through a history of previously visited windows. The `Frame` and `NavigationWindow` classes can provide this type of navigation much as a web browser does.

The `PageFunction` class also supports a special form of navigation. It allows one page to treat another as if it were calling a function. The first page can *call* the second, passing it input parameters. When the called page finishes, the first page receives a "return result" from the called page.

Later chapters have more to say about using these different forms of navigation.

GOALS AND BENEFITS

WPF has several important goals including:

- Better use of graphics hardware
- Property binding to provide animation
- Property inheritance
- Styles
- Templates
- Consistent control containment
- Separate user interface and code-behind
- New controls
- Declarative programming

The following sections describe these goals and how they are achieved by WPF in greater detail.

Better Use of Graphics Hardware

Windows Forms controls are built using the GDI (graphics device interface) and GDI+ (.NET's version of GDI) programming interfaces. These native (non-.NET) APIs provide many flexible and powerful features for drawing graphics, but they have become a bit outdated. In particular, GDI and GDI+ do not take advantage of the tremendous power available in modern computer graphics hardware. Even a fairly inexpensive modern desktop system has vastly more graphical power than the computers that were available when GDI was created.

WPF is based on the more recent DirectX library rather than GDI or GDI+. *DirectX* is a library of tools that provide high-performance access to graphic and multimedia hardware.

By using DirectX, WPF can draw objects more quickly and flexibly than before. DirectX also provides a few secondary benefits practically for free including better multimedia support, transformations, 3D graphics, retained-mode drawing, and high-resolution vector graphics.

Better Multimedia Support

DirectX includes routines that play multimedia files efficiently. WPF has taken advantage of some of those routines to make displaying media files quick and easy.

For example, the following XAML snippet makes all of the buttons in a window play the sound file speech_on.wav when clicked:

```
<EventTrigger RoutedEvent="ButtonBase.Click" >
    <SoundPlayerAction Source="speech_on.wav"/>
</EventTrigger>
```

The `MediaElement` control plays video files almost as easily. In the XAML code, you can set the control's `Source` property to the file that you want to play. At run time, the control provides simple methods and properties such as `Play`, `Pause`, `Stop`, and `Position` to control the video playback.

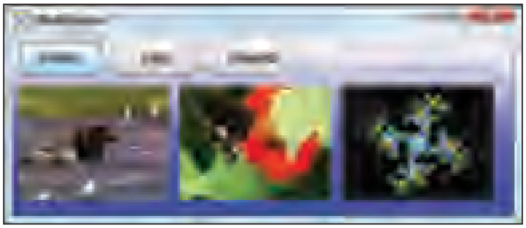


FIGURE 1-6

Figure 1-6 shows a small application playing three video files simultaneously. The buttons use `MediaElement` methods and properties to control the video. The program also uses the previous XAML snippet to play a sound whenever you click a button.

Transformations

DirectX provides methods that make it easy to move, stretch, skew, and rotate graphics. WPF gives you access to those methods so you can easily transform any graphics that you draw. In fact, you can also apply those same transformations to objects that WPF draws including controls such as `Labels`, `TextBoxes`, and `Buttons`.

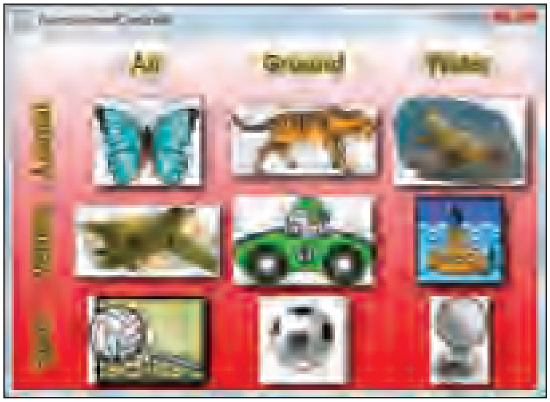


FIGURE 1-7

Figure 1-7 shows a program that displays `Labels` rotated 90 degrees on the left.

You can even transform a `MediaElement`, and it will display video moved, stretched, skewed, and rotated! Figure 1-8 shows a program displaying three videos in `MediaElements` that have been rotated, skewed, and stretched, respectively. The three `Buttons` at the top are also skewed.

It's hard to imagine a business application where you would need to display rotated and stretched video, but using WPF, you could do it if you really had to.

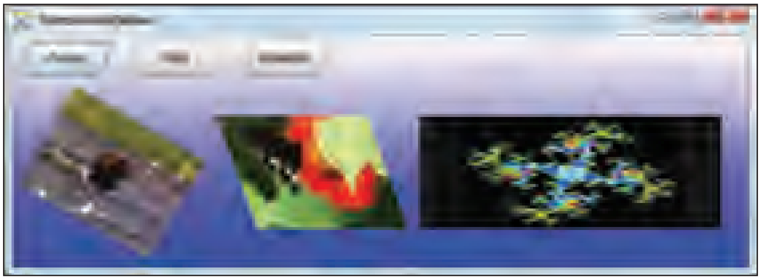


FIGURE 1-8

3D Graphics

Many modern computers include graphics hardware that supports 3D graphics, and DirectX (specifically, the Direct3D part of DirectX) takes advantage of that support. WPF, in turn, uses Direct3D to provide support for 3D graphics.

If your program uses Direct3D directly, you need to write a fair amount of code that deals with initializing Direct3D, discovering what hardware is available on the computer, handling device errors, and other administrative chores. WPF handles these details for you so it's easier to build 3D scenes with WPF than it is working with Direct3D itself.

In addition to letting you avoid some annoying details, using WPF to produce 3D graphics also gives you some hope that your application will still work when new versions of Direct3D become available. When some previous versions of Direct3D came out, programmers had to change their programs to work with the new version. If you use WPF to make 3D programs, WPF should adjust to accommodate any Direct3D changes in the future.

There's still plenty of work to do, however. Building a 3D scene requires that you create lights, materials, and surfaces. Depending on the project, it may require that you define texture coordinates and surface normals (vectors indicating a direction away from the surface) to indicate how the surfaces are oriented.

In fact, even with WPF's help, building complex scenes is tough enough that you'll probably want to use code to do it rather than using XAML, but at least it's possible.

The Gasket3D example program shown in Figure 1-9 uses WPF code to display a rotating cube full of holes. This program draws a fairly complicated shape and requires a lot of resources so you may not get very good performance if you set Level greater than 2.

FIGURE 1-9

Retained-Mode Drawing

When you place a control on a form, the control draws itself whenever necessary. If you cover the form with another window and then show the form again, the control redraws itself.

Unlike Windows Forms, WPF provides this capability to all visible objects, not just controls. Your code creates objects such as arcs and polygons that can then draw themselves whenever necessary.

Furthermore, if you need to move an object, you can simply change its coordinates and it will redraw itself accordingly.

The following XAML code defines a `Polygon` object that draws a star. The program doesn't need any code to redraw the star when the window repaints. The `Polygon` object does this automatically.

```
<Polygon Stroke="Red" StrokeThickness="5"
 Points="20,20 120,40 30,70 80,10 110,90" />
```

Figure 1-10 shows the Star example program displaying this object.

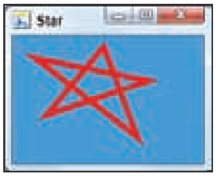


FIGURE 1-10

High-Resolution Vector Graphics

One approach to drawing graphics is to make a bitmap image of your picture and then display the bitmap. A system that draws images based on bitmaps is called a raster-based system.

This works well if you only need to display the image at its original size, but if you scale, skew, rotate, or otherwise transform the bitmap, the results are often blocky and pixelated.

To avoid these problems, WPF takes a different approach. Instead of drawing bitmap-based graphics, it uses *vector graphics*. A vector drawing system stores the data (points, line segments, and other drawing shapes) needed to draw a picture instead of a bitmap image of the original picture. If the picture needs to be transformed, the graphics system transforms the drawing data and then redraws the picture, giving a high-resolution result no matter how the drawing is transformed.

The Zoom example program shown in Figure 1-11 demonstrates the difference between raster and vector graphics. In the top row, the program uses XAML objects (ellipses and an elliptical arc for the mouth) to draw the exact same Smiley Face at different scales. In the bottom row, the program draws a bitmap image of the smallest face at different scales. As you zoom in, you can see how blocky the result becomes.

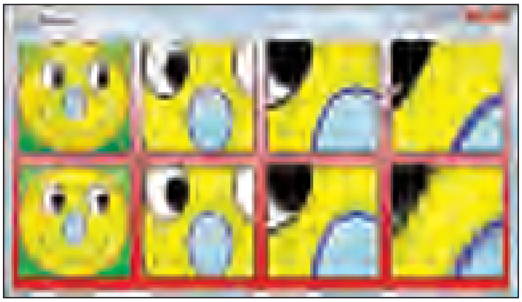


FIGURE 1-11

LINES ENLARGED

Notice that the lines in the vector-drawn image are also scaled. When the drawing is enlarged, lines become thicker and fonts become larger, just as if the original image were magnified, only smoother.

Property Binding to Provide Animation

In addition to using DirectX, another fundamental change with WPF is the way it handles properties. WPF properties seem similar to good old Windows Forms properties at first, and, in many cases, you can treat them similarly.

For example, to set a property in code, you simply create the value that the property should have and assign it to the property. The following C# code creates a `LinearGradientBrush` that shades from white to green. It then sets the `grdMain` control's `Background` property to the brush.

```
LinearGradientBrush bg =
    new LinearGradientBrush(Colors.White, Colors.Green, 90);
grdMain.Background = bg;
```

Behind the scenes, however, WPF properties are very different from the simple properties a program normally gives to the classes that it creates. WPF uses *dependency properties*, a special kind of property that is registered with the WPF property system. Dependency properties support several features that normal properties do not, including default values, inheritance (described further in the next section), WPF-style data binding, and property change notification.

Change notification allows the WPF system to notice when a property changes and take action if necessary. The WPF-style data binding allows a property's value to be tied to some other value. Together these two features allow a program to animate many of the properties that define the appearance of its interface.

For example, the GrowingButtons program shown in Figure 1-12 defines several `Buttons` scaled to 75 percent of their normal sizes. When the mouse moves over a `Button`, the `Button`'s `IsMouseOver` property value changes from `False` to `True`. The program detects that change and starts an animation that increases the `Button`'s scale to 150 percent. When the mouse leaves the `Button`, the property's value changes to `False` and the program uses another animation to decrease the button's scale to 75 percent again.

By providing change notification and supporting the type of data binding used by animations, WPF properties make property animation possible.

FIGURE 1-12

Property Inheritance

A WPF control inherits property values from the control that contains it. For example, if you place a `Button` on a `Grid` that has defined font properties, then the `Button` inherits those properties.

> **A DIFFERENT KIND OF INHERITANCE**
>
> The idea of *property inheritance* is different from the normal concept of inheritance used in object-oriented programming (OOP). In OOP, a derived class inherits the property, method, and event definitions of the parent class but an object in the child class does not inherit property values from the parent class. In WPF, a control inherits some of the property values of the control that contains it.

A control's property values actually depend on a sequence of possible sources that are applied in a specific order of precedence. For example, a `Button` may inherit font properties from the `Grid` that contains it, but the `Button`'s own properties can override those values.

The following list shows the precedence of the simplest places where a property can get its value. The items at the top have the highest precedence, so, for example, a `Button`'s own properties (local values) override inherited values, but an animation can override the local values, at least while the animation is running.

1. Animated values
2. Local value
3. Style
4. Inheritance
5. Default

For a more complete list of places where properties can get there values, see `msdn.microsoft.com/ms743230.aspx`.

Styles

A style lets you define a package of property values for later use. For example, a style can define values for `Width`, `Height`, `BitmapEffect`, and `Background` properties for a `Button`. Later, you can apply the style to a `Button` to give it those property values.

Styles let you easily define a common appearance for a group of controls. Later, if you want to change the appearance of the application, you can simply modify the style, and all of the controls that use it automatically pick up the change. This makes it a lot easier to maintain complex user interfaces and to give controls a similar appearance.

You can use one style as a starting point for another style. For example, you could make a style that defines a common `Background` for the entire application. Then you could make more refined styles to determine the appearances of `Buttons`, `Labels`, and other types of controls.

> **CONTESTED INHERITANCE**
>
> Some controls block inheritance. For example, `Button` and `ListBox` controls do not inherit `Background` property values.

Templates

WPF has a couple of kinds of templates.

A *control template* lets you customize the behavior of a control by determining the objects that make up the control and how they behave.

For example, a normal button displays a rounded rectangle with a certain visual appearance including text in the middle. If you wanted, you could define a control template that made a `Button` draw itself as an ellipse with the text extending outside the left and right sides. You could then define the actions that the control takes when significant events occur such as the mouse moving over the button, the user pressing the button, the button being disabled, and so forth.

A second type of template is a data template. *Data templates* let you define how data-bound controls display their data. For example, suppose you want to display data about students in a `ListBox`. You could use a data template to make the `ListBox` display each entry as a `StackPanel` that holds a picture of the student followed by the student's name and contact information.

Styles let you modify control appearance. Templates let you modify control structure and behavior.

Consistent Control Containment

Many types of Windows Forms controls can contain only a single type of content. For example, `TextBox`, `Label`, `Button`, and `GroupBox` controls all have a `Text` property that determines what text the control displays.

In contrast, many WPF controls can contain just about anything either directly or indirectly. For example, a `Button` control can contain a single object. Normally that object is a string, but it could be something else such as a `StackPanel`. The `StackPanel` can then include whatever you like.

Figure 1-13 shows a program displaying three `Buttons`. Each `Button` holds a `StackPanel` that contains an `Image` and a `Label`.

FIGURE 1-13

This ability for controls to hold practically anything makes WPF controls far more flexible than Windows Forms controls.

Separate User Interface and Code-Behind

One of the biggest goals of WPF was to separate the user interface from the code that lies behind it. Using XAML to define the user interface and C# or Visual Basic code to provide the application's functions lets you separate the two tasks of interface construction and programming.

In theory, graphic designers using Expression Blend can build the interface without knowing anything about programming. Then programmers can use Visual Studio to add the application code.

In practice, not every company can afford dedicated graphic designers and Expression Blend. In that case, the same people may end up building the user interface and writing the code behind it, either with a combination of Expression Blend and Visual Studio, or with Visual Studio alone. Even in that case, however, separating the user interface from the code behind it makes working on the two pieces easier.

A GOOD, OLD IDEA

Good developers have been separating UI development and the code behind it for years. In Windows Forms applications, one developer can build the user interface, while another writes code to perform application tasks. Then you can stitch the two together.

Using XAML makes the separation a bit stricter, however, so it provides additional benefit.

Windows Forms developers also often make the code attached to the user interface as *thin* as possible, making simple event handlers that call routines in other modules to do all of the heavy lifting. That strategy still works for WPF applications.

Another drawback to the "separate designers" theory is that it assumes that graphic designers can build the interfaces you need without knowing a lot about programming. Unfortunately, some of the more advanced XAML and WPF techniques are quite complicated, and both Expression Blend and Visual Studio have trouble with them. To get the most out of WPF, a UI designer might need a PhD in WPF and XAML programming, not a topic that's typically covered in graphics design courses (at least not so far).

This has led to my non-purist WPF philosophy. Do what's reasonable in XAML. If something is easier to handle in code, do so.

New Controls

WPF comes with several new controls for arranging child controls. For example, `StackPanel` arranges its children in a single row or column, `DockPanel` attaches its children to its edges, `WrapPanel` arranges its children in a row or column and wraps to a new row or column when necessary, and `Grid` arranges its children in rows and columns.

A NEW TAKE ON OLD CONTROLS

Some of these controls have been around in other forms for a while now. For example, the old `FlowLayoutPanel` does roughly what the new `WrapPanel` does and the old `TableLayoutPanel` has a similar purpose to the new `Grid`, although it's not as flexible.

Some have claimed that these arranging controls should change the way you design user interfaces by allowing you to take best advantage of whatever space is available on the form. Good developers have already been using Windows Forms controls such as `FlowLayoutPanel` and properties such as `Anchor` to do this, so it's not a revolutionary new concept, although the new controls do give you some new options.

One new control that deserves special mention is `FlowDocument`. A `FlowDocument` is a flexible document object that can hold content much like a web page does. It can hold text in various styles grouped by paragraphs. It can hold lists, tables, and images. It can hold figures and *floaters* that hold content around which the text flows. A `FlowDocument` can hold WPF controls such as `Buttons`, `TextBoxes`, and 3D drawings. It can even run WPF animations so the figures in the document move.

WPF comes with three controls for viewing `FlowDocuments`:

- `FlowDocumentScrollViewer` displays a `FlowDocument` in a long vertically scrolling page much like a web browser displays web pages.
- `FlowDocumentPageViewer` displays the content one page at a time. It provides scrolling and zooming features so the user can see the whole document.
- `FlowDocumentReader` can display the content in one of three ways: like the `FlowDocumentScrollViewer`, like a `FlowDocumentPageViewer`, or in a two-page mode that shows the document two side-by-side pages at a time.

Figure 1-14 shows a `FlowDocumentPageViewer` displaying a `FlowDocument` that contains text, tables, lists, shapes, and even a `Button` control. The floater on the right holds a polygon and a 3D viewport displaying the text *VB Helper* rotating in three dimensions. The `FlowDocumentPageViewer` is displaying the document in TwoPage mode so it shows two pages of the document side-by-side.

Declarative Programming

Many developers add declarative programming as a benefit of using XAML. Intuitively this makes some sense. If the user interface just sits there, it seems reasonable that the code that implements it should be declarative and just sit there, too.

Of course, many WPF user interfaces do far more than just sit there. Once you start adding triggers, storyboards, templates, and styles, the interface is doing much more than just sitting there. In some cases, an interface built in XAML alone is much harder to understand than one that uses a little code at key moments.

Declarative programming is a nice feature of XAML, but it's not clear that it's beneficial in itself.

FIGURE 1-14

DISADVANTAGES

Despite all of its advantages, WPF does have some disadvantages, the biggest of which is its steep learning curve. Using WPF to build a simple form filled with `Labels`, `TextBoxes`, and simple shapes is easy, but taking advantage of all of the features it has to offer is hard.

WPF includes many inconsistencies such as exceptions to property inheritance. The fact that XAML uses a hierarchical format to store information that is not always hierarchical means it must include some rather confusing syntax.

While WPF includes new controls that Windows Forms doesn't have, it is also missing some old ones such as DateTimePicker, MonthCalendar, PropertyGrid, and BackgroundWorker.

The mere fact that WPF is relatively new means that it is less stable than the more established Windows Forms. Changes, hopefully for the better, are inevitable.

Finally, the Expression Blend and Visual Studio XAML Editors leave much to be desired. Neither of them can handle all of the flexibility provided by WPF, and they sometimes generate cumbersome XAML code that you may need to fix by hand. IntelliSense support is also weak for XAML editing, particularly for providing help while entering values for attributes.

Still, WPF provides many advantages and it's easy enough to use for simple interfaces. You can put `TextBoxes` and `Labels` on a window with little difficulty. You can add more complex animations and special effects later when you've had a chance to digest them and you decide that they are necessary for your application.

SUMMARY

WPF contains an assortment of objects that build user interfaces. XAML is an XML-based language that you can use to specify WPF objects. At run time, a program can read a XAML file to create a user interface.

WPF has many advantages including:

- Better use of graphics hardware
- Property binding to provide animation
- Property inheritance
- Styles
- Templates
- Consistent control containment
- Separate user interface and code-behind
- New controls
- Declarative programming

It also has some drawbacks, three of the biggest being a difficult learning curve, increased expense (if you want to use the Expression Blend tool), and mediocre support in the design tools Expression Blend and Visual Studio.

However, WPF is easy enough to use if you don't need all of its more advanced features. You can use WPF to make simple forms containing textboxes, labels, and basic decorations such as frames. Then you can add more advanced features such as styles, templates, and animations when you decide you need them.

As long as you don't get carried away with animation, sound effects, and other fun but distracting features, you can use WPF to build engaging, responsive interfaces that will both keep users interested and let them do their jobs.

Now that you understand what WPF is and what its strengths and weaknesses are, you're ready to start building WPF applications. The next two chapters describe the two main tools for building these applications: Visual Studio and Expression Blend.

2

WPF in Visual Studio

Visual Studio provides everything you need to build WPF applications. It has a WYSIWYG ("what you see is what you get") Window Designer that lets you create a XAML (Extensible Markup Language) interface by dragging controls onto a window. (If you skipped Chapter 1 and want more detail about what XAML, see the section "What Is XAML?" in Chapter 1. Its Code Editors let you write C# or Visual Basic code to sit behind the user interface (UI). It can also run the WPF application that you're building so you can see it in action.

This chapter explains how you can use Visual Studio to build WPF applications. It does not explain how to use Visual Studio in its entirety. Visual Studio is a huge application with lots of powerful features for writing applications in C#, Visual Basic, Visual C++, and other languages, and this book only covers the bare minimum necessary to use Visual Studio to build WPF applications. For more information on programming in those other languages, see a book that covers those topics.

> *For more details on Visual Basic programming, see my book* Visual Basic 2010 Programmer's Reference *(Rod Stephens, Wiley, 2010).*

> *If you have lots of previous experience with Visual Studio, possibly building Windows Forms applications, then most of this material will be familiar and easy to understand. If you're comfortable building forms, setting control properties, and performing other basic Visual Studio chores, then you may want to skim much of this chapter and focus on the parts that are most specific to WPF development, particularly the section "Code-Behind."*

NEW PROJECTS

Starting a new WPF project in Visual Studio is easy. Open the File menu and select New Project to display the New Project dialog shown in Figure 2-1.

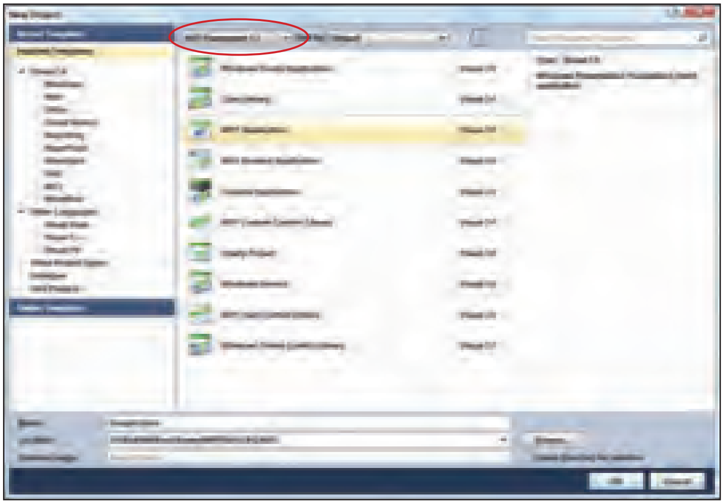


FIGURE 2-1

Select the type of project that you want to build. If you want to use C# to build a stand-alone application, select "WPF Application" as shown in Figure 2-1. If you want to build a XAML Browser Application (XBAP), select "WPF Browser Application."

Note that Expression Blend 3 cannot open programs created for .NET Framework version 4, at least not as of this writing. If you want to be able to edit the program with Expression Blend 3, select a .NET Framework version that it can understand such as 3.5. In Figure 2-1, the Framework version is circled in red.

ALL MAY NOT BE AS IT APPEARS

The exact appearance of the New Project dialog shown in Figure 2-1 and where menu commands appear may depend on how you have Visual Studio configured. For example, in Figure 2-1 it is assumed that Visual Studio is configured for C# development. If it is configured for Visual Basic development, then Visual Basic will be listed as the first category, and C# will appear in the "Other Languages" category.

Also depending on the version of Visual Basic you have installed and how it is configured, your development environment may not look exactly the same as mine. Windows may be moved to new locations or hidden, and even menus may be rearranged. You may need to spend a little extra time searching through Visual Studio to find something, but it should all be there somewhere.

If you want to use Visual Basic for the code behind the user interface, expand the "Other Languages" category and select "Visual Basic" before you pick "WPF Application" or "WPF Browser Application."

Enter a good name for the project and click OK.

Figure 2-2 shows Visual Studio displaying a newly created WPF application. If the Properties window (#5 in the annotated figure) isn't visible, use the View menu's "Properties Window" command to display it.

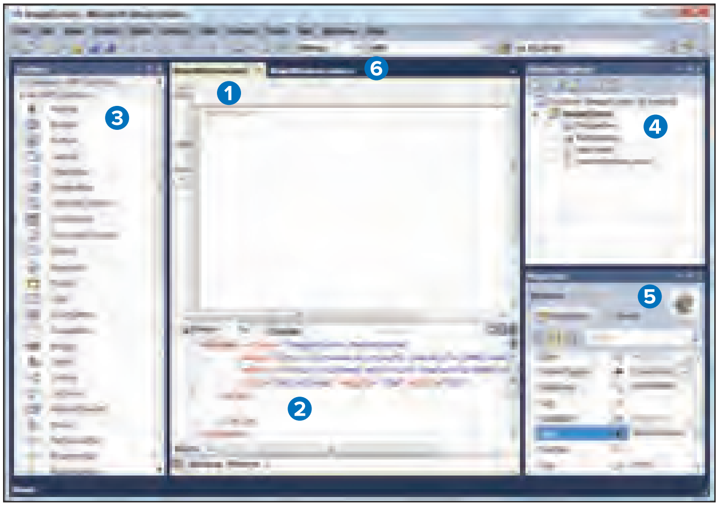


FIGURE 2-2

The following list names the parts that are numbered in Figure 2-2:

1. Window Designer
2. XAML Editor
3. Toolbox
4. Solution Explorer
5. Properties window
6. Window tabs

The following sections describe each of these parts of Figure 2-2 in greater detail.

WINDOW DESIGNER

The Window Designer lets you build windows graphically. It lets you add controls to the window, resize and arrange controls, place controls inside other controls, set control properties, and more.

Notice the scale slider on the left edge of the Window Designer in Figure 2-2. This lets you change the Designer's level of magnification so you can zoom in to work on a small area or zoom out to see everything. Click on the little box with arrows pointing in four directions below the slider to scale the window to fit the available area.

Use the Toolbox (described shortly) to select a control type. Then click and draw on the Window Designer to place the control on it. Alternatively, you can double-click on a tool in the Toolbox to place the control on the window at a default size and location.

After you place a control on the Designer, you can click on it and drag it into a new position. If you create or drag a control on top of a container control such as a Grid, StackPanel, or Frame, the control will be placed inside the container.

CONTENT INTENT

Be careful when you position controls on the window. It's easy to accidentally drag a control inside a container.

When in doubt, look at the XAML code to see which control contains other controls. You can also use the XAML Editor to move controls into other containers if you have trouble dragging them where you want them to be.

When you select a control, the Designer highlights it as shown in Figure 2-3.

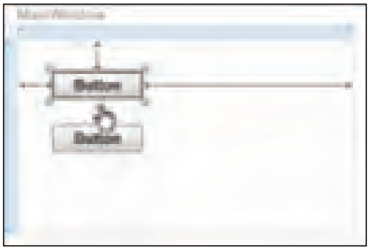


FIGURE 2-3

In Figure 2-3 the upper button's left, top, and right edges are attached to the edges of its container by arrows. This indicates that the control's corresponding edges will remain a fixed distance from the container's edges if the container is resized. In this case, that means if the container is made wider, then the button grows wider, too.

The following code shows the XAML generated by the Window Designer for the top button. The `Margin` attribute sets the distance between the control's and the container's left, top, and right edges. The value `VerticalAlignment="Top"` keeps the button's bottom edge from attaching to the container's.

```
<Button Height="23" Margin="46,30,157,0" Name="button1"
    VerticalAlignment="Top">Button</Button>
```

If you hover over one of the attachment arrows or over the circle on any unattached side, then the cursor changes to a pointing hand, as shown in Figure 2-3. If you click while that cursor is displayed, the control toggles whether the clicked edge is attached to its container's edge. This is an easy way to turn edge attachment on and off.

FIGURE 2-4

You can click-and-drag the grab handles at the control's corners to resize it. As you resize the control, the Window Designer displays the control's width and height, as shown in Figure 2-4. The small, gray line between the two buttons indicates that the Designer snapped the height of the upper button so it is a standard distance away from the lower button (6 pixels).

You can also click-and-drag the control's body to move it. As you drag the control, snap lines appear to show when the control lines up with other controls in various ways such as along the left edges, right edges, top edges, content, and so forth.

Once in a while, the Window Designer may get confused and display the message shown in Figure 2-5 instead of redrawing. Simply follow the message's instructions and click on the message to make the Error List appear (at the bottom in Figure 2-5). Double-click an error message to make the XAML editor go to that line so you can fix it.

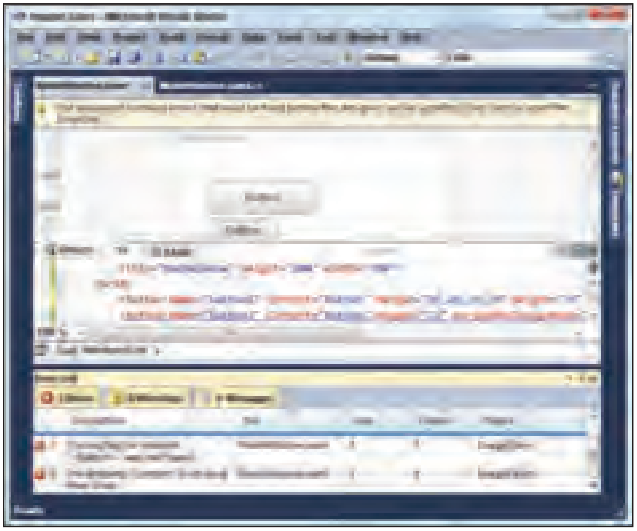


FIGURE 2-5

The Window Designer may also display the message shown in Figure 2-6. Use the View menu's Error List command to open the Error List. Then you can double-click on an error to go to the broken line in the XAML editor as before.

If you look closely at the XAML code in Figure 2-6, you'll see that there are two `Button` controls with the name *button1*. In this example, the error message is correct. If you rename one of the buttons, the Window Designer will reload correctly.

XAML EDITOR

The XAML Editor lets you manually edit the window's XAML code. Usually it's easier to use the Window Designer to place controls on the window and let the designer generate the XAML for you, but there are occasions when you'll need to edit the XAML code directly. The Window Designer just plain can't do some things, and it does a poor job of doing others.

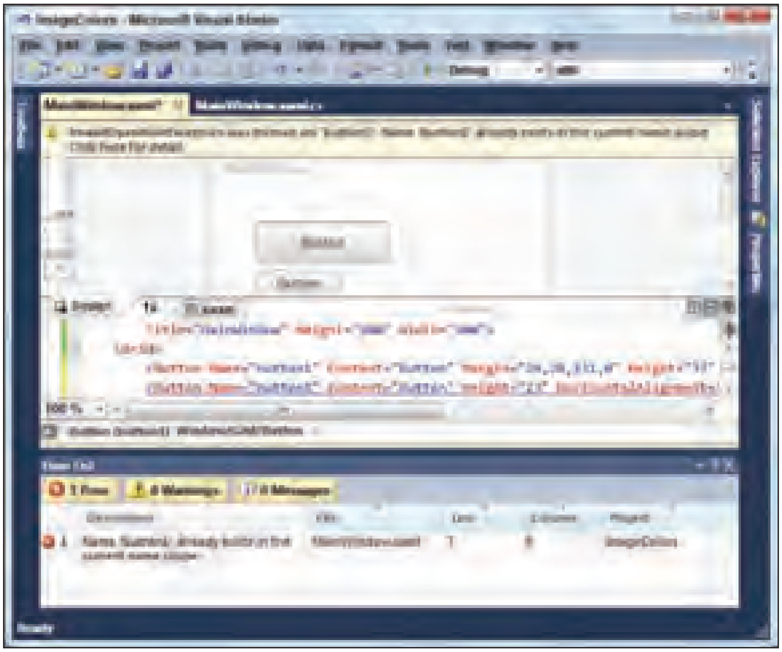


FIGURE 2-6

For example, the Window Designer will not create or apply styles, templates, or storyboards, although if you enter them in the XAML code by hand, it will do its best to honor them.

Sometimes it's easier to edit the XAML than to use the Window Designer. For example, setting exact row and column sizes in a `Grid` can be tricky in the Window Designer, but it's easy in XAML. Define some row and columns in the Window Designer. Then open the XAML Editor, find the `RowDefinition` and `ColumnDefinition` elements, and set their `Height` and `Width` attributes as needed.

For example, the following code gives a Grid two rows, one of height 30 pixels and one using all of the remaining vertical space. It gives the Grid three columns of equal widths.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="30" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="33*" />
        <ColumnDefinition Width="33*" />
        <ColumnDefinition Width="33*" />
    </Grid.ColumnDefinitions>
</Grid>
```

You could very carefully set the Grid's column widths in the Window Designer, but it would be tricky to make them exactly equal. The Window Designer also can't set a row height to a fixed value such as 30 pixels.

COLORIZED CODE

The XAML Editor colorizes code to show its structure. If you don't mess with the default color settings, it displays angle brackets in blue, object names in brown, attribute names in red, and attribute values in blue. The colors make it a bit easier to pick out different parts of an object's definition such as the attribute values.

I recommend that you leave the default colors alone as much as possible so your code looks like other developers' code (and the code in this book), but if you really need to change the colors (for example, if you have trouble telling the default colors apart) you can. Open the Tools menu and select Options. Expand the Environment folder and click the Fonts and Colors tab.

ROW AND COLUMN PROPERTIES

While the Window Designer cannot set fixed row or column sizes, you can do it by clicking the ellipsis next to the `Grid`'s `ColumnDefinitions` and `RowDefinitions` properties in the Properties window. Using the XAML Editor is easier still.

It's also sometimes easier to make a lot of copies of a group of controls using the XAML Editor. Simply highlight the controls' code, press [Ctrl]+C to copy, and [Ctrl]+V to paste.

The XAML Editor provides IntelliSense so as you type it displays a list of possible things you might be trying to type. For example, Figure 2-7 shows IntelliSense after I typed **<Lab** in the XAML Editor. The list has *Label* highlighted because that's the only choice that it knows about that matches what I typed.

After IntelliSense opens, you can use the mouse or arrow keys to select a choice and then press the [Tab] key to make IntelliSense insert the full command. If IntelliSense goes away and you want it back (e.g., if you pressed [Esc] or moved to another line and then moved back), press [Ctrl]+Space to make it open again.

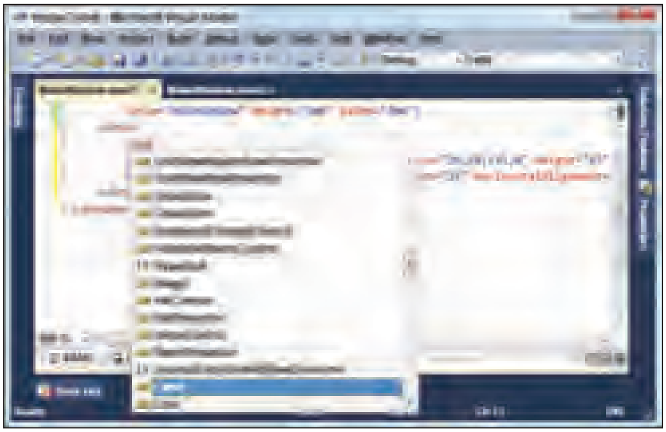


FIGURE 2-7

SPLIT WINDOWS

You can drag the slider between the Window Designer and the XAML Editor to make one bigger and the other smaller. When you're initially placing controls on a window, you may want a large Window Designer and a small XAML Editor. Later, when you're fine-tuning the XAML code, you may want a big XAML Editor.

You can also click on the up/down arrow (⇅) between the two panes to switch which is on top and which is on bottom. If you keep the top window large and the bottom one small, this makes it easy to switch between the two.

TOOLBOX

The Toolbox holds controls that you can place on the window. Many of the controls, such as `Label` and `TextBox`, are self-explanatory, and you should have little trouble figuring out how to use them. Many of the controls are also similar to Windows Forms controls, so using them will be easy if you have experience with Windows Forms controls. Chapters 5 through 8 describe individual controls in more detail.

To use the Toolbox, click on a tool to select it. Move to the Window Designer and click-and-drag to place an instance of the control on the window. You can then drag the control around on the Designer to move it on the window. You can also click on the control and use its grab handles to resize it.

If you [Ctrl]+click on a control in the Toolbox, then that control remains selected even after you click-and-drag to place an instance on the form. This lets you make several controls quickly and can be useful, for example, if you need to make a form containing a lot of labels and textboxes.

Click on the Pointer tool to deselect any previously selected control (including if you [Ctrl]+clicked one). Then you can click on controls in the Window Designer to select them.

If you double-click on a control on the Toolbox, Visual Studio adds an instance of the control to the window at a default size and location. You can then drag it into position and resize it as needed.

The Toolbox contains tabs that group related controls. In Figure 2-2, the Controls tab is open to display all of the available controls. Figure 2-8 shows the Toolbox with the Common tab expanded. This tab contains only the most frequently used controls, which include (on my system at least) `Border`, `Button`, `CheckBox`, `ComboBox`, `Grid`, `Image`, `Label`, `ListBox`, `RadioButton`, `Rectangle`, `StackPanel`, `TabControl`, and `TextBox`.

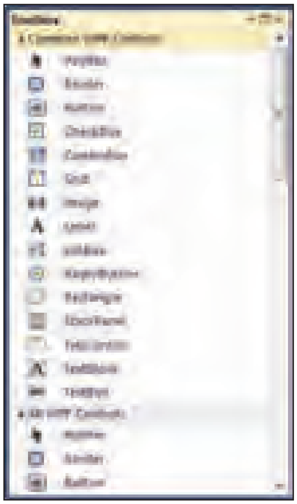


FIGURE 2-8

SOLUTION EXPLORER

The Solution Explorer lists the application's files. Double-click on a file to open it in the appropriate editor. For example, if you double-click Window1.xaml as shown in Figure 2-9, Visual Studio opens the file in the Window Designer.

You can use the Solution Explorer to find other project files such as code-behind files. For example, to open the code-behind file for Window1.xaml, click on the hollow triangle to the left of that file in the Solution Explorer if necessary, and double-click on the MainWindow.xaml.cs (for C#) or MainWindow.xaml.vb (for Visual Basic) file that lies beneath. (Figure 2-9 shows the file expanded so the code-behind file is visible.)

If you add resource files to the project such as images, video, or audio files, you can change the files' properties. Click on a file in the Solution Explorer to select it and then use the Properties window to set properties such as "Build Action" and "Copy to Output Directory."

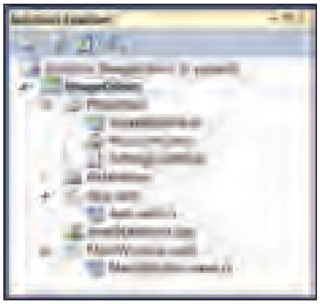


FIGURE 2-9

EDITORS EVERYWHERE

Visual Studio includes a lot of different kinds of editors. It can edit C#, Visual Basic, XAML, images, icons, and even cursors.

In fact, that gives you a relatively easy way to create and edit images and icons. Open the Project menu and select "Add New Item." Select the type of item that you want to add, give it a good name, and click Add. Initially Visual Studio opens the new file for editing. Later you can double-click on the file in Solution Explorer to edit it.

The editors include all sorts of useful tools such as color palettes, drawing tools (pens, brushes, spray, flood), and specialized tools for different file types. For example, the Cursor Editor lets you set the cursor's hotspot, and the Icon Editor lets you add new sizes and color depths to the file. (Icons look best on different systems if you provide lots of different sizes and depths.) The tools may not be as powerful as commercial editors that you can buy, but they are functional and handy.

PROPERTIES WINDOW

The Properties window lets you view and edit the properties values for the currently selected control in the Window Designer. Figure 2-10 shows the Properties window when a `Grid` control is selected.

The Properties window can only set relatively simple kinds of properties. For example, it can set a control's `Background`, `Fill`, or `Stroke` property to solid colors such as Red, Light Blue, or #FFFF8000 (a dark orange). It can even set these properties to gradient brush but it cannot set them to more complex objects such as visual brushes.

To set more complex property values in Visual Studio, you need to type the brush's code in the XAML Editor. For example, the following code fills a `Rectangle` with a `VisualBrush`:

```
<Rectangle>
    <Rectangle.Fill>
```

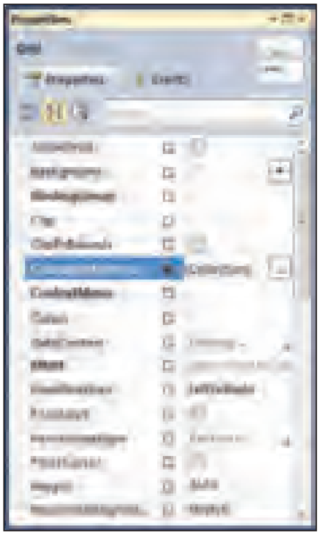


FIGURE 2-10

```
        <VisualBrush Viewport="0,0,0.25,0.25" TileMode="FlipXY">
            <VisualBrush.Visual>
                <Label Content="Hello"/>
            </VisualBrush.Visual>
        </VisualBrush>
    </Rectangle.Fill>
</Rectangle>
```

PROPERTY ODDITY

Sometimes the Properties window doesn't show the properties that you expect. For example, if you select more than one control at the same time (use [Shift]+click to select more than one control or [Ctrl]+click to toggle whether a control is selected), then the Properties window displays only properties that are common to all of the selected controls. For example, `Grids` have a `Background` property but `Rectangles` use `Fill` for the same purpose, so if you select a `Grid` and a `Rectangle`, then you'll see neither of those properties.

Similarly, if you click on a file in the Solution Explorer, you'll see the properties of the file, not the currently selected controls.

The Properties window does provide editors for a few complex properties. For example, it provides editors that let you modify a `Grid` control's `ColumnDefinitions` and `RowDefinitions` properties, and to define gradient brushes.

NECESSARY NAMES

In Figure 2-10, the selected `Grid` control has no name. You can see the Name textbox near the top of the Properties window below the control's type, `System.Windows.Controls.Grid`.

If you need to refer to a control in code, either the XAML code or the C# or Visual Basic code behind the interface, then you should give the control a good name.

Developers often don't give names to controls that are never referred to by the code such as `Labels` that don't change, `GroupBoxes` and other decorative controls, layout controls such as `StackPanels`, and so forth.

Many C# and Visual Basic developers give controls a prefix that tells the kind of control followed by a descriptive name. For example, `txtFirstName` is a `TextBox` that holds a first name. Some links you can follow for specific conventions include:

- Microsoft's Visual Basic naming conventions at `msdn.microsoft.com/aa263493.aspx`

continues

continued

- Microsoft Consulting Service's Visual Basic naming conventions at `support.microsoft.com/kb/110264`
- More general Microsoft naming guidelines at `http://msdn.microsoft.com/xzf533w0.aspx`

Using a prefix lets IntelliSense help you quickly find the control you want. For example, if you type **txt** in a Code Editor, IntelliSense will bring you a list of every available `TextBox` so you can easily pick the one you want. (I find this particularly useful when I teach and I don't know the names of a student's controls. If they use the right prefix, I can type **txt** and find the control whether they named it `txtFirstName`, `txtCustomerName`, or `txtName`.)

Some developers put the type of control at the end of the name as in `FirstNameTextBox` or `firstNameTextBox`. This makes the control's purpose obvious, but it removes the IntelliSense benefit. (I suspect those developers are used to an environment that doesn't have good IntelliSense support.)

WINDOW TABS

Figure 2-11 shows the Window tabs at the top of Visual Studio's editing area.

FIGURE 2-11

These tabs let you select among the various windows that are open for editing. In Figure 2-11, those include MainWindow.xaml (the Window Designer/XAML Designer currently selected), the MainWindow.xaml.cs code-behind file, App.xaml.cs (the code-behind file for the main application), and App.xaml (the XAML file for the main application).

CODE-BEHIND

Unless your application consists solely of UI elements (a loose XAML page might), you'll eventually need to associate program code with the UI elements. For example, the application will need to take action when the user clicks buttons and selects commands from menus.

Visual Studio makes this easy. Depending on what you're trying to accomplish and which language you're using, you have several options.

EVENT HANDLER EXAMPLES

The ImageColors example program, which is shown in Figure 2-12, demonstrates the techniques for attaching controls to event handlers that are described in the following sections. The program's different buttons are attached to event handlers in different ways. Download the C# or Visual Basic version of this program from the book's web site depending on which language you are using.

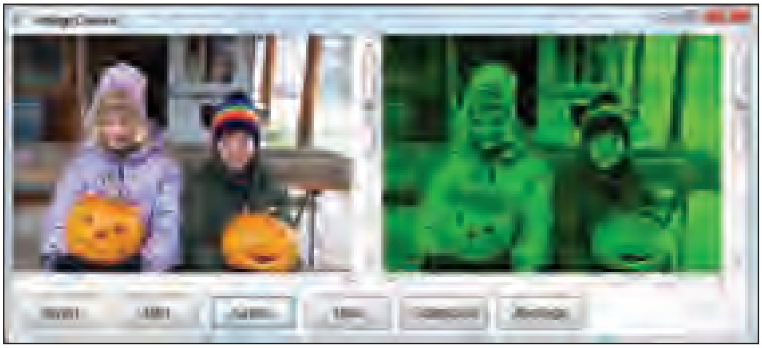


FIGURE 2-12

Default Event Handlers

The simplest way to attach code to the user interface is through Window Designer. Open the window in Window Designer. If you want to add code to a control's default event handler (e.g., a `Button`'s `Click` event), simply double-click on the control. Visual Studio automatically creates an event handler for the event and opens it in the Code Editor (either the C# or Visual Basic Editor depending on the language you are using). It also adds code to the XAML element to attach the control to the event handler.

For example, the following code shows the event handler generated when you double-click on a `Button` named `btnGreen` while using C#.

```
private void btnGreen_Click(object sender, RoutedEventArgs e)
{

}
```

The following XAML code shows the button's definition. Visual Studio automatically added the `Click="btnGreen_Click"` attribute to tie the button to its event handler.

Available for download on Wrox.com

```
<Button Height="23" HorizontalAlignment="Left"
 Margin="10,10,0,0" Name="btnGreen" VerticalAlignment="Top"
 Width="75" Click="btnGreen_Click">Button</Button>
```

ImageColors

The `Click` attribute is the only connection between the XAML code and the code-behind. If you need to detach the button from the code, simply remove that attribute. You don't even need to remove the event-handler code, although leaving it in will clutter the code.

Non-Default Event Handlers

If you want to attach code to an event other than a control's default event (e.g., a `Button`'s `MouseOver` event), select the control in the Window Designer. At the top of the Properties window, click on the Events button to view the control's events.

Figure 2-13 shows the Properties window displaying the events for `btnGreen`. Notice that the `Click` event handler is already filled in with the `btnGreen_Click` routine.

If you select an event, a dropdown appears to the right that will let you pick an appropriate event handler for that event.

If you want to create a new event handler, simply double-click on the event, and Visual Studio will create an appropriate event handler and open it in the Code Editor.

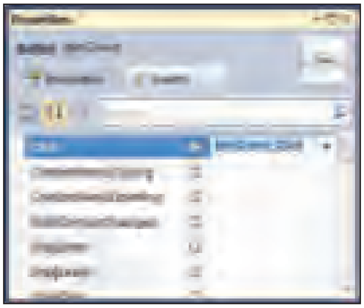


FIGURE 2-13

Handmade Event Handlers

Double-clicking on a control and using the Properties window are the easiest ways to build event handlers, but there's nothing stopping you from doing it yourself by hand.

Add an appropriate attribute to the control's XAML code. For example, to add a `ValueChanged` event to a `Slider` control, add an attribute similar to `ValueChanged="slider1_ValueChanged"`.

Then add the event handler to your code. You'll need to give it the appropriate parameters for the type of event you are catching. For C#, the `Slider` control's `ValueChanged` event handler would look similar to the following:

```
private void slider1_ValueChanged(object sender,
    RoutedPropertyChangedEventArgs<double> e)
{

}
```

In Visual Basic, the code would be similar to the following:

```
Private Sub Slider1_ValueChanged(ByVal sender As System.Object, _
  ByVal e As System.Windows.RoutedPropertyChangedEventArgs( _
  Of System.Double)) _
  Handles Slider1.ValueChanged

End Sub
```

The trick to building event handlers by hand lies in knowing what parameters they need. Because these are often complicated, it's usually easier to double-click on the control or use the Properties window.

Runtime Attached Event Handlers

A Window's controls are objects in the project's code, so you can add event handlers to them in the code at run time. In this case, you don't need to add an attribute to the XAML code to connect the control to its event handler.

The following C# code attaches the `btnRed` control's `Click` event to the `btnRed_Click` event handler.

```
btnRed.Click += btnRed_Click;
```

Here's the Visual Basic version:

```
AddHandler btnRed.Click, AddressOf btnRed_Click
```

It's usually easier to connect controls to event handlers at design time, but this technique lets you add, remove, or change the code associated with an event at run time.

Other Visual Basic Event Handlers

Visual Basic gives you one additional option for connecting controls to event handlers. Do not include an event attribute in the XAML code. Then add a `Handles` clause to the event handler's declaration.

The following Visual Basic code shows the declaration for an event handler. The `Handles` clause at the end indicates that this routine handles the `btnGreen` control's `Click` event.

```
Private Sub btnGreen_Click(ByVal sender As System.Object, _
  ByVal e As System.Windows.RoutedEventArgs) _
  Handles btnGreen.Click
```

The trick again is in knowing what parameters to pass the routine. Happily, Visual Basic's Relaxed Delegates feature lets you simplify the parameter list.

This feature lets you replace the type of a parameter with a different type as long as you know that the actual value will match the new type at run time. For example, if this event handler only handles button clicks, then the `sender` parameter is actually a button. If the event handler doesn't need to use the second parameter, it can declare it using the simpler type `Object`. The simplified version of the event handler is:

```
Private Sub btnBlue_Click(ByVal btn As Button, _
  ByVal a As Object) Handles btnBlue.Click
```

Finally, if the event handler doesn't need to use the parameters at all, you can omit them as in the following code:

```
Private Sub btnGrayscale_Click() Handles btnGrayscale.Click
```

This technique isn't quite as simple as double-clicking on the control in the Window Designer, but it does make the Visual Basic code very simple. Note that you can double-click on the control to create the event handler and then remove its parameters, giving you the best of both worlds.

SUMMARY

Visual Studio provides the basic tools that you need to build WPF applications. The Window Designer lets you build the XAML interface, and the Code Editors let you create the C# or Visual Basic code that sits behind the controls.

While you can get the job done with Visual Studio, it makes some tasks rather cumbersome. For example, it doesn't provide editors for complex objects such as visual brushes, and it doesn't provide any help for creating property animations. To build these things in Visual Studio, you need to write the XAML code yourself.

The Expression Blend tool described in the next chapter provides some useful tools that are missing from Visual Studio. For example, it lets you create complex visual brushes and property animations interactively rather than by writing XAML code. You'll still want to use Visual Studio to write the code behind the interface, but Expression Blend can make some chores a lot easier.

3

Expression Blend

Visual Studio provides an excellent environment for writing C# or Visual Basic code but a mediocre environment for creating XAML interfaces. Expression Blend provides the opposite experience: It is a fine environment for creating XAML interfaces but a lackluster tool for creating C# and Visual Basic code-behind.

Fortunately, Expression Blend is nicely integrated with Visual Studio, so you can switch back and forth between the two tools as needed.

This chapter explains how you can use Expression Blend to build Windows Presentation Foundation (WPF) user interfaces (UI). It explains how to start a new project, add controls to a window, specify basic control properties, and create simple animations.

INSTALLING EXPRESSION BLEND

To get the most out of this chapter, you'll need to install Expression Blend. Unfortunately Expression Blend is rather expensive and, unlike Visual Studio, has no free Express edition. Fortunately you can download a 60-day trial version. You can learn more about Expression Blend or download the trial version at `expression.microsoft.com/cc136530.aspx`.

Expression Blend is a huge and complicated application with lots of powerful features for building XAML user interfaces, and this book only covers the bare minimum necessary to build WPF applications. For more detailed information about the intricacies of Expression Blend, consult a book that focuses solely on Expression Blend such as *Foundation Expression Blend 3 with Silverlight* (Victor Gaudioso, friends of ED, 2009) or *Microsoft Expression Blend Unleashed* (Brennon Williams, Sams, 2008).

If you have lots of previous experience with Expression Blend, then most of this material will be familiar and easy to understand, so you might want to skim this chapter.

NEW PROJECTS

Starting a new WPF project in Expression Blend is easy. Open the File menu and select "New Project" to display the New Project dialog shown in Figure 3-1.

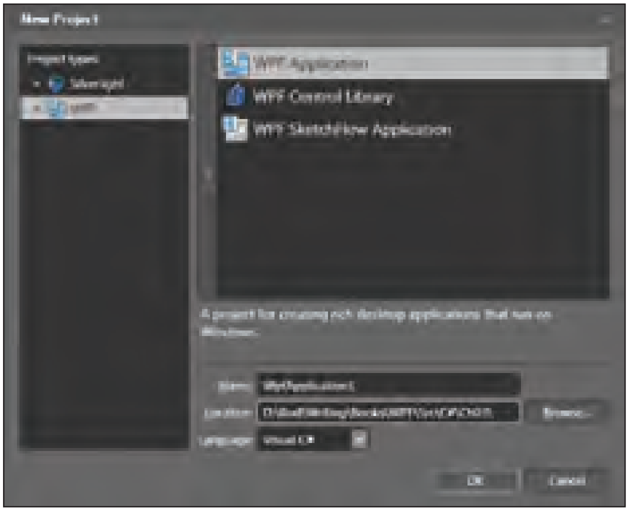


FIGURE 3-1

Select the type of project that you want to build. Enter a descriptive name for the project and the directory where you want to build it. Select the programming language that you want to use for code-behind (C# or Visual Basic) and click OK to create the project.

> *If you only want to make loose XAML pages, select the WPF Application project type and pick either programming language. Later you can copy the XAML file that you build out of the project's directory and discard the rest.*

Figure 3-2 shows Expression Blend displaying a typical WPF application.

> **ALL MAY NOT BE AS IT APPEARS**
>
> Expression Blend may not appear exactly as shown in the figures in this chapter. While it's not as configurable as Visual Studio, there are still plenty of ways to resize, collapse, hide, or otherwise change the appearance of Expression Blend's windows. If you can't find a window, look for its title next to an arrow indicating that it has been collapsed. If you still can't find it, look in the Window menu.

FIGURE 3-2

The following list names the numbered parts that are highlighted in Figure 3-2.

1. Assets window
2. Projects window tab
3. Window Designer, Design view
4. Window Designer, XAML view
5. Properties window
6. Resources window tab
7. Objects and Timeline
8. Triggers window tab
9. Control Toolbox

The following sections describe each of the parts of Figure 3-2 in greater detail.

> **TERMINOLOGY**
>
> The Window Designer's Design view is also called the *artboard*.

ASSETS WINDOW

The Assets window lists tools that you can use while building a project. Select a category on the left (Project, Controls, Styles, etc.) to see items that fit the category on the right. For example, in Figure 3-2 the Controls category is selected so the list on the right shows common controls. You can expand the Controls category to pick from the sub-categories All, Data Visualization, and Panels.

PROJECTS WINDOW TAB

The Project window lists the files in the project. These include application files, resource files such as images and video, the XAML window definition files, and the code-behind for those files.

The Project window gives you several ways to edit different kinds of files. If you double-click on a XAML file, Expression Blend opens it in the Window Designer (described next).

If you double-click on a code-behind file, Expression Blend opens it in an internal code editor complete with IntelliSense.

MISSING EDITORS

Unfortunately, Expression Blend doesn't have some of the extra editors that Visual Studio has to let you edit bitmaps, icons, cursors, and so forth.

If you double-click on an image file in Expression Blend and you have a window open in the Window Designer, Expression Blend adds an Image control containing the image to the currently selected control. If you don't have a window open in the Window Designer, Expression Blend opens the file using its default application, usually Microsoft Paint.

Alternatively, you can right-click on the file and select "Edit Externally" to open it in an external editor like Microsoft Paint.

Best of all, you can right-click on a XAML or code-behind file and select "Edit in Visual Studio" to open the entire WPF project in Visual Studio. This lets you easily edit the code-behind and even the XAML with the full benefits of Visual Studio.

WINDOW DESIGNER

The Window Designer lets you edit windows and define animations. The Storyboards section later in this chapter explains how you can use the Window Designer to build animations. This section explains how you can use it to build a window's basic user interface.

When you want to concentrate on the Window Designer, Expression Blend lets you hide the side panels that normally contain the Project window, Triggers window, and other secondary windows. To hide or restore the panels, open the Window menu and select “Auto-Hide All Panels.”

Figure 3-3 shows the Window Designer with the side panels hidden.

FIGURE 3-3

The upper part of this figure shows the design surface (*artboard*). Here you can add, select, resize, move, and delete controls. The Window Designer highlights the currently selected control with blue lines and displays the control’s name in its upper-left corner (“rectContent” in Figure 3-3).

Below the design surface is the XAML Editor. Here you can review and edit XAML code. In Figure 3-3 the `Rectangle` that is selected in the designer surface is highlighted in the XAML Editor.

The upper-right corner of the design surface holds three sideways buttons (circled in red in Figure 3-3) that let you view the design surface only, the XAML Editor only, or a split view similar to the one shown in Figure 3-3.

Between the design surface and the XAML Editor is a zoom dropdown that says “100%” in Figure 3-3. Use the dropdown to select different scale levels while using the Designer.

The buttons to the right of the zoom dropdown let you turn off rendering effects (for performance), show or hide the snap grid (shown in Figure 3-3), turn snapping to gridlines on and off, turn snapping to snaplines on and off, and show or hide annotations.

The left edge of the screen holds the Control Toolbox. For more information on this area, see the section “Control Toolbox” later in this chapter.

PROPERTIES WINDOW

The Properties Window lets you view and modify the properties that determine the appearance and behavior of a window's controls. Figure 3-4 shows the Properties window for the Rectangle rectContent.

This window groups the control's properties into categories that vary slightly depending on the type of control selected. The following list summarizes the categories shown in Figure 3-4:

- **Brushes** — Sets the control's fill and outline colors.
- **Appearance** — Sets the control's opacity, visibility (visible or hidden), and some aspects of geometry (the radii of the rectangle's corner curves in this example). Advanced properties set dash style and bitmap effects.
- **Layout** — Sets the control's width, height, and alignment. If the control is inside a `Grid`, sets the row and column.
- **Common Properties** — Sets properties that are common to many controls such as `Cursor`, `IsEnabled`, and `ToolTip`.
- **Transform** — Sets transformations that can move, scale, rotate, and skew the control.
- **Miscellaneous** — Holds advanced properties such as `ContextMenu` that don't fit anywhere else.

FIGURE 3-4

Another common category, Text, is used by controls that contain text to specify font properties.

Click on the right-pointing triangle (▶) next to a category to expand it. In Figure 3-4, the Layout, Common Properties, Transform, and Miscellaneous categories are collapsed so they show this symbol.

Click on the downward-pointing triangle (▼) next to a category to collapse it. In Figure 3-4, the Brushes and Appearance categories are expanded so they show this symbol.

Note that some categories also have advanced properties that are hidden by default. In Figure 3-4, the Appearance category has advanced properties that are hidden. Click on the downward pointing arrow with a dot in it (☒) at the bottom of the category to see the advanced properties.

Unfortunately these categories and their advanced sections are fairly confusing. If you are a beginner and sort of know what you want to do but don't really know what property to look for, the sections can help. For example, if you want to change a grid's fill color but don't know the name of the property that does that, you can look in the Brushes and Appearance categories until you figure out that the property is `Background`.

However, if you know that you want to change a control's `MinWidth` property to 100, then you may need to wander around until you discover that `MinWidth` is in the advanced properties part of the Layout category.

To make finding properties easier, the Properties window provides a search box above the property categories. In Figure 3-4, it contains the italicized text *Search*. Click in this box, enter the name of the property you want (or part of the name), and the Properties window will display the matching properties.

The following sections say a bit more about making and editing brushes and using property resources.

Brushes

In WPF, a brush determines how an object is filled. WPF also uses brushes to "fill" items that seem too thin to fill such as the lines that make up a polygon or the edges of a rectangle.

Figure 3-5 shows a rectangle that is filled with a pale yellow brush. Its `Stroke` property, which is used to draw its edges, is a gradient brush that shades from yellow at the top to orange at the bottom. The star-shaped `Polygon` has no fill color (so the `Rectangle` shows through), and its `Stroke` is a gradient brush that shades from blue to white.

FIGURE 3-5

Making Brushes

To make a brush, select the object that should use it on the Window Designer. Then in the Properties window, expand the Brushes category. Figure 3-6 shows the Brushes properties for the rectangle shown in Figure 3-5.

FIGURE 3-6

The bars across the top of the Brushes area show the different kinds of brushes the control can use. For the `Rectangle` shown in Figure 3-5, this includes `Fill` (the `Brush` used to fill the `Rectangle`), `Stroke` (the `Brush` used to draw the `Rectangle`'s outline), and `OpacityMask` (a `Brush` used to determine how opaque or transparent the `Rectangle` is at different points).

Click on one of these bars to select the brush that you want to modify. In Figure 3-6, the Stroke brush is selected.

Now you can use the Brush definition area below to define the brush. The small images across the top of this area are tabs that select different styles of brushes. The following list summarizes these tabs in left-to-right order:

- **No Brush** — The control is not filled so whatever lies below shows through.
- **Solid Brush** — The control is filled with a single solid color. (The rectangle shown in Figure 3-5 is filled with this type of brush.)

- **Gradient Brush —** The control is filled with a gradient brush that shades between colors. The following section says more about gradient brushes.
- **Tile Brush —** The control is filled with a repeating image. The section "Tile Brushes" later in this chapter says more about tile brushes.
- **Brush Resources —** This tab lists predefined brush resources. They include solid system-color brushes such as `ControlBrushKey` and `HighlightBrushKey`, in addition to any brush resources that you have defined. The section "Tile Brushes" later in this chapter explains how to make some kinds of brush resources.

Pick the brush tab that you want, and then define the brush that you want to use. The following sections provide a little more detail about how to make some of the more complicated brushes.

Gradient Brushes

A *gradient brush* fills an area with colors that flow smoothly from one value to another. Depending on the type of brush, the colors may flow in one direction or radially away from a center point. A gradient brush can also include any number of colors so, for example, it might shade from red to green to blue.

To create a gradient brush, select the Brush area's "Gradient brush" tab to see a Brush Editor similar to the one shown in Figure 3-7.

FIGURE 3-7

Below the tabs lies a color square that you can click on to select colors. Along the right edge of the square is a rainbow-like hue selector that you can click on to change the color square's hue. In Figure 3-7, a pure green is selected so the color square shows different shades of pure green. If you clicked on the yellow part of the hue selector, then the color square would display shades of yellow.

Beneath the color square is a horizontal bar showing the colors that make up the brush's gradient. In Figure 3-7, the gradient starts light green on the left, shades to dark green in the middle, and then shades back to light green on the right.

The little house-shaped things below the gradient bar show key colors along the gradient. If you click on one to select it, then you can use other parts of the Brush Editor to change the color at that position. For example, you can click on the color square to pick the key color's value.

To the right of the color square and hue selector are R, G, and B textboxes where you can type a color's red, green, and blue components as values between 0 and 255.

The A textbox holds the color's alpha component. This value tells how opaque the color is and can take values between 0 percent (transparent) and 100 percent (opaque).

Below the A textbox is the color's representation in hexadecimal.

Click-and-drag a key color to change its position in the gradient. Click on the color gradient bar to add more key colors.

The final pieces of the Brush Editor described here let you decide whether the brush should be a linear gradient brush or a radial gradient brush. Click on the button in the Editor's lower-left corner (it shades from black on the left to white on the right) to select a linear gradient brush where colors shade from one to another in a single direction. Click on the next button to the right (it shades from black in the center to white around the edges) to select a radial gradient brush.

Figure 3-8 shows a program that fills two `Rectangles` with gradient brushes. The two brushes are the same (and use the colors specified in Figure 3-7) except the one on the left is a linear gradient brush and the one on the right is a radial gradient brush.

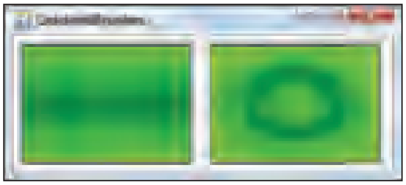


FIGURE 3-8

Transformed Gradient Brushes

More advanced options let you change the appearance of gradient brushes. A particularly useful brush editing tool is the Brush Transform tool. This tool looks like a fat arrow in the Control Toolbox and is shown in the middle of Figure 3-9.

FIGURE 3-9

If you select this tool and then click on a control that uses a brush, the Window Designer displays an arrow that lets you change the brush's geometry.

In Figure 3-10, the rectangle on the left is filled with a linear gradient brush. The arrow shows the gradient's direction.

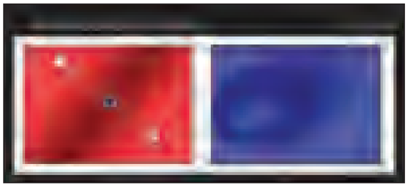


FIGURE 3-10

When the Brush Transform tool is active for a linear gradient brush, you can use the mouse to:

- Grab the middle dot and move the whole arrow without rotating it.
- Grab the arrow's head or tail and move it.
- Hover beyond either end of the arrow to get a rotation pointer. Then click-and-drag to rotate the arrow without moving it.

In Figure 3-11, the rectangle on the right is filled with a radial gradient brush. The gradient's colors shade across the arrow's length from the tail to the head. (This example shades from light blue at the tail to dark blue in the middle to light blue again at the head.)

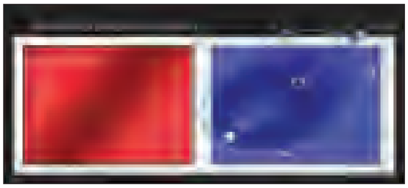


FIGURE 3-11

When the Brush Transform tool is active for a radial gradient brush, you can use the mouse to:

- Drag the arrow's tail to change where the first color begins.
- Drag the small white grab handles to resize the brush. (For example, you could make it shorter and wider.)
- Drag the brush's edge to move the whole brush without otherwise changing it.
- Hover beyond a side's grab handle to get a rotation pointer. Then click-and-drag to rotate the brush without moving it.

Tile Brushes

A *tile brush* fills an area with a repeating pattern. That pattern can be an image, a drawing, or a dynamic drawing that animates at run time. The following sections describe the three kinds of tile brushes: image brush, drawing brush, and visual brush.

Image Brush

An *image brush* fills an area with a picture. You can use the brush's properties to display some or all of the picture on some or all of the filled area, stretching the picture if necessary.

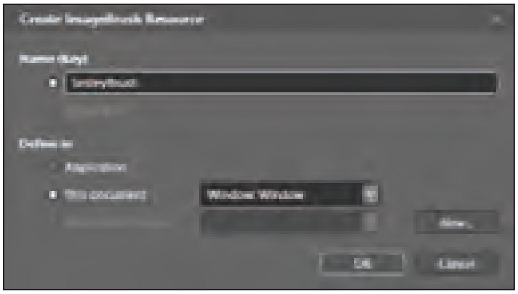


FIGURE 3-12

To make an image brush, first create an `Image` control containing the picture that you want to use. In the Objects and Timeline area, select the new `Image` control.

On the Tools menu, open the "Make Brush Resource" submenu, and select "Make Image Brush Resource" to open the Create ImageBrush Resource dialog shown in Figure 3-12.

Give the brush a good name and click OK.

At this point, Expression Blend has created the brush resource.

> **EXPENDABLE IMAGE**
>
> After creating the image brush, Expression Blend no longer needs the `Image` control that you used to create it. You can delete that control if you like.

To see the new brush, click on the Resources tab on Expression Blend's right panel. Click on the dropdown arrow to the right of the brush to open the Brush Property dialog shown in Figure 3-13.

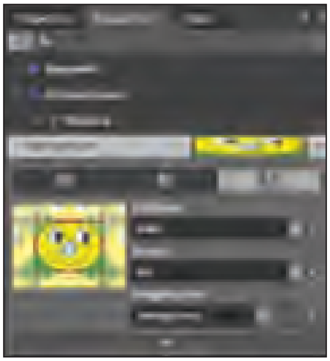


FIGURE 3-13

The "Tile mode" property determines how the image is repeated if necessary to fill the area and can take the following values:

- **None** — The image is not repeated. Parts of the area may be unfilled if a single copy of the image doesn't fill the area completely.
- **FlipX** — The image is flipped horizontally to create new columns of the image.
- **FlipY** — The image is flipped vertically to create new rows of the image. (If you look closely at Figure 3-13, you'll see

that the copy of the image above the central one is flipped vertically so the eyes are at the bottom. Similarly the image on the bottom has been flipped to the smile is on top.)

- **FlipXY** — The image is flipped both horizontally and vertically to create new copies of the image.
- **Tile** — The image is repeated as needed without any flipping.

The `Stretch` property determines how the image is scaled or stretched when needed and can take the following values:

- **None** — The image is not stretched.
- **Fill** — The image is stretched to fill the area.
- **Uniform** — The image is stretched to be as big as possible in the area but is not distorted. For example, if the image is relatively tall and thin compared to the area, it will be stretched until it fills the area vertically, and parts of the area to the sides may remain unfilled.
- **UniformToFill** — The image is stretched uniformly until it completely fills the area. If parts of the image extend beyond the edges of the filled area, they are clipped off.

FIGURE 3-14

To use the new brush resource, place a control such as a `Rectangle` on the window. Open the Properties window, find the Brushes category, and select the brush type that you want to set (`Fill` for a `Rectangle`). Among the tabs for different kinds of brushes (none, solid, gradient, etc.), select the rightmost Brush Resources tab.

Find the new brush in the list shown in Figure 3-14 and select it.

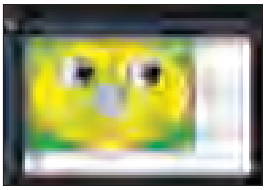


FIGURE 3-15

Figure 3-15 shows the result. Notice that the image is stretched to fill the entire rectangle. Since the original image was circular, the result is stretched out of shape.

This is the default appearance that Expression Blend assumes you want to use. It works well if you want a large area to display a picture as a background and either the image fits the area or you don't mind some stretching.

It doesn't work as well if you want to tile an area with repeated copies of an image. To do that, you need to understand viewboxes and viewports.

A brush's *viewbox* determines the part of the image that is used by the brush. The *viewport* determines the part of the filled area that should correspond to a single copy of the viewbox image.

In the image shown in Figure 3-15, the viewbox includes the entire image, and the viewport includes the entire rectangle so the image is stretched to fill the rectangle.

> **PERPLEXING PROPERTIES**
>
> Expression Blend does not provide an interactive tool for setting a brush's viewbox or viewport properties. To set these values, you must manually edit the XAML code.

The brush's `Viewbox` and `Viewport` properties contain four values that give the left, top, width, and height of the area you want to use. These values can be in one of two coordinate systems: RelativeToBoundingBox or Absolute.

RelativeToBoundingBox coordinates are measured as a fraction of the relevant area's size. For example, setting the left value to 0 and the width to 0.5 would select the left half of the area.

Absolute coordinates are measured in pixels. They are often useful for tiling images, although you need to know the exact size of the image to make everything fit perfectly.

Figure 3-16 shows four rectangles filled with the same image but with different `Viewbox` and `Viewport` values.

The upper-left rectangle uses the default settings, so the image is stretched to fill the rectangle.

The upper-right rectangle uses `Viewbox` values 0, 0, 1, 1, so it uses the entire brush image. Its `Viewport` values are 0, 0, 0.5, 0.5, so a single copy of the image is used to fill the upper-left quadrant of the rectangle. In this brush, `Stretch` is set to `Uniform`; thus the image is scaled uniformly to be as big as possible within the upper-left quadrant. Since the `TileMode` property is `Tile`, the image is repeated without flipping.

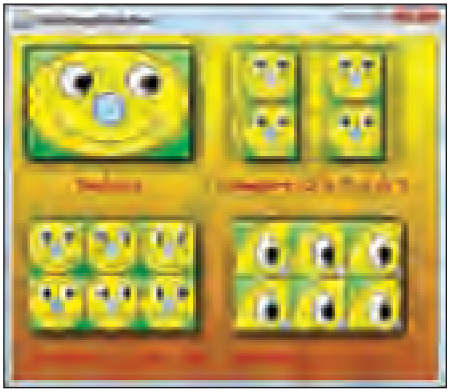


FIGURE 3-16

> **SHADOWY TILES**
>
> The rectangles in Figure 3-16 all have drop shadows. Notice how the upper-right rectangle displays the drop shadow for each tiled copy of the image, not the rectangle as a whole.

Since the lower-left rectangle sets `ViewportUnits` to `Absolute`, the `Viewport` values are specified in pixels. Its `Viewport` values are 0, 0, 50, 50; thus the first tiled image is sized to fill the upper-left 50×50-pixel square in the rectangle. This rectangle is 150 pixels wide and 100 pixels tall and thus is filled exactly by the six images shown in Figure 3-16.

Finally, the lower-right rectangle sets its `Viewbox` values to 0, 0, 0.5, 0.5, so it selects the image's upper-left quadrant. Since it uses the same viewport as the lower-left rectangle, this piece of the image is used to exactly tile the rectangle.

> **TRUE TILING**
>
> The technique shown in the lower left of Figure 3-16 is very useful for tiling areas. Set the brush's `Viewbox` to 0, 0, 1, 1. Set `ViewportUnits` to `Absolute` so the `Viewport` values are specified in pixels and then set the `Viewport` values to match the image's size in pixels.

The following code snippets show the brushes used by the UseImageBrushes example program shown in Figure 3-16. Note that it shows only the brushes, not the rest of the XAML code.


```
<ImageBrush ImageSource="Smiley.bmp" />

<ImageBrush Stretch="Uniform" TileMode="Tile"
 ImageSource="Smiley.bmp"
  Viewbox="0,0,1,1" Viewport="0,0,0.5,0.5"
/>

<ImageBrush Stretch="Uniform" TileMode="Tile"
 ImageSource="Smiley.bmp"
  Viewbox="0,0,1,1"
  Viewport="0,0,50,50" ViewportUnits="Absolute"
/>

<ImageBrush Stretch="Uniform" TileMode="Tile"
 ImageSource="Smiley.bmp"
 Viewbox="0,0,0.5,0.5"
 Viewport="0,0,50,50" ViewportUnits="Absolute"
/>
```

UseImageBrushes

Drawing Brush

A *drawing brush* fills an area with a drawing that may contain shapes, labels, video, and other controls.

To make a drawing brush, add some sort of container control to the window such as a `Grid`, `Canvas`, or `StackPanel`. Then add the controls to it that you want to be part of the brush.

Next, select the container control, open the Tools menu, expand the "Make Brush Resource" submenu, and select "Make DrawingBrush Resource." On the Create DrawingBrush Resource dialog, give the brush a good name and click OK.

Now you can modify the brush as explained in the previous section for image brushes. For example, you can open the Resources window in Expression Blend's right panel and click on the brush's dropdown to set its `TileMode` and `Stretch` properties.

Review the previous section "Image Brush" for more information on modifying the brush. In particular, you may want to review how to set the brush's viewbox and viewport.

DRAWING DELETED

After creating the drawing brush, Expression Blend no longer needs the drawing that you used to create it. You can delete the container control that you used if you like.

Figure 3-17 shows a rectangle filled with a drawing brush. The brush contains a centered image and four labels that are rotated and aligned to the brush's sides.

FIGURE 3-17

The following code snippet shows the brush's opening element with its `TileMode`, `Viewbox`, `Viewport`, and `ViewportUnits` properties:

```
<DrawingBrush x:Key="DangerBrush" TileMode="Tile"
 Viewbox="0,0,1,1"
 Viewport="0,0,100,100" ViewportUnits="Absolute">
```

The rest of the brush's definition isn't shown here because it's long, very complicated, and not the focus of this chapter.

BEWILDERING BRUSH

The XAML definition for this seemingly simple brush contains 810 lines of code and more than 5,000 parts (tags, attributes, and other values).

One of the reasons this drawing brush is so complicated is that it contains a large object hierarchy. It contains a `DrawingGroup`, which in turn contains a series of `GeometryDrawing` and other `DrawingGroup` objects with appropriate transformations and geometry objects. By far the biggest part of the brush is the code that draws the text, because Expression Blend converts the `Label` controls into `PathGeometry` objects that draw each letter stroke-by-stroke.

Visual Brush

A *visual brush* fills an area with a copy of a user interface element. That element may contain other elements and may even include animations.

To make a visual brush, create the control that defines the brush, including any controls that it should contain. Select the control, open the Tools menu, expand the "Make Brush Resource" submenu, and select "Make VisualBrush Resource." On the Create DrawingBrush Resource dialog, give the brush a good name and click OK.

Now you can modify the brush as explained in the previous sections for image brushes. For example, you can open the Resources window in Expression Blend's right panel and click on the brush's dropdown to set its `TileMode` and `Stretch` properties.

Review the earlier section "Image Brush" for more information on modifying the brush. In particular, you may want to review how to set the brush's viewbox and viewport.

KEEP THE CONTROLS

Unlike the case for the image brush and drawing brush, you must not delete the controls that define a visual brush. WPF uses those controls at run time to build the image that the brush uses; thus, if you delete those controls, then it cannot make the brush's image.

If you don't want the visual brush's controls to be visible, you can hide them behind another control.

Figure 3-18 shows a program using a visual brush. The grid in the upper-left corner contains a radial gradient background and a label that uses a property animation to rotate continuously. The rectangle on the right is filled with a visual brush that uses this grid as its visual source. As the label in the grid rotates, so do the copies displayed in the rectangle. (Note that this would be a very annoying user interface!)

FIGURE 3-18

Figure 3-19 shows a common parlor trick that uses visual brushes. Below each of the textboxes is a rectangle that uses the corresponding textbox as a visual brush. The rectangles are scaled by a factor of –1 vertically so their contents are flipped upside down, making them look like reflections of the textboxes.

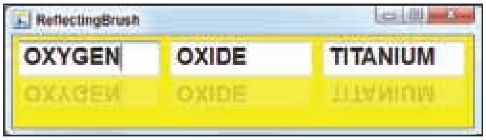


FIGURE 3-19

When you type in a textbox at run time, the corresponding visual brush also updates, so the rectangle below shows the updated reflection. In Figure 3-19, you can even see the reflection of the cursor in the first textbox.

RIDDLE ME THIS

All three of the rectangles shown in Figure 3-19 are drawn using the same technique. So why isn't the text in the middle rectangle upside down like the others? See the footnote for the answer.

The text in the middle textbox happens to include only letters that look the same right side up as upside down, at least in that font.

The program shown in Figure 3-20 demonstrates all three types of tile brush. The text in the visual brush on the right spins continuously.

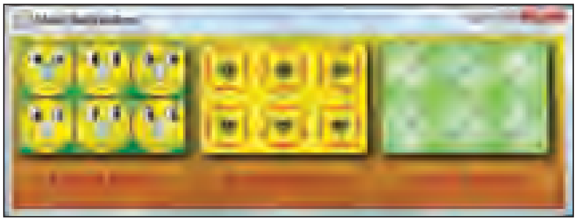


FIGURE 3-20

The program contains a grid that holds two other grids. The first contains a brush that provides the visual brush used in the rightmost rectangle. The second grid covers the first (so the visual brush's defining grid is hidden). It contains a horizontal stack panel that contains three vertical stack panels, each holding a filled rectangle and a label.

Pens

Brushes determine how an area such as the interior of a rectangle is filled. They also determine how lines such as the borders of a rectangle or polygon are filled.

Pens determine the *style* of lines such as the edges of rectangles and polygons. For example, they determine the line's thickness, dash pattern, and corner style.

The WPF properties that define a pen all begin with the word *Stroke* and include `StrokeThickness`, `StrokeDashArray`, `StrokeDashCap`, and `StrokeMiterLimit`. You can find these properties in the Properties window's Appearance section, mostly in the advanced properties area.

STROKE

In fact, the brush used to fill a line is stored in the line's `Stroke` property, so all of the properties that define a line's appearance are stroke properties.

Figure 3-21 shows a polygon that demonstrates three of the `Stroke` properties. Its `StrokeThickness` property is set to 20, so the polygon's lines are very thick; its `StrokeMiterLimit` property is set to 1, so the extra pointy corners on the right edge of the polygon are chopped off; and its `Stroke` property is set to a brush that shades from red to yellow.

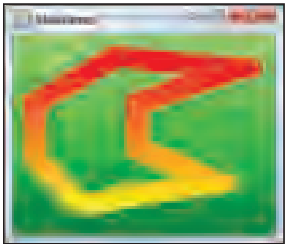


FIGURE 3-21

The following XAML code shows the polygon's definition:

Available for download on Wrox.com

```
<Polygon StrokeThickness="20"
 Points="20,110 20,60 90,20 200,30
         110,60 110,100 180,130 60,150"
 StrokeMiterLimit="2">
    <Polygon.Stroke>
        <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
            <GradientStop Color="#FFFF0000" Offset="0"/>
            <GradientStop Color="#FFFFFF00" Offset="1"/>
        </LinearGradientBrush>
    </Polygon.Stroke>
</Polygon>
```

UsePens

Chapter 9 has more to say about how pens and brushes work. In this section, I just want to mention that Expression Blend provides editors for stroke properties that are missing in Visual Studio.

Property Resources

A *resource* makes it easy to ensure that several controls all use the same values. Rather than setting the `Background` property for a group of buttons individually, you can make a resource representing the background and then make the buttons all use that resource for their backgrounds.

Resources also make it easier to change shared values. To make the buttons use a different brush, you only need to change the `Brush` resource.

The Properties window allows you to convert many properties into resources. Then you can use those resources to set similar property values on other controls.

FIGURE 3-22

For example, suppose you want to give every button on a form the same background. First give one button the background that you want. Next click the tiny “Advanced property options” button under the mouse in Figure 3-22 and select the “Convert to New Resource” command to make a resource holding the brush’s properties.

FIGURE 3-23

Now you can select the other buttons and set their background properties to use this resource. Select another button, open the Properties window’s Brushes section, and select the Background brush. Click the “Brush resources” tab on the right (under the mouse in Figure 3-23) and pick the new resource from the list. Figure 3-23 shows a `Brush` resource named `btnBrush`.

The Properties window lets you use resources for other properties in slightly different ways. For example, to make a resource to hold a control’s `Width` property, click on the little box to the right of the `Width` property and select “Convert to New Resource” from the dropdown menu.

To set another control’s property to this value, select the control and click on the same little box to the right of the control’s property. In the dropdown menu, expand the Local Resource submenu and select the resource, as shown in Figure 3-24.

Styles

A *style* lets you group properties, triggers, and other attributes to be shared by several controls. For example, the previous section explained how you could make several buttons share a common background resource. You can use a style to make the buttons share a common background, width, height, and other properties.

To create a style for a group of buttons, make one button with the properties that you want in the style. Then select the button, open the Object menu, expand the Edit Style submenu, and select “Edit a Copy.”

This creates a style and opens it for editing. Use the Properties window to set any additional property values for the style. When you are finished, click the “Return scope to Window” button under the mouse in Figure 3-25.

Having created a style, you can apply it to other controls. Select a control, open the Object menu, expand the Edit Style submenu, expand its Apply Resource menu, and select the style that you want to apply.

Note that Expression Blend will only let you apply a style to an object that is compatible with the one you used to define the style. For example, if you define a `Button` style, you cannot apply it to a `Label`.

FIGURE 3-24

FIGURE 3-25

EDITING STYLES

If you need to edit a style later, open the Resources window, find the style that you want to edit, and click the “Edit resource” button to its right.

RESOURCES WINDOW

The Resources window lets you view and edit resources. Figure 3-26 shows the Resources window.

The Resources window lets you edit some values directly. For example, in Figure 3-26 you can type a new value for `btnWidth` in its textbox.

The Resources window provides editors for other values. If you click on the colored area to the right of the `btnBrush` resource, a brush editing area appears similar to the one you use to define a brush in the Properties window.

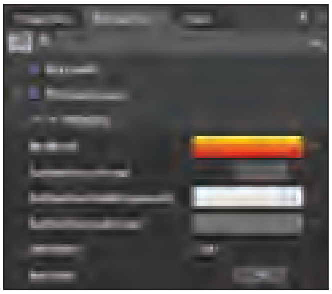


FIGURE 3-26

OBJECTS AND TIMELINE

The Objects and Timeline area shown in Figure 3-27 lets you edit objects and storyboards. Usually you use this area to select controls so you can change their properties in the Properties window.

FIGURE 3-27

When it is displaying the window's controls, this area shows the hierarchical arrangement of controls. For example, in Figure 3-27, the window contains a `Grid` named `OuterGrid`, which contains two `Grids` named `VBHGrid` and `InnerGrid`. Those controls contain a bunch of others. (The square brackets around a control's type means that it has no name.)

One of the more confusing features of this area is in how you select controls. If you click on a control, it turns gray and is selected for property editing, so changes you make to the Properties window apply to the control. In Figure 3-27, the `Label` that says "Image Brush" is selected for property editing.

If the control you click is a container, it is also highlighted with a blue box. If the control is not a container, then its container is highlighted. In Figure 3-27 the `StackPanel` containing the selected `Label` is highlighted.

If you double-click on a control in the Toolbox, a new instance of that control is added to the container that is highlighted with the blue box. In Figure 3-27, if you were to double-click the Button tool, a new `Button` would be added to the highlighted `StackPanel`.

The Objects and Timeline area has two useful icons to the right of each control. The first, which looks like an eye, lets you make the control visible or invisible. Even if you want a control to be visible at run time, it's sometimes useful to hide it for a while at design time to reduce clutter so it's easier to see the other controls.

The second icon looks like a dot or a padlock depending on whether you have clicked it. When the padlock is visible, the control's properties are locked so you cannot accidentally change them, for example, by dragging the control on the Window Designer.

The Objects and Timeline area has a few other useful features. You can click-and-drag controls to new positions within a container. For example, that would let you reorder the controls in a `StackPanel`. It also lets you change the stacking order because the controls are painted in order with those listed first drawn first.

One final trick is worth special mention here. If you select a group of controls, right-click on them, and select "Group Into," then Expression Blend lets you select a container control such as a `Grid` or `StackPanel`. When you pick a container, Expression Blend moves the controls that you selected into a new instance of the container.

Storyboards

In addition to letting you edit control properties, the Objects and Timeline area lets you build *storyboards* that control property animations.

To create a storyboard, click on the plus sign at the top of the area. Give the storyboard a descriptive name and click OK.

Figure 3-28 shows the Objects and Timeline area while it is editing the storyboard named *SpinButton*.

To edit a storyboard, drag the yellow time indicator to a time. In Figure 3-28, the indicator is set at 0.5 second. Now change the properties of the controls that you want to animate. Expression Blend creates a storyboard that sets those properties to the desired values at the selected time.

You can use the video-control style buttons at the top of the timeline area to test the animation. For example, click the Play button (▶) to run the animation.

FIGURE 3-28

> **SUSPEND TIME**
>
> While you are editing a storyboard, the Window Designer is outlined in red and displays a label at the top that says "Timeline recording is on." Click the red dot beside this label to turn timeline recording off. Then you can modify control properties without the changes becoming part of the storyboard.

Figure 3-28 shows the timeline for the SpinButton storyboard in the SpinningButtons example program. The storyboard sets a button's angle of rotation to 0 degrees at 0 seconds and sets the angle of rotation to 359.9 degrees at 0.5 second. This makes the button rotate through one complete revolution in half a second.

TRIGGERS

After you define a storyboard, you can determine which events trigger it. Initially, Expression Blend hooks the storyboard to the window's `Loaded` event so the storyboard executes when the program starts.

You can change this behavior by using the Triggers area shown in Figure 3-29.

Click on the control dropdown next to the right of the word "When" and select the control that should trigger the storyboard. In Figure 3-29, this is set to the `Button` named `btnSpinMe`.

FIGURE 3-29

> **CONTROL CONFUSION**
>
> The control dropdown only lists the control that is currently attached to the storyboard and the control that is currently selected in the Objects and Timeline area. If the dropdown doesn't show the control that you want, be sure it is selected in the Objects and Timeline area.

Next, use the second dropdown to select the control's event that should trigger the storyboard. In Figure 3-29, this is the `Button`'s `Click` event.

In the dropdowns below, you can select the storyboard that you want to control with the trigger and the action that you want the storyboard to perform. In Figure 3-29, the SpinButton storyboard begins.

CONTROL TOOLBOX

The Control Toolbox shown in Figure 3-30 lets you put controls on a window. Double-click on a control in the Objects and Timeline area to select it for adding children. Then you can double-click on a control in the Toolbox to put a new instance of that control in the selected control.

FIGURE 3-30

The Control Toolbox only shows a few controls at a time. If you don't see the control that you want, you need to take action to make it appear.

If the control you want is one of the more common controls, you can find it in the Toolbox's control groups. For example, the third control from the top in Figure 3-30 represents the `Grid` control. If you want some other container control, you can right-click this one and select `Canvas`, `StackPanel`, `WrapPanel` or some other container from a dropdown list. That replaces the Grid control with the control you selected so you can use it.

If the control that you want isn't in any of the control groups or you just can't find it, click on the double right arrow at the bottom of the Toolbox to open the Asset Library, shown in Figure 3-31.

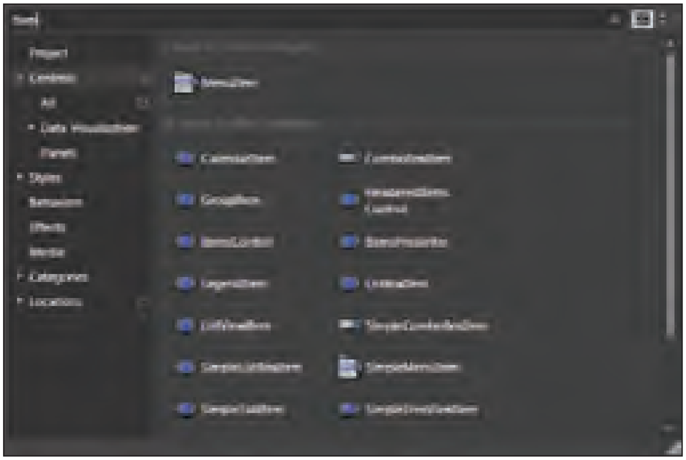


FIGURE 3-31

The Asset Library contains a lot of controls so, if you don't immediately see the control that you want, type part of its name in the search box at the top. In Figure 3-31, the search box contains the text *Item* so the Asset Library is showing the controls that have the word *Item* in their names.

When you select a control, the Control Toolbox adds that control to the bottom in a recently used controls group. In Figure 3-30, that area holds the `Border` control just below the double arrow.

CODE-BEHIND

Expression Blend is not a programming tool and isn't really intended to be used to edit the code behind the user interface.

To add code to a window, open the Project window. Right-click on the XAML file that should have code and select "Edit in Visual Studio." Edit the code there, and then close Visual Studio to return to Expression Blend.

SYNCHING SECRETS

When you switch between Expression Blend and Visual Studio, the two applications need to ensure that the files you modify are synchronized. For example, if you start Expression Blend, move to Visual Basic, modify a file, and then return to Expression Blend, Expression Blend asks if you want to reload the modified file to pick up the changes you made.

That all makes sense, but there's one situation that can be confusing. If you open Expression Blend, modify a file, and then open Visual Studio, the changes you made in Expression Blend are not automatically saved first, so Visual Studio does not see the changes. If you edit the file in Visual Studio and save the changes, you may have two versions of the file floating around, one being used by Expression Blend and one being used by Visual Studio. If you save the two versions, you'll lose one or the other, depending on which you save first.

To avoid confusion, always save all changes before you switch between the two applications.

SUMMARY

This chapter explains the basics of Expression Blend. Expression Blend is a tool for building XAML interfaces. It provides many editors for creating and using properties, resources, styles, and storyboards.

While Expression Blend is not a free product like Visual Studio, it provides many useful editors that are missing from Visual Studio. For example, Expression Blend provides an easy-to-use editor for specifying gradient backgrounds that is more powerful than the one available in Visual Studio.

Creating storyboards by hand in Visual Studio is confusing enough that Expression Blend's storyboard editing system alone may be worth the expense, at least if you do a lot of property animation. Download the trial version of Expression Blend at `expression.microsoft.com/cc136530.aspx` and see if its benefits outweigh its cost.

Expression Blend is not a programming tool, so it doesn't give you much help for attaching C# or Visual Basic code to the interface. But since Visual Studio provides the tools that you need to do that, Expression Blend and Visual Studio are a powerful combination.

This chapter and the previous one describe the two main tools for building WPF applications: Visual Studio and Expression Blend. Having read these chapters, you should be able to create a new project and add controls to build a user interface. However, it will be difficult to pick the right control for a particular task unless you know what controls are available and what they can do.

The following chapters describe different kinds of controls and discuss their capabilities. After you read them, you'll know just what controls to use under different circumstances.

4

Common Properties

All of the WPF control classes inherit from the `Control` class, and that common ancestry gives them many properties in common. For example, most of them have properties that determine their size and position, foreground and background colors, transformations, and font properties.

This chapter describes some of the properties that many controls share. Later chapters mention some of these properties again where they are particularly relevant, but if you learn about them now, you will already know a lot about the controls described in the following chapters.

The following section describes a particularly important and confusing topic: sizing and positioning properties. Every WPF developer must understand these properties to arrange controls effectively.

SIZE AND POSITION

Sizing and positioning properties can be very confusing because a control's geometry depends on several different properties, the way in which those properties interact, and even the control's container. For example, the same `Button` looks very different when it is placed inside a `Grid`, a `StackPanel`, or a `Canvas`.

The following section describes the most important properties for determining a control's size and position — its alignment properties. The next section then describes some simpler properties that also influence a control's size and position.

Alignment

Four of the most important properties that determine a control's size are `Width`, `Height`, `HorizontalAlignment`, and `VerticalAlignment`.

ASSORTED ALIGNMENTS

Don't confuse `HorizontalAlignment` and `VerticalAlignment` with `HorizontalContentAlignment` and `VerticalContentAlignment`. The former determine the control's alignment, while the latter determine the alignment of the control's content.

The `Width` and `Height` properties are fairly self-explanatory. They determine the control's size directly.

Often, however, you don't really want to set a control's size explicitly. Instead, it would be better for the control to resize itself in some manner to fit either its contents or the area that contains it. That lets the control use as little space as possible or take best advantage of the space available.

That's where the `HorizontalAlignment` and `VerticalAlignment` properties come into play. `HorizontalAlignment` can take the values `Left`, `Right`, `Center`, and `Stretch`. `VerticalAlignment` can take the values `Top`, `Bottom`, `Center`, and `Stretch`.

The `Left`, `Right`, `Top`, `Bottom`, and `Center` values determine the control's position in a fairly obvious way. For example, if a control has these properties set to `Left/Top`, then it is placed in the upper-left corner of its container.

The `Stretch` value is a bit less intuitive. In general, this value means that the control should expand to fill its container, but the exact result depends on the type of the control's container.

For example, if the container is a `Grid`, then the control stretches to fill its `Grid` cell as expected. In contrast, if the container is a `StackPanel` oriented vertically, then setting `HorizontalAlignment = Stretch` makes the control expand to fill the width of the `StackPanel`, but the `VerticalAlignment` property is ignored. If you don't explicitly set the control's `Height` property, then the control will be as short as possible while still displaying its contents.

The SizeInContainers example program shown in Figure 4-1 displays `Button` controls inside various types of containers. The `Labels` tell what kind of containers they are, and the containers have yellow backgrounds where possible so you can see them. The `Buttons` in each container are the same, and all have `HorizontalAlignment` and `VerticalAlignment` set to `Stretch`.

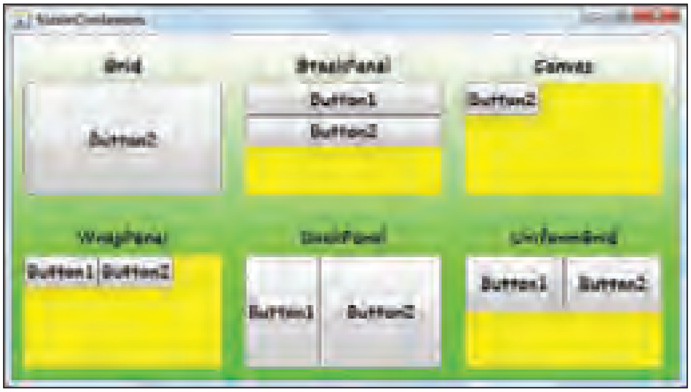


FIGURE 4-1

The following XAML code shows how the program creates its `Grid` container together with its `Label` and `Buttons`. Notice that the `Grid`'s size is explicitly set with its `Width` and `Height` properties. The code for the other containers is similar except they use other controls in place of the `Grid`.

Available for download on Wrox.com

```
<StackPanel Margin="10">
    <Label Content="Grid" HorizontalAlignment="Center"/>
    <Grid Background="Yellow" Width="175" Height="100">
        <Button Content="Button1"/>
```

```
            <Button Content="Button2"/>
        </Grid>
    </StackPanel>
```

SizeInContainers

The following list summarizes the results in the different containers:

- `Grid` — The `Buttons` stretch to fill the `Grid`. The `Buttons` both fill the `Grid` so you only see the one on top: `Button2`.
- `StackPanel` — The `Buttons` stretch to fill the `StackPanel`'s width (because its orientation is vertical) and are as short as they can be while holding their content.
- `Canvas` — The `Buttons` are as small as possible while holding their content. Without other positioning information, the `Canvas` places them both in its upper-left corner so you can only see the one on top: `Button2`.
- `WrapPanel` — The `Buttons` are as small as possible while holding their content. The `WrapPanel` arranges them in order so you can see them both.
- `DockPanel` — By default, the first `Button` is docked to the control's left edge, so it is stretched to fill the `DockPanel` vertically, and it is as thin as possible while still holding its content. By default, the last control is stretched to fill the `DockPanel`'s remaining space.
- `UniformGrid` — The `UniformGrid` divides its space up evenly into cells, places each `Button` in a cell, and stretches them to fill their cells.

The SizeInSingleChildContainers example program shown in Figure 4-2 displays controls inside containers that can hold only a single child. Each container contains a `StackPanel` that holds two `Buttons`.

FIGURE 4-2

The following XAML code shows how the program creates its `Border` container together with its `Label`, `StackPanel`, and `Buttons`. The `StackPanel` has a yellow background so it's easy to see. The code for the other containers is similar except they use other controls in place of the `Border`.

Available for download on Wrox.com

```
<StackPanel Margin="10" Background="LightBlue">
    <Label Content="Border" HorizontalAlignment="Center"/>
    <Border Width="175" Height="100"
     BorderBrush="Black" BorderThickness="3">
        <StackPanel Background="Yellow">
            <Button Content="Button1"/>
            <Button Content="Button2"/>
```

```
        </StackPanel>
    </Border>
</StackPanel>
```

SizeInSingleChildContainers

Notice that the `StackPanel`'s size is not set in the XAML code. Because its `HorizontalAlignment` and `VerticalAlignment` properties default to `Stretch`, the `StackPanel` stretches to fill the `Border` and `ScrollViewer`.

In the `Viewbox`, however, the `StackPanel` remains as small as possible while holding its content. The `Viewbox`'s job is to stretch its contents to fill itself so it doesn't give the `StackPanel` a target area to stretch into. The `StackPanel` remains small, and then the `Viewbox` stretches it to fill the `Viewbox`.

SERIOUS STRETCHING

By default, the `Viewbox` stretches its contents uniformly to be as big as possible while fitting inside the `Viewbox`. You can change this behavior by setting the `Viewbox`'s `Stretch` property. For more information on the `Viewbox` control, see Chapter 6.

The size and position properties are really quite confusing when combined with different kinds of containers. I strongly encourage you to download the SizeInContainers and SizeInSingleChildContainers example programs from the book's web site and experiment with them to get a better sense of how these properties work.

Other Size and Position Properties

While `Width`, `Height`, `HorizontalAlignment`, and `VerticalAlignment` play the biggest roles in determining a control's size, a few other properties play smaller roles.

The `Margin` property determines how much extra space is added around the control. That influences the control's position, and, if the control is stretching to fit its container, it also reduces the control's size.

`Margin` can take the form of one, two, or four numbers. If the value includes a single number, then all four edges of the control are given that much extra space. If the value includes two numbers, the first gives the left and right margins, and the second gives the top and bottom margins. If the value includes four numbers, they give the left, top, right, and bottom margins respectively.

The Margins example program shown in Figure 4-3 displays buttons with different `Margin` values contained in `Grid` controls.

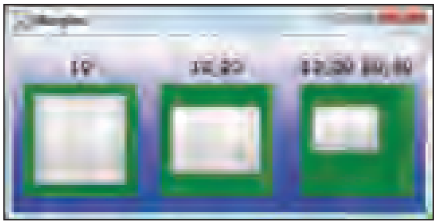


FIGURE 4-3

PERFECT PADDING

Some controls such as `Border`, `Button`, and `TextBox` also provide a `Padding` property. This property has a syntax similar to that of `Margin`, but it specifies padding to add inside the control around its contents. For example, setting `Padding = "10"` adds 10 pixels of extra space around the text inside a `TextBox`.

Four other properties that influence a control's size are `MaxWidth`, `MaxHeight`, `MinWidth`, and `MinHeight`. As you can guess from their names, they determine how big and how small a control can be when its container is resized and its properties make it resize, too.

The MaxMinSizes example program shown in Figure 4-4 displays four rectangles with rounded corners. Labels show the rectangles' `MinHeight` and `MaxHeight` values.

FIGURE 4-4

The first rectangle's `Height` is set to 75, so it always has that height.

The second rectangle's `MaxHeight` value is set to 100. The window is big enough for the rectangle to be taller than this, but its `MaxHeight` value limits it.

The third rectangle's `MinHeight` value is 150. That's too tall to fit completely on the window, so this rectangle is clipped.

The final rectangle does not have any `MinHeight` or `MaxHeight` values, so it grows and shrinks to fit its part of the window.

You might take a few minutes to download the MaxMinSizes example program from the book's web site and run it to see what happens to the rectangles when you change the window's size at run time.

FONT

It makes sense that text displaying controls like `Label` and `TextBox` would provide font properties that let you determine the text's appearance. It's less obvious that some other controls that don't display text provide the same properties.

For example, the `Menu`, `ListBox`, and `Window` classes provide these properties even though they don't directly display text. These objects generally contain other controls that do display text. (`Menus` generally contain `MenuItems`, `ListBoxes` generally contain `ListBoxItems`, and `Windows` can contain anything.) Giving these objects `Font` properties allows the controls they contain to inherit their values, making it easier to give all of those controls the same fonts.

The following list summarizes the most useful font properties:

- `FontFamily` — Determines the text's font face. Some common values include Segoe (pronounced *see-go*, this is Microsoft's preferred font for Windows Vista), Times New Roman, `Courier New`, and Arial. I use Comic Sans MS for many of the figures in this book.

- `FontSize` — Determines the size of the font in pixels. This gives the size of the characters and doesn't include leading and internal space that may be drawn above and below a line of text. You can add the unit specifiers "px" or "pt" to indicate pixels or printer's points ($^1/_{72}$ inch). For example, the value "20pt" means that the text should be 20 points tall.
- `FontStyle` — Determines the text's style. This property can take the values `Normal` and `Italic`.
- `FontWeight` — Determines the text's *density* (whether it's bold or not). This property can take the values (in increasing order of density) `Thin`, `ExtraLight`, `Light`, `Normal`, `Medium`, `SemiBold`, `Bold`, `ExtraBold`, `Black`, and `ExtraBlack`.
  The FontProperties example program shown in Figure 4-5 demonstrates an assortment of font property values. Notice how much taller the 20-point Segoe font is than the 20-pixel Times New Roman font.

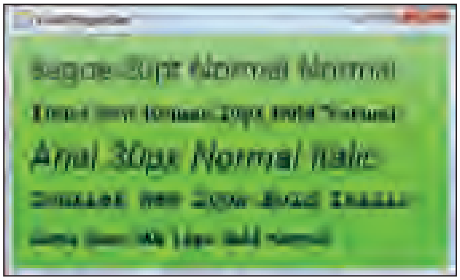


FIGURE 4-5

For a particular font, some `FontWeight` values may produce the same results. For example, on my computer in the Segoe font (and most fonts), the values `Thin` through `Medium` produce a normal font and `SemiBold` through `ExtraBlack` produce the same bold font. In contrast, my computer displays the Arial font's `ExtraBold` through `ExtraBlack` values with a slightly darker bold than the `Bold` value.

The FontWeights example program shown in Figure 4-6 displays text in 20-pixel Segoe and Arial fonts at different weights. You can tell that Arial gives three different weights by looking closely at the widths of the text *Arial* in the column on the right.

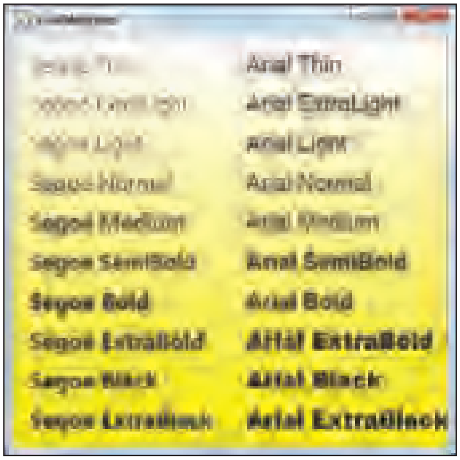


FIGURE 4-6

COLOR

Several properties determine a control's appearance. For drawing controls such as `Ellipse`, `Rectangle`, and `Path`, the `Stroke` property determines the brush used to draw the shape's outline, and the `Fill` property determines how the shape is filled.

For non-drawing controls such as `Label`, `TextBox`, and `StackPanel`, the `Background` property determines how the control's interior is filled, and the `Foreground` property determines how any objects displayed on top of the control appear.

A NEEDLESS INCONSISTENCY

Yes, it might have been nice if these objects consistently used either `Foreground/Background` or `Stroke/Fill` — but that's not the way WPF works.

For example, a `GroupBox`'s `Foreground` property determines the color of its header text, and its `Background` property determines how its client area is filled.

The GroupBoxColors example program shown in Figure 4-7 displays a `GroupBox` with `Foreground = Green` and `Background = LightBlue`. Notice how the property values are inherited by some controls (the `TextBlock`) but not others (the `Label`).

FIGURE 4-7

The following XAML code shows how the GroupBoxColors program creates its controls:

Available for download on Wrox.com

```
<GroupBox Background="LightBlue" Foreground="Green" Header="GroupBox">
    <StackPanel Margin="10">
        <Label Content="Foreground is not inherited by a Label."/>
        <TextBlock Text="Foreground is inherited by a TextBlock."/>
    </StackPanel>
</GroupBox>
```

GroupBoxColors

IMAGE SHAPE

Normally an Image is rectangular because the picture it displays is rectangular, but there are ways that you can give an `Image` control (or any control) another shape.

One method to change a control's shape is to set its `OpacityMask` property to an *opacity mask*. An *opacity mask* is a brush that determines the opacities of the parts of the control.

Note that the red, green, and blue color components used by the opacity mask are irrelevant. Only a pixel's *alpha component*, which determines the pixel's opacity, matters. This alpha value can be between 0 (transparent) and 255 (opaque).

Two common methods for building an opacity mask are using a gradient brush and using an image.

Gradient Opacity Masks

The GradientOpacityMask example program shown in Figure 4-8 uses a radial gradient brush for an opacity mask.

FIGURE 4-8

The following code shows how the program uses its `OpacityMask`:

Available for download on Wrox.com

```
<Image Margin="5" Source="Flowers.jpg" Stretch="None">
    <Image.OpacityMask>
        <RadialGradientBrush>
            <GradientStop Color="#FF833C3C" Offset="0"/>
            <GradientStop Color="#FFE4D6D6" Offset="0.8"/>
            <GradientStop Color="#00FFFFFF" Offset="1"/>
        </RadialGradientBrush>
    </Image.OpacityMask>
</Image>
```

GradientOpacityMask

The code displays an Image that shows a picture. The control's `Image.OpacityMask` property element defines the Image's opacity mask.

The `Image.OpacityMask` element contains a `RadialGradientBrush`. The brush's `GradientStops` define the colors the brush uses.

The first `GradientStop`, with `Offset = 0`, has an opacity of 255 (the FF hexadecimal = 255 decimal in the first color component), so the colors at the center of the brush are opaque. (Remember that the other color components are irrelevant.) When used as an opacity mask, this means that the Image's picture at the center is opaque.

The second `GradientStop` at `Offset = 0.8` also uses an opaque color, so the Image up to 80 percent away from the center is opaque.

The final `GradientStop` at `Offset = 1` uses a completely transparent color. That means that the Image at the edge of the brush is transparent. The brush smoothly shades between the opaque and transparent colors so the parts of the Image between the second and third `GradientStop` smoothly change from opaque to transparent. Figure 4-8 shows the result.

Image Opacity Masks

The second common method for making opacity masks is to use an image.

First, create a mask image that defines opacities for its pixels. If you want a pixel in the result to be transparent, set the alpha value for the corresponding mask pixel to 0. If you want a pixel to be opaque, set the corresponding pixel's alpha value to 255. (As before, the other color components are irrelevant.) Use values between 0 and 255 for partially transparent pixels.

Next, in your WPF application, either in XAML or program code, use the mask image to create an `ImageBrush`.

BUILDING BRUSHES

The "Image Brush" section in Chapter 3 explains how to make an `ImageBrush` in Expression Blend. Chapter 10 has more to say about `ImageBrushes` and `Brushes` in general.

Finally, set the `OpacityMask` for the original image or control to this brush.

The ImageOpacityMask example program shown in Figure 4-9 demonstrates this technique.

The picture on the left of the top row is the original image.

The second picture shows the mask image. It contains an opaque black ellipse on top of a transparent background.

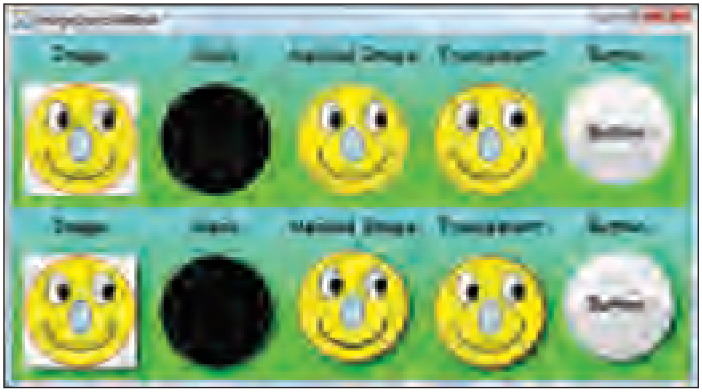


FIGURE 4-9

The third picture shows the original image using the mask image as its `OpacityMask`. Only the parts of the original image that correspond to opaque parts of the mask are visible.

The next picture shows a simpler way to achieve a similar result. This image is similar to the original image except its background pixels are transparent so it doesn't need an opacity mask.

The final item on the first row shows a `Button` using the same opacity mask. Note that the button is still rectangular; you just can't see its corners. If you click next to the round area that's visible where those corners should be, you can click the `Button`.

The following XAML code shows how the program uses the opacity mask with the original image:

Available for download on Wrox.com

```
<Button Content="Button" Height="105" Width="105">
    <Button.OpacityMask>
        <ImageBrush ImageSource="PlainSmileyMask.png" Stretch="None"/>
    </Button.OpacityMask>
</Button>
```

ImageOpacityMask

The bottom row in Figure 4-9 shows the same images displayed in the top row but with drop shadows. Chapter 20 explains bitmap effects such as this in greater detail.

MISCELLANEOUS

WPF controls and objects certainly share a lot of other properties, and I don't want to take up a bunch of space covering them all exhaustively, but there are a few that are so useful they deserve special mention.

The `Name` property identifies a control for use by XAML code and for use by C# or Visual Basic code-behind. You do not need to give a control a name, but if you plan to use it in your code, you should.

The `Parent` property returns a control's logical parent. For example, if you place a `Button` inside a `Grid`, then the `Button`'s parent is the `Grid`. You can use this property to climb around in the application's logical tree and learn about its structure.

WHAT'S IN A NAME?

Sometimes you can get around naming a control, but you probably should give it a name anyway. For example, you can use a `Button`'s `Click` attribute to indicate the event handler that should execute when the user clicks the `Button` without giving the `Button` a name. That makes the code more confusing, however, so I recommend that you give the `Button` a descriptive name (such as `btnExit`) and then you give the event handler a corresponding name if possible (in this case, `btnExit_Click`).

PRACTICAL PARENTHOOD

One useful trick that uses the `Parent` property is to copy a control's property values from its `Parent` so they match.

The `Tag` property can hold an arbitrary object that you can use for anything you want. For example, you could store a number or string here to identify the control (if the control is a `MenuItem` or `ListBoxItem`, it might indicate the item's position in the menu or list box).

You could even use code-behind to store an object in the `Tag` property. For example, you might build a `ListBox` that shows employee names and set each `ListBoxItem`'s `Tag` property to the corresponding `Employee` object.

The `ContextMenu` property determines the context menu that a control displays when the user right-clicks on it. You can use context menus to provide actions that make sense for specific parts of the application's user interface.

The `ToolTip` property determines what text is displayed in a tooltip when the mouse hovers over a control.

The `IsEnabled` property determines whether a control will interact with the user. If `IsEnabled = False`, then the control completely ignores the user. Most controls also change their appearances to indicate that they are disabled (usually by graying themselves out).

ENABLING GROUPS

If you set `IsEnabled` to `True` or `False` for a container such as a `StackPanel` or `GroupBox`, all of the container's children inherit that value. That makes it easy to enable or disable a group of related controls quickly and easily.

Note, however, that the container may not display itself as disabled. For example, a `GroupBox` looks the same whether `IsEnabled` is `True` or `False`.

The `Visibility` property determines whether a control is visible on the screen. This property can take three values: `Visible` (the control is visible as usual), `Hidden` (the control is not visible, but the window's layout saves room for it), and `Collapsed` (the control is not visible, and the window's layout does not include room for it).

SUMMARY

WPF controls have many shared properties. Many you can safely ignore most of the time, but some are important, and a few are confusing. The sizing and positioning properties `Width`, `Height`, `HorizontalAlignment`, and `VerticalAlignment` play particularly important and confusing roles in determining a control's size and location.

This chapter describes the most important of these common properties and helps demystify the sizing and positioning properties. Later chapters provide additional details as they cover specific properties in depth.

This chapter covers properties that apply to every kind of control. The following four chapters turn to specific groups of controls. They describe the controls you can use to display information, arrange other controls, interact with the user, and draw shapes. These chapters describe specific controls and discuss their capabilities. After you read these chapters, you'll know just what controls to use under different circumstances.

5

Content Controls

WPF provides a nice assortment of controls that let the user view, modify, and otherwise interact with a program's data. This chapter describes one group of those controls: content controls.

Content controls are primarily intended to hold content that the user should see. They display something that the user should view but generally won't modify.

In contrast, Chapter 6 describes layout controls that arrange the controls that they contain, and Chapter 7 describes user interaction controls that let the user interact with the application.

CONTROL CATEGORIES

Many controls fall into multiple categories. For example, a `TextBox` can display output, but the user can also enter values in it. Similarly, a `ListBox` can display items and even highlight a particular item, but the user can also select items in the list.

In this chapter and the next two, if a control usually contains or arranges other controls (e.g., a `StackPanel` or `TabControl`), then it's a layout control and is described in Chapter 6. If the control lets the user enter data, adjust something, select an item, or otherwise manipulate the program, then it's a user interaction control and is described in Chapter 7.

(I thought long and hard about the `ListView` and `TreeView` because they arrange their contents in very specific ways such as grid-like and hierarchical displays. When you use them in their simplest forms, however, they display only text and the controls they contain are mostly hidden, so I finally decided to include them here.)

Even the simplest of controls provides a multitude of properties, methods, and events. To be as useful as possible, this chapter doesn't cover every last detail of every control.

Instead, it focuses on what each control does and how you can use it in your applications. It also describes the properties, methods, and events that are most important for a particular control.

PROPERTIES, METHODS, AND EVENTS

In case you're not familiar with object-oriented terminology, all objects provide three kinds of features: properties, methods, and events.

A *property* is a value that determines the object's appearance or behavior. For example, a control's `Background` property determines how it is filled, and its `IsEnabled` property determines whether the user can interact with the control. In XAML code, you use attributes and object property syntax to set properties.

A *method* is a routine that a control can execute to do something. For example, a `TextBox`'s `Clear` method makes the control erase any text that it contains, and a `Window`'s `Close` method makes the `Window` close itself. XAML code cannot invoke an object's methods.

An *event* is something that a control *raises* to let your program know that something important has happened. Your program can *catch* the event and take appropriate action. For example, a `TextBox` raises a `TextChanged` event whenever its text has changed, and a `Button` raises a `Click` event whenever it is clicked. XAML code can indicate that a particular code-behind routine should handle an event. For example, the `Click` attribute at the end of the following XAML statement means that the routine `btnClear_Click` should handle the `Button`'s `Click` event.

```
<Button Name="btnClear" Content="Button" Click="btnClear_Click" />
```

Note that this book does not cover every control and object provided by WPF. WPF includes a huge number of objects, many of which you don't work with directly. For example, the following XAML code assigns a `ToolTip` object to a `TextBox` control's `ToolTip` property:

```
<TextBox ToolTip="The customer's ZIP code."/>
```

The following code does the same thing with a separate `ToolTip` object. Which do you think is easier?

```
<TextBox>
    <TextBox.ToolTip>
        <ToolTip>
            The customer's ZIP code.
        </ToolTip>
    </TextBox.ToolTip>
</TextBox>
```

Figure 5-1 shows the ToolTip at run time.

Because the first method is easier, this book only covers objects like `ToolTip` used by the second method very briefly or not at all.

FIGURE 5-1

CONTROL OVERVIEW

The following table briefly lists the controls described in this chapter together with their purposes. You can use this table to help decide which control you need for a particular purpose.

CONTROL	PURPOSE
`Border`	Draws a border around or background behind a single child control.
`BulletDecorator`	Displays an item and bullet in a bulleted list.
`DocumentViewer`	Displays `FixedDocument` content such as XPS (XML Paper Specification) files.
`FlowDocumentPageViewer`	Displays a `FlowDocument` in page viewing mode.
`FlowDocumentReader`	Displays a `FlowDocument` in Page, Scroll, or TwoPage mode.
`FlowDocumentScrollViewer`	Displays a `FlowDocument` in Scroll mode.
`GroupBox`	Displays a header and a border around a single child.
`Image`	Displays a graphical image.
`Label`	Displays text that the user cannot modify.
`ListView`	Displays a set of items in one of several layouts such as a grid or as icons.
`MediaElement`	Plays an audio or video file.
`Popup`	Displays an area floating over a window much as a `ContextMenu` or `ToolTip` does.
`ProgressBar`	Displays progress information to the user.
`Separator`	Displays a horizontal separator line in a menu.
`TextBlock`	Displays read-only text much as a `Label` does but with additional features such as line wrapping, italics, and bold text.
`ToolTip`	Displays a tooltip when the mouse hovers over its parent.
`TreeView`	Displays hierarchical data in a tree form.

The following sections describe these controls in greater detail and provide XAML examples demonstrating the controls. The controls are grouped into three categories: graphical, textual, and spatial.

GRAPHICAL CONTROLS

The main purpose of the graphical controls is, as you can probably guess, to display something graphical. This includes lines and shapes, images, and multimedia output.

Usually you can completely specify these controls at design time and don't need to modify them at run time. They also usually don't interact with the user.

Image

The `Image` control displays a graphical image.

To easily use an `Image` control, first add the file to the project. In either Expression Blend or Visual Studio, open the Project menu, select "Add Existing Item," find the file, and click Open.

Now create an `Image` control and set its `Source` property to the file.

The `Image` control's two other most useful properties are `Stretch` and `StretchDirection`. `Stretch` can take the following values:

- `None` — The picture is not stretched. It is displayed at its original size.
- `Fill` — The picture is stretched to fill the `Image` control, possibly distorting the picture.
- `Uniform` — The picture is stretched by the same amount vertically and horizontally to make it as big as possible while still fitting in the `Image` control.
- `UniformToFill` — The picture is stretched by the same amount vertically and horizontally until it completely fills the `Image` control. If the picture doesn't have the same width-to-height ratio as the `Image` control, then some of the picture will stick out past the edges of the control and will be clipped.

The `StretchDirection` property determines whether the picture can be enlarged or shrunk to satisfy the `Stretch` property. `StretchDirection` can take the following values:

- `UpOnly` — The picture can only be stretched to make it larger.
- `DownOnly` — The picture can only be shrunk to make it smaller.
- `Both` — The picture can be stretched or shrunk.

FIGURE 5-2

The following XAML code creates the `Image` control shown in Figure 5-2:

Available for download on Wrox.com

```
<Image Width="Auto" Height="Auto"
 HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
 Source="Flatirons.jpg" Stretch="UniformToFill"/>
```

UseImage

MediaElement

The `MediaElement` control plays an audio or video file. At design time, the control's most important property is `Source`, which gives the name of the media file to play.

By default, when the `MediaControl` loads, it begins playing its media file. Unless you want it to play some sort of startup sound or video, that's probably not what you want. Instead, you probably want the program to control the media.

To let the program control the media, set the control's `LoadedBehavior` property to `Manual`. That prevents the control from playing when it loads and lets the program take charge.

Next, write code to call the control's methods to play the media.

The following XAML code defines two `MediaElement` controls, one that plays a .wmv video file and one that plays a .wav audio file. Notice that `LoadBehavior` is set to `Manual`.

Available for download on Wrox.com

```
<MediaElement x:Name="mmJulia" Source="EvolvingJuliaSet.wmv"
 LoadedBehavior="Manual"
 Margin="283.556,24,0,0" HorizontalAlignment="Left"
 VerticalAlignment="Top" Height="88" />

<MediaElement x:Name="mmAudio" Source="Windows XP Startup.wav"
 LoadedBehavior="Manual"
 Margin="0,116,106.444,28" HorizontalAlignment="Right"
 Width="114" />
```

UseMediaElement

STRANGE SIZES

While working on one program, I set an `Image` control's `Stretch` property to `None`, and the (fairly large) picture shrank to postage-stamp size. I reassigned the picture and got the same result. I opened the picture file in Microsoft Paint, and it was at its normal size, but when I opened it in an `Image` control, it was tiny.

It turned out that I had taken that picture with a digital camera. The imaging software that came with it expected me to print the image on a printer so it set the image's resolution to 480 dpi (dots per inch), which is much higher than the resolution of the computer screen. The `Image` control was smart enough (or stupid enough, depending on how you look at it) to realize that the image was intended to display more pixels per inch, so it resized the image accordingly.

I solved the problem by opening the file in Microsoft Paint, selecting all of the pixels, and copying and pasting them into a new image. Fancier image processing applications such as Paint Shop Pro can also set an image's dpi settings.

The following code shows how the `UseMediaElement` example program controls the video control:

Available for download on Wrox.com

```
// Play the video file.
private void btnPlayVideo_Click(object sender, RoutedEventArgs e)
{
    mmJulia.Play();
}

// Pause the video file.
private void btnPauseVideo_Click(object sender, RoutedEventArgs e)
{
    mmJulia.Pause();
}

// Rewind the video file.
private void btnRewindVideo_Click(object sender, RoutedEventArgs e)
{
    mmJulia.Position = new TimeSpan(0);
}
```

UseMediaElement

Figure 5-3 shows the UseMediaElement program in action.

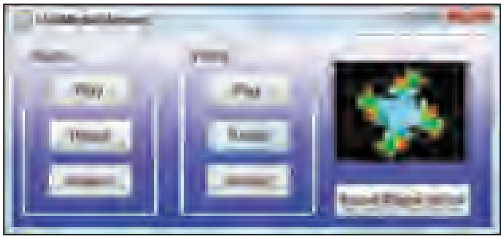


FIGURE 5-3

Unfortunately, the `MediaElement` control doesn't provide any dependency properties that let a XAML file control its playback. For example, it doesn't provide an `IsPlaying` or `State` property that XAML code could set to make the control play or pause. That means you need to write code-behind to control the media.

However, XAML does include a `SoundPlayerAction` command that lets you play an audio file to the end without pausing or stopping. You can write XAML trigger code to invoke this command when an event takes place.

The following XAML code defines a trigger that executes when the `btnSoundPlayerAction` button fires its `Click` event. When the code executes, the `SoundPlayerAction` plays the file notify.wav.

Available for download on Wrox.com

```
<Window.Triggers>
    <EventTrigger RoutedEvent="ButtonBase.Click"
     SourceName="btnSoundPlayerAction">
        <SoundPlayerAction Source="notify.wav"/>
    </EventTrigger>
</Window.Triggers>
```

UseMediaElement

MEDIA MANIPULATIONS

It seems like an odd omission that the `MediaElement` has no properties that let XAML code control the media. If you search the Web, you can find people who have written controls that wrap `MediaElement` and provide the properties you need. It seems likely that Microsoft will add these properties to the `MediaElement` at some point.

TEXTUAL CONTROLS

The textual controls deal primarily with displaying text. While you can place other things inside some of these controls, they are most often used to display text. For example, you can place a `Button` inside a `Label` but normally a `Label` displays text.

The `FixedDocument` and `FlowDocument` objects can contain all sorts of things such as pictures, tables, and controls in addition to text, but they are still basically intended to display text formatted in various ways so I have grouped them here.

DocumentViewer

The `FlowDocument` object (described later in this chapter) displays information that can rearrange itself as needed at run time. Just as a web page displayed in a browser can reflow text and pictures when the browser resizes, a `FlowDocument` can rearrange its contents when it is resized.

In contrast, a `DocumentViewer` displays fixed content that does not reflow at run time. The content in a `DocumentViewer` is fixed with every item positioned exactly as it will appear to the user, much as the content of a PDF file is fixed.

The following code fragment creates a `FixedDocument` object containing two pages of content. The surrounding code and most of the actual content have been omitted to save space. The styles (which have also been omitted) set properties such as font sizes for headings and paragraph text.

Available for download on Wrox.com

```
<DocumentViewer Width="Auto" Height="Auto"
 HorizontalAlignment="Stretch" VerticalAlignment="Stretch">
    <DocumentViewer.Background>
        <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
            <GradientStop Color="#FF000000" Offset="0"/>
            <GradientStop Color="#FF06B7FF" Offset="1"/>
        </LinearGradientBrush>
    </DocumentViewer.Background>
    <FixedDocument>
        <PageContent>
            <FixedPage Width="600" Height="400">
                <StackPanel Width="500" Height="300"
                 Background="#FFFFFF90" Margin="50">
                    <Label Style="{StaticResource styHeader}">
                        FixedDocument Elements
                    </Label>
```

```
                    <TextBlock Style="{StaticResource styPara}">
                        Unlike the FlowDocument, a FixedDocument
                        contains items that cannot be moved by the
                        user at run time.
                        It is useful for creating precisely
                        Positioned documents in a manner similar
                        to PDF files.
                    </TextBlock>
                     ... Other TextBlocks and Labels omitted...
                </StackPanel>
            </FixedPage>
        </PageContent>
        <PageContent>
            <FixedPage Width="600" Height="400">
                <StackPanel Width="500" Height="300"
                 Background="#FFFFFF90" Margin="50">
                 ... Other TextBlocks and Labels omitted...
                </StackPanel>
            </FixedPage>
        </PageContent>
    </FixedDocument>
</DocumentViewer>
```

UseDocumentViewer

The code begins with a `DocumentViewer` that contains a `FixedDocument`. The `FixedDocument` object contains two `PageContent` objects representing the program's two pages.

Each `PageContent` object holds a `FixedPage` that contains other WPF controls that create the actual content.

Figure 5-4 shows the result. The `DocumentViewer` lets the user zoom in and out, print, and display two pages (as shown in Figure 5-4).

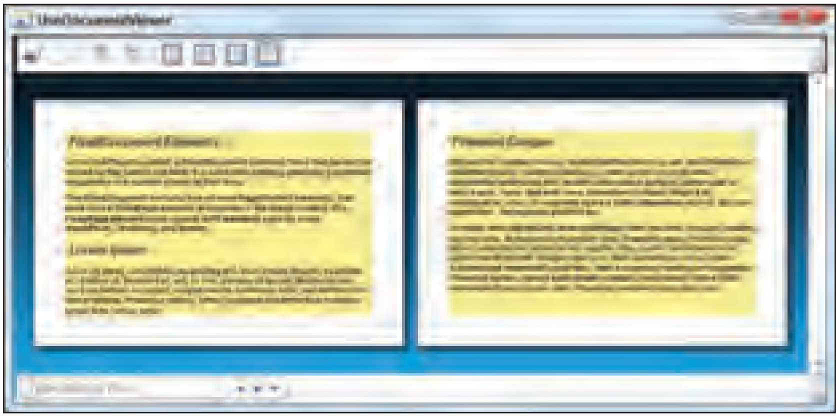


FIGURE 5-4

FlowDocument

The `FlowDocument` control contains objects that make up a document. These objects may include:

- Paragraphs
- Tables
- Lists
- Floaters
- Figures
- User interface elements such as `Buttons` and `TextBoxes`
- Three-dimensional (3D) objects

For more information on building `FlowDocuments`, see Chapter 21.

The following three sections describe controls that display `FlowDocuments` in different styles. If you place a `FlowDocument` directly in a `Window`, then the control acts as if it were inside a `FlowDocumentReader`. See the section describing that control for information about how it behaves.

FlowDocumentPageViewer

The `FlowDocumentPageViewer` displays a `FlowDocument` in page viewing mode. In this mode, the user can see a single page at a time, and controls at the bottom of the viewer allow the user to move to other pages.

The size of a page and the amount displayed by the viewer are determined by the `FlowDocument`'s `PageHeight` and `PageWidth` properties, and by the viewer's size.

By default, the document scales to fit the viewer's width. Then a page is defined by however much of the document can fit vertically in the viewer at that width. If you set the document's `PageHeight` and `PageWidth` properties, then the viewer may chop the document into more pages to honor those values.

Figure 5-5 shows a `FlowDocumentPageViewer` displaying a `FlowDocument`. The arrows in the viewer's bottom center let you scroll through the document's pages. The plus and minus buttons on the bottom right let you zoom in and out.

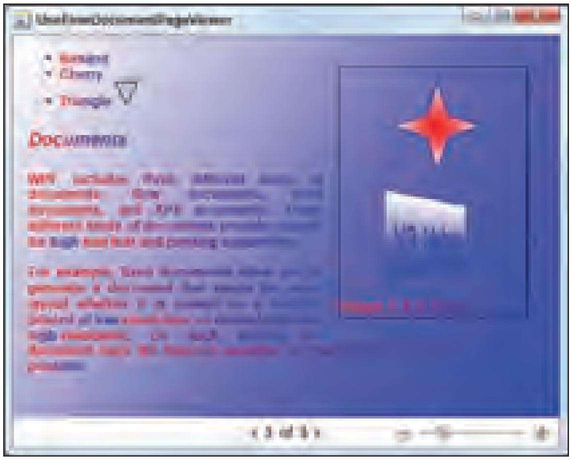


FIGURE 5-5

FlowDocumentReader

The `FlowDocumentReader` displays a `FlowDocument` in one of three modes: Page, Scroll, or TwoPage.

In Page mode, the control works like a `FlowDocumentPageViewer`. See the preceding section for information about that control.

In Scroll mode, the control works like a FlowDocumentScrollViewer. See the following section for information about that control.

In TwoPage mode, the control displays the document two pages at a time side-by-side. As is the case with the FlowDocumentPageViewer, the size of a page is determined by the document's PageHeight and PageWidth properties and the available space in the viewer. By default, the viewer divides its area in two, scales the document to fit the width of the two halves, and displays as much of the document as will fit at that width for each page.

Figure 5-6 shows a FlowDocumentReader displaying a FlowDocument in TwoPage mode. The arrows in the viewer's bottom center let you move between pages. The icons to the right let you pick between Page, TwoPage, and Scroll mode. The plus and minus buttons on the bottom right let you zoom in and out.

FIGURE 5-6

FlowDocumentScrollViewer

The FlowDocumentScrollViewer displays a FlowDocument as a tall, continuously scrolling document much as a web browser displays the contents on a long web page.

Figure 5-7 shows a FlowDocumentScrollViewer displaying a FlowDocument.

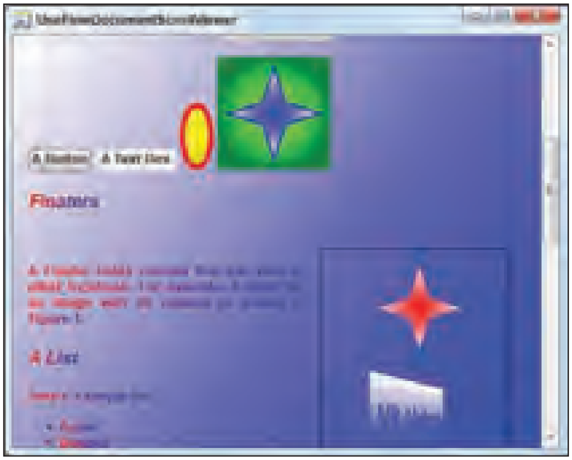


FIGURE 5-7

Label

The Label control normally displays text that the user cannot modify, although you can give the Label control a different kind of child such as a Grid or Button if you want to for some strange reason.

When used to display ordinary text, the `Label` control is relatively simple. Its `Content` property determines the text that it displays. It provides `Background` and `Foreground` properties to determine its colors. Standard font properties (`FontFamily`, `FontSize`, `FontWeight`, etc.) determine the appearance of the text.

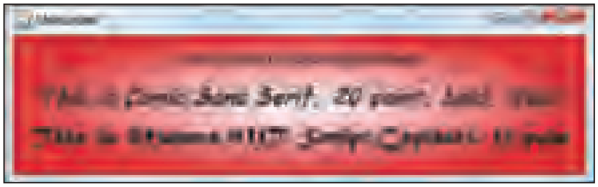


FIGURE 5-8

The following XAML code creates the middle `Label` control shown in Figure 5-8:

Available for download on Wrox.com

```
<Label Content="This is Comic Sans Serif, 20 point, bold, italic"
 FontFamily="Comic Sans MS" FontSize="20"
 FontWeight="Bold" FontStyle="Italic"
 HorizontalAlignment="Center"/>
```

UseLabel

LABEL OR TEXTBOX?

Usually if you want to display text that the user cannot modify, you should use a `Label` (or a `TextBlock`, described later in this chapter). However, sometimes you can make the user's life easier by displaying read-only text in a `TextBox` instead.

The `Label` control won't let the user change the text it displays, but it won't let the user select that text either. That means that the user cannot copy and paste the text.

If you want to prevent the user from changing some text but you want to allow copy and paste, consider using a `TextBox` with its `IsReadOnly` property set to `True`. For example, if a program's About dialog displays the application's version and serial numbers, you might want to allow the user to copy those values but not modify them.

Pop-Up

The `Popup` control displays an area floating over a window much as a `ContextMenu` or `ToolTip` does. However, the `Popup` does not automatically display itself as the `ContextMenu` and `ToolTip` do. Instead, the program must set the control's `IsOpen` property to display and hide the pop-up.

PAINLESS POP-UPS

The `ContextMenu` and `ToolTip` controls are easier to use than a `Popup` because they automatically display themselves when appropriate. If you only need features that a `ContextMenu` or `ToolTip` can provide, use those controls instead of a `Popup`.

The following table describes properties that determine where the pop-up appears when it is visible:

PROPERTY	PURPOSE
`PlacementTarget`	The object relative to which the `Popup` is positioned. Typically, this is a control, and the `Popup` is placed next to it. In XAML code, this might look like: `PlacementTarget="{Binding ElementName=image1}"`. If you omit this property, then the `Popup` is positioned relative to its container.
`Placement`	Determines how the `Popup` is positioned relative to the `PlacementTarget` object. See the following text for more information.
`PlacementRectangle`	If you include this, it determines a rectangle within the `PlacementTarget` relative to which the `Popup` should be positioned.
`HorizontalOffset`	Sets a horizontal offset for the `Popup`'s placement.
`VerticalOffset`	Sets a vertical offset for the `Popup`'s placement.

The UsePopup example program shown in Figure 5-9 demonstrates the `PlacementTarget` and `IsOpen` properties. When the mouse enters a small picture, the program's code-behind uses those properties to display the larger picture in a `Popup`.

FIGURE 5-9

The following XAML code shows how the program creates its interface:

Available for download on Wrox.com

```
<StackPanel>
    <StackPanel Orientation="Horizontal" Height="35" Margin="4">
        <Label Width="200" Content="George Washington"/>
        <Image Name="img1" Stretch="Uniform"
         Source="GeorgeWashington.jpg"
         MouseEnter="img_MouseEnter"
         MouseLeave="img_MouseLeave" />
    </StackPanel>
    ... Other rows omitted ...
    <Popup Name="popFullScale" Placement="Right"
     HorizontalOffset="5" VerticalOffset="5">
        <Image Name="imgFullScale" Stretch="None"/>
    </Popup>
</StackPanel>
```

UsePopup

The `Window` contains a `StackPanel` that holds a series of horizontal `StackPanels`. Each of those contains a `Label` and an `Image` that shows the small preview picture.

After the horizontal entries, the main `StackPanel` holds a `Popup` control that contains an `Image`.

The program uses the following code to show and hide its `Popup` control:

Available for download on Wrox.com

```
// The mouse has entered an image. Display the popup.
private void img_MouseEnter(object sender, MouseEventArgs e)
{
    Image img = (Image)sender;
    imgFullScale.Source = img.Source;

    popFullScale.PlacementTarget = img;
    popFullScale.IsOpen = true;
}

// The mouse has left an image. Hide the popup.
private void img_MouseLeave(object sender, MouseEventArgs e)
{
    popFullScale.IsOpen = false;
}
```

UsePopup

When the mouse enters an `Image` control, the `img_MouseEnter` event handler figures out which `Image` raised the event. It sets the `Popup`'s `Image` control to display the same image as the small preview control but at full scale. It sets the `Popup`'s `PlacementTarget` to the control that raised the event and sets `IsOpen` to `True`.

When the mouse leaves an `Image` control, the `img_MouseLeave` event handler simply sets the `Popup` control's `IsOpen` property to `False`.

The following table describes the values that the `Placement` property can take:

VALUE	POSITION IS RELATIVE TO:
`Absolute`	Upper-left corner of the screen. If the pop-up won't fit, it is moved to the edge of the screen.
`Relative`	Upper-left corner of `PlacementTarget`
`Bottom`	Lower edge of `PlacementTarget`
`Center`	Center of `PlacementTarget`
`Right`	Right edge of `PlacementTarget`
`AbsolutePoint`	Upper-left corner of the screen. If the pop-up won't fit, it is extended from an axis defined by `HorizontalOffset` or `VerticalOffset`.
`RelativePoint`	Upper-left corner of `PlacementTarget`. If the pop-up won't fit, it is extended from an axis defined by `HorizontalOffset` or `VerticalOffset`.
`Mouse`	The mouse's position

continues

(continued)

VALUE	POSITION IS RELATIVE TO:
`MousePoint`	The mouse's position. If the pop-up won't fit, it is extended from an axis defined by `HorizontalOffset` or `VerticalOffset`.
`Left`	Left edge of `PlacementTarget`
`Top`	Top edge of `PlacementTarget`
`Custom`	A position specified by a `CustomPopupPlacementCallback`

The PopupPlacement example program shown in Figure 5-10 demonstrates the most common `Placement` values. All four pop-ups have their `PlacementTarget` values set to the `Image` control.

FIGURE 5-10

PUSHY POP-UPS

Pop-ups float over the application. If you move the application, they remain where they were originally positioned and do not move with the application.

In fact, pop-ups tend to float above all other applications, too. If you switch to another program, the pop-ups remain visible floating over the new program.

By default, a pop-up remains visible until its `IsOpen` property is set to `False`. If you set the control's `StaysOpen` property to `False`, then the `Popup` automatically captures the mouse and hides when a mouse event occurs outside of the `Popup`. It also hides if you switch focus to another application.

The final `Popup` property mentioned here is `PopupAnimation`. You can set this property to `None` (the pop-up simply appears), `Fade` (the pop-up fades in and out), `Scroll` (the pop-up scrolls in from its upper-left corner), and `Slide` (the pop-up slides down from the top or up from the bottom if it won't fit sliding down). To use any of these values other than `None`, you must set the control's `AllowsTransparency` property to `True`.

TextBlock

The `TextBlock` displays read-only text much as a `Label` does but with greater flexibility. The `TextBlock` has additional features that let you:

- Change some text's appearance inline by making it **bold**, *italic*, or underlined.
- Add line breaks.
- Change line spacing.
- Expand or condense the text.
- Truncate text with an ending ellipsis.
- Apply text decorations such as ~~strikethrough~~, baseline, underline, and overline.
- Use superscripts and subscripts.

The following table summarizes some of the more useful properties that the `TextBlock` class itself provides.

PROPERTY	PURPOSE
`LineHeight`	Determines the spacing between lines.
`LineStackingStrategy`	Can be `BlockLineHeight` (line height is determined by `LineHeight`) or `MaxHeight` (each line's height is set to fit its contents)
`TextTrimming`	Determines how the `TextBlock` handles words that don't fit. Can be `None` (words are truncated), `WordEllipsis` (text is broken at a word boundary and replaced with an ellipsis), or `CharacterEllipsis` (text is broken at a character boundary and replaced with an ellipsis). See Figure 5-11.
`TextWrapping`	Determines whether text is wrapped at the end of the `TextBox`. This can be `NoWrap`, `Wrap`, or `WrapWithOverflow`. If the text contains a very long word and the control cannot figure out how to break the text into lines, then `WrapWithOverflow` may make part of the long word stick out beyond the edge of the `TextBlock` (and get chopped off), whereas `Wrap` will break the word in the middle and continue it on the next line.

Each row in Figure 5-11 demonstrates a different `TextTrimming` value. The `TextBlocks` on the right all contain the text *C:\Program Files\CoolApplications* and have their `TextWrapping` properties set to `NoWrap`. When `TextTrimming` is `CharacterEllipsis`, the control ends the text at the last character that can fit and finishes with an ellipsis. When `TextTrimming` is `WordEllipsis`, the control ends the text at the last word break that can fit and finishes with an ellipsis. When `TextTrimming` is `None`, the control simply truncates the text.

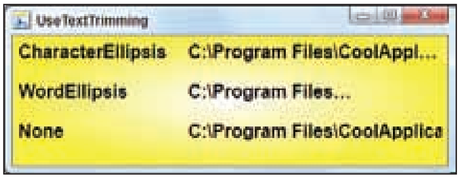


FIGURE 5-11

In addition to ordinary text, the `TextBlock` can contain *inlines*, objects that change the text's appearance for a short while. The following table summarizes the most useful inlines.

INLINE	PURPOSE
`Bold`	Makes the enclosed text bold.
`Hyperlink`	Displays text as a hyperlink. Can raise `Click` events and can automatically navigate if contained in a navigation host such as a NavigationWindow, Frame, or browser.
`InlineUIContainer`	Holds other controls.
`Italic`	Makes the enclosed text italic.
`LineBreak`	Starts a new line.
`Run`	Contains a run of text. The `Run` inline also has its own properties such as `Foreground`, `Background`, and `FontSize`, so you can use it to change many text features.
`Span`	Groups other inline elements. Like `Run`, `Span` has properties that can greatly change the enclosed text's appearance.
`Underline`	Makes the enclosed text underlined.

For example, the following XAML code italicizes the word *much*:

```
<TextBlock>
    A TextBlock gives <Italic>much</Italic> greater control
    than a Label.
</TextBlock>
```

A `TextBlock` can also directly contain some controls, which it includes within the flow of the text.

The following XAML code demonstrates many `TextBlock` features. The result is shown in Figure 5-12.

Available for download on Wrox.com

```
<TextBlock HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
 FontSize="16" LineHeight="30" Margin="0,0,0,0"
 TextWrapping="Wrap" TextTrimming="WordEllipsis"
 LineStackingStrategy="BlockLineHeight">
    The text in a TextBlock can include <Italic>inlines</Italic>
    that make the text <Bold>bold</Bold>, <Italic>italic</Italic>,
    or <Underline>underline</Underline>.
    They can also provide
    <Hyperlink>hyperlinks</Hyperlink>
    and can include
    <LineBreak/>
    line breaks and even controls such as
    <Button Content="Buttons"/>.
    They can include both
    <Run BaselineAlignment="Superscript"
```

```
            FontSize="12">superscripts</Run>
        and <Run BaselineAlignment="Subscript"
            FontSize="12">subscripts</Run>.
        <LineBreak/>
        <Run Foreground="Red" FontWeight="Bold">
            <Run.Background>
                <LinearGradientBrush>
                    <GradientStop Offset="0" Color="White"/>
                    <GradientStop Offset="1" Color="Orange"/>
                </LinearGradientBrush>
            </Run.Background>
            You can use a Run to give text different properties such as
            Foreground and Background colors.
        </Run>
    </TextBlock>
```

UseTextBlock

ToolTip

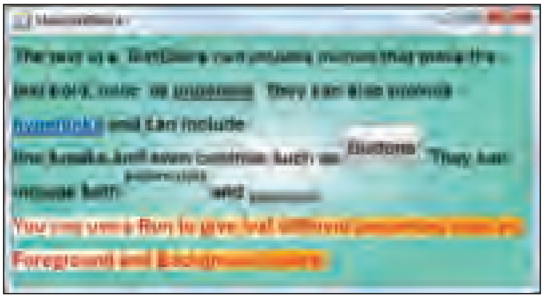


FIGURE 5-12

The ToolTip displays a tooltip for another object. Figure 5-13 shows a program displaying a tooltip for the "First Name" textbox that's sitting under the mouse.

While you can give a control a tooltip by building a ToolTip object, it's generally easier to simply set the control's ToolTip property as shown in the following code:

Available for download on Wrox.com

```
<TextBox Margin="80,10,12,0" VerticalAlignment="Top"
 Text="Prog" ToolTip="The customer's first name"
 Name="txtFirstName" Height="21.96" />
```

UseToolTip

SPATIAL CONTROLS

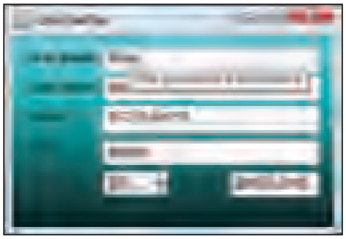


FIGURE 5-13

The spatial controls have an important spatial component. Some, such as BulletDecorator and GroupBox, provide spatial arrangement and decoration for other controls. Others, such as ListView and TreeView, use spatial positioning to show relationships among other objects.

Border

The Border control displays a border or background. It's a spatial control in the sense that it creates a space to hold its child. While that may seem like a minor accomplishment, creating separate areas on a window can be very useful for defining groups of controls. For example, if a window contains many radio buttons or checkboxes, grouping them with `Borders` can make the window easier to read and understand.

The control can have only a single child, but that child can be a container such as a `Grid` or `StackPanel`, so this isn't really much of a restriction.

The following table summarizes the `Border` control's most useful properties:

PROPERTY	PURPOSE
`Background`	The control's background brush
`BorderBrush`	The brush used to draw the `Border`'s edges
`BorderThickness`	Determines the thicknesses of the `Border`'s edges. If this is a single value, then all four edges use the same thickness.
`CornerRadius`	Determines the radii of curvature for the `Border`'s four rounded corners. If this is a single value, then all four corners use the same radius. Set this to 0 for square corners.

The following XAML code creates the `Border` shown in Figure 5-14:

Available for download on Wrox.com

```
<Border HorizontalAlignment="Center" VerticalAlignment="Center"
 Width="150" Height="100"
 CornerRadius="20"
 BorderBrush="#FFFF8000" BorderThickness="5">
    <Border.BitmapEffect>
        <DropShadowBitmapEffect/>
    </Border.BitmapEffect>
    <Border.Background>
        <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
            <GradientStop Color="#FF00FF00" Offset="0"/>
            <GradientStop Color="#FF004000" Offset="1"/>
        </LinearGradientBrush>
    </Border.Background>
    <TextBlock Width="100" Height="50" FontWeight="Bold"
     TextWrapping="Wrap" TextAlignment="Center"
     Text="This TextBlock is contained inside the Border."/>
</Border>
```

UseBorder

FIGURE 5-14

BulletDecorator

The `BulletDecorator` displays an item and bullet in a bulleted list. The control can have only a single child (other than the bullet), but that child can be a container such as a `Grid` or `StackPanel`, so this isn't really much of a restriction.

The `BulletDecorator` control's most important property is `Bullet`, which should contain the object to be used as the item's bullet. Normally this should be something that looks like a bullet such as a small image or character, but in theory it can be something else such as a label or button.

FIGURE 5-15

The following XAML code creates the first item in the bulleted list shown in Figure 5-15. The three `BulletDecorator` controls are contained in a `StackPanel` to make the list.

Available for download on Wrox.com

```
<BulletDecorator>
    <BulletDecorator.Bullet>
        <Image Width="20" Height="20"
         Source="Bullet.bmp" Stretch="None"
         HorizontalAlignment="Left" VerticalAlignment="Center" >
            <Image.BitmapEffect>
                <DropShadowBitmapEffect/>
            </Image.BitmapEffect>
        </Image>
    </BulletDecorator.Bullet>
    <Label Content="Choice 1" Foreground="Red"/>
</BulletDecorator>
```

UseBulletDecorator

GroupBox

A `GroupBox` displays a header and a border around a single child. Like the `Border`, the `GroupBox` is useful for creating distinct areas on a window.

The `GroupBox`'s `Foreground` property determines the brush used to draw the header text, while its `BorderBrush` property determines the brush used to draw the border. The `Background` property determines how the `GroupBox` is filled.

BROKEN BORDERS

For some reason, the `GroupBox`'s border is partially obscured if the control's `BorderThickness` property is larger than `2`.

Note also that you can hide the border by setting `BorderThickness = 0`.

The following XAML code fragment creates the `GroupBox` shown in Figure 5-16. The `GroupBox` contains a `Grid` that holds the program's `Label` and `TextBox` controls. To save space, they are not shown in the following code:

Available for download on Wrox.com

```
<GroupBox HorizontalAlignment="Stretch" VerticalAlignment="Stretch"
 Header="Customer Information" Margin="8" BorderBrush="#FFD5DFE5">
    <Grid Margin="5">
        ...
    </Grid>
</GroupBox>
```

UseGroupBox

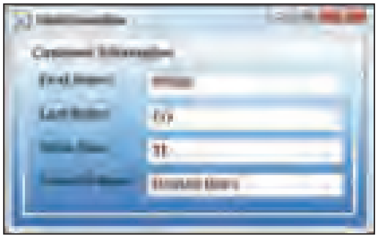


FIGURE 5-16

ListView

The `ListView` displays a set of data items in one of several layouts. You can define new custom layouts for the control, but probably the most common standard layout is the grid-like view shown in Figure 5-17.

FIGURE 5-17

Although displaying a fixed set of data in a grid with resizable columns may sometimes be useful, the real power of the `ListView` is its ability to display data from a data source such as a database or a list of objects.

Data binding is covered in greater detail in Chapter 18; this section covers only a few of the basics so you can see how the `ListView` works.

To really understand this example, you need to jump back and forth between the code-behind that defines the data and the `ListView` control's XAML code.

The following C# code shows how the UseListView example program defines its `BookInfo` class. The class's constructor simply initializes the new object's four properties: `Author`, `Title`, `Year`, and `Price`. Since the properties are all implemented in a very similar way, only the code for the `Author` property is shown here.

Available for download on Wrox.com

```
public class BookInfo
{
    // Initialize a new object's fields.
    public BookInfo(string new_Author, string new_Title,
        string new_Year, string new_Price)
    {
```

```
            Author = new_Author;
            Title = new_Title;
            Year = new_Year;
            Price = new_Price;
        }

        private String m_Author;
        public string Author
        {
            get { return m_Author; }
            set { m_Author = value; }
        }

        ... Code for the other properties omitted ...
    }
```

UseListView

When the program starts, it uses the following code to create an `ObservableCollection` of `BookInfo` objects containing the data that should be displayed. It then sets the `ListView` control's `DataContext` property to the collection so that the control has access to all of the data.

Available for download on Wrox.com

```
// Create a list of BookInfos.
ObservableCollection<BookInfo> books =
    new ObservableCollection<BookInfo>();
books.Add(new BookInfo("Daniel Pinkwater",
    "5 Novels", "1997", "9.56"));
books.Add(new BookInfo("Glen Cook",
    "Cold Copper Tears (Garrett Files)", "2007", "6.99"));
books.Add(new BookInfo("Simon R. Green",
    "The Man With the Golden Torc", "2008", "7.99"));
books.Add(new BookInfo("Tad Williams",
    "The War of the Flowers", "2004", "8.99"));
books.Add(new BookInfo("Tom Holt",
    "You Don't Have to Be Evil to Work Here, But it Helps",
    "2007", "10.00"));

// Set the ListView's data context to this list.
lvwBooks.DataContext = books;
```

UseListView

In addition to assigning basic properties such as `Name` and `Background` to the `ListView`, the XAML code must tell the control what pieces of data to display where. The following XAML code fragment creates the `ListView` shown in Figure 5-17:

Available for download on Wrox.com

```
<ListView Name="lvwPeople"
 Background="{x:Null}"
 ItemsSource="{Binding}">
    <ListView.View>
        <GridView>
            <GridViewColumn Header="Author" Width="100"
             DisplayMemberBinding="{Binding Path=Author}"/>
            <GridViewColumn Header="Title" Width="300"
```

```
                DisplayMemberBinding="{Binding Path=Title}"/>
               <GridViewColumn Header="Year" Width="50"
                DisplayMemberBinding="{Binding Path=Year}"/>
               <GridViewColumn Header="Price" Width="50"
                DisplayMemberBinding="{Binding Path=Price}"/>
           </GridView>
       </ListView.View>
   </ListView>
```

UseListView

In this code, the ListView's ItemsSource property is set to {Binding}. That tells the control that it should use all of the data to which it is bound. In this example, that means it should use the entire ObservableCollection created by the code and attached to the control's DataContent property.

The ListView's View property (the ListView.View element shown in the code) describes how the control's data should be displayed. The GridView that the View property contains produces the grid-like arrangement shown in Figure 5-17.

The GridView element contains a series of GridViewColumn objects that define the view's columns. The Width and Header attributes simply set the text displayed at the top of a column and the column's initial width.

The GridViewColumn's DisplayMemberBinding attribute tells the column how to find the data that it should display in that column. The Path part of the binding tells what part of the data to display. For example, the Author column's DisplayMemberBinding attribute is set to {Binding Path=Author}. That tells the column to take the next piece of data bound to the ListView (a BookInfo object) and display its Author member. In this example, that means displaying the BookInfo object's Author property.

The other GridViewColumns are defined similarly. Look back at Figure 5-17 to see the result.

The ListView control is fairly powerful, and, together with XAML triggers and code-behind, it can provide many special features such as multiple views, formatted cells (in this case, it might be nice to right-justify the Year and Price columns), cells that are colored differently depending on the values of their contents, editable data, and so forth.

Unfortunately, many of these features are fairly complicated and therefore this chapter doesn't say anything more about them. For a list of about a dozen How-To examples, see Microsoft's web page msdn.microsoft.com/ms752071.aspx.

ProgressBar

The ProgressBar displays progress information to the user. As your program performs some time-consuming task, it can use a ProgressBar to let the user know that the program is still working and to indicate roughly how much it has done.

The control's Minimum, Maximum, and Value properties determine the control's appearance. For example, suppose your program needs to process 17 large text files. You could set the ProgressBar's Minimum property to 0 and its Maximum property to 17. Then after processing each file, the program could set the control's Value property to the number of files processed so far. As the program worked its way through the files, the ProgressBar would fill in a corresponding percentage of itself.

Unfortunately in WPF, the situation isn't quite that simple. As the program works on the files, it doesn't give up control of the CPU long enough for the `ProgressBar` to refresh itself on the screen so the user doesn't see any change in the control's status.

One way to solve this problem is to perform the lengthy task on a different thread. A thread is like a lightweight process that can run independently of the main user interface. Now as the thread works, it can update the `ProgressBar`'s `Value` property. Because the `ProgressBar` runs in a different thread, it can update itself independently of the lengthy task.

Sadly, even this more complicated approach isn't as easy as it sounds. In WPF (and Windows Forms programming, too), a control can only be accessed by code running in the same thread that created the control. That means the file processing code cannot directly update the `ProgressBar` because it is running in a separate thread.

The almost final solution is to have the file processing thread use a `Dispatcher` object to invoke code on the user interface thread when necessary.

If you think this all sounds pretty complicated for as simple a task as showing progress, you're right! Fortunately, there's a simpler solution.

The `BackgroundWorker` class can automatically perform a task on a separate thread and provide notification to the user interface thread when necessary. Behind the scenes the `BackgroundWorker` class handles all of the details about starting itself on a new thread, using a Dispatcher to invoke code on the user interface thread, and so forth, but all of the details are nicely hidden from you.

The UseProgressBar example program takes this approach to show how it's doing as it simulates a long task. The following C# code shows how the program works:

Available for download on Wrox.com

```
// The BackgroundWorker that will perform the long task.
private BackgroundWorker m_BackgroundWorker;

public Window1()
{
    this.InitializeComponent();

    // Insert code required on object creation below this point.

    // Prepare the BackgroundWorker.
    m_BackgroundWorker = new BackgroundWorker();
    m_BackgroundWorker.WorkerReportsProgress = true;
    m_BackgroundWorker.DoWork += BackgroundWorker_DoWork;
    m_BackgroundWorker.ProgressChanged +=
        BackgroundWorker_ProgressChanged;
    m_BackgroundWorker.RunWorkerCompleted +=
        BackgroundWorker_RunWorkerCompleted;
}

// Start the long task.
private void btnDoSomething_Click(object sender, RoutedEventArgs e)
{
    // Display the progress controls.
    prgWorking.Value = 0;
    prgWorking.Visibility = Visibility.Visible;
    lblWorking.Visibility = Visibility.Visible;
```

```
        btnDoSomething.Visibility = Visibility.Hidden;

        // Asynchronously start the routine
        // that does the long calculation.
        m_BackgroundWorker.RunWorkerAsync();
    }

    // Simulate a long task.
    private void BackgroundWorker_DoWork(Object sender,
        System.ComponentModel.DoWorkEventArgs e)
    {
        int value = 0;
        Random rand = new Random();
        while (value < 100)
        {
            // Wait a little while.
            Thread.Sleep(200);

            // Add a bit to the progress.
            value += rand.Next(10, 20);

            // Update the UI thread.
            m_BackgroundWorker.ReportProgress(value);
        }
    }

    // Display the progress.
    private void BackgroundWorker_ProgressChanged(Object sender,
        System.ComponentModel.ProgressChangedEventArgs e)
    {
        prgWorking.Value = e.ProgressPercentage;
    }

    // Reset the UI.
    private void BackgroundWorker_RunWorkerCompleted(Object sender,
        System.ComponentModel.RunWorkerCompletedEventArgs e)
    {
        prgWorking.Visibility = Visibility.Hidden;
        lblWorking.Visibility = Visibility.Hidden;
        btnDoSomething.Visibility = Visibility.Visible;
    }
```

UseProgressBar

The code starts by declaring the `BackgroundWorker` object. The `Window`'s constructor initializes the object and sets its `WorkerReportsProgress` property to `True` so it can generate progress events. It then assigns event handlers to the object's `DoWork`, `ProgressChanged`, and `RunWorkerCompleted` events.

When the user clicks on the `btnDoSomething` button defined by the program's XAML code, the `btnDoSomething_Click` event handler executes. That code displays the `ProgressBar` and a progress `Label` and hides the `Button`. It then calls the `BackgroundWorker`'s `RunWorkerAsync` method to start the lengthy task.

When the program calls the BackgroundWorker's RunWorkerAsync method, the object raises its DoWork event. The attached event handler sets a value variable to 0 and then enters a loop that runs as long as the variable's value is less than 100.

Each time through the loop, the program sleeps for 200 milliseconds to simulate a long task. At this point, a real program would be frantically reading files, generating output, buying and selling stocks, and performing other calculations.

After wasting some time, the program generates a random number between 10 and 20 and adds it to the value variable. It then calls the BackgroundWorker's ReportProgress method, passing it the new value. This makes the worker raise its ProgressChanged event on the user interface thread.

The loop continues until the value variable reaches 100.

When the worker raises its ProgressChanged event, the program's BackgroundWorker_ProgressChanged event handler executes. That routine runs on the user interface thread that created the worker so it can directly access the program's controls. It simply updates the ProgressBar's Value property.

When the loop ends, the BackgroundWorker_DoWork event handler ends, and the BackgroundWorker raises its RunWorkerCompleted event. The event handler hides the ProgressBar and the progress Label and redisplays the program's button.

Figure 5-18 shows the program about 70 percent of the way through its simulated calculation.

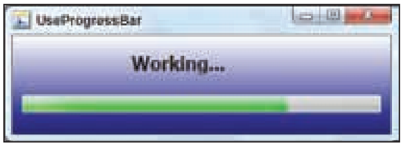


FIGURE 5-18

Separator

The Separator object draws a line between two items in a menu or toolbar. The UseSeparator example program shown in Figure 5-19 has a Separator between the File menu's Open and Exit commands, and between the second and third toolbar buttons in the two toolbars near the bottom of the window.

FIGURE 5-19

The following XAML code builds the File part of the program's menu structure. The code that builds the Edit and Help menus has been omitted to save space.

Available for download on Wrox.com

```
<Menu HorizontalAlignment="Stretch"
 DockPanel.Dock="Top" VerticalAlignment="Top">
    <MenuItem Header="File">
        <MenuItem Header="New">
            <MenuItem.Icon>
                <Image Source="New.ico"/>
            </MenuItem.Icon>
        </MenuItem>
        <MenuItem Header="Open">
            <MenuItem.Icon>
                <Image Source="Open.ico"/>
            </MenuItem.Icon>
        </MenuItem>
```

```
            <Separator/>
            <MenuItem Header="Exit"/>
        </MenuItem>
        ...
    </Menu>
```

UseSeparator

This program also has two `ToolBar` controls on the bottom. The upper `ToolBar` directly contains a series of `Buttons` with a `Separator` between the second and third `Buttons`. The result is the light vertical line shown in Figure 5-19.

The bottom `ToolBar` contains a `ToolBarPanel` that holds the `Buttons`. Normally the `ToolBarPanel` changes the appearance of the `Buttons` so the `Separator` is invisible. To make the `Separator` visible again, this program uses the following code to give the `Separator` some width and to remove its usual vertical line:

```
<Separator Width="10" Margin="0,0,0,0" Background="{x:Null}"/>
```

Note that you could use any blank object as a `Separator` in a `Toolbar`. For example, the following `Label` also makes a horizontal gap in the `Toolbar`:

```
<Label Width="10"/>
```

TreeView

The `TreeView` displays hierarchical data. For example, the UseTreeView example program shown in Figure 5-20 displays the organizational chart for a fictitious company. The dark triangles beside the Board of Directors, President/CEO, and other items indicate that those items are expanded to show the subitems that they contain. The light, hard to see triangle next to the CFO item indicates that the CFO item is collapsed to hide its subitems.

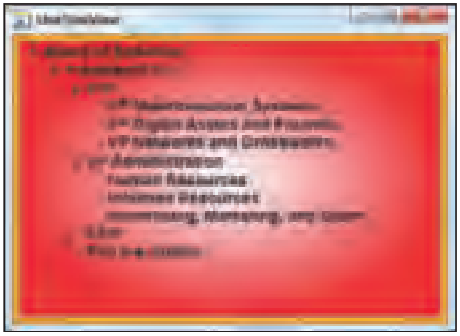


FIGURE 5-20

The `TreeView` object contains `TreeViewItem` objects that hold the values shown on the control. A `TreeViewItem` can contain other controls, but if you just want to display text as shown in Figure 5-20, you can simply set the `TreeViewItem`'s `Header` property to the text you want it to display.

You give the `TreeView` its hierarchical structure by placing `TreeViewItems` inside other `TreeViewItems`.

The following XAML code shows how the `UseTreeView` program builds its `TreeView` control:

Available for download on Wrox.com

```
<TreeView Margin="5">
    <TreeView.Background>
        <RadialGradientBrush>
            <GradientStop Color="#FFFFFFFF" Offset="0"/>
            <GradientStop Color="#FFFF3030" Offset="1"/>
```

```
            </RadialGradientBrush>
        </TreeView.Background>
        <TreeViewItem Header="Board of Directors" IsExpanded="True">
            <TreeViewItem Header="President/CEO" IsExpanded="True">
                <TreeViewItem Header="CIO" IsExpanded="True">
                    <TreeViewItem Header="VP Misinformation Systems"/>
                    <TreeViewItem
                     Header="VP Digital Assets and Paranoia"/>
                    <TreeViewItem Header="VP Networks and Grossworks"/>
                </TreeViewItem>
                <TreeViewItem
                 Header="VP Administration" IsExpanded="True">
                    <TreeViewItem Header="Human Resources"/>
                    <TreeViewItem Header="Inhuman Resources"/>
                    <TreeViewItem
                     Header="Advertising, Marketing, and Spam"/>
                </TreeViewItem>
                <TreeViewItem Header="CFO" IsExpanded="False">
                    <TreeViewItem Header="VB Resource Mismanagement"/>
                    <TreeViewItem Header="General Accounting"/>
                    <TreeViewItem Header="Colonel Accounting"/>
                    <TreeViewItem
                     Header="Internal &quot;Auditing&quot;"/>
                    <TreeViewItem Header="VB Perks and Boondoggles"/>
                </TreeViewItem>
                <TreeViewItem Header="Phil the Janitor"/>
            </TreeViewItem>
        </TreeViewItem>
    </TreeView>
```

UseTreeView

Like the `ListView` control, the `TreeView` control has some advanced data-binding capabilities. Chapter 18 explains some `TreeView` data-binding techniques.

SUMMARY

This chapter describes WPF's most useful content controls. These are the controls that a program uses mainly to display content for the user to view but not to modify or manipulate.

This chapter provides a brief overview of the controls and gives enough detail for you to get started using them. However, some of the controls are incredibly complex. For example, the `ListView` and `TreeView` controls provide all sorts of features that let you customize the ways in which they bind to and display data. With enough work, you can customize these controls until they are hardly recognizable.

Unfortunately, there isn't room to cover all of these advanced techniques in a single book. For more information on specific controls, start by looking at Microsoft's documentation at `msdn.microsoft.com/ms752324.aspx`.

The following chapter describes another group of controls: layout controls. These are controls such as `Grid` and `StackPanel` that help you by arranging the controls that they contain.

6

Layout Controls

Layout controls are primarily intended to contain and arrange other controls. They position the controls they contain in different ways to make it easier to design various kinds of user interfaces.

Layout controls are not quite the same as container controls. While all layout controls do contain other controls, the converse is not true: Not all controls that contain other controls arrange those controls in a nontrivial way.

For example, the `GroupBox` control draws a header and a border around a single child. Because its main purpose is to draw the header and border and because it contains only a single child control, it doesn't really do much arranging.

These non-arranging container controls are described elsewhere. For example, the `GroupBox` is described in the previous chapter.

CONTROL OVERVIEW

The following table briefly lists the controls described in this chapter together with their purposes. You can use this table to help decide which control you need in a particular situation.

CONTROL	PURPOSE
`Canvas`	Lets you position child controls explicitly by specifying the distances between their left, top, right, and bottom edges and those of the `Canvas`.
`DockPanel`	Docks its children to its edges.
`Expander`	Displays a header and an icon that the user can click on to show or hide a single child control.
`Grid`	Displays children in rows and columns. You can also ignore the rows and columns and position children somewhat as you can in a `Canvas`.

continues

(continued)

CONTROL	PURPOSE
`ScrollViewer`	Displays a single child in a scrollable region.
`StackPanel`	Displays children in a single row or column.
`StatusBar`	Creates an area at the bottom of the window where you can display status information to the user.
`TabControl`	Displays a series of tabs that the user can click to select different children.
`ToolBar`	Displays a vertical or horizontal area where you can place buttons and other tools for easy access.
`ToolBarTray`	Handles sizing, dragging, and other arrangements of `ToolBar` controls.
`UniformGrid`	Displays a grid where all rows have the same size and all columns have the same size.
`Viewbox`	Stretches its contents in various ways.
`WindowsFormsHost`	Contains Windows Forms controls.
`WrapPanel`	Displays children in a row/column that wraps to a new row/column when necessary.

The following sections describe these controls in greater detail and provide XAML examples demonstrating the controls.

CANVAS

The `Canvas` control lets you explicitly position child controls by specifying the distances between their left, top, right, and bottom edges and those of the `Canvas`. To do that, the `Canvas` control provides `Left`, `Top`, `Right`, and `Bottom` attached properties.

FIGURE 6-1

For example, the following code creates a `Canvas` containing four buttons that are 20 pixels from the edges of the `Canvas`. Figure 6-1 shows the result.

Available for download on Wrox.com

```
<Canvas>
    <Button Content="TopLeft" Width="85" Height="30"
     Canvas.Top="20" Canvas.Left="20"/>
    <Button Content="TopRight" Width="85" Height="30"
     Canvas.Top="20" Canvas.Right="20"/>
    <Button Content="BottomLeft" Width="85" Height="30"
```

```
        Canvas.Bottom="20" Canvas.Left="20"/>
    <Button Content="BottomRight" Width="85" Height="30"
        Canvas.Bottom="20" Canvas.Right="20"/>
</Canvas>
```

UseCanvas

The Canvas always gives its children as much space as they want.

If a child specifies both the Left and Right or the Top and Bottom attached properties, then the Left or Top values take precedence and the Right or Bottom values are ignored.

EDGE EFFECTS

Because of the way the attached properties work, the Canvas control's children stick to its edges and keep their original sizes when the Canvas resizes. If you want the children to stretch when their parent resizes, consider a different container such as a Grid and use the children's Margin properties.

DOCKPANEL

The DockPanel control docks its children to its edges. This control defines the attached property Dock, which can take the values Left, Right, Top, and Bottom.

During layout, the DockPanel loops through its child controls and checks their Dock values. It attaches each child to the appropriate edge of whatever space is currently unclaimed in the DockPanel.

If the DockPanel control's LastChildFill property is True, then it makes its last child fill whatever space remains.

For example, the following code creates a series of Border controls that are docked to the DockPanel's edges and that contain Label controls.

Available for download on Wrox.com

```
<DockPanel LastChildFill="True">
    <Border DockPanel.Dock="Top" Background="LightGreen">
        <Label Content="1, Top"/>
    </Border>
    <Border DockPanel.Dock="Left" Background="#FFFF00FF">
        <Label Content="2, Left" >
            <Label.LayoutTransform>
                <RotateTransform Angle="-90"/>
            </Label.LayoutTransform>
        </Label>
    </Border>
    <Border DockPanel.Dock="Right" Background="#FFFB400">
        <Label Content="3, Right" >
```

```
                <Label.LayoutTransform>
                    <RotateTransform Angle="90"/>
                </Label.LayoutTransform>
            </Label>
        </Border>
        <Border DockPanel.Dock="Bottom" Background="Yellow">
            <Label Content="4, Bottom"/>
        </Border>
        <Border DockPanel.Dock="Bottom" Background="#FF80FFFF">
            <Label Content="5, Bottom"/>
        </Border>
        <Border Background="White">
            <Label Content="6, None"/>
        </Border>
    </DockPanel>
```

UseDockPanel

The `DockPanel` considers its first child. That control has `Dock` value `Top`, so the `DockPanel` attaches it to the top of its area.

Next the `DockPanel` considers its second child. That control has `Dock` value `Left`, so the `DockPanel` attaches it to the left edge of the space that is not occupied by the first child.

The `DockPanel` continues positioning its children until it reaches the last one. If the control's `LastChildFill` property is `True` (which it is by default), the control makes its final child fill the remaining space.

Figure 6-2 shows the results.

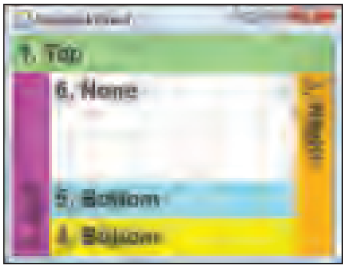


FIGURE 6-2

SIZE SPECIFICATIONS

By default, the `DockPanel` tries to stretch its child controls so they cover the edges to which they are attached. For example, in Figure 6-2, it stretches the top `Border` so it fills the `DockPanel`'s entire width.

At the same time, the `DockPanel` lets the control try to grow to its natural size in the other dimension. The top `Border` in Figure 6-2 is about 37 pixels tall because that's how tall it wants to be.

You can override either of a child's dimensions by explicitly setting its `Width` and `Height` properties.

EXPANDER

The `Expander` control displays a header and an icon that the user can click on to show or hide a single child control.

Figure 6-3 shows four expanders with headers "Personal Information," "Address," "Equipment Stats," and "Medical Information." The first `Expander` is expanded to show the `Grid` it contains and the other controls inside the `Grid` (an `Image`, three `Labels`, and three `TextBoxes`).

The following table summarizes the `Expander`'s most useful properties.

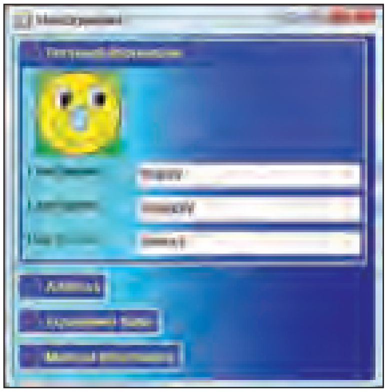


FIGURE 6-3

PROPERTY	PURPOSE
`Background`	The color of any exposed parts of the `Expander`, including the header area. In Figure 6-3, the background includes the blue header and thin blue border. The light-to-dark blue shaded area is the `Grid` inside the `Expander`.
`BorderBrush`	The color of the border drawn around the entire `Expander` (white in Figure 6-3)
`BorderThickness`	The thickness of the border drawn around the entire `Expander` (2 pixels in Figure 6-3)
`ExpandDirection`	Determines the direction in which the `Expander` opens to show its contents. This can be `Down`, `Left`, `Right`, and `Up` (`Down` in Figure 6.3).
`Foreground`	The color of the `Expander`'s header text (yellow in Figure 6-3)
`Header`	The text that the `Expander` displays ("Personal Information" for the first `Expander` in Figure 6.3)
`IsExpanded`	Determines whether the `Expander` is currently expanded.

GRID

The `Grid` control arranges its children in rows and columns.

The `Grid`'s `ColumnDefinitions` property is a collection of `ColumnDefinition` objects that determine the widths of the `Grid`'s columns. Similarly, the `Grid`'s `RowDefinitions` property is a collection of `RowDefinition` objects that determine the heights of the `Grid`'s rows.

The `ColumnDefinition` and `RowDefinition` objects have properties that determine their sizes (`Width` for columns, `Height` for rows).

These `Width` and `Height` properties can take absolute values or proportional values. To use an absolute number of pixels, you can simply use a number. For example, if a column's `Width` property is set to 50, that column is 50 pixels wide.

You can include a unit indicator after a number to use a unit other than pixels. The following table lists the unit indicators that you can use:

INDICATOR	MEANING	EQUIVALENT
px	pixels	1 pixel
in	inches	96 pixels
cm	centimeters	96/2.54 ª 37 pixels
pt	points	1/72 inch

To use a proportional width or height, assign the property a number followed by an asterisk (*). (The value * is equivalent to 1*.) The `Grid` calculates the amount of space it has that is not used up by absolute values and then divides it among the rows or columns with proportional sizes, weighting them according to their numeric values.

For example, the following XAML code defines a grid with three columns and two rows.

```
<Grid Margin="5">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="0.75 in"/>
        <ColumnDefinition Width="2*"/>
        <ColumnDefinition Width="1*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="30"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
</Grid>
```

The first column has a width of 0.75 inch = 72 pixels, so it takes up the leftmost 72 pixels in the `Grid`. The second and third columns' widths are set to 2* and 1*, so the second column is twice as wide as the last column. In other words, the second column gets 2/(2 + 1) = 2/3 of the remaining width in the `Grid`, and the last column gets 1/(2 + 1) = 1/3 of the remaining width.

The first row has a height of 30 pixels. The second row has height set to *, so it gets all of the remaining height in the `Grid`.

By convention, many developers set proportional values so they add up to either 100 or 1. That lets you think of the sizes as percentages of the remaining space. For example, in the following XAML code, the first two columns each use 25 percent of the `Grid`'s width and the third column uses the remaining 50 percent of the `Grid`'s width.

```
<Grid Margin="5">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="25*"/>
        <ColumnDefinition Width="25*"/>
        <ColumnDefinition Width="50*"/>
    </Grid.ColumnDefinitions>
</Grid>
```

To let you place child controls in its rows and columns, the `Grid` control defines `Row` and `Column` attached properties. For example, the following XAML code creates a `Label` and places it in row 0 column 2 of a `Grid`.

```
<Label Grid.Row="0" Grid.Column="2" Content="Hello!"/>
```

The `Grid` also defines `RowSpan` and `ColumnSpan` attached properties, so you can make a child cover more than one row or column. For example, set `RowSpan = 2` to make a child span 2 rows.

Figure 6-4 shows a `Grid` containing `Borders` and `Labels` that demonstrate various `Row`, `Column`, `RowSpan`, and `ColumnSpan` values. Since the `Grid`'s `ShowGridLines` property is set to `True`, it displays dashed lines between its rows and columns. Download the example program from the book's web site to see how the XAML code produces this result.

FIGURE 6-4

SCROLLVIEWER

The `ScrollViewer` control displays a single child inside a scrollable region. Its most important properties are `HorizontalScrollBarVisibility` and `VerticalScrollBarVisibility`. These can take one of the following values:

- `Auto` — The scrollbar is displayed when needed and hidden otherwise.
- `Visible` — The scrollbar is displayed even when it isn't needed.
- `Disabled` — The scrollbar does not appear even when it is needed. The child control's size is set to the available size of the `ScrollViewer`. For example, a `Grid` might be resized to fit.
- `Hidden` — The scrollbar does not appear even when it is needed. The child's size is not set to the available size of the `ScrollViewer`. For example, a `Grid` would keep its original size even if it sticks out beyond the bounds of the `ScrollViewer` (where it is clipped).

Figure 6-5 illustrates the difference between the `Disabled` and `Hidden` values. The program contains a `StackPanel` holding two nearly identical `ScrollViewers`. Each `ScrollViewer` holds a `Grid` with two equally sized rows and columns. Since the `Grids`' `ShowGridLines` properties are set to `True`, you can see the dashed lines between the rows and columns.

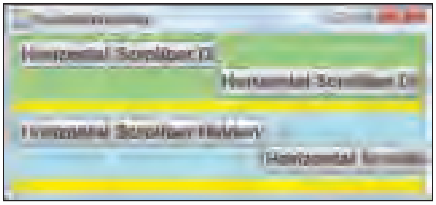


FIGURE 6-5

Since the upper `ScrollViewer`'s `HorizontalScrollBarVisibility` property is set to `Disabled`, the horizontal scrollbar isn't shown and the `Grid` is resized to fit within the `ScrollViewer`. The large buttons don't fit in the `Grid` so they are truncated.

The lower `ScrollViewer`'s `HorizontalScrollBarVisibility` property is set to `Hidden`. This time, since the `Grid` is not resized to fit within the `ScrollViewer`, it remains large enough to display its buttons. The first button fits in the `ScrollViewer`, but the second doesn't.

The UseScrollViewer example program uses the following code to demonstrate the most typical use of the `ScrollViewer`: to let the user scroll over something that's too big to fit.

Available for download on Wrox.com

```
<ScrollViewer
 HorizontalScrollBarVisibility="Auto"
 VerticalScrollBarVisibility="Auto">
    <Image Source="ColoradoFlowers.jpg" Stretch="None"/>
</ScrollViewer>
```

UseScrollViewer

Figure 6-6 shows the UseScrollViewer program in action.

FIGURE 6-6

STACKPANEL

The `StackPanel` control displays its children in a single row or column. If any children don't fit, they are clipped.

The most useful property of the `StackPanel` is `Orientation`, which can take the values `Vertical` (the default) or `Horizontal`.

The UseStackPanel example program shown in Figure 6-7 uses three `StackPanels`. The first fills the window and contains the other two. The second is a horizontal `StackPanel` that contains the blue buttons. The last is a vertical `StackPanel` that contains the yellow buttons.

FIGURE 6-7

The following XAML code shows how the program builds the `StackPanel` containing the blue buttons. (Don't worry about the `Style` for now. That just sets the button sizes, colors, and margins.)

Available for download on Wrox.com

```
<StackPanel Orientation="Horizontal">
    <Button Content="Drinks" Style="{StaticResource HStyle}"/>
    <Button Content="Appetizers" Style="{StaticResource HStyle}"/>
    <Button Content="Entrees" Style="{StaticResource HStyle}"/>
    <Button Content="Desserts" Style="{StaticResource HStyle}"/>
</StackPanel>
```

UseStackPanel

STATUSBAR

The `StatusBar` control creates an area at the bottom of the window where you can display status information to the user. Ideally, a `StatusBar` should contain relatively simple read-only information such as labels, icons, and progress bars, although some programs put buttons, combo boxes, menus, and all sorts of other content in them.

Figure 6-8 shows the UseStatusBar application displaying a `StatusBar`. A timer in the code-behind updates the status icon, progress bar, and clock.

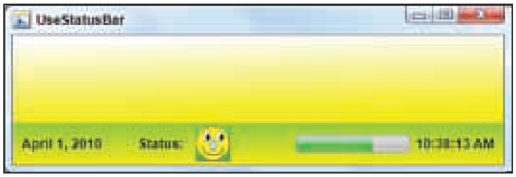


FIGURE 6-8

A `StatusBar` should contain `StatusBarItem` objects to hold the status items. A `StatusBarItem` can contain only a single child, but that child can be a container such as a `Grid` or `StackPanel`, so you can display just about anything you like.

The following XAML fragment shows how the UseStatusBar program colors the `StatusBar` and makes its date label. The timer in the code-behind resets the date to the current date when the program starts.

Available for download on Wrox.com

```
<StatusBar HorizontalAlignment="Stretch" VerticalAlignment="Bottom"
 FontWeight="Bold">
    <StatusBar.Background>
        <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
            <GradientStop Color="#FF80FF00" Offset="0"/>
            <GradientStop Color="#FFFFFF00" Offset="1"/>
        </LinearGradientBrush>
    </StatusBar.Background>
    <StatusBarItem>
        <Label Name="lblDate" Content="4/1/2010"/>
    </StatusBarItem>
    ... Other StatusBarItems omitted ...
</StatusBar>
```

UseStatusBar

TABCONTROL

The `TabControl` displays a series of tabs that the user can click to select the control's children.

> **CONTROL CONTROLS**
>
> Yes, this control is called `TabControl`, not just *Tab*, *TabContainer*, or some other more appealing name. This can lead to awkward phrases such as "Insert a TabControl control here."
>
> For some reason, it seems that one or two controls always slip through that have the word *control* in their names, perhaps for historical reasons.

The `TabControl` object contains `TabItem` objects that hold the control's content. A `TabItem` can contain only a single child, but that child can be a container such as a `Grid` or `StackPanel`, so you can put just about anything you want in the `TabItem`.

Figure 6-9 shows the UseTabControl example program with its first tab selected. Compare this interface to the `Expanders` used in Figure 6-3.

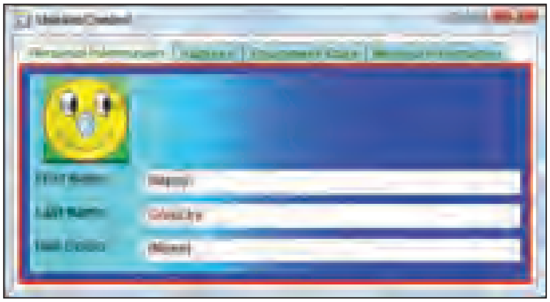


FIGURE 6-9

The following table summarizes the `TabControl`'s most useful properties:

PROPERTY	PURPOSE
`BorderBrush`	Sets the color for the control's border (red in Figure 6-9).
`BorderThickness`	Sets the thickness of the border (3 pixels in Figure 6-9).
`SelectedIndex`	The index of the currently selected tab (0 in Figure 6-9)
`SelectedItem`	The selected object (normally a `TabItem` object)
`TabStripPlacement`	Determines where the control places its tabs. Can take the values `Left`, `Right`, `Top` (the default), or `Bottom`.

The `TabItem`'s `ForeGround` and `BackGround` properties determine the colors used to draw the item's tab. In Figure 6-9, these are yellow and blue, respectively. The light-to-dark blue shaded area is the `Grid` contained inside the `TabItem`.

Usually the `TabItem`'s `Header` property is a simple string, but if you want to place something more complicated in the tab, you can do so by using property element syntax.

> **BAD BACKGROUNDS**
>
> When you click on a `TabControl`'s tab, the control changes the background of that tab to show that it is selected. In Figure 6-9, you can see that the first tab's background is different from those of the other tabs.
>
> If you change the tabs' colors, be sure that the foreground color you use will contrast with the selected tab's background color, or the result may be hard to read.

For example, in the following XAML fragment, the `Header` property contains a horizontal `StackPanel` that holds an `Ellipse` and a `Label`. The `TabItem`'s body holds a `Grid` that could contain other controls.

```
<TabItem>
    <TabItem.Header>
        <StackPanel Orientation="Horizontal">
            <Ellipse Width="30" Height="20"
             Stroke="Black" Fill="Yellow"/>
            <Label Content="Round Things" />
        </StackPanel>
    </TabItem.Header>
    <Grid>
        ... Content omitted ...
    </Grid>
</TabItem>
```

TOOLBAR AND TOOLBARTRAY

The `ToolBar` control displays an area where you can place buttons, combo boxes, and other tools that the user can easily find.

To give the user the full `ToolBar` experience, place one or more `ToolBars` inside a `ToolBarTray`. The `ToolBarTray` arranges the `ToolBars` and lets the user drag them around within the tray.

Finish by putting `Buttons`, `TextBoxes`, `ComboBoxes`, and whatever other tools you want to display inside the `ToolBar`.

The `ToolBarTray` defines two attached properties that you can use to control the positioning of the `ToolBars`: `Band` and `BandIndex`.

`Band` is a number that determines the order of the horizontal rows or bands in the `ToolBarTray` that hold the `ToolBars`. The `ToolBarTray` uses the `Band` value to sort the bands, but it doesn't care whether they start with 0, 1, or something else, or if they are consecutive.

`BandIndex` is a number that determines a `ToolBar`'s position within its band.

Figure 6-10 shows the UseToolBar example program displaying one `ToolBarTray` containing three `ToolBars`.

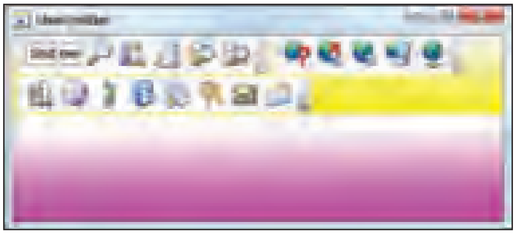


FIGURE 6-10

The first `Toolbar`, which contains a textbox and search-related buttons, has `Band = 1` and `BandIndex = 1`, so it appears first in the first (top) band.

The second `ToolBar`, which contains various globe-related buttons, has `Band = 1` and `BandIndex = 2`, so it appears second in the first band.

The final `ToolBar`, which contains miscellaneous buttons, has `Band = 2` and `BandIndex = 1`, so it appears first in the second band.

At run time, the user can drag `ToolBars` to new positions in the `ToolBarTray`, even in different bands.

TOOLBAR TURMOIL

In my tests, Expression Blend wasn't very good at updating `ToolBars`. Occasionally it got confused and stopped displaying a tool or put `ToolBars` in the wrong bands. Switching the designer to XAML view and then switching back to Design or Split view usually fixed it, and it always displayed correctly at run time.

If a `ToolBar` is too small to display all of its items (e.g., if you make the window too narrow), it creates an overflow area that is available to the user via a dropdown arrow.

The `ToolBar` defines an `OverflowMode` attached property that lets you determine how an item in a `ToolBar` behaves when the `ToolBar` uses an overflow area. You can set `OverflowMode` to `Always` (always put the item in the overflow area), `Never` (never put the item in the overflow area, even if it gets chopped off), and `AsNeeded` (put the item in the overflow area if it won't fit in the `ToolBar`).

Figure 6-11 shows the UseToolBar program again, this time with the overflow area in the third `ToolBar` open. The four items in this area have `OverflowMode` set to `Always`, so they appear only in the overflow area (not shown in Figure 6-10).

FIGURE 6-11

The tools in the first `ToolBar` all have `OverflowMode` set to `Never`, so they are never moved into that `ToolBar`'s overflow area. If you make the form too narrow to show the whole `ToolBar`, it sticks off the side of the window and is clipped.

The following XAML code shows how the UseToolBar program builds parts of its `ToolBars`:

Available for download on Wrox.com

```
<ToolBarTray VerticalAlignment="Top" HorizontalAlignment="Stretch">
    <ToolBarTray.Background>
        <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
            <GradientStop Color="White" Offset="0"/>
            <GradientStop Color="Yellow" Offset="1"/>
        </LinearGradientBrush>
    </ToolBarTray.Background>
    <ToolBar Band="1" BandIndex="1"> <!--  Search  -->
        <TextBox BorderBrush="Gray" Width="50"
         ToolBar.OverflowMode="Never"/>
        <Button ToolBar.OverflowMode="Never">
            <Image Source="search.ico" Height="24" Width="24"/>
        </Button>
        <Button ToolBar.OverflowMode="Never">
            <Image Source="search_reference.ico" Height="24" Width="24"/>
        </Button>
        <Button ToolBar.OverflowMode="Never">
            <Image Source="search4doc.ico" Height="24" Width="24"/>
        </Button>
        ... Other Buttons omitted ...
    <ToolBar Band="1" BandIndex="1"> <!--  Search  -->
    ... Other ToolBars omitted ...
</ToolBarTray>
```

UseToolBar

The `ToolBarTray` control has one final property worth mentioning here: `Orientation`. You can set this property to `Horizontal` or `Vertical` to determine whether the tray arranges its `ToolBars` vertically or horizontally.

Figure 6-12 shows a program that arranges its `ToolBarTray` vertically. The program's XAML code uses a `LayoutTransform` to rotate the textbox sideways. Note that the third `ToolBar`'s overflow area opens horizontally even though the `ToolBar` itself is arranged vertically.

FIGURE 6-12

UNIFORMGRID

The `UniformGrid` control displays items in a grid in which all rows have the same size and all columns have the same size. You set the control's `Rows` and `Columns` properties to the number of rows and columns you want. The control divides its area up evenly to form that many rows and columns.

The control places its children in the resulting grid cells. If a child is too big to fit in its cell, it is clipped at the cell's boundaries.

In some ways, a `UniformGrid` acts like a `WrapPanel`. Both arrange their children in a row until they run out of room, and then they begin a new row. However, there are two big differences between the controls.

First, the `WrapPanel` gives each item in a row the same height, but the items in different rows may have different widths so they don't necessarily line up in columns. In contrast, the `UniformGrid` gives every item the same area vertically and horizontally.

Second, the `WrapPanel` has an `Orientation` property that lets you determine whether it arranges its children in rows or columns. The `UniformGrid` always arranges its children in row-first order.

Figure 6-13 shows the UseUniformGrid example program displaying three rows and four columns. Since the buttons stretch to fit their cells, they all have the same size, but the control would work just as well if its children all had different sizes.

FIGURE 6-13

The following XAML code fragment shows the key pieces of the UseUniformGrid program. I've omitted the `UniformGrid`'s background definition to save space.

Available for download on Wrox.com

```
<UniformGrid Columns="4" Rows="3">
    <UniformGrid.Background>
        ... Omitted to save space ...
    </UniformGrid.Background>
    <Button Content="Button 1" Margin="5"/>
    <Button Content="Button 2" Margin="5"/>
    <Button Content="Button 3" Margin="5"/>
    <Button Content="Button 4" Margin="5"/>
    <Button Content="Button 5" Margin="5"/>
    <Button Content="Button 6" Margin="5"/>
    <Button Content="Button 7" Margin="5"/>
    <Button Content="Button 8" Margin="5"/>
    <Button Content="Button 9" Margin="5"/>
    <Button Content="Button 10" Margin="5"/>
    <Button Content="Button 11" Margin="5"/>
    <Button Content="Button 12" Margin="5"/>
</UniformGrid>
```

UseUniformGrid

VIEWBOX

The `Viewbox` control stretches its contents in various ways. That not only lets you stretch images (the `Image` control can do that by itself anyway), but it also lets you stretch other controls.

The `Viewbox` can contain only a single child, but that child can be a container such as a `Grid` or `StackPanel`, so you can put anything you want in the child.

The `Viewbox`'s `Stretch` property determines how it stretches its child. The `Stretch` property can take one of the following values:

- `None` — The child is not stretched.
- `Fill` — The child is stretched to fill the `Viewbox` even if that causes distortion.
- `Uniform` — The child is stretched uniformly (by the same amount vertically and horizontally) until it is as big as possible while still fitting within the `Viewbox`.
- `UniformToFill` — The child is stretched uniformly until it fills the entire `Viewbox` even if parts of the child stick out and are clipped.

Figure 6-14 shows the UseViewbox example program demonstrating the different `Stretch` property values.

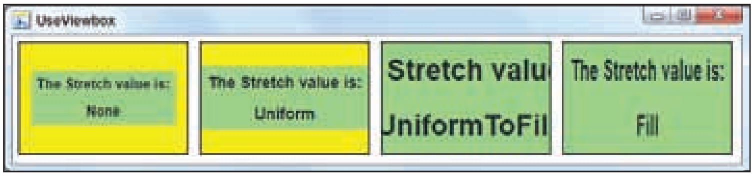


FIGURE 6-14

WINDOWSFORMSHOST

The `WindowsFormsHost` control can hold Windows Forms controls. This lets you use a Windows Forms control if there is no equivalent WPF version. (It can even host older COM controls, although that's not covered explicitly here.)

For example, WPF doesn't have a `DateTimePicker` control, but Windows Forms does. The following XAML code fragment adds a `DateTimePicker` control to a WPF window.

Available for download on Wrox.com

```
<GroupBox Header="Appointment Info" Margin="5">
    <StackPanel Margin="10,5,0,0">
        <CheckBox Content="Make an appointment" Margin="5"
         Name="chkMakeAppt" Checked="chkMakeAppt_Checked"
         Unchecked="chkMakeAppt_Unchecked" />
        <WindowsFormsHost Margin="5" x:Name="wfhAppt">
            <WindowsFormsHost.Child>
                <wf:DateTimePicker x:Name="dtpAppt"
                 Enabled="False" />
            </WindowsFormsHost.Child>
        </WindowsFormsHost>
    </StackPanel>
</GroupBox>
```

UseWindowsFormsHost

The code starts with a `GroupBox` that displays the header "Appointment Info." The `GroupBox` contains a `StackPanel` that holds a `CheckBox` and the `WindowsFormsHost`. The `WindowsFormsHost`'s `Child` property contains a `DateTimePicker` control.

Figure 6-15 shows the result.

FIGURE 6-15

The program uses the following C# code-behind to enable the `DateTimePicker` only when the program's `CheckBox` is checked. When the control is enabled, you can click on its parts to change the month, date, and year, or you can click on the dropdown arrow to display the month selection area shown in Figure 6-15.

Available for download on Wrox.com

```
// Enable the DateTimePicker so the user can make an appointment.
private void chkMakeAppt_Checked(object sender, RoutedEventArgs e)
{
    dtpAppt.Enabled = true;
}

// Disable the DateTimePicker.
private void chkMakeAppt_Unchecked(object sender, RoutedEventArgs e)
{
    dtpAppt.Enabled = false;
}
```

UseWindowsFormsHost

Before you can use a `WindowsFormsHost`, you must perform three steps:

1. Add a reference to the WindowsFormsIntegration library. In Visual Studio, you can open the Project menu, select "Add Reference," and select the library from the .NET tab on the Add Reference dialog. In Expression Blend, you can open the Project menu, select Add Reference, and then browse to the library file. On my system, it's installed at `C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\WindowsFormsIntegration.dll`.
2. Add a reference to the System.Windows.Forms.dll library. Again, in Visual Studio, you can use the Project menu's Add Reference command and select the library from the resulting dialog. In Expression Blend, you can again use the Project menu's Add Reference command and then browse to the library file. On my system, it's installed at `C:\Windows\Microsoft.NET\Framework\v2.0.50727\System.Windows.Forms.dll`.
3. Add a namespace declaration for the Windows Forms namespace to the top of the XAML file. The declaration should look something like this:

```
xmlns:wf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms"
```

This makes *wf* the abbreviation for the namespace, so you can refer to Windows Forms controls as in `wf:DateTimePicker`.

HOST HINTS

Visual Studio seems to do a better job of displaying the `WindowsFormsHost` control than Expression Blend does, at least in the versions I'm using now. Visual Studio seems to understand Windows Forms controls well enough to calculate the size of the host and its contents and display an outline for the host. Expression Blend doesn't seem to look inside the host, so it doesn't use its content to give the host the proper size.

If the host's contents should help determine its size, you may want to lay out the host in Visual Studio rather than Expression Blend.

WRAPPANEL

The `WrapPanel` control works much like a `StackPanel`, arranging controls in a row. The difference between the two is that the `WrapPanel` starts a new row when the current row runs out of room. It keeps arranging its children in rows, starting new rows when necessary, until it has positioned all of its children.

Like the `StackPanel`, the `WrapPanel` has an `Orientation` property that can take the values `Horizontal` or `Vertical`. If you set `Orientation` to `Vertical`, the control arranges its children in columns instead of rows.

Figure 6-16 shows the UseWrapPanel example program displaying two `WrapPanels` that demonstrate the two orientations.

FIGURE 6-16

The `WrapPanel` is ideal when it's more important for the user to see all of the items than it is to have each item in a specific position.

SUMMARY

This chapter describes WPF's most useful layout controls. These are the controls that a program uses mainly to arrange other controls in various ways.

This chapter provides a brief overview of the controls and gives enough detail for you to get started using them. However, some of the controls are fairly complex. In particular, the `Grid` control provides a lot of features for arranging its children, and you may need a bit of practice with it before you fully master its capabilities.

The following chapter describes another group of controls: user interaction controls. These are controls such as `TextBox`, `Button`, and `Menu` that let the user enter data, execute code, and otherwise control the application.

7

User Interaction Controls

The two preceding chapters described two categories of WPF controls: content controls and layout controls. This chapter describes a third category: user interaction controls.

User interaction controls are the objects that the user manipulates to interact with the application. Where content controls let the program give information to the user, user interaction controls let the user give information to the program.

The types of information the user gives to the program include:

- Textual values entered in textboxes
- Choices selected from list boxes, combo boxes, radio buttons, checkboxes, and so forth
- Values picked from a value selection control such as a slider or numeric up/down button

The user also tells the program when to perform actions by clicking buttons and selecting menu items.

This chapter describes these controls and gives examples to help you get started using them. Because the whole point of these controls is to give information and instructions to the program, this chapter contains a bit more code-behind than the previous ones.

Chapter 2 includes several sections that explain different ways of building event handlers. See the section "Code-Behind" for details.

CONTROL OVERVIEW

The following table briefly lists the controls described in this chapter together with their purposes. You can use this table to help decide which control you need for a particular purpose.

CONTROL	PURPOSE
`Button`	Lets the user click to tell the program to do something.
`CheckBox`	Lets the user select or deselect a non-exclusive option.

continues

(continued)

CONTROL	PURPOSE
`ComboBox`	Lets the user pick from a dropdown list of choices.
`ContextMenu`	Displays a pop-up menu associated with a control.
`Frame`	Displays Web or XAML content and provides simple navigation buttons.
`GridSplitter`	Lets the user resize the rows and columns in a `Grid`.
`ListBox`	Displays a list of items to the user and lets the user pick one or more, depending on how the control is configured.
`Menu`	Displays a main menu for a window.
`PasswordBox`	Lets the user enter text while hiding the text on the screen
`RadioButton`	Lets the user select one of an exclusive set of options.
`RepeatButton`	Lets the user fire `Click` events repeatedly as long as the mouse is pressed over the control.
`RichTextBox`	Lets the user enter formatted text with such features as multiple fonts, multiple colors, paragraphs, hanging indentation, bulleted lists, and so forth.
`ScrollBar`	Lets the user select a numeric value from a range of values. Usually the value is used to adjust something graphically such as the position of a large image within a viewing area.
`Slider`	Lets the user select a numeric value from a range of values.
`TextBox`	Lets the user enter text without fancy formatting.

BUTTON

The `Button` control lets the user click to tell the program to do something.

This is an extremely simple control to use. When the user clicks a `Button`, the `Button` raises its `Click` event. The program catches the event and takes whatever action is necessary.

The following code shows one of the simplest ways to attach an event handler to a `Button` in XAML code. When the user clicks on this button, the code-behind executes the routine named `btnShowTime_Click`:

```
<Button Content="Show Time" x:Name="btnShowTime"
 Width="100" Height="30" Margin="5" Click="btnShowTime_Click" />
```

The following code shows the C# code behind for this button:

Available for download on Wrox.com

```
// Show the time.
private void btnShowTime_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show(DateTime.Now.ToString());
}
```

UseButtons

The UseButtons example program uses similar code to display the three buttons shown on the left in Figure 7-1.

A `Button`'s `Content` property can contain a single child, but that child can be a container such as a `Grid` or `StackPanel`, so you can put just about anything you want in a `Button`. The large button on the right in Figure 7-1 contains a `Grid` that holds an `Image` and five `Labels`.

FIGURE 7-1

CHECKBOX

A `CheckBox` control lets the user select or unselect an option. The user can check or uncheck the option independently of any other `CheckBox`. (In contrast, only one `RadioButton` in a group can be selected at one time.)

A `CheckBox` can also operate in a tri-state mode where it can be checked, unchecked, or in an indeterminate state that is neither checked nor unchecked.

The UseCheckBoxes example program, shown in Figure 7-2, contains four `CheckBoxes`. In the figure, the first and third are checked, and the last has an indeterminate state.

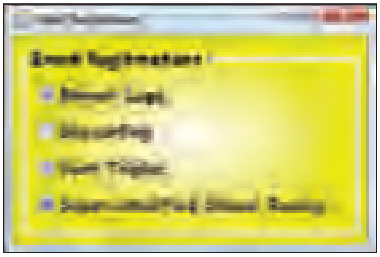


FIGURE 7-2

INDETERMINATE CONFUSION

Many users don't understand tri-state `CheckBoxes` very well, largely because their use isn't very common and the meaning of the indeterminate state isn't really standardized. Some programs use this state to mean parts of an option are selected. For example, when picking code libraries to install, you might select all of them (checked), none of them (unchecked), or some of them (indeterminate). In this case, the program needs some method for the user to specify which items should be selected and that further complicates the interface.

continues

(continued)

If you use tri-state `CheckBoxes` in a program, expect to spend a little extra time and documentation explaining. You might want to consider using a `ComboBox` or other multi-value control so that the user can select the three possible values by name (*Used*, *Unused*, and *Sort of Used*?) instead of using checks and filled boxes.

Two of the `CheckBox` class's most useful events are `Checked` and `Unchecked`. These events fire when (you guessed it) the user checks or unchecks the control. If you want to take action whenever a `CheckBox` is either checked or unchecked, then you can catch the `Click` event instead.

COVERT CLICKS

Despite its name, the `Click` event handler doesn't only fire when a `CheckBox` is clicked. If the user tabs to the control and presses the [Enter] or Space key, the control also fires its `Click` event.

Often, however, a program doesn't need to catch these events and can simply check the control's state later when it must perform some action, perhaps triggered by a `Button`.

The control's most useful property (other than the usual assortment of colors, fonts, etc.) is `IsChecked`. This property can take the values `True` (checked), `False` (unchecked), and `null` (indeterminate). The program can use `IsChecked` to get or set the control's current state.

The `IsThreeState` property determines whether the control uses two states (checked and unchecked) or three states (checked, unchecked, and indeterminate).

COMBOBOX

The `ComboBox` control lets the user pick from a dropdown list of choices. The choices are represented by `ComboBoxItem` objects inside the `ComboBox`. Each `ComboBoxItem` can hold a single child, although that child can be a container such as a `Grid` or `StackPanel`.

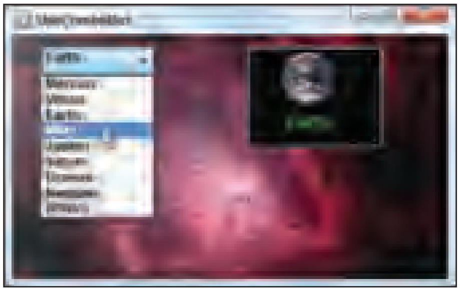


FIGURE 7-3

The UseComboBox example program shown in Figure 7-3 has two `ComboBoxes`. In the figure, the combo box on the left is open and displaying its list of choices.

The following XAML code shows how the program builds its left `ComboBox`. This is a fairly typical text-only `ComboBox` and is quite simple to use.

Available for download on Wrox.com

```
<ComboBox IsSynchronizedWithCurrentItem="True"
 Grid.Row="0" Grid.Column="0"
 VerticalAlignment="Top" Margin="10"
 SelectedIndex="2" Height="25" Width="100">
  <ComboBoxItem Content="Mercury"/>
  <ComboBoxItem Content="Venus"/>
  <ComboBoxItem Content="Earth"/>
  <ComboBoxItem Content="Mars"/>
  <ComboBoxItem Content="Jupiter"/>
  <ComboBoxItem Content="Saturn"/>
  <ComboBoxItem Content="Uranus"/>
  <ComboBoxItem Content="Neptune"/>
  <ComboBoxItem Content="(Pluto)"/>
</ComboBox>
```

UseComboBox

Figure 7-4 shows the UseComboBox program with its right `ComboBox` open. Each `ComboBoxItem` contains a `StackPanel` that holds an `Image` and a `Label`.

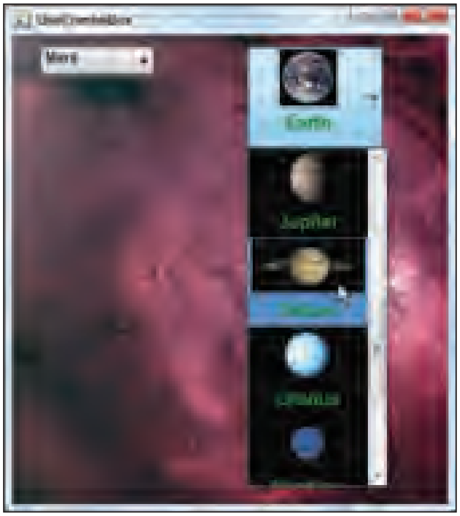


FIGURE 7-4

The following XAML code shows how the program builds the first item in its right `ComboBox`:

Available for download on Wrox.com

```
<ComboBox IsSynchronizedWithCurrentItem="True"
 Grid.Row="0" Grid.Column="1"
 VerticalAlignment="Top" Background="Black"
 SelectedIndex="2" Height="90" Width="120" Margin="10"
 FontWeight="Bold" FontSize="16">
  <ComboBoxItem Background="Black">
    <StackPanel HorizontalAlignment="Stretch" Width="100">
      <Image Source="Mercury.jpg" Height="50" Width="Auto"
       HorizontalAlignment="Center"/>
```

```
            <Label Content="Mercury" HorizontalAlignment="Center"
             Foreground="Green"/>
          </StackPanel>
        </ComboBoxItem>
        ... Other ComboBoxItems omitted ...
      </ComboBox>
```

UseComboBox

The ComboBox's Background property sets the background used for the ComboBox itself, not the items. You can see in Figure 7-3 that the right ComboBox's background is black.

A ComboBoxItem's Background property sets the background used for the ComboBoxItem. For example, in Figure 7-4, you can see that the background behind the Earth, Jupiter, and Saturn items is black.

Notice the blue background behind Mars in Figure 7-4. When the mouse hovers over an item in the dropdown list (Mars in Figures 7-3 and 7-4), the ComboBox changes its background color so the user can see that it is selected.

The ComboBox can only change the background color of an item where it is not covered by its contents. For example, in the previous code, if you set the ComboBoxItem's StackPanel's Background property to blue, then almost all of the dropdown items would have a blue background. The ComboBox would only be able to change the color of two small slices to the left and right of the dropdown item, so the user would have very little clue when the item was selected.

The morale of the story is to not try to fill the dropdown items completely but make sure that some background shows through.

The following table summarizes the ComboBox's most useful properties:

PROPERTY	PURPOSE
Background	The background used to draw the ComboBox itself
IsEditable	Determines whether the user can type a new value into the ComboBox. (This works best if the items are plaintext values as on the left in Figure 7-3.)
MaxDropdownHeight	The maximum height of the dropdown list
SelectedIndex	The index of the currently selected item
SelectedItem	The currently selected ComboBoxItem

CONTEXTMENU

The ContextMenu control displays a pop-up menu associated with a control. For example, the UseContextMenu example program shown in Figure 7-5 contains a ListBox with an associated ContextMenu. When you right-click on the ListBox, the ContextMenu automatically appears, as shown in the figure.

FIGURE 7-5

Strangely, neither Expression Blend nor Visual Studio has a `ContextMenu` in its Control Toolboxes. Fortunately, a `ContextMenu` is easy to build in XAML.

The following XAML code shows how the UseContextMenu program builds its `ListBox` and `ContextMenu`. After the `ListBox`'s `Background` element, a `ListBox.ContextMenu` element defines the `ContextMenu`. That element must contain a `ContextMenu` object that contains `MenuItem` objects.

Available for download on Wrox.com

```
<ListBox Name="lstWebSites" Grid.Row="1" HorizontalAlignment="Stretch" Width="Auto"
 IsSynchronizedWithCurrentItem="True" Margin="5">
  <ListBox.Background>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
      <GradientStop Color="#FFFFFFFF" Offset="0"/>
      <GradientStop Color="#FFFFFF00" Offset="1"/>
    </LinearGradientBrush>
  </ListBox.Background>
  <ListBox.ContextMenu>
    <ContextMenu>
      <MenuItem Name="mnuAddNewSite" Header="Add New Site"/>
      <MenuItem Name="mnuDeleteSite" Header="Delete Site"/>
    </ContextMenu>
  </ListBox.ContextMenu>
  <ListBoxItem>
    <StackPanel Orientation="Horizontal">
      <Label Content="VB Helper" Width="150"/>
      <Label Content="www.vb-helper.com" Width="150"/>
    </StackPanel>
   </ListBoxItem>
   ... Other ListBoxItems omitted ...
</ListBox>
```

UseContextMenu

You can easily create submenus in a `ContextMenu` by simply adding `MenuItems` inside other `MenuItems`. For example, the following `ContextMenu` contains a submenu that holds items Copy, Cut, and Paste:

```
<ContextMenu>
  <MenuItem Name="mnuAddNewSite" Header="Add New Site"/>
  <MenuItem Name="mnuDeleteSite" Header="Delete Site"/>
  <MenuItem Header="Submenu">
    <MenuItem Name="mnuCopy" Header="Copy"/>
    <MenuItem Name="mnuCut" Header="Cut"/>
    <MenuItem Name="mnuPaste" Header="Paste"/>
  </MenuItem>
</ContextMenu>
```

HANDLING MENUITEMS

When the user selects a menu item, the `MenuItem` object raises a `Click` event just as a `Button` does. You can catch it and handle it in the same way.

`ContextMenus` are one kind of `Menu` object. The "Menu" section later in this chapter has more to say about `Menu` and `MenuItem` objects.

FRAME

The `Frame` control displays content and provides a simple navigation model. It can display web pages or XAML files and follow links in those files.

The control's `Source` property determines the web page or XAML file that the control should display. A program can also call the control's `Navigate` method to make it display a new page. For example, the following C# code makes the `Frame` named `fraGo` open the web page `www.vb-helper.com/index_categories.html`:

```
fraGo.Navigate(new Uri("http://www.vb-helper.com/index_categories.html"));
```

When the `Frame` navigates to a new page, either because its `Source` property changed or the program called the `Navigate` method, the `Frame` stores its previous location in its navigation history. The user can click on the control's forward and backward buttons to move through the history.

COVERT NAVIGATION

If the frame moves to a new page because the user clicks a link on the current page, the `Frame` does *not* add the new page to the navigation history.

The UseFrame example program shown in Figure 7-6 contains a `ComboBox` and a `Frame`. When you select a web page from the `ComboBox`, the program displays the page.

FIGURE 7-6

In Figure 7-6, you can see the navigation buttons in the `Frame` control's upper-left corner. If you click on the small dropdown arrow to the right of the buttons, the control displays a list of the pages in the navigation history so you can quickly jump to one of them.

GRIDSPLITTER

The `GridSplitter` control lets the user adjust the widths of the rows and columns in a `Grid`. To use a `GridSplitter`, you add it to a cell in the `Grid`. At run time, the user can grab the `GridSplitter` and drag it to resize the rows or columns on either side of the control.

Figure 7-7 shows the UseGridSplitter example program. The blue and red bars are `GridSplitters` that let you resize the `Grid`'s columns and rows, respectively.

FIGURE 7-7

The following XAML code shows how the UseGridSplitter program makes its blue `GridSplitter`:

Available for download on Wrox.com

```
<GridSplitter Grid.Column="1" Grid.RowSpan="2"
  HorizontalAlignment="Left"
  VerticalAlignment="Stretch"
  Background="Blue"
  ShowsPreview="False"
  Width="5"/>
```

UseGridSplitter

The following list explains each of this control's property values:

- `Grid.Column="1"` — The `GridSplitter` will resize columns 0 and 1. This property puts the control in column 1.
- `Grid.RowSpan="2"` — This makes the control span both of the `Grid`'s rows. If you leave this out, the control only extends through the first row. It would still let the user resize the columns, but it looks weird.
- `HorizontalAlignment="Left"` — This makes the control stick to the left edge of column 1 so it appears to be between columns 0 and 1.

- `VerticalAlignment="Stretch"` — This makes the control fill the `Grid` vertically.
- `Background="Blue"` — This just makes the control stand out.
- `ShowsPreview="False"` — If this is `True`, the control shows a gray ghost of itself as the user drags it. If this is `False`, the control actually resizes the `Grid`'s columns as the user drags it.
- `Width="5"` — This determines the control's width. Make this big enough that the user can click on it easily but small enough that it doesn't waste too much room.

To make a `GridSplitter` that lets the user resize rows instead of columns, switch the roles of the rows and columns and the vertical and horizontal alignment. The following XAML code shows how the UseGridSplitter program makes its red `GridSplitter`:

Available for download on Wrox.com

```
<GridSplitter Grid.Row="1" Grid.ColumnSpan="2"
  HorizontalAlignment="Stretch"
  VerticalAlignment="Top"
  Background="Red"
  ShowsPreview="True"
  Height="5"/>
```

UseGridSplitter

MAKE ROOM FOR SPLITTERS

Because the `GridSplitter` actually sits inside the `Grid`'s cells, it takes up room in them. If you want the other controls in the cells to be completely visible and not cover the `GridSplitter`, be sure to set their `Margin` properties so they don't overlap with the `GridSplitter`.

LISTBOX

The `ListBox` control displays a list of items to the user and lets the user pick one or more, depending on the control's `SelectionMode` property. `SelectionMode` can take the three values described in the following list:

- `Single` — You can pick only one item. If you click on a new item, the old one is deselected.
- `Multiple` — You can pick multiple items by clicking on them. Click on an item a second time to deselect it.
- `Extended` — You can pick multiple items. If you just click on an item, all others are deselected. [Ctrl]+click on an item to toggle whether it is selected without changing other selections. Click on an item and [Shift]+click on a second item to select them and all of the items between. [Ctrl]+[Shift]+click does the same thing as [Shift]+click except it doesn't remove previous selections. (This is easier to use than to visualize from a written description. Give it a try to get a better sense of how it works.)

Two other useful properties are `HorizontalContentAlignment` and `VerticalContentAlignment`, which let you control how items in the list are aligned.

The UseListbox example program shown in Figure 7-8 contains a `ListBox` with `SelectionMode` set to `Extended` and the third, fifth, and sixth items selected.

Each `ListboxItem` must contain a single child, but that child can be a container such as a `Grid`, so you can effectively put anything you want in the list.

FIGURE 7-8

The following XAML code shows how the program builds this `ListBox`. The "Mountain Unicycling" item includes a `StackPanel` holding a `TextBlock` and an `Image`.

Available for download on Wrox.com

```
<ListBox Grid.Row="1"
 HorizontalAlignment="Stretch" VerticalAlignment="Stretch" Height="Auto"
 IsSynchronizedWithCurrentItem="True"
 FontSize="16" FontWeight="Bold"
 Foreground="Blue" Background="Pink"
 HorizontalContentAlignment="Left" VerticalContentAlignment="Top"
 SelectionMode="Extended">
  <ListBoxItem Content="Bossaball"/>
  <ListBoxItem Content="Ga-Ga"/>
  <ListBoxItem>
    <StackPanel Orientation="Horizontal">
      <TextBlock Text="Mountain Unicycling" VerticalAlignment="Center"/>
      <Image Source="Unicycle.jpg" Stretch="Uniform"
       Height="50" Margin="30,0,0,0"/>
    </StackPanel>
  </ListBoxItem>
  <ListBoxItem Content="Pesapallo"/>
  <ListBoxItem Content="Hornussen"/>
  <ListBoxItem Content="Trugo"/>
</ListBox>
```

UseListBox

COLOR CAUTION

The `ListBox` control changes an item's colors when it is selected. To ensure that the user can tell when an item is selected, don't cover the entire item with a control that has explicitly set foreground and background colors.

For example, in the previous code, if you set the `StackPanel`'s `Background` to red and the `TextBlock`'s `Foreground` to black, then it's harder to tell when the Mountain Unicycling item is selected.

To make it easier for the `ListBox` to change items' colors, the control doesn't inherit its background color from its container.

MENU

The `Menu` control displays a main menu for a window. The `Menu` can provide submenus, shortcuts (such as [Ctrl]+O for the Open command), and accelerators (e.g., opening the File menu when you press [Alt]+F).

The `Menu` control contains `MenuItem` objects that represent the commands within the menu. You can build submenus by placing `MenuItems` inside other `MenuItems`.

When the user selects a menu item, the corresponding `MenuItem` object raises its `Click` event. Your program can catch and handle that event just as it handles `Button Click` events.

The `MenuItem`'s `Header` property determines what the item displays. Often the `Header` value is simple text.

Place an underscore in front of the Header's letter that you want the `MenuItem` to use as an accelerator key. For example, if you set the `Header` to *_File*, then the `MenuItem` displays *File*. (Note that this is different from the way Windows Forms applications handles this. In a Windows Forms application, you place an ampersand in front of the character that you want underlined in a menu.)

ACTIVATING ACCELERATORS

In recent Windows operating systems, accelerator characters are not normally underlined until the user presses the [Alt] key. Then the accelerator characters are underlined, and the user can press the corresponding key to trigger the item.

In a typical application, for example, when the user presses [Alt], the File menu's underline appears. If the user presses **F**, the menu opens to show its items with their accelerator keys underlined. In many programs, the File menu contains a New command. If the user presses **N** while the accelerators are visible, that command executes.

You can set a menu item's shortcut keys by setting the `MenuItem`'s `InputGestureText` property. For example, you might set the `InputGestureText` for the `New` command to *[Ctrl]+N*.

Support for accelerators is automatic. You only need to define the accelerator keys by adding underscores, and the application automatically does the rest.

Unfortunately, since support for shortcuts is not automatic, your program must catch the appropriate keystrokes and take action when necessary. WPF provides a command architecture that allows you to handle keystrokes for menu items in a very general way, but since it is fairly complicated, it isn't described in any more detail here. Chapter 19, "Commanding," covers commands more completely.

Figure 7-9 shows a program with its File menu open. In this figure, I pressed [Alt] and then **F** to open the File menu.

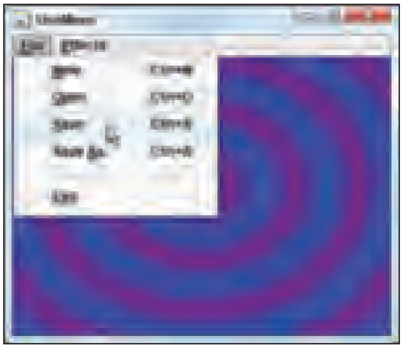


FIGURE 7-9

The following XAML code builds the menu shown in Figure 7-9. Notice the `Separator` object between the "Save As" and Exit `MenuItems`.

Available for download on Wrox.com

```
<Menu VerticalAlignment="Top">
  <MenuItem HorizontalAlignment="Left" VerticalAlignment="Top" Header="_File">
    <MenuItem Header="_New" InputGestureText="Ctrl+N"/>
    <MenuItem Header="_Open" InputGestureText="Ctrl+O"/>
    <MenuItem Header="_Save" InputGestureText="Ctrl+S"/>
    <MenuItem Header="Save _As" InputGestureText="Ctrl+A"/>
    <Separator/>
    <MenuItem Header="E_xit"/>
  </MenuItem>
  <MenuItem Header="_Effects">
    ... MenuItems omitted ...
  </MenuItem>
</Menu>
```

UseMenu

A `MenuItem`'s `Header` property is often plaintext but it can be any single control including a container control. Each of the menu items shown in Figure 7-10 contains a horizontal `StackPanel` that holds an `Image` and a `Label`.

The `MenuItems` shown in Figure 7-10 also demonstrate two new properties. The `IsCheckable` property determines whether the menu item can display a checkbox. The `IsChecked` property determines whether the item is actually checked. In Figure 7-10, every `MenuItem` has `IsCheckable = True`, but only the "Drop Shadow" `MenuItem` has `IsChecked = True`.

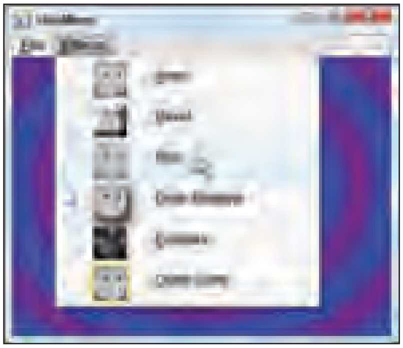


FIGURE 7-10

The following code shows how the UseMenu example program defines the "Drop Shadow" `MenuItem` shown in Figure 7-10:

Available for download on Wrox.com

```
<MenuItem IsCheckable="True" IsChecked="True">
  <MenuItem.Header>
    <StackPanel Orientation="Horizontal">
      <Image Stretch="None" Source="GraySmiley.bmp" Margin="0,0,20,0">
        <Image.BitmapEffect>
          <DropShadowBitmapEffect/>
        </Image.BitmapEffect>
      </Image>
      <Label Content="_Drop Shadow"/>
    </StackPanel>
  </MenuItem.Header>
</MenuItem>
```

UseMenu

UNLIMITED FREE CHECKING

Support for checkable items is almost automatic in WPF. If a `MenuItem` has `IsCheckable = True`, then the program will automatically check and uncheck the item when the user clicks it. The `MenuItem` even provides `Checked` and `Unchecked` events to let the program know what's happening.

If that's all you need the item to do, then you're done. If you need menu items to be exclusive (e.g., you only want one of the items shown in Figure 7-10 to be checked at a time), then you need to add extra code to uncheck the other items when an item is checked.

MENUS, MENUS EVERYWHERE

Normally menus stretch across the top of an application, but there's nothing to prevent you from putting them in other places. For example, you could add a menu inside a `GroupBox` or in the tabs of a `TabControl` to provide commands that only apply to the contained controls. You could even apply transformations to menus to make them rotated or skewed.

Have pity on your users, however, and don't go crazy dropping menus all over the place. If you make the menus too confusing, you'll have to spend a lot of time explaining things to confused users.

One final property is worth mentioning here. You can set a `MenuItem`'s `Icon` property to determine the picture that is displayed next to it while the item is not checked. When the item is checked, it shows a checkmark similar to the one shown next to the "Drop Shadow" item in Figure 7-10 instead of the `Icon`.

PASSWORDBOX

The `PasswordBox` is a relatively simple textbox that replaces the characters that the user types with dots (•) or some other special character on the screen. Internally the control keeps track of what's being typed but prevents the user from seeing it. More to the point, it prevents Phil the office prankster from peeking over the user's shoulder to steal the password.

The UsePasswordBox example program shown in Figure 7-11 demonstrates the `PasswordBox`. Enter a username and password. As you type the password, the `PasswordBox` displays a series of dots. When you click the OK button, the program displays the username and password that you entered.

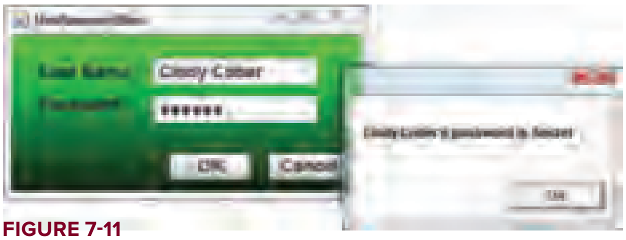


FIGURE 7-11

A program can use the control's `Password` property to see what password is really stored in the control. The following code shows how the UsePasswordBox program displays the username and password that you enter:

Available for download on Wrox.com

```
// Display the entered username and password.
private void btnOk_Click(object sender, RoutedEventArgs e)
{
  MessageBox.Show(txtUserName.Text + "'s password is " +
    pwdPassword.Password);
}
```

UsePasswordBox

After `Password`, the control's second most useful property is `PasswordChar`. This property determines the character that the control displays as the user types. Usually the default value (which produces the dots shown in Figure 7-11) is fine, but if you don't like it, you can set `PasswordChar` to a simple character such as a question mark (?), or you can copy and paste a character into the XAML code.

You can also set `PasswordChar` to a numeric HTML character code. For example, the following code creates a `PasswordBox` that uses character 167 (the section symbol §) as its password character:

```
<PasswordBox Password="Secret" PasswordChar="&#167;"/>
```

RADIOBUTTON

The `RadioButton` control lets the user select one of an exclusive set of options. If the user selects a `RadioButton`, all other `RadioButtons` in the same container automatically deselect.

(In contrast, any number of `CheckBoxes` in a group can be selected at one time. If the user should be able to select more than one option at a time, use `CheckBoxes` instead of `RadioButtons`.)

Often it is helpful to place related `RadioButtons` in a `GroupBox` so the user can easily see that they are related. The UseRadioButton example program shown in Figure 7-12 displays `RadioButtons` arranged in three `GroupBoxes`. If the user clicks a `RadioButton`, the others in that `GroupBox` deselect.

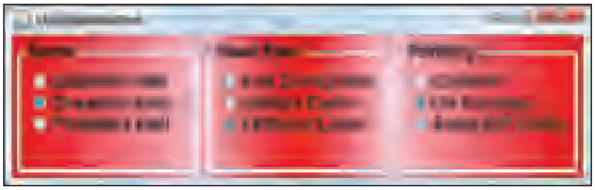


FIGURE 7-12

If you don't want `GroupBoxes` to surround related `RadioButtons`, place them in a `StackPanel`, `Grid`, or some other control that doesn't normally display a border.

RELIABLE RADIOBUTTONS

If you don't use `GroupBoxes` to group related `RadioButtons`, then you should use some other method to make it easy for the user to figure out which buttons go together. For example, you might place different groups of `RadioButtons` in rows or columns.

The `RadioButton`'s properties are very similar to those of the `CheckBox`. (Behind the scenes, they're closely related controls.)

The `IsChecked` property determines whether the control is selected. If the control's `IsThreeState` property is `True`, then `IsChecked` can be `True`, `False`, or `null` to indicate that the control is checked, unchecked, or in an indeterminate state, neither checked nor unchecked.

> **STATE OF CONFUSION**
>
> Three-state `RadioButtons` are even more confusing to users than three-state `CheckBoxes`. Visibly, the difference between the checked and indeterminate states is very small. To avoid confusion, you might want to do without three-state `RadioButtons`. See the tip "Indeterminate Confusion" earlier in this chapter for more discussion of the indeterminate state.

Two of the `RadioButton`'s most useful events are `Checked` and `Unchecked`, which fire when the user selects or deselects the control, respectively. The `Click` event fires both when the user selects and when the user deselects the control.

REPEATBUTTON

The `RepeatButton` control lets the user fire `Click` events repeatedly as long as the mouse is pressed over the control. By firing these events, the user can tell the program to do something many times very quickly.

The `RepeatButton` is as easy for a program to use as a `Button` control. It simply catches the control's `Click` event and takes the appropriate action. The control automatically handles the details of firing its `Click` event as long as it is pressed.

When the user first presses it, the button fires a `Click` event. It then pauses briefly before it starts firing more `Click` events.

The `Delay` property determines the length of the delay between the first and second `Click` events in milliseconds. The `Interval` property determines how much time passes between the subsequent `Click` events.

By default, `Delay` and `Interval` are 500 and 33, so the button waits half a second (500 milliseconds) after the first `Click` event and then fires another `Click` event every 33 milliseconds (about 30 times per second) after that.

The UseRepeatButton example program shown in Figure 7-13 demonstrates the `RepeatButton`. Each time the button raises its `Click` event, the program increments the Messages Sent counter.

FIGURE 7-13

INCONSISTENT INTERVALS

The `RepeatButton`'s `Interval` property determines the amount of time the control tries to pause between `Click` events, but the actual time may vary, particularly if `Interval` is small. It takes a certain amount of time for the control to raise a `Click` event and for your program to catch and handle it.

Use the `Delay` and `Interval` properties to let the user fire `Click` events at a reasonable speed, but don't rely on the exact timing.

RICHTEXTBOX

The `RichTextBox` control lets the user enter richly formatted text with such features as multiple fonts, multiple colors, paragraphs, hanging indentation, bulleted lists, and so forth. The `RichTextBox` control can even hold other content such as user interface elements (buttons, shapes, etc.), although there's no good way for the user to enter those items at run time.

Unfortunately, using XAML code to place content in a `RichTextBox` is complicated. The `RichTextBox`'s `Document` property contains the control's contents. The `Document` property should be a `FlowDocument` that holds other objects such as `Paragraphs` and `Tables`.

The following XAML code shows how a program might create a `RichTextBox` and its content. Note that the word "enobled" in the control's contents is intentionally misspelled so you can see the control's built-in spell-checking feature described shortly.

Available for download on Wrox.com

```
<RichTextBox>
  <FlowDocument>
    <Paragraph>
      <Run>If spell checking is enobled and you left-click
over a misspelled word, then the context menu also includes a list
of possible correct spellings and an Ignore All command.</Run>
    </Paragraph>
  </FlowDocument>
</RichTextBox>
```

UseRichTextBox

Chapter 21 has more to say about documents.

The `RichTextBox` is a very complex control and contains dozens of properties and methods that let you control its behavior and the format of its content. The following sections describe the `RichTextBox` control's most useful features. For a complete description of the control, see `msdn.microsoft.com/aa970779.aspx`.

Editing Commands

The `RichtextBox` control automatically provides features that let the user format text. For example, if the user selects some text in the control and presses [Ctrl]+B, the `RichTextBox` makes the selected text bold.

The `EditingCommands` class also makes these editing commands available through code. For example, the following C# code toggles whether the current selection in the `RichTextBox` named `rchBody` is part of a numbered list:

Available for download on Wrox.com

```
// Toggle the selection's number state.
private void mnuParaNumber_Click(object sender, RoutedEventArgs e)
{
  EditingCommands.ToggleNumbering.Execute(null, rchBody);
}
```

UseRichTextBox

Instead of executing an editing command in code-behind, you can indicate the command that a button or menu item should execute in its XAML code.

For example, the `Command` attribute in the following XAML code makes the `MenuItem` execute the `AlignLeft` editing command when it is selected:

```
<MenuItem Name="mnuParaAlignLeft" Header="_Left"
 Command="EditingCommands.AlignLeft"/>
```

The following table lists the most useful of the editing commands and the keys the user can press to invoke them. The command name is the name of the `EditingCommand` class method that invokes the command.

COMMAND NAME	SHORTCUT
`AlignCenter`	[Ctrl]+E
`AlignJustify`	[Ctrl]+J
`AlignLeft`	[Ctrl]+L
`AlignRight`	[Ctrl]+R
`DecreaseFontSize`	[Ctrl]+OemOpenBracket
`DecreaseIndentation`	[Ctrl]+[Shift]+T
`EnterLineBreak`	[Shift]+[Enter]
`EnterParagraphBreak`	[Enter]
`IncreaseFontSize`	[Ctrl]+OemCloseBracket
`IncreaseIndentation`	[Ctrl]+T
`ToggleBold`	[Ctrl]+B
`ToggleBullets`	[Ctrl]+[Shift]+L
`ToggleItalic`	[Ctrl]+I

COMMAND NAME	SHORTCUT
`ToggleNumbering`	[Ctrl]+[Shift]+N
`ToggleSubscript`	[Ctrl]+OemPlus
`ToggleSuperscript`	[Ctrl]+[Shift]+OemPlus
`ToggleUnderline`	[Ctrl]+U

The OemOpenBracket, OemCloseBracket, and OemPlus keys are the OEM (Original Equipment Manufacturer) [,], and + characters. These are usually placed in the keypad area and, depending on your keyboard hardware, similar keys in other parts of the keyboard may not work. For example, the + key in the keypad area might work but the + key next to the number keys might not.

For more information about editing commands and a more complete list, see `msdn.microsoft.com/system.windows.documents.editingcommands.aspx`.

Spell Checking

One of the `RichTextBox` control's most exciting features is support for spell checking. If you set the control's `SpellCheck.IsEnabled` property to `True`, then the control automatically highlights likely spelling errors.

By default, the `RichTextBox` control provides a context menu containing the Copy, Cut, and Paste commands. If spell checking is enabled and you left-click over a misspelled word, then the context menu also includes a list of possible correct spellings and an "Ignore All" command. If you select one of the spelling suggestions, the control replaces the misspelled version with your selection. If you select "Ignore All," the control treats all occurrences of the misspelled word as correctly spelled.

Figure 7-14 shows the UseRichTextBox example program displaying its context menu.

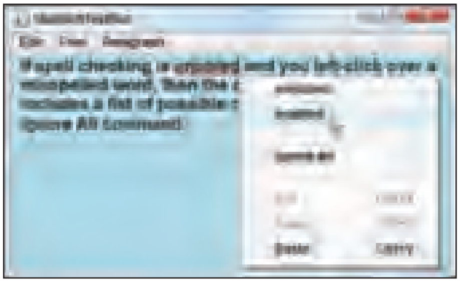


FIGURE 7-14

Undo and Redo

The `RichTextBox` control automatically provides undo and redo features. Each time the user presses [Ctrl]+Z, the control undoes the most recent change. Each time the user presses [Ctrl]+Y, the control reapplies the most recently undone change.

The control also provides tools that you can use to manage the undo stack programmatically. The `CanUndo` and `CanRedo` properties indicate whether there is an action that the control can undo or redo. The `Undo` and `Redo` methods make the control undo or redo an action.

The UseRichTextBox program's Edit menu contains `Undo` and `Redo` commands. When the user opens the Edit menu, the following code checks the `RichTextBox` control's `CanUndo` and `CanRedo` properties to see which of the menu items should be enabled. For example, if the user has not undone a command recently, then CanRedo returns False and the mnuEditRedo menu is disabled.

Available for download on Wrox.com

```
// Enable or disable the Undo and Redo commands.
private void mnuEdit_SubmenuOpened(object sender, RoutedEventArgs e)
{
  mnuEditUndo.IsEnabled = (rchBody.CanUndo);
  mnuEditRedo.IsEnabled = (rchBody.CanRedo);
}
```

UseRichTextBox

The following code shows how the program invokes the control's Undo and Redo methods.

Available for download on Wrox.com

```
// Undo.
private void mnuEditUndo_Click(object sender, RoutedEventArgs e)
{
  rchBody.Undo();
}

// Redo.
private void mnuEditRedo_Click(object sender, RoutedEventArgs e)
{
  rchBody.Redo();
}
```

UseRichTextBox

Other Features

The RichTextBox control provides too many other features to mention them all here. Before moving on, however, the following table summarizes some key RichTextBox properties.

PROPERTY	PURPOSE
AcceptsReturn	Determines whether [Enter] key strokes are placed in the control's content or whether they are sent to the window. If AcceptsReturn is False and the window contains a Button with IsDefault set to True, then pressing [Enter] inside the control triggers the Button.
AcceptsTab	Determines whether [Tab] key strokes are placed in the control's content or whether they are sent to the window. If AcceptsTab is False, then pressing [Tab] moves focus to the window's next control.
HorizontalScrollBarVisibility	Determines whether the horizontal scrollbar is visible. This can be Auto, Disabled, Hidden, or Visible.

PROPERTY	PURPOSE
`IsEnabled`	Determines whether the control will interact with the user. If this is `False`, the user can look but cannot change, highlight, or scroll the control's contents.
`IsReadOnly`	Determines whether the user can modify the control's contents. If `IsReadOnly` is `True`, the user can still scroll the control's contents, select text, and copy text to the clipboard. If you want to display something that the user cannot change, it is often better to set `IsReadOnly = True` instead of setting `IsEnabled = False`.
`VerticalScrollBarVisibility`	Determines whether the vertical scrollbar is visible. This can be `Auto`, `Disabled`, `Hidden`, or `Visible`.

The UseRichTextBox example program demonstrates some other simple formatting commands that let you change:

- Font style (**bold**, *italic*, underline)
- Font size (small, medium, large)
- Text color
- Background color
- Font family (e.g., Times New Roman, Arial, `Courier New`)
- Bulleted and numbered paragraphs
- Paragraph alignment (left, center, right, justify)

Download the example from the book's web site and look at the code to see how it works.

For a more complete description of the control and its features, see these links:

- **RichTextBox Overview** — `msdn.microsoft.com/aa970779.aspx`
- **RichTextBox How-To Topics** — `msdn.microsoft.com/ms744813.aspx`
- **RichTextBox Class** — `msdn.microsoft.com/system.windows.controls.richtextbox.aspx`

Microsoft's web site contains several sample programs that demonstrate `RichTextBox` features. Two particularly useful samples are:

- **EditingCommands Sample** — Demonstrates most of the `RichTextBox`'s editing commands: `http://msdn.microsoft.com/ms771634.aspx`.
- **Notepad Demo** — A WPF notepad application: `http://msdn.microsoft.com/aa972157.aspx`

SCROLLBAR

The `ScrollBar` control lets the user select a numeric value from a range of values. Often a program doesn't need to use `ScrollBars` directly, instead relying on controls that display them as needed such as `ScrollViewer`, `TextBox`, and `FlowDocumentReader`.

Occasionally, however, a `ScrollBar` is handy for letting the user select a value that will be used in some way other than to scroll content.

For example, the UseScrollBar example program shown in Figure 7-15 uses three `ScrollBars` to let the user select red, green, and blue color values between `0` and `255`. When any of the values change, the program displays the new value to the right of its `ScrollBar` and displays a sample of the color with those component values on the left.

FIGURE 7-15

The `ScrollBar` control is relatively simple. The following table summarizes the control's most useful properties:

PROPERTY	PURPOSE
`LargeChange`	The amount by which the control's value changes when the user clicks between the draggable thumb and one of the control's arrows
`Orientation`	`Vertical` or `Horizontal`
`Maximum`	The largest value the user can select
`Minimum`	The smallest value the user can select
`SmallChange`	The amount by which the control's value changes when the user clicks on one of the control's arrows
`Value`	The control's current value

The `ScrollBar`'s most useful event is `ValueChanged`, which — as you can probably guess — fires whenever the control's value changes.

The following C# code shows how the UseScrollBar program handles `ValueChanged` events for all three of its `ScrollBars`. The code uses the `ScrollBars`' values to make a solid brush and sets the `lblSample` control's `Background` property to the brush. It then displays the color values in labels.

Available for download on Wrox.com

```
// Display a color sample.
private void sbar_ValueChanged(
  object sender, RoutedPropertyChangedEventArgs<double> e)
{
  byte r = (byte)sbarRed.Value;
  byte g = (byte)sbarGreen.Value;
  byte b = (byte)sbarBlue.Value;
  Color clr = Color.FromRgb(r, g, b);
  SolidColorBrush br = new SolidColorBrush(clr);
```

```
        lblSample.Background = br;

        lblRed.Content = r;
        lblGreen.Content = g;
        lblBlue.Content = b;
    }
```

UseScrollBar

> **AUGMENTED ARROWS**
>
> The `ScrollBar`'s arrows are actually `RepeatButtons`. If you click on one, the control's value changes by the `SmallChange` value. If you hold an arrow down, the value changes by that amount repeatedly.

SLIDER

Like the `ScrollBar` control, the `Slider` control lets the user select a numeric value from a range of values. The `Slider` works in almost exactly the same way as the `ScrollBar` but with a different appearance.

Figure 7-16 shows the UseSlider example program. Notice how similar this program is to the UseScrollBar program shown in Figure 7-15. The main differences are that the `Sliders` display tick marks and don't have arrows.

FIGURE 7-16

The `Slider` has many of the same properties as the `ScrollBar`. See the table in the previous section for a description of the most important of those properties.

The following table summarizes new properties that the `Slider` uses to define its tick marks:

PROPERTY	PURPOSE
`IsSnapToTickEnabled`	If this is `True`, then the control snaps its value to the nearest tick mark.
`IsSelectionRangeEnabled`	Determines whether the user can select a range of values rather than only a single value.
`TickFrequency`	Determines the numeric spacing between tick marks.
`TickPlacement`	Determines the position of the tick marks. This can be `Both` (both top and bottom if horizontal or left and right if vertical), `BottomRight` (bottom if horizontal or right if vertical), `TopLeft` (top if horizontal or left if vertical), or `None`.

continues

(continued)

PROPERTY	PURPOSE
`Ticks`	An explicit list of numeric positions where there should be tick marks. For example, the value `"0,16,128,255"` would put tick marks at values `0`, `16`, `128`, and `255`.

TICK TROUBLE

If you specify the `Ticks` property but don't include the minimum and maximum values in it, the user may not be able to click on the Slider's body to jump to those values. This is a minor issue, but you may as well avoid it by including the minimum and maximum values in the `Ticks` property.

TEXTBOX

The `TextBox` control lets the user enter text without fancy formatting. The `TextBox` control is a good choice if you need to get plain old textual input from the user without multiple fonts, styles, colors, and so forth.

The `TextBox` control provides some of the same features as the `RichTextBox` control, but it doesn't have the powerful formatting capabilities. All of the text in a `TextBox` must use the same font family (Times New Roman, Arial, `Courier New`), font size, font style (*italic*, **bold**, underscore), and color.

If the control's `AcceptsReturn` property is `True`, the user can enter carriage returns in the control to create paragraphs, but the control does not format paragraphs specially. It cannot create paragraphs with hanging indent, bullets, or numbered lists. The control can justify text or align it on the left, right, or center, but the alignment applies to all of the text — not individual paragraphs.

Features that the `TextBox` shares with the `RichTextBox` include:

- Spell checking
- A context menu containing Copy, Cut, and Paste commands as well as spelling suggestions
- Undo and redo
- Properties: `AcceptsReturn`, `AcceptsTab`, `HorizontalScrollBarVisibility`, `IsEnabled`, `IsReadOnly`, `VerticalScrollBarVisibility`

See the descriptions of these features in the earlier "RichTextBox" section for details.

The `TextBox` control does not provide the editing commands supported by the `RichTextBox`. (Of course, that also means it doesn't support the keyboard shortcuts that let users apply those commands at run time.)

The UseTextBox example program is similar to the UseRichTextBox program except it demonstrates the `TextBox` control instead of the `RichTextBox` control. Its menus let you change the properties of all of the control's text, not just the selected text as program `UseRuchTextBox` does. Because the `TextBox` doesn't support bulleted or numbered paragraphs, the `UseTextBox` program's `Paragraph` menu doesn't provide those commands.

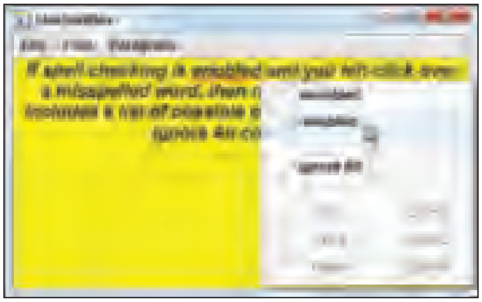


FIGURE 7-17

Figure 7-17 shows the UseTextBox program in action. Notice how the font styles, colors, and alignment apply to the control's whole contents, not just parts of the text.

SUMMARY

This chapter describes WPF's most useful user interaction controls. These are the controls that allow the user to control and provide input to a program.

This chapter provides a brief overview of the controls and gives enough detail for you to get started using them. However, some of the controls are fairly complex. The `RichTextBox` control is remarkably complicated and provides a wide variety of formatting options and editing commands. You'll need to spend some time experimenting with the `RichtextBox` control before you can use its capabilities to their fullest.

This chapter and the previous two describe major categories of WPF controls. Chapter 5 describes content controls — controls that display output for the user to view. The following chapter provides more detail on a specific kind of content control: those that produce two-dimensional shapes. These include Line, Ellipse, Rectangle, Polygon, Polyline, and Path. While many applications rely primarily on more textual controls such as `Labels` and `TextBoxes`, these drawing controls can be very useful for visualizing data.

8

Two-Dimensional Drawing Controls

Chapter 5, "Content Controls," describes controls that display information to the user. Most of these controls display textual data or images.

This chapter describes controls that display more graphical output: two-dimensional (2D) drawing controls. These controls draw lines, ellipses, curves, and other 2D shapes.

This chapter also describes the Path mini-language that you can use to concisely make the `Path` object produce complex drawings.

CONTROL OVERVIEW

The following table briefly lists the controls described in this chapter together with their purposes. You can use this table to help decide which control to use to satisfy your needs.

CONTROL	PURPOSE
`Ellipse`	Draws an ellipse.
`Line`	Draws a line.
`Path`	Draws a series of lines and curves.
`Polygon`	Draws a series of line segments that connect a series of points. It finishes by connecting the last point to the first.
`Polyline`	Draws a series of line segments that connect a series of points.
`Rectangle`	Draws a rectangle.

STROKE PROPERTIES

The controls described in this chapter draw linear features. Whether they draw a single line segment, a closed ellipse, or a complex sequence of curves, lines, and polygons, they all basically draw lines.

WPF provides a special set of properties to control the way those lines are drawn. These properties control the lines' colors, dash style, and thickness.

The following table summarizes these controls' most important drawing properties:

PROPERTY	PURPOSE
`Fill`	The control's background brush (This is similar to the `Background` property used by many other controls.)
`Stroke`	The brush used to draw the control's edges (This is similar to the `Foreground` property used by many other controls.)
`StrokeDashArray`	Array of values that tell how many pixels are drawn and then skipped to make a dash pattern The numbers are scaled by the line's thickness, so a value of 1 for a 5-pixel-wide line makes the line draw or skip 5 pixels. This makes dashes bigger in bigger lines and just looks better. The line shown in Figure 8-1 uses the `StrokeDashArray` value (3, 3) and therefore draws three times the line's width (5 pixels) and then skips three times the line's width.
`StrokeDashCap`	Determines the shape of the ends of dashes. This can be `Flat`, `Round`, `Square`, or `Triangle`. The `Flat` style ends a dash exactly where it should end. The `Round`, `Square`, and `Triangle` styles place a shape (circle, square, or triangle) at the end of the dash so it extends slightly into the space between dashes.
`StrokeDashOffset`	Determines the distance from the beginning of a line to the start of the first dash.
`StrokeEndLineCap`	Determines the shape of the end of a line. This can be `Flat`, `Square`, `Round`, or `Triangle`. If the line ends at a blank space between dashes, then it ends with the dash's end cap style instead of the line's. If the line ends close to the end of a dash, the `Round`, `Square`, and `Triangle` end caps may extend slightly beyond the end of the segment. (See the description of the `StrokeDashCap` property.)
`StrokeLineJoin`	Determines how adjacent lines are joined in a `Rectangle`, `Polyline`, `Polygon`, or `Path`. This can be `Miter` (edges extend until they intersect), `Bevel` (corners are cut off symmetrically), or `Round` (corners are slightly rounded). The effect is subtle unless the `StrokeThickness` is relatively large.
`StrokeStartLineCap`	Determines the shape of the start of a line. This is similar to `StrokeEndLineCap`.
`StrokeThickness`	The width of the line

The following XAML code creates the `Ellipse` and `Line` shown in Figure 8-1:

Available for download on Wrox.com

```
<Ellipse Width="100" Height="50"
 Stroke="Blue" StrokeThickness="5" Fill="Cyan"
 HorizontalAlignment="Left" VerticalAlignment="Top"
 Margin="10,10,0,0"/>

<Line X1="200" Y1="20" X2="50" Y2="100" Stroke="Red" StrokeThickness="5"
 StrokeDashArray="3,3"/>
```

UseEllipseLine

Chapter 10, "Pens and Brushes," describes stroke attributes in greater detail and provides examples.

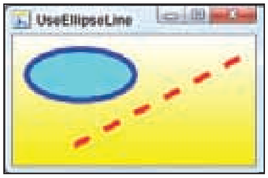


FIGURE 8-1

ELLIPSE

The `Ellipse` control draws a simple ellipse that cannot contain any children. Normally an `Ellipse` control doesn't interact with the user, although it provides `MouseDown`, `MouseEnter`, `MouseLeave`, and other events that you can catch if you like.

The `Ellipse`'s `Width` and `Height` properties determine its size. The control's location is determined by the container that holds it. For example, the `Ellipse` shown in Figure 8-1 is contained in a `Grid`, and the location of its upper-left corner is determined by its `HorizontalAlignment` (`Left`), `VerticalAlignment` (`Top`), and `Margin` (`10,10,0,0`) properties.

If you omit the `Width` and `Height` properties, the `Margin` property can make the `Ellipse` resize as its container resizes.

LINE

The `Line` control draws a line segment that cannot contain any children. Instead of `Width` and `Height` properties, the `Line` control's size and location are determined by its `X1`, `Y1`, `X2`, and `Y2` properties. The line starts at the point (`X1`, `Y1`) and extends to the point (`X2`, `Y2`).

Like the `Ellipse`, the `Line` control normally doesn't interact with the user, but it provides a set of events that you can catch just in case.

PATH

The `Path` control draws a series of lines, arcs, Bézier curves, and other shapes. You can use child objects to define a `Path`'s shapes, but it's usually easier to use the Path mini-language.

Path Mini-Language

To use the Path mini-language, you give the control a `Data` attribute. That attribute includes a list of single-letter abbreviations for drawing commands interspersed with point coordinates that determine where to draw.

Figure 8-2 shows a program that draws a `Path`, a `Polygon`, and a `Polyline`, all using the same point coordinate data.

FIGURE 8-2

The following XAML code shows how the program draws its `Path` object. The `Data` attribute contains a series of commands in the Path mini-language. In this example, the `M41,3` command makes the object move the drawing position to the point (41, 3). The `L` followed by a series of coordinates makes the object draw a series of connected lines.

Available for download on Wrox.com

```
<Path Margin="15" HorizontalAlignment="Center"
 VerticalAlignment="Top" Width="Auto" Height="Auto"
 Fill="#FFFF0000" Stroke="#FF000000" StrokeThickness="5"
 Data="M41,3 L2,70 78,32 3,19 59,69 41,3"/>
```

UsePathPolygonPolyline

Many of the Path mini-language's commands are followed by one or more points that are used as parameters. You can separate the points or the coordinates within a point with commas or spaces. To make it easier to read the code, I use commas to separate a point's X and Y coordinates and spaces to separate different points.

Several of the mini-language's commands have uppercase and lowercase versions. The uppercase version means that the following points are in absolute coordinates, while the lowercase version means that the following points are relative to the previous points.

The following table describes the Path mini-language's commands:

COMMAND	MEANING
`F0`	Use odd/even fill rule (see Figure 8-3).
`F1`	Use nonzero fill rule (see Figure 8-3).
`M` or `m`	Move to the following point.
`L` or `l`	Draw lines to the following points.
`H` or `h`	Draw a horizontal line to the given X coordinate.
`V` or `v`	Draw a vertical line to the given Y coordinate.
`C` or `c`	Draw a cubic Bézier curve. This command takes three points as parameters: two control points and an endpoint. The curve starts at the current point moving toward the first control point and ends at the endpoint moving away from the second control point. (See Figure 8-4.)

COMMAND	MEANING
S or s	Draw a smooth Bézier curve. This command takes two points as parameters: a control point and an endpoint. The curve defines an initial control point by reflecting the final control point from the previous S command. It then draws a cubic Bézier curve using the newly defined control point and the two parameter points. This makes the second curve smoothly join with the previous one. (See Figure 8-4.)
Q or q	Draw a quadratic Bézier curve. This command takes two points as parameters: a control point and an endpoint. The curve starts at the current point moving toward the control point and ends at the endpoint moving away from the control point. (See Figure 8-4.)
T or t	Draw a smooth Bézier curve defined by a single point. This command takes a single point as a parameter and draws a smooth curve to that point. It reflects the previous T command's control point to define a control point for the new section of curve and uses it to draw a quadratic Bézier curve. The result is a smooth curve that passes through the points sent to consecutive T commands. (See Figure 8-4.)
A or a	Draws an elliptical arc starting at the current point and defined by five parameters: ➤ size — The X and Y radii of the arc ➤ rotation_angle — The ellipse's angle of rotation in degrees ➤ large_angle — 0 if the arc should span less than 180 degrees; 1 if it should span 180 or more degrees ➤ sweep_direction — 0 for counterclockwise; 1 for clockwise ➤ end_point — The point where the arc should end
Z or z	Close the figure by drawing a line from the current point to the first point.

Figure 8-3 shows the difference between the odd/even and nonzero fill rules.

The PathBezier example program, shown in Figure 8-4, demonstrates the Path mini-language's Bézier curve commands. The program draws each curve with a thick blue line and then draws thin black line segments over the curve to show where its control points lie.

FIGURE 8-3

A Path Holding Objects

The Path mini-language isn't the only way to make the Path object draw. Instead of using the mini-language, you can place other objects inside the Path object to represent the shapes.

FIGURE 8-4

The PathObjects example program shown in Figure 8-5 uses the following XAML code to draw two curves. The code first uses the Path mini-language to draw a curve and then draws the same curve by placing objects inside a `Path` control.

FIGURE 8-5

Available for download on Wrox.com

```
<!--  Draw using the Path mini-language.  -->
<Path Stroke="Red" StrokeThickness="5"
 Data="M60,20 Q140,150 140,50 140,0 30,100"/>

<!--  Draw using objects.  -->
<Path Stroke="Red" StrokeThickness="5">
 <Path.Data>
  <PathGeometry>
   <PathGeometry.Figures>
    <PathFigureCollection>
     <PathFigure StartPoint="60,20">
      <PathFigure.Segments>
       <PathSegmentCollection>
         <PolyQuadraticBezierSegment
          Points="140,150 140,50 140,00 30,100"/>
       </PathSegmentCollection>
      </PathFigure.Segments>
     </PathFigure>
    </PathFigureCollection>
   </PathGeometry.Figures>
  </PathGeometry>
 </Path.Data>
</Path>
```

PathObjects

The two methods produce the same output, but the Path mini-language is much simpler and more concise.

POLYGON

The `Polygon` object draws lines segments that connect a series of points. It finishes by connecting the first point to the last point.

The following XAML code draws the polygon shown in the middle of Figure 8-2. The `Points` attribute at the end lists the coordinates of the points that the polygon connects.

Available for download on Wrox.com

```
<Polygon Margin="15" HorizontalAlignment="Center"
 VerticalAlignment="Top" Width="Auto" Height="Auto"
 Fill="#FFFF8000" Stroke="#FF000000" StrokeThickness="5"
 Points="41,3 2,70 78,32 3,19 59,69"/>
```

UsePathPolygonPolyline

POLYLINE

The `Polyline` object draws line segments that connect a series of points. Unlike the `Polygon` object, it does not finish by connecting the first point to the last point.

In fact, even if you repeat the first point at the end so the `Polyline` finishes at the first point, you still don't get exactly the same result you would get from the `Polygon` object.

A `Polygon` considers the join between the last and first lines as a corner in the shape. In contrast, the `Polyline` considers the last and first lines as two separate lines, even if they happen to end at the same point.

The PolygonPolylineDifference example program shown in Figure 8-6 draws a `Polygon` and a `Polyline` that connect the same points. The first and last points are the same for the `Polyline`, so it closes the shape.

FIGURE 8-6

If you look closely at Figure 8-6, you'll see that the top point on the left triangle (drawn with `Polygon`) isn't quite the same as the top point on the triangle on the right (drawn with `Polyline`). The `Polygon` treats this join as a corner, so it gets a nice point. The `Polyline` treats this as where the last and first line segments happen to start, so it draws both of their ends.

The following XAML code shows how the PolygonPolylineDifference program draws its `Polygon` and `Polyline`:

Available for download on Wrox.com

```
<StackPanel Orientation="Horizontal">
  <StackPanel>
    <Polygon Margin="15,5,0,0"
     HorizontalAlignment="Center" VerticalAlignment="Top"
     Fill="Yellow" Stroke="Black" StrokeThickness="10"
     Points="50,10 100,100 0,100"/>
    <Label HorizontalAlignment="Center" Content="Polygon"/>
  </StackPanel>
  <StackPanel>
    <Polyline Margin="25,5,0,0"
     HorizontalAlignment="Center" VerticalAlignment="Top"
     Fill="Yellow" Stroke="Black" StrokeThickness="10"
     Points="50,10 100,100 0,100 50,10"/>
    <Label HorizontalAlignment="Center" Content="Polyline"/>
  </StackPanel>
</StackPanel>
```

PolygonPolylineDifference

RECTANGLE

The `Rectangle` control draws a simple rectangle that cannot contain any children. The control's location and position are determined much as those of an `Ellipse`. The `Rectangle`'s size is determined by its `Width` and `Height` properties, and its location is determined by its container and its `Margin`, `HorizontalAlignment`, and `VerticalAlignment` properties.

The UseRectangle example program shown in Figure 8-7 uses the following XAML code to draw two `Rectangles`:

Available for download on Wrox.com

```
<Rectangle Width="Auto" Height="Auto" Margin="10"
 Stroke="Red" StrokeThickness="10"
 StrokeDashArray="3,3" StrokeDashCap="Triangle"/>

<Rectangle Width="100" Height="50"
 Stroke="Blue" StrokeThickness="5" Fill="Cyan"
 HorizontalAlignment="Center" VerticalAlignment="Center"/>
```

UseRectangle

In the first `Rectangle`, since the `Width` and `Height` are set to `Auto`, the control's `Margin` property determines the control's size and location. Because the `Margin` is 10 in this example, the `Rectangle` is positioned so its edges are 10 pixels from the edges of the `Grid` that contains it.

In the second `Rectangle`, the `Width` and `Height` are set explicitly. Since the control's `HorizontalAlignment` and `VerticalAlignment` properties are both set to `Center`, the control is given its specified size and centered in its container.

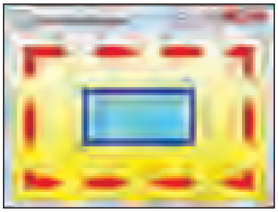


FIGURE 8-7

SUMMARY

This chapter describes WPF's 2D drawing controls. These controls let you draw ellipses, lines, rectangles, and other 2D shapes. Although these controls draw very different shapes, they have much in common such as their `Stroke` and `Fill` properties.

Most of these controls are relatively simple, although the `Path` control with its specialized mini-language can be quite complex. It's so complex, in fact, that you'll probably only use it for relatively simple shapes. For complex shapes, you may want to use program code to generate the `Path`'s drawing commands.

This chapter finishes the description of WPF controls. The controls described in Chapters 5 through 8 should meet most of your everyday programming needs.

The chapters that follow turn away from specific controls and discuss more general issues. Chapter 9, "Properties," explains in general terms how XAML code can set control properties. Since the examples so far in this book have used properties extensively, you probably already know how to set values for simple properties. Chapter 9 provides more background and explains how to set properties that take values that are objects themselves.

9

Properties

The preceding chapters in this book include close to 100 examples that demonstrate all sorts of control properties, methods, events, and other features provided by WPF. To keep those examples as simple and self-contained as possible, I sometimes glossed over some techniques used in the code and deferred them to later chapters.

This chapter covers one of those topics: *properties*. Many properties are relatively straightforward, and XAML, C#, and Visual Basic code can easily work with them.

Others, however, are more confusing. Some properties don't take values that don't have simple data types such as `Integer` or `String`, and using them can be confusing, particularly in XAML code.

This chapter describes WPF properties. It explains how you can set even complicated property values in XAML, C#, and Visual Basic code.

PROPERTY BASICS

The idea of a *property* is fairly simple: a *property* is a value that belongs to an object and that determines its behavior, appearance, or the data that it represents.

For example, if you create a `Person` class, it probably has properties like `FirstName`, `LastName`, and `StreetAddress` that define the data that a `Person` object represents.

Using this kind of simple property is easy. For example, the following C# code creates a `Person` object and sets its `FirstName` and `LastName` properties:

```
// Create a new Person object.
Person author = new Person();

// Set the Person's name properties.
author.FirstName = "Rod";
author.LastName = "Stephens";
```

XAML code also handles these kinds of simple-valued properties easily. The following XAML code sets a `Label` control's `Background` property to yellow:

```
<Label Content="Hello" Background="Yellow"/>
```

Using this kind of simple property is easy in your code, but behind the scenes, WPF is doing a bit more work than you might think. To translate XAML values into something comprehensible behind the scenes, WPF uses type converters.

TYPE CONVERTERS

While your XAML code simply sets a `Label` control's `Background` property to the string `"Yellow"`, the `Background` property is really not a simple string. It is actually an object that is an instance of the `System.Windows.Media.Brush` class.

The following C# code sets the `grdMain` control's `Background` property. Notice that the code doesn't set the property to a string value. Instead, it uses a static value provided by the `Brushes` class. The class's `Yellow` property returns a `SolidColorBrush` object (which is a type of `Brush`) that has the color yellow.

```
grdMain.Background = Brushes.Yellow;
```

Fortunately, XAML provides a set of *type converters* that can convert strings like `"Yellow"` into objects like brushes.

XAML provides type converters for many simple property types such as numbers (`Width`, `Height`), points, arrays of points (the `Polygon`'s `Points` property), other numeric arrays (`DashArray`), strings (the `Content` property when set to a string), simple colors (`Background`, `Stroke`, `Fill`), Booleans (`IsChecked`, `IsEnabled`), simple brushes (the `Button`'s `BorderBrush`), and many others.

XAML even has a type converter to translate a complex sequence of commands in the Path mini-language into a series of objects that produce lines, arcs, and curves. For example, the following XAML code draws the shape shown in Figure 9-1. The type converter translates the complex series of arc commands in the `Path` object's `Data` attribute into the curves shown in the figure. (For more information on the Path mini-language, see the section "Path Mini-Language" in Chapter 8.)

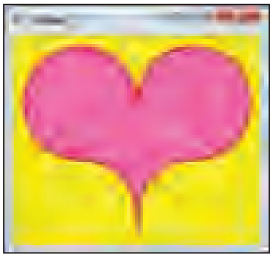


FIGURE 9-1

Available for download on Wrox.com

```
<Path Fill="HotPink" Stroke="Red" StrokeThickness="3"
 Data="M110,60
   a 50,50 90 0 1 50,-50
   a 50,50 90 0 1 50,50
   a 50,50 90 0 1 -50,50
   a 50,50 90 0 0 -50,50
   a 50,50 90 0 0 -50,-50
   a 50,50 90 0 1 -50,-50
   a 50,50 90 0 1 50,-50
   a 50,50 90 0 1 50,50"
/>
```

Heart

Unfortunately, XAML does not provide type converters for other, more complicated properties. For example, it's easy to describe a single-color `Brush` object with a string such as `Yellow`, `LightGreen`, or `#FF0C0C0` (a sort of dark turquoise). But what if you want to fill a `Grid` with the brush shown in Figure 9-2? It would be difficult to represent this complicated object as a simple string.

To use more complex objects in XAML code, you need to build the actual objects instead of describing them in a simple string. You can do that with property element syntax.

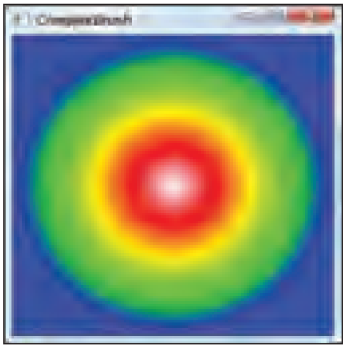


FIGURE 9-2

PROPERTY ELEMENT SYNTAX

Property element syntax lets XAML code represent a property as a separate element with its own start and end tags nested inside the object that will use it. In this example, the `Grid` that will use the background contains a new element that defines the background.

The property element's name consists of the control's type (`Grid`), followed by a dot (`.`), followed by the property's name (`Background`); thus, in this example, the element's name is `Grid.Background`.

Inside this new property element, the XAML code defines the object.

The following XAML code shows how the ComplexBrush example program displays the background shown in Figure 9-2:

Available for download on Wrox.com

```
<Grid>
  <Grid.Background>
    <RadialGradientBrush>
      <GradientStop Color="#FFFFFFFF" Offset="0.00"/>
      <GradientStop Color="#FFFF0000" Offset="0.25"/>
      <GradientStop Color="#FFFFFF00" Offset="0.50"/>
      <GradientStop Color="#FF00FF00" Offset="0.75"/>
      <GradientStop Color="#FF0000FF" Offset="1.00"/>
    </RadialGradientBrush>
  </Grid.Background>
</Grid>
```

ComplexBrush

The `Grid.Background` element is the `Grid`'s `Background` property element. It contains a `RadialGradientBrush` object. That object contains five `GradientStop` objects that indicate the colors that the brush should use at various positions. Finally, the `GradientStop` objects' `Color` and `Offset` properties are relatively simple values (simple colors and numbers), so they are represented as attributes that XAML's type converters can handle.

The following C# code shows how a program could build the same brush object in code-behind:

Available for download on Wrox.com

```
RadialGradientBrush br = new RadialGradientBrush();
br.GradientStops.Add(new GradientStop(Colors.White, 0.00));
br.GradientStops.Add(new GradientStop(Colors.Red,   0.25));
br.GradientStops.Add(new GradientStop(Colors.Yellow, 0.50));
```

```
br.GradientStops.Add(new GradientStop(Colors.Lime,  0.75));
br.GradientStops.Add(new GradientStop(Colors.Blue,  1.00));
grdBackground.Background = br;
```

MakeComplexBrush

This code creates a `RadialGradientBrush`. It then adds `GradientStop` objects to the brush's `GradientStops` collection.

NO TYPE CONVERTERS REQUIRED

Notice that the values passed to the `GradientStop` objects' constructor are a color and a numeric value. Since neither value is a string as it is in the XAML file, it doesn't need a type converter.

Property elements let you build quite complex objects in XAML code. Other common property elements that have been used in previous examples include:

- `Grid.ColumnDefinitions` — A collection of `ColumnDefinition` objects that determine how wide the `Grid`'s columns are
- `Grid.RowDefinitions` — A collection of `RowDefinition` objects that determine how tall the `Grid`'s rows are
- `BitmapEffect` — A control's `BitmapEffect` property determines whether the control displays a drop shadow, embossed surface, bevel, or other special effect. `BitmapEffect` must be an object, so XAML code must set it as a property element.

EFFICIENT EFFECTS

Microsoft *could* have defined a simple type converter for `BitmapEffect`. For example, it would convert the string `"Bevel"` into a `BevelBitmapEffect` object. Unfortunately, they didn't do this — so you're stuck using a property element.

- `Header` — The item displayed in a `MenuItem`, as the caption in a `GroupBox`, or on a `TabItem`'s tab is defined by the `Header` property. If you want to display simple text, then you can use an ordinary attribute to set `Header` to a string value. If you want to display something else (such as an `Image` or a `StackPanel`), then you need to use a property element.
- `ContextMenu` — A control's `ContextMenu` property defines the pop-up menu that appears when you right-click on the control. It's an object, so you must define it with a property element.
- `LayoutTransform` — A control's `LayoutTransform` property determines how the control is moved, stretched, or rotated before it is laid out and drawn. These transformations are relatively complex objects and thus you must define them with property elements.

The PropertyElements example program shown in Figure 9-3 demonstrates each of these property elements.

The following XAML code shows how the PropertyElements program works. The discussion following the code walks through what each of the property elements does. As you read the discussion, refer to Figure 9-3 to see the results.

Available for download on Wrox.com

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="Window1"
  x:Name="Window"
  Title="PropertyElements"
  Width="400" Height="300"
  FontWeight="Bold" FontSize="16">
  <Window.Background>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
      <GradientStop Color="Red" Offset="0"/>
      <GradientStop Color="Black" Offset="1"/>
    </LinearGradientBrush>
  </Window.Background>
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="*"/>
      <ColumnDefinition Width="*"/>
      <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="*"/>
      <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <Image Grid.Row="0" Stretch="Uniform"
     Grid.Column="0" Width="100" Height="Auto"
     Source="Lighthouse.jpg">
       <Image.BitmapEffect>
        <BevelBitmapEffect BevelWidth="10"/>
      </Image.BitmapEffect>
      <Image.ContextMenu>
        <ContextMenu>
          <MenuItem>
            <MenuItem.Header>
              <StackPanel Orientation="Horizontal">
                <TextBlock Text="Phone"
                 VerticalAlignment="Center"/>
                <Image Width="30"
                 Source="cellphone.ico">
                  <Image.BitmapEffect>
                    <DropShadowBitmapEffect/>
                  </Image.BitmapEffect>
                </Image>
              </StackPanel>
            </MenuItem.Header>
          </MenuItem>
          <MenuItem Header="Exit"/>
        </ContextMenu>
```

```
          </Image.ContextMenu>
        </Image>
        <GroupBox Grid.Row="0" Grid.Column="1" Grid.ColumnSpan="2" Margin="10">
          <GroupBox.Header>
            <StackPanel Orientation="Horizontal">
              <Label Content="Keys"/>
              <Image Width="30" Source="Keys.ico">
                <Image.BitmapEffect>
                  <DropShadowBitmapEffect/>
                </Image.BitmapEffect>
              </Image>
            </StackPanel>
          </GroupBox.Header>
        </GroupBox>
        <Image Grid.Row="1" Grid.Column="0" Width="75" Source="Frog.jpg">
          <Image.LayoutTransform>
            <RotateTransform Angle="10"/>
          </Image.LayoutTransform>
        </Image>
        <Label Content="Stretched Text" FontSize="25" Foreground="Yellow"
         HorizontalAlignment="Center" VerticalAlignment="Top" Margin="0,-40"
         Grid.Row="1" Grid.Column="1" Grid.ColumnSpan="2">
          <Label.RenderTransform>
            <TransformGroup>
              <ScaleTransform ScaleY="3"/>
              <SkewTransform AngleX="20"/>
              <RotateTransform Angle="20"/>
            </TransformGroup>
          </Label.RenderTransform>
        </Label>
      </Grid>
    </Window>
```

PropertyElements

The main `Window` element contains a `Window.Background` property element to make the `Window` display a red gradient brush.

FIGURE 9-3

The `Grid` control contains `Grid.ColumnDefinitions` and `Grid.RowDefinitions` property elements to define its three columns and two rows.

The `Image` displaying a lighthouse contains an `Image.BitmapEffect` property element to give the image a beveled appearance so it looks sort of like a tall button. The original image doesn't have the shaded edges — those are added by the `BevelBitmapEffect` object.

That `Image` also has an `Image.ContextMenu` property element to define its context menu. The `ContextMenu` contains two `MenuItems`.

The first MenuItem has a MenuItem.Header property element that makes the item display a StackPanel containing a TextBlock and an Image. Like the previous Image, this one uses an Image.BitmapEffect property element, this time to display a drop shadow.

If you unwind all of the closing tags until you reach the end of the first Image control, you'll find a GroupBox element. Its GroupBox.Header property element makes the control display a StackPanel containing a Label and an Image in its header. The Image uses an Image.BitmapEffect property element to display a drop shadow.

The program's next Image control uses an Image.LayoutTransform property element to transform the image. In this case, it uses a RotateTransform to rotate the image 10 degrees.

The program's final control is a Label. It uses a Label.RenderTransform object to transform itself. That object contains a TransformGroup that contains several transformations that are applied in sequence. Those transforms scale the Label by a factor of 3 vertically, skew the Label in the X direction by 20 degrees, and then rotate the Label by 20 degrees.

TRICKY TRANSFORMS

The PropertyElements example program uses LayoutTransform and RenderTransform objects, both of which transform a control.

The difference is that WPF applies any LayoutTransforms before it determines how it should lay out a window's controls. For example, suppose you rotate a very tall, thin rectangle by 90 degrees. The resulting rectangle is short and wide. Now WPF can use the rectangle's new transformed dimensions to determine how it should be arranged with the other controls.

In contrast, a RenderTransform decides where controls will be positioned first and only transforms the object afterward, right before it is drawn on the screen.

PROPERTY INHERITANCE

In object-oriented programming, you can define a *subclass* that represents a particular kind of class. For example, if you define a Person class, you could then define an Employee subclass that was a particular kind of Person.

The subclass inherits the properties, methods, and events defined by the *parent class*, and it may define new ones. For example, if the Person class defines the FirstName and LastName properties, then the Employee class inherits them. The Employee class may also add new properties such as EmployeeId, StartDate, and Salary that don't apply to all Person objects.

In addition to this object-oriented concept of class inheritance, WPF controls support another type of *property inheritance* in which a control inherits the property values of its container.

For example, suppose a StackPanel sets the property values FontName = Comic Sans MS, FontSize = 20, and FontWeight = Bold. Then if you place a Label inside the StackPanal, the Label inherits those property values, so it draws its text in 20-point bold Comic Sans MS.

Unfortunately, WPF property inheritance includes many exceptions; thus, you cannot always count on it. For example, the `ListBox` control highlights its currently selected item by making the item's background blue and its foreground white. Suppose you set the window's `Background` property to the same blue color. If the `ListBox` and the items it contains inherited that property, then all of the items in the `ListBox` would have blue backgrounds and the user wouldn't be able to tell which item was selected. To prevent this, the `ListBox` doesn't inherit its container's `Background` property value.

Of course, since you can explicitly set the `ListBox`'s `Background` property or change the `Background` properties of the items that it contains, this doesn't really solve the problem. It just makes it less likely that the problem will occur. You can still get into trouble by selecting poor background colors, but it won't happen unless you take extra action.

The InheritedProperties example program shown in Figure 9-4 demonstrates some properties that are inherited and some that are not.

FIGURE 9-4

The program's window sets its font properties to 18-point bold Comic Sans MS, and you can see in the figure that the program's `ComboBox` and `ListBox` items, `Label`, `Button`, and `GroupBox` all inherit this font.

The program's window also sets its `Background` property to a green gradient brush. The program's `Label`, `StackPanels`, and `GroupBox` have transparent backgrounds so the brush shows through, but the `Button`, `ComboBoxes`, and `ListBox` display their own backgrounds.

The window sets its `Foreground` property to yellow. The program's `GroupBox` inherits this value, but the other controls do not.

One item in each of the program's `ComboBoxes` and in its `ListBox` has its `Background` set to yellow and its `Foreground` set to red, so you can see that you can explicitly set these colors if you want.

If you download the example program from the book's web site and experiment with it, you'll see that these explicitly set colors do interfere with the `ListBox` and `ComboBox`'s abilities to highlight their selected items.

The moral is that you should be aware that controls sometimes inherit property values from their containers — but not always.

ATTACHED PROPERTIES

Sometimes a control might need to have a property that contains information for use by another control. For example, consider a `Button` inside a `Grid` control. The `Grid` needs to determine how to arrange the `Button`. In particular, it needs to know what row and column should hold the `Button`. Although the `Grid` needs to know this information, the row and column really belong to the `Button`.

So how do you attach this information to the `Button` control? You could give `Button` the properties `Row` and `Column`. If you took that approach, then you would need to similarly define the same properties for every other kind of control on the off chance that instances were placed inside a `Grid`.

Even that might work (you could put the properties in the `Control` class and let every other control class inherit them), but there are lots of other properties that you might need to handle similarly. For example, the `DockPanel` control needs to know how to dock its children; the `Canvas` control needs to know `Top`, `Left`, and other position properties for its children; and the `ToolBar` needs to know what `OverflowMode` to use for the items it contains.

Adding all of these properties to the `Control` class would not only clutter the class, but it would also not give you a general solution. If you later created a new control that needed to know something about other controls (e.g., you make a new container class that needs to know how to arrange its children), you would need to rebuild the `Control` base class to accommodate the new control. I doubt that Microsoft is likely to recompile the WPF libraries and release a special build just to add your property.

OBJECT-ORIENTED OBJECTION

Adding all of these properties to the `Control` class would also mess up the classes' encapsulation. It would make the `Control` class know a lot about other classes when it really shouldn't. Object-oriented purists would say that disqualifies this approach from the start.

Instead, WPF takes a different approach, which allows a control to provide an *attached property* for use by other controls. For example, the `Grid` control actually implements the `Grid.Row` and `Grid.Column` attached properties, and other controls can use them.

The syntax for an attached property is the name of the property provider, followed by a dot, followed by the name of the property. For example, the `Grid` control's `Row` property is called `Grid.Row`.

PROPERTIES WITHOUT RESTRICTIONS

The control providing the attached property does not check to make sure that the property makes sense in context. For example, you can give a `TextBox` control a `Grid.Row` property even if it is not contained in a `Grid`. In that case, the property is simply ignored.

Unfortunately, the syntax for an attached property is very similar to the syntax for a property element. For instance, the AttachedProperties example program shown in Figure 9-5 uses the following XAML code. This code contains several attached properties and property elements.

Available for download on Wrox.com

```
<Grid>
  <Grid.Background>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
      <GradientStop Color="Yellow" Offset="0"/>
      <GradientStop Color="Red" Offset="1"/>
    </LinearGradientBrush>
  </Grid.Background>
  <Grid.RowDefinitions>
```

```
        <RowDefinition Height="40"/>
        <RowDefinition Height="*"/>
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
      </Grid.ColumnDefinitions>
      <Label Content="Select Transaction" Foreground="Yellow"
        HorizontalAlignment="Stretch" HorizontalContentAlignment="Center"
        Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2">
        <Label.Background>
          <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
            <GradientStop Color="Yellow" Offset="0"/>
            <GradientStop Color="Red" Offset="1"/>
          </LinearGradientBrush>
        </Label.Background>
      </Label>
      <Button Margin="10" Grid.Row="1" Grid.Column="0">
        <StackPanel>
          <Label Content="Credit" HorizontalAlignment="Center"
           Grid.Row="1" Grid.Column="1"/>
          <Image Source="Credit.jpg" Height="60" />
        </StackPanel>
      </Button>
      <Button Margin="10" Grid.Row="1" Grid.Column="1">
        <StackPanel>
          <Label HorizontalAlignment="Center" Content="Debit" />
          <Image Source="Debit.jpg" Height="60" />
        </StackPanel>
      </Button>
    </Grid>
```

AttachedProperties

FIGURE 9-5

The `Grid` control contains three property elements: `Grid.Background`, `Grid.RowDefinitions`, and `Grid.ColumnDefinitions`. These describe properties of the `Grid` control that happen to have objects for their values.

The first `Label` has three attached properties: `Grid.Row`, `Grid.Column`, and `Grid.ColumnSpan`. These properties are provided by the `Grid` class and tell the `Grid` containing the `Label` how to position it.

This `Label` also has a property element, `Label.Background`, that defines the `Label`'s background brush.

The first `Button` control has two attached properties, `Grid.Row` and `Grid.Column`, that tell the `Grid` where to place it. The `Button` contains a `StackPanel` that holds a `Label` and an `Image`.

The `Button`'s `Label` has attached attributes `Grid.Row` and `Grid.Column`. Because the `Label` is not directly contained in a `Grid`, those properties are ignored.

The second `Button` also has `Grid.Row` and `Grid.Column` properties to tell the `Grid` where to position it.

SUMMARY

This chapter explains how to use properties in XAML code. It explains how to use simple property values, values that are translated by type converters, property elements, and attached properties. These three kinds of properties let you define the property values that WPF controls need.

The next chapter describes a particularly useful set of properties: pen and brush properties. It explains how to build simple and complex pens and brushes that determine the colors used to outline and fill graphical objects such as text, backgrounds, lines, rectangles, and curves.

10

Pens and Brushes

Pens and brushes determine some of the most visually obvious pieces of a graphical application — its colors.

In graphical programming terms, a *pen* determines the color and style of a linear feature such as a line or the edge of a circle. A *brush* determines how an area is filled. For example, the ellipse drawn in Figure 10-1 is filled with a red brush and outlined with a thick, dashed, purple pen.

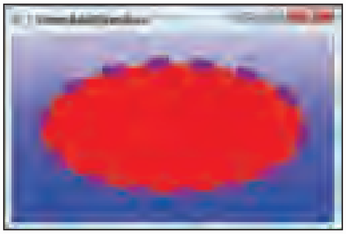


FIGURE 10-1

This chapter describes pen and brush properties and tells how to set them in XAML code. It explains the different kinds of pens and brushes and the properties that determine their appearance.

PENS

A *pen* defines the color and geometry of a linear feature such as a line, a curve defined by a `Path` control, or the edges of an ellipse or rectangle.

Interestingly, some colors that you might think would be determined by a pen are actually determined by a brush. In particular, text is rendered by filling the font's shapes with a brush rather than drawing the shapes with a pen. This is a bit counterintuitive if you think of drawing text by hand, where you usually use a pen (or a pencil or crayon, but probably not a brush unless you're practicing your Japanese calligraphy).

A pen has several different properties — all with names beginning with the word *stroke* — that determine its color and geometry.

A CLOSER LOOK

Many of the stroke properties don't make a huge difference to appearance unless the linear feature you're drawing is quite thick. For example, `StrokeDashCap` determines whether the ends of dashes are drawn as square, rounded, or triangular. If a line is only 1 pixel wide, you won't be able to tell the difference.

Remember, however, that WPF can draw objects that are scaled, so, if your program allows it, the user can zoom in on linear features until these properties make a big difference. The MagnifiedLines example program shown in Figure 10-2 draws the exact same line at five different scales. Only in the thickest lines can you tell that the dash caps are triangular.

FIGURE 10-2

The following code shows how the MagnifiedLines program draws its thickest line. Because the line isn't as wide as the `ViewBox` that contains it, the `ViewBox` enlarges it to make it fit:

Available for download on Wrox.com

```
<Viewbox Stretch="Uniform" Height="20" Width="300" Margin="10,10,10,0">
  <Line X1="0" Y1="0" X2="18.75" Y2="0" Stroke="Blue"
   StrokeDashArray="3" StrokeDashCap="Triangle"/>
</Viewbox>
```

MagnifiedLines

Stroke

The `Stroke` property determines the color of a linear feature. You can specify the color in a couple of ways. First, you can give the color's name. Names that WPF understands include *Red*, *Chartreuse*, *LavenderBlush*, and about 150 others.

The following XAML code shows how the PensAndBrushes example program draws the ellipse shown in Figure 10-1. The `Stroke` property makes the ellipse's border purple.

```
<Ellipse Fill="Red" Stroke="Purple"
 Margin="20" StrokeThickness="10" StrokeDashArray="2,2"/>
```

The second way to specify a solid color is by giving its *color components*. This value can have one of two forms: #RRGGBB or #AARRGGBB. In both cases, the RR, GG, and BB parts include two hexadecimal digits giving the color's red, green, and blue color components between 00 and FF (255). For example, the value *#FF0000* is red, and the value *#FFFF00* is yellow (equal parts red and green).

In the second color component format, *AA* indicates the color's *alpha value* or *opacity*. This is also a two-digit hexadecimal value between 00 and FF, where *00* means that the color is completely transparent, and *FF* means that it is completely opaque.

The Opacity example program shown in Figure 10-3 uses the following XAML code to demonstrate opacity:

Available for download on Wrox.com

```
<Canvas>
  <Label Content="Opacity!" Canvas.Top="60" Canvas.Left="2"
    FontFamily="Times New Roman" FontSize="46" FontWeight="Bold"/>
  <Ellipse Fill="#80FF0000" Stroke="Black" Height="100" Width="100"
    Canvas.Top="10" Canvas.Left="10"/>
  <Ellipse Fill="#8000FF00" Stroke="Black" Height="100" Width="100"
    Canvas.Top="10" Canvas.Left="70"/>
  <Ellipse Fill="#800000FF" Stroke="Black" Height="100" Width="100"
    Canvas.Top="70" Canvas.Left="40"/>
</Canvas>
```

Opacity

The `Canvas` control contains a `Label` followed by red, green, and blue `Ellipses` with opacity set to 80. Since the value 80 in hexadecimal (128 in decimal) is halfway between 00 and FF, the colors are 50 percent opaque. That means that the lower ellipses show through those above and the text shows through all of them.

FIGURE 10-3

While these solid colors (whether opaque or not) seem like simple things, they are not. Even though they are logically pen colors, WPF implements them as brushes for drawing purposes. The solid colored pens described so far are actually provided by the `SolidColorBrush` class.

The fact that these colors are really defined by brushes means that you can use other types of brushes to determine pen color. Figure 10-4 shows the GradientPens example program displaying an ellipse, a rectangle, and a line with borders filled with gradient brushes.

The "Brushes" section later in this chapter says more about different kinds of brushes you can build, and you can use any of those brushes to define the way a pen is drawn.

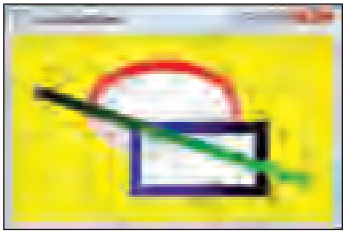


FIGURE 10-4

StrokeThickness

The `StrokeThickness` property is the simplest of the pen properties. It determines the thickness of a line in pixels. For example, the shapes drawn in Figure 10-4 all have `StrokeThickness` set to 10.

StrokeDashArray

The `StrokeDashArray` property is an array of values indicating the number of pen units that should be drawn and skipped while drawing. A *pen unit* is equal to the thickness of the linear feature set by the `StrokeThickness` property.

For example, suppose a `Line` has `StrokeThickness = 10`. Then the `StrokeDashArray` value "`2,2,4,2`" means to draw 20 pixels, skip 20 pixels, draw 40 pixels, skip 20 pixels, and repeat as needed to finish the `Line`.

Figure 10-5 shows the StrokeDashArrays example program displaying several dash patterns. The labels on the left indicate the lines' `StrokeDashArray` values. The first two lines are 1 pixel wide, while the others are 10 pixels wide. If you look closely, you can see how the `StrokeThickness` value helps determine the size of the dashes.

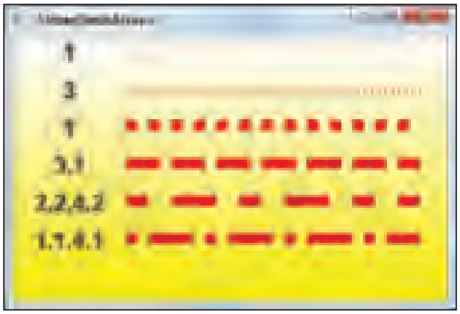


FIGURE 10-5

StrokeDashCap

The `StrokeDashCap` property determines how the ends of dashes are drawn. This property can take the values `Flat`, `Square`, `Round`, and `Triangle`. Figure 10-6 shows the results of each of these values.

FIGURE 10-6

StrokeDashOffset

The `StrokeDashOffset` property determines how far into the first dash a line segment starts drawing. The property's value indicates a distance in pen units (the thickness of the line).

The StrokeDashOffsets example program shown in Figure 10-7 displays similar `Line` controls drawn with various `StrokeDashOffset` values. The labels on the left indicate the `StrokeDashOffset` values.

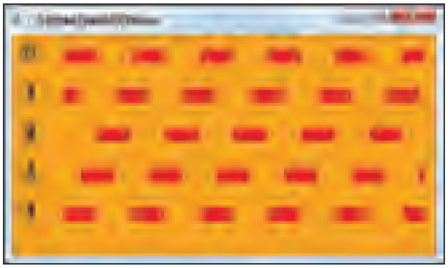


FIGURE 10-7

StrokeEndLineCap and StrokeStartLineCap

The `StrokeStartLineCap` and `StrokeEndLineCap` properties determine how the start and the end of a line are drawn. Like the `StrokeDashCap` property, these properties can take the values `Flat`, `Square`, `Round`, and `Triangle`. The StrokeLineCaps example program shown in Figure 10-8 shows the results of these values.

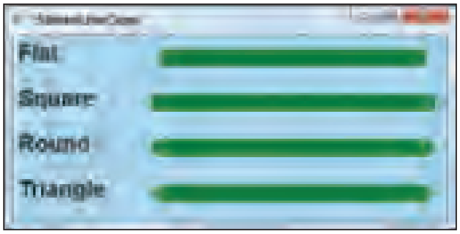


FIGURE 10-8

StrokeLineJoin

The `StrokeLineJoin` property determines how lines are connected in rectangles, polylines, polygons, and other shapes that draw connected line segments. This property can take the values `Miter`, `Bevel`, and `Round`.

FLAT AND SQUARE

Notice that the `Flat` and `Square` dash caps don't quite line up. `Flat` makes the dash end abruptly after its proper length. `Square`, `Round`, and `Triangle` all extend the end of the dash half of its width further to draw their end effects.

END CAP EDIFICATION

Note that a line only draws its start or end caps if it starts or ends on a drawn piece of line. For example, if the line ends in the middle of a gap between dashes, then the line stops with the previous dash. If the `StrokeDashCap` and `StrokeEndLineCap` properties are different, the result may look a bit strange.

`Miter` makes the edges of the lines extend until they meet, possibly resulting in a sharp point; `Bevel` cuts off the sharp point; and `Round` makes the corners rounded.

The StrokeLineJoins example program shown in Figure 10-9 demonstrates these values.

FIGURE 10-9

StrokeMiterLimit

The `StrokeMiterLimit` property limits how long the sharp point on a mitered corner can be before it is chopped off in a bevel. Technically it limits the ratio of miter length divided by half the `StrokeThickness`. In other words, if `StrokeMiterLimit` is 3, then the point at a corner can be at most three times half the line's thickness past the center of the line.

Intuitively, it's just easier to look at a picture and understand that larger `StrokeMiterLimit` values allow pointier corners.

The StrokeMiterLimits example program shown in Figure 10-10 shows the same curve with `StrokeMiterLimit` set to 1, 2, and 3. Notice that second corner near the middle of each shape is not beveled when `StrokeMiterLimit` is 2 or 3 because the corner does not reach its miter limit.

FIGURE 10-10

BRUSHES

A *brush* determines how an area is filled. The area may be simple (such as a rectangle or ellipse) or complex (such as a polygon, overlapping pieces of a polyline, or a shape drawn by a `Path` control).

As mentioned earlier in this chapter, brushes also determine how text is filled. That means that you can fill text with a solid color, a color gradient, or even a repeating pattern or picture. The PictureFilledText example program shown in Figure 10-11 draws some text filled with a picture of a field full of flowers.

FIGURE 10-11

Before you learn about the different kinds of brushes that you can create, you should learn about two properties that affect the way any brush is used: `FillRule` and `SpreadMethod`. The following two sections describe these properties. The sections after those describe the brush classes.

FillRule

If the lines that define an area cross each other, then the object's `FillRule` property determines which pieces of the area are filled. If `FillRule` is `Nonzero`, then any piece of the screen that is enclosed by the shape is filled. If `FillRule` is `EvenOdd`, then only pieces of the screen that are enclosed an odd number of times by the shape are filled.

The difference is hard to understand when described in words, but it's fairly easy to understand if you look at an example. In Figure 10-12, the FillRules example program shows the difference between the two `FillRule` values.

FIGURE 10-12

SpreadMethod

If a brush is not big enough to fill a drawn area, then its `SpreadMethod` property determines how the remaining area is filled. For example, if you're filling a large rectangle with a small picture, the brush isn't big enough to fill the rectangle. The `SpreadMethod` property can take the values `Pad`, `Reflect`, and `Repeat`.

The value `Pad` makes the brush fill the remaining area with its final color. For example, if the brush smoothly shades from white to green, then any remaining area is filled with green.

The value `Reflect` makes the brush reverse itself and continue filling the area. For example, suppose a brush shades from white to green. Then a larger area would be filled with white shading to green followed by green shading to white. This pattern would repeat until the entire area was filled.

The value `Repeat` makes the brush start over and repeat itself. If a brush shades from white to green, then a large area would be filled with white shading to green followed by an abrupt jump back to white shading to green again.

FILLRULE DETAILS

To better understand how `FillRule` works and what its values mean, pick a point somewhere on Figure 10-12 and mentally draw a ray from that point infinitely far to the right (off the page is far enough in this example).

For the `EvenOdd` rule, count the number of times the ray intersects one of the shape's segments. If the number of intersections is odd, then the point is "inside" the shape and is colored. If the number of intersections is even, then the point is "outside" of the shape and is not colored.

For the `Nonzero` rule, you need to consider the orientation of the shape's segments where they cross the ray. Each time a segment crosses the ray from left to right (as seen by the ray), add 1 to a counter. Each time a segment crosses the ray from right to left, subtract 1 from the counter. When you're done, if the counter is nonzero, then the point is "inside" the shape and is colored. If the count is zero, then the point is "outside" of the shape and is not colored.

Experiment with new points inside and outside of the shapes in Figure 10-12 and see what happens.

The SpreadMethods example program shown in Figure 10-13 demonstrates `LinearGradientBrushes` and `RadialGradientBrushes` using each of the three `SpreadMethod` values. Notice the abrupt change of colors when `SpreadMethod` is `Repeat`.

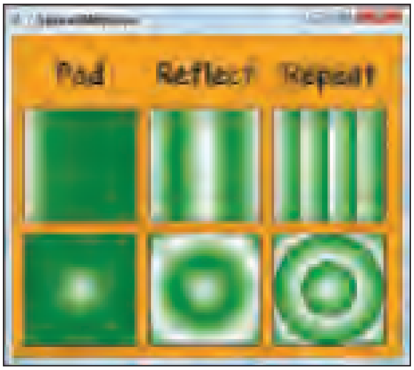


FIGURE 10-13

SolidColorBrush

The `SolidColorBrush` class represents a single solid color. In XAML code, you can specify the brush's color by name (e.g., *Red* or *HotPink*) or by hexadecimal value (e.g., *#FFFF0000* or *#FFFF69B4*).

Although it's usually easier to use a color's name or value, you can also explicitly build a `SolidColorBrush` object if you like.

The UseSolidBrush example program uses the following XAML code to demonstrate both methods. First, it uses the color name *Blue* to draw a blue ellipse. Then it draws a second ellipse, filling it with an explicitly created red `SolidColorBrush`.

Available for download on Wrox.com

```
<Ellipse Margin="10,20" StrokeThickness="5" Stroke="Black"
 Fill="Blue"/>
<Ellipse Margin="75,0" StrokeThickness="5" Stroke="Black">
  <Ellipse.Fill>
    <SolidColorBrush Color="Red"/>
  </Ellipse.Fill>
</Ellipse>
```

UseSolidBrush

LinearGradientBrush

A `LinearGradientBrush` fills an area with a sequence of colors that blend smoothly from one to another in a linear direction. For example, the brush might start blue on the left and gradually turn to white on the right.

The `LinearGradientBrush`'s `StartPoint` and `EndPoint` properties determine where the gradient starts and ends its colors. The coordinates of these points use a scale, where (0, 0) is the brush's upper-left corner and (1, 1) is its lower-right corner.

The brush contains a collection of `GradientStop` objects that determine how it fills its area. The `GradientStop` object's `Color` property defines the color that the brush should use at a particular point. Its `Offset` property determines how far through the brush from its `StartPoint` to its `EndPoint` the color should be positioned (on a 0 to 1 scale).

The brush can contain many `GradientStop` objects to define intermediate colors within the gradient.

The UseLinearGradientBrush example program shown in Figure 10-14 uses the following code to draw its rectangles:

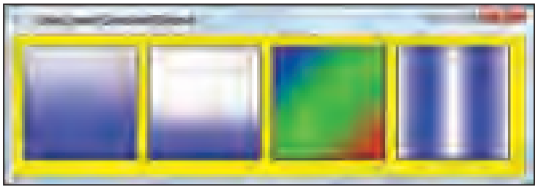


FIGURE 10-14

Available for download on Wrox.com

```
<Rectangle Margin="5" Width="100" Height="100" Stroke="Black">
  <Rectangle.Fill>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
      <GradientStop Color="White" Offset="0"/>
      <GradientStop Color="Blue" Offset="1"/>
    </LinearGradientBrush>
  </Rectangle.Fill>
</Rectangle>
<Rectangle Margin="5" Width="100" Height="100" Stroke="Black">
  <Rectangle.Fill>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
      <GradientStop Color="White" Offset="0"/>
      <GradientStop Color="White" Offset="0.5"/>
      <GradientStop Color="Blue" Offset="1"/>
    </LinearGradientBrush>
  </Rectangle.Fill>
</Rectangle>
<Rectangle Margin="5" Width="100" Height="100" Stroke="Black">
  <Rectangle.Fill>
    <LinearGradientBrush EndPoint="1,1" StartPoint="0,0">
      <GradientStop Color="Blue" Offset="0"/>
      <GradientStop Color="Lime" Offset="0.5"/>
      <GradientStop Color="Red" Offset="1"/>
    </LinearGradientBrush>
  </Rectangle.Fill>
</Rectangle>
<Rectangle Margin="5" Width="100" Height="100" Stroke="Black">
```

```
      <Rectangle.Fill>
        <LinearGradientBrush EndPoint="0.25,0.5" StartPoint="0,0.5"
         SpreadMethod="Reflect">
          <GradientStop Color="White" Offset="0"/>
          <GradientStop Color="Blue" Offset="1"/>
        </LinearGradientBrush>
      </Rectangle.Fill>
    </Rectangle>
```

UseLinearGradientBrush

The first rectangle shades vertically from white at the top point (0.5, 0) to blue at the bottom point (0.5, 1).

The second rectangle adds an additional `GradientStop` object to make the gradient remain white until halfway through the brush before it starts shading to blue.

The third rectangle's `Brush` uses different `StartPoint` and `EndPoint` values to make the gradient move from the upper-left corner (0, 0) to the lower-right corner (1, 1). It uses three `GradientStop` objects to make the colors blend from blue to lime to red.

The final rectangle's `Brush` has `EndPoint` (0.25, 0.5), so the brush only covers a quarter of the rectangle's width. The `Brush`'s `SpreadMethod` is set to `Reflect`, so the `Brush` reflects to cover the rest of the rectangle.

RadialGradientBrush

The `RadialGradientBrush` blends colors smoothly radiating away from a central point.

Instead of using `StartPoint` and `EndPoint` properties to determine the `brush`'s shape, the `RadialGradientBrush` uses the properties `GradientOrigin`, `RadiusX`, and `RadiusY`.

`GradientOrigin` determines the point from which the colors radiate. By default, this is (0.5, 0.5), so the colors radiate from the center of the `Brush`'s (0, 0) to (1, 1) coordinate system.

`RadiusX` and `RadiusY` determine how far the brush extends horizontally and vertically from the center, again using the (0, 0) to (1, 1) coordinate system.

REMEMBER THE RADIUS

Remember that the radius is *half* of the width of a circle, so `RadiusX` and `RadiusY` values of 0.5 make the gradient fill the brush.

Note also that values of 0.5 make the gradient only reach to the *sides* of the brush not all the way into the corners. That means, for example, that the corners of a rectangle are not covered by the gradient, so how they are filled depends on the `SpreadMethod` property.

The UseRadialGradientBrush example program shown in Figure 10-15 draws several rectangles filled with `RadialGradientBrushes`. From left-to-right and top-to-bottom, these brushes have the following properties:

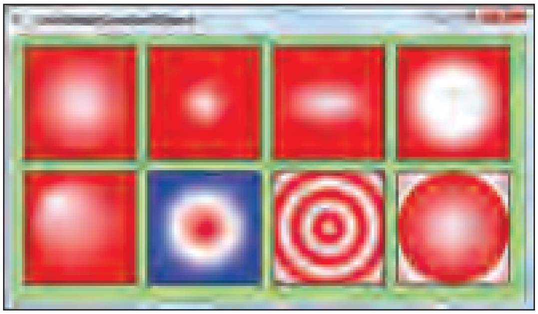


FIGURE 10-15

- A default brush shading from white to red
- `RadiusX` = 0.25 and `RadiusY` = 0.25
- `RadiusX` = 0.5 and `RadiusY` = 0.25
- An extra `GradientStop` object that makes the brush remain white until it is halfway through the brush
- `GradientOrigin` = 0.25, 0.25
- An extra `GradientStop` object that makes the brush shade from red to white to blue
- `RadiusX` = 0.1, `RadiusY` = 0.1, and `SpreadMethod` = `Reflect`
- `SpreadMethod` = `Repeat` (so you can see that the gradient doesn't reach the rectangle's corners)

Download the UseRadialGradientBrush program from the book's web site to view the complete source code.

TileBrush

`TileBrush` is a base class that represents a brush that fills an area with a repeating pattern. The `ImageBrush`, `DrawingBrush`, and `VisualBrush` subclasses fill areas with pictures, drawings, and user interface elements, respectively.

The section "Tile Brushes" and the sections that follow it in Chapter 3 describe the properties that you can use to define these types of brushes.

The following sections provide a bit more detail and some examples of these kinds of brushes.

ImageBrush

The `ImageBrush` class fills an area with a picture. See the section "Image Brush" in Chapter 3 for a general description of this class's most useful properties and instructions for building `ImageBrushes` in Expression Blend.

The ImageBrushTileModes example program shown in Figure 10-16 demonstrates the control's `TileMode` property values. This property determines how the brush

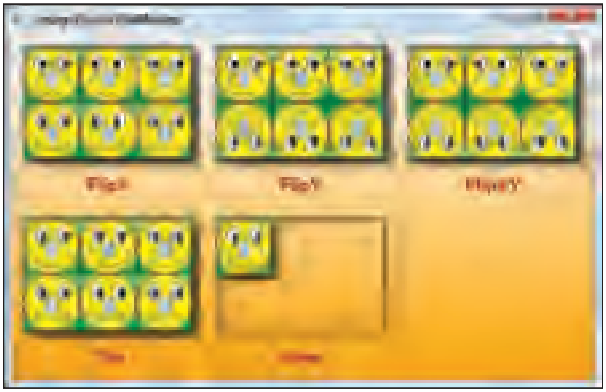


FIGURE 10-16

repeats itself to fill the area if necessary, similar to the way the `SpreadMethod` property determines how gradient brushes fill areas they don't cover.

DrawingBrush

The `DrawingBrush` class fills an area with a drawing. The drawing may contain labels, lines, polygons, or other controls.

Unfortunately, building a `DrawingBrush` in XAML code is rather difficult. The brush should contain a `Drawing` property element that defines the drawing.

The `Drawing` element can contain a drawing object such as a `GeometryDrawing`, a `GlyphRunDrawing`, or an `ImageDrawing`. Those objects define properties for the objects they contain. For example, the `GeometryDrawing` object's `Brush` and `Pen` properties apply to any objects drawn inside it.

If you want to include more than one drawing object, you can give the `Drawing` element a `DrawingGroup` child, which can contain other drawing objects such as a `GeometryDrawing`, a `GlyphRunDrawing`, or an `ImageDrawing`.

Creating a complex picture within a `DrawingBrush` is quite a chore. In fact, it's so difficult that you may wonder why you don't just use an `ImageBrush`, which is much simpler. But the advantage of the `DrawingBrush` is that it defines its contents as drawing commands so if you zoom in on a `DrawingBrush` you still see nice smooth curves. In contrast, if you zoom in on an `ImageBrush`, the brush becomes pixelated and blocky.

The MagnifiedDrawingBrush example program shown in Figure 10-17 displays a sequence of rectangles filled with the same `DrawingBrush`. Each rectangle is displayed inside a `Viewbox` that stretches its contents so you can see the brush at different scales.

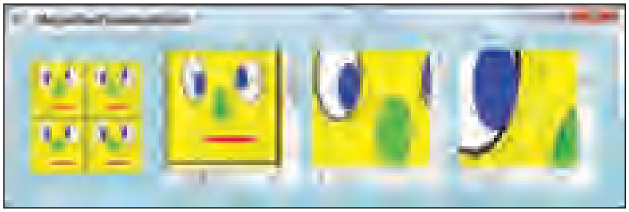


FIGURE 10-17

The following code shows how the MagnifiedDrawingBrush program creates its brush. Different parts of the brush are outlined and filled with different colors, so the brush contains a `DrawingGroup` that holds a series of `GeometryDrawing` objects that define the drawing's pieces.

Available for download on Wrox.com

```
<DrawingBrush x:Key="brFace" TileMode="Tile" ViewportUnits="Absolute"
 Viewbox="0,0,1,1" Viewport="0,0,50,50">
  <DrawingBrush.Drawing>
    <DrawingGroup>
      <!--  Background  -->
      <GeometryDrawing Brush="Yellow">
        <GeometryDrawing.Pen>
          <Pen Brush="Black"/>
        </GeometryDrawing.Pen>
        <GeometryDrawing.Geometry>
          <RectangleGeometry Rect="0,0,100,100"/>
        </GeometryDrawing.Geometry>
      </GeometryDrawing>
      <!--  Left eye  -->
      <GeometryDrawing Brush="White">
```

```
          <GeometryDrawing.Pen>
            <Pen Brush="Black"/>
          </GeometryDrawing.Pen>
          <GeometryDrawing.Geometry>
            <EllipseGeometry Center="25,25" RadiusX="10" RadiusY="20"/>
          </GeometryDrawing.Geometry>
        </GeometryDrawing>
        <!--  Left pupil  -->
        <GeometryDrawing Brush="Blue">
          <GeometryDrawing.Geometry>
            <EllipseGeometry Center="30,30" RadiusX="5" RadiusY="10"/>
          </GeometryDrawing.Geometry>
        </GeometryDrawing>
        <!--  Right eye  -->
        <GeometryDrawing Brush="White">
          <GeometryDrawing.Pen>
            <Pen Brush="Black"/>
          </GeometryDrawing.Pen>
          <GeometryDrawing.Geometry>
            <EllipseGeometry Center="75,28" RadiusX="12" RadiusY="18"/>
          </GeometryDrawing.Geometry>
        </GeometryDrawing>
        <!--  Right pupil  -->
        <GeometryDrawing Brush="Blue">
          <GeometryDrawing.Geometry>
            <EllipseGeometry Center="68,25" RadiusX="5" RadiusY="10"/>
          </GeometryDrawing.Geometry>
        </GeometryDrawing>
        <!--  Nose  -->
        <GeometryDrawing Brush="Lime">
          <GeometryDrawing.Pen>
            <Pen Brush="Green"/>
          </GeometryDrawing.Pen>
          <GeometryDrawing.Geometry>
            <EllipseGeometry Center="50,50" RadiusX="7" RadiusY="15"/>
          </GeometryDrawing.Geometry>
        </GeometryDrawing>
        <!--  Mouth  -->
        <GeometryDrawing>
          <GeometryDrawing.Pen>
            <Pen Brush="Red" Thickness="5"/>
          </GeometryDrawing.Pen>
          <GeometryDrawing.Geometry>
            <LineGeometry StartPoint="35,80" EndPoint="80,80"/>
          </GeometryDrawing.Geometry>
        </GeometryDrawing>
      </DrawingGroup>
    </DrawingBrush.Drawing>
  </DrawingBrush>
```

MagnifiedDrawingBrush

As you can see, building even a simple `DrawingBrush` in XAML by hand isn't easy. It is likely that most of the `DrawingBrushes` used by developers will be either very simple or defined by automated tools such as Expression Blend.

For instructions explaining how to make Expression Blend create a `DrawingBrush` for you, see the section “Drawing Brush” in Chapter 3.

VisualBrush

The `VisualBrush` class fills an area with a copy of a user interface element. That element can contain other controls such as `Lines`, `Labels`, `TextBoxes`, `Polygons`, and just about anything else that you want to add.

In fact, if you fill an area with a `VisualBrush` defined by a control and that control changes at run time, then the `Brush` automatically updates to match the control’s new appearance.

Unfortunately, the control you use to define the `Brush` cannot be invisible because the `Brush` needs to use its drawn image, and that image doesn’t exist if the control’s `Visibility` property is not set to `Visible`. That means you cannot make the control completely hidden and then use it only to fill areas on the screen.

You can, however, place the control behind another control or move it so it is off the program’s window.

The MagnifiedVisualBrush example program uses a `VisualBrush` to produce exactly the same result as the MagnifiedDrawingBrush program but uses a `VisualBrush` instead of a `DrawingBrush`.

The following XAML code shows how the MagnifiedVisualBrush program builds the `Canvas` control that defines its `VisualBrush`. Note how the code sets the `Canvas` control’s `Margin` property to `"-120,0,0,0`” to move the control off the left edge of the window so it is invisible at run time.

Available for download on Wrox.com

```
<Canvas HorizontalAlignment="Left" Margin="-120,0,0,0"
 Name="cvsFace" Width="100" Height="100" Background="Yellow">
  <Rectangle Canvas.Left="0" Canvas.Right="0" Width="100" Height="100"
   Stroke="Black" Fill="Yellow"/>
  <!-- Left eye -->
  <Ellipse Canvas.Left="15" Canvas.Top="5" Width="20" Height="40"
       Stroke="Black" Fill="White"/>
  <!-- Left pupil -->
  <Ellipse Canvas.Left="25" Canvas.Top="20" Width="10" Height="20"
       Fill="Blue"/>
  <!-- Right eye -->
  <Ellipse Canvas.Left="63" Canvas.Top="10" Width="24" Height="36"
       Stroke="Black" Fill="White"/>
  <!-- Right pupil -->
  <Ellipse Canvas.Left="63" Canvas.Top="15" Width="10" Height="20"
       Fill="Blue"/>
  <!-- Nose -->
  <Ellipse Canvas.Left="43" Canvas.Top="35" Width="14" Height="30"
       Stroke="Green" Fill="Lime"/>
  <!-- Mouth -->
  <Line X1="35" Y1="80" X2="80" Y2="80" Stroke="Red" StrokeThickness="5"/>
</Canvas>
```

MagnifiedVisualBrush

At 22 lines, this version is much shorter than the 65 lines used by the previous version to define a `GeometryGroup`. (If you remove the comments, which are much less important in this simpler version, it’s only 16 easy-to-follow lines long.)

This version also uses simpler controls that are easier to understand.

Having built the control that defines the `Brush`, the MagnifiedVisualBrush program must still actually create the `Brush`. Fortunately, that's relatively short and easy.

Available for download on Wrox.com

```
<VisualBrush x:Key="brFace" Visual="{Binding ElementName=cvsFace}"
  TileMode="Tile" Viewbox="0,0,1,1"
  Viewport="0,0,50,50" ViewportUnits="Absolute" />
```

MagnifiedDrawingBrush

BRUSHES THE EASY WAY

I don't know about you, but I think the `VisualBrush` is a whole lot easier to build than the `DrawingBrush`. Unless I have a *really* simple drawing, I use the `VisualBrush`.

SUMMARY

`Pens` and `Brushes` determine the appearance of just about everything that your WPF application draws. `Pens` determine how lines are drawn, and `Brushes` determine how areas are filled.

This chapter describes pens and brushes. It explains `Pen` properties and the `Brush` classes `SolidColorBrush`, `LinearGradientBrush`, `RadialGradientBrush`, `ImageBrush`, `DrawingBrush`, and `VisualBrush`.

With these properties and brushes, you can produce just about any drawing you can imagine.

WPF program development involves two distinct steps: creating the user interface and writing the code that sits behind it. Most of the chapters up until this point in the book explain how to build the user interface. The next chapter explains the other half of the equation: how to put code behind the controls.

11

Events and Code-Behind

Unless your application consists solely of user interface (UI) elements (a loose XAML page might), you'll eventually need to associate program code with the UI elements. For example, the program will need to take action when the user clicks buttons, selects menu items, and clicks tools in the toolbar.

In WPF, the code that sits behind the user interface, responding to control events and performing other processing, is called the *code-behind*. Visual Studio makes writing code-behind easy. This chapter explains how to write code that handles control events so the application can respond to the user at run time.

Depending on what you're trying to accomplish and which language you're using, you have several options for connecting XAML objects to code-behind:

- Using an event name attribute in XAML code
- Adding event handlers at run time
- Using the `Handles` clause (Visual Basic only)

The following sections describe these options and provide examples. Before you learn about event handlers and how to create them, however, it's worth taking a little time to learn about code-behind files.

CODE-BEHIND FILES

Whenever you add a window to a project, Expression Blend or Visual Studio adds a corresponding code-behind file. The file has the same name as the XAML file with the extra extension *cs* (for C#) or *vb* (for Visual Basic). For example, when you create a new WPF project in

C#, the project includes the initial window in the file Window1.xaml and the corresponding code-behind file Window1.xaml.cs.

If you're using Visual Studio, then you can edit the code-behind file and take full advantage of Visual Studio's powerful code-editing features such as keyword highlighting and IntelliSense.

If you're using Expression Blend, then you have two options for editing the code-behind file.

First, you can right-click on the file's name in the Project file list, and select "Edit Externally." This opens the file for editing in whatever application your system has associated with this type of file. For example, if you're using C# and you don't have Visual Studio installed, the system may open the file in Notepad, or it may ask you to select an application for editing the file. If you have Visual Studio installed, then your system will probably open the file in Visual Studio.

If you use this method to open the code-behind file in Notepad, WordPad, or some other text editor, then you don't get any of the advantages of Visual Studio. You can enter and edit code in the file, but you have to do all of the typing correctly by yourself without the benefits of IntelliSense. (There are even some third-party Code Editors such as the Antechinus C# Editor. See `www.c-point.com/c_sharp_editor.php` for more information.)

The second way in which you can edit a code-behind file is to right-click on it and select "Edit in Visual Studio." This opens the whole project in Visual Studio so you can take advantage of all of Visual Studio's capabilities. When you switch back and forth between Visual Studio and Expression Blend, the two applications synchronize your changes so they stay up to date.

SAVE YOUR CHANGES

Before you switch from Visual Studio to Expression Blend or vice versa, save any changes you have made so that the other application can see them. For example, if you make changes in Visual Studio and then switch to Expression Blend without saving your changes, Expression Blend will not see the changes. If you then make more changes in Expression Blend, the two programs will have inconsistent views of the project. The result can be very confusing and will probably result in some lost changes. Avoid this by always saving before you switch applications.

Whether you program in C# or Visual Basic, Visual Studio sets up the namespace references you need to access the WPF controls in your code. For example, the following XAML code defines a `TextBox` named *txtFirstName*:

```
<TextBox Name="txtFirstName" />
```

Your code can refer to this control simply as `txtFirstName`. You don't need to do anything extra to use the control in code.

EXAMPLE CODE

This chapter's source code, which is available for download on the book's web site at `www.wrox.com`, includes several example applications that all do the same thing but use different techniques for attaching code-behind to the user interface.

FIGURE 11-1

Figure 11-1 shows one of the programs in action. You can drag the scrollbars or enter values in the textboxes to set the red, green, and blue components of the color displayed on the right. If you click on the Apply button, the program sets the window's background color to match the current sample.

The ImageColors example program, which is shown in Figure 11-2, demonstrates all of the different techniques for attaching UI elements to code-behind in a single program. The program's different buttons are attached to event handlers in different ways. Download the C# or Visual Basic version of this program from the book's web site to see the details.

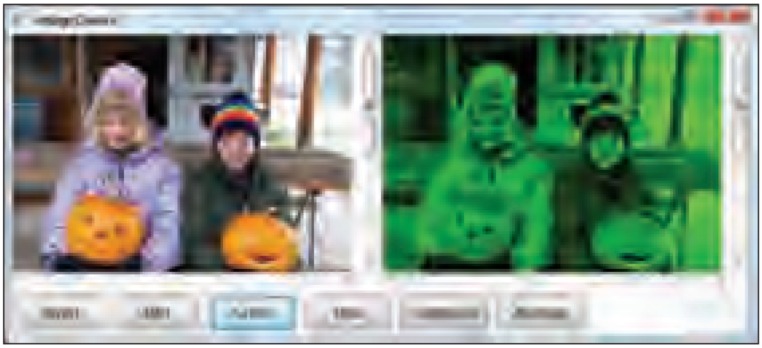


FIGURE 11-2

EVENT NAME ATTRIBUTES

The first way to attach code-behind to a control's events uses an attribute in the XAML code. The attribute's name is the same as the event that you want to catch. Its value is the name of the event handler in the code-behind.

For example, the following XAML code defines a `Button` named `btnApply`. The `Click` attribute indicates that the routine `btnApply_Click` catches the control's `Click` event handler.

```
<Button Content="Apply" IsDefault="True"
 Name="btnApply" Click="btnApply_Click"/>
```

Unfortunately, if you simply add the event handler declaration to the XAML code, the program won't run. If you try to execute it in Expression Blend or Visual Studio, you'll receive an error that says your program doesn't contain a definition for the event handler routine.

INCOGNITO CONTROLS

You don't really need to give a control a name to give it an event handler. For example, the `Button` in the previous code doesn't need to be called *btnApply* or anything at all. The `Click` attribute tells the program what event handler to execute. If you're never going to refer to the `Button` in code (e.g., to enable or disable it), then you can omit the name.

Some development projects have standards for naming controls, however, and some may insist that any control with an event handler have a name.

SEPARATE NO MORE

The fact that the program won't run without the event handler highlights a pretty serious breakdown in the separation between UI construction and the code-behind. It means that the UI designer cannot test the interface until the event handlers exist, but the event handlers are not part of the interface so they should be none of the interface designer's business. You could argue that the programmer should make the event name attribute in the XAML code after writing the event handlers, but the programmer shouldn't need to touch the interface code.

In practice, this is probably not a huge problem — it just means that the interface designer and programmer need to speak to each other.

The techniques for attaching code-behind described later in this chapter — adding event handlers at run time and using the `Handles` clause (in Visual Basic) — avoid this problem.

The following C# code handles the event declared in the previous XAML code:

```
private void btnApply_Click(object sender, RoutedEventArgs e)
{
  this.Background = borSample.Background;
}
```

This code catches the `btnApply` `Button`'s `Click` event and sets the window's `Background` property to the value used by the `borSample` control's `Background` property (more details on this shortly).

Your code doesn't need to do anything special to wire up the event handler to catch the event — Expression Blend and Visual Studio take care of that for you.

The following XAML fragment shows how the EventNameAttributes example program determines which event handlers catch the key events raised by the program's Apply button, and the scrollbar and textbox that control the color's red component. The code for the other scrollbars and textboxes is similar.

Available for download on Wrox.com

```
<ScrollBar Orientation="Horizontal" Minimum="0" Maximum="255"
 Grid.Row="0" Grid.Column="0" Value="255"
 Name="scrRed" ValueChanged="scrRed_ValueChanged"/>

<TextBox Margin="2" Grid.Row="0" Grid.Column="1" Text="255"
 HorizontalContentAlignment="Right" VerticalContentAlignment="Center"
 Name="txtRed" TextChanged="txtRed_TextChanged" />

<Button Grid.Row="3" Grid.Column="0" Grid.ColumnSpan="3" Width="100"
Height="40"
 Content="Apply" IsDefault="True"
 Name="btnApply" Click="btnApply_Click"/>
```

EventNameAttributes

The following C# fragment shows how the EventNameAttributes program responds when you use the first scrollbar or textbox to change the red color component, and when you click on the Apply button. The event handlers for the other scrollbars and textboxes are similar.

Available for download on Wrox.com

```
// A ScrollBar has been changed. Update the corresponding TextBox.
private void scrRed_ValueChanged(object sender,
 RoutedPropertyChangedEventArgs<double> e)
{
  if (txtRed == null) return;
  txtRed.Text = ((int)scrRed.Value).ToString();
  ShowSample();
}

// A TextBox has been changed. Update the corresponding ScrollBar.
private void txtRed_TextChanged(object sender,
 TextChangedEventArgs e)
{
  // Keep the value within bounds.
  int value;
  try {
    value = int.Parse(txtRed.Text);
  } catch {
    value = 0;
  }

  if (value < 0) value = 0;
  if (value > 255) value = 255;
  txtRed.Text = value.ToString();

  if (scrRed == null) return;
  scrRed.Value = value;
  ShowSample();
}

// Display a sample of the color.
```

```
private void ShowSample()
{
  if (borSample == null) return;

  byte r, g, b;
  try {
    r = byte.Parse(txtRed.Text);
  } catch {
    r = 0;
  }
  try {
    g = byte.Parse(txtGreen.Text);
  } catch {
    g = 0;
  }
  try {
    b = byte.Parse(txtBlue.Text);
  } catch {
    b = 0;
  }
  borSample.Background = new SolidColorBrush(Color.FromRgb(r, g, b));
}
```

EventNameAttributes

When the scrollbar's value changes, the `ValueChanged` event handler converts the new value into an integer (to throw away any fractional part) and displays the result in the red component's textbox. It then calls `ShowSample` to display a sample of the new color.

When the textbox's value changes, the `TextChanged` event handler converts the new text value into an integer. It makes sure that the result is between 0 and 255, the allowed values for a color component, and redisplays the result. It then sets the corresponding scrollbar's value to this new value and calls `ShowSample` to display a sample of the color.

The `ShowSample` method gets the values in the red, green, and blue textboxes; uses them to create a color; and displays it in the `Background` property of the `Border` control `borSample`.

CONSISTENT CODE-BEHIND

This code-behind is more or less the same in later versions of the example program. Only the XAML code changes significantly.

Creating Event Handlers in Expression Blend

To create an event handler in Expression Blend, create the event name attribute in the XAML code. Then edit the code-behind file and type in the event handler code.

Unfortunately, Expression Blend will complain loudly if you make a mistake but won't give you any help in typing this code, so you need to type it correctly by yourself. You need to get all of the syntax and the event handler's parameters correct.

Creating Event Handlers in Visual Studio

Visual Studio provides three methods for creating event handlers that are much easier than the methods provided by Expression Blend: double-clicking a control, using the Properties window, or using XAML IntelliSense.

Double-Clicking a Control

The first method for creating an event handler is to simply double-click on the control on the Window Designer. This creates an event handler for the control's default event. For example, the default event for a `Button` is `Click`.

If you're using C#, Visual Studio adds an appropriate name attribute to the XAML code and creates a stub for the event handler in the code-behind file. (If you're using Visual Basic, then Visual Studio uses a `Handles` clause as described later in this chapter.)

NAME FIRST

Visual Studio uses the control's name to build a name for the event handler. For example, if you double-click on a `Button` named `btnSave`, then Visual Studio creates an event handler named `btnSave_Click`.

If the control doesn't have a name, Visual Studio gives it a name and then names the event handler after it. For example, it might give the `Button` the name `button1` and then name the event handler `button1_Click`.

If you want the control and event handler to have nice names, give the control a good name before you let Visual Studio create the event handler.

Using the Properties Window

The second way to make an event handler uses the Properties window. First select the control in the Window Designer. Next click on the Events button in the Properties window (it looks like a lightning bolt) to see a list of that control's events and find the event that you want to handle.

Now you have three options:

1. To create a new event handler with a default name, double-click on the event.
2. To create a new event handler with a name of your choosing, type the name next to the event and then double-click on the event.
3. To make the control use an existing event handler (e.g., if you want several `TextBoxes` to all fire the same event handler), click on the dropdown arrow to the right of the event and select an existing event handler from the list.

Figure 11-3 shows the Visual Studio Properties window displaying the events for the ScrollBar control scrGreen. In this figure, the control's ValueChanged event's dropdown is listing the code-behind methods that have the right signatures for the ValueChanged event. (Note the lightning bolt button that makes the window display events rather than properties.)

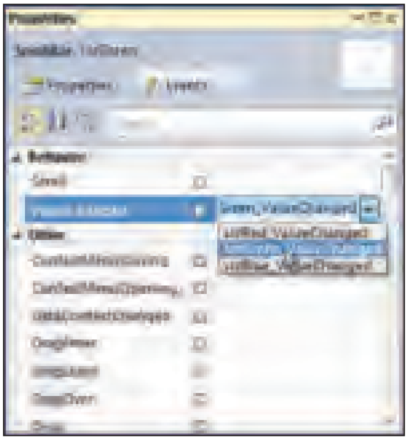


FIGURE 11-3

Using XAML IntelliSense

The third way to make an event handler in Visual Studio is to use the XAML Code Editor's IntelliSense. Start typing the event name attribute in the XAML Code Editor. When you type the equals sign, IntelliSense displays a list of existing event handlers that could catch the event. At the top of the list is the special entry <New Event Handler>. If you select this entry, Visual Studio invents an event handler name and creates an event handler stub.

> **NO HANDLES CLAUSE**
>
> If you're using Visual Basic, then this last method creates an event name attribute instead of using the Handles clause. That makes sense because you're creating the event handler by typing the event name attribute in the XAML Code Editor.

Figure 11-4 shows IntelliSense displaying the code-behind methods that have the right signature for the TextBox named txtBlue.

FIGURE 11-4

Relaxed Delegates

If you're using C#, you can skip this section, but if you're using Visual Basic, you can sometimes simplify event handlers.

Visual Basic supports *Relaxed Delegates*. These let you replace the data types of a routine's parameters with other data types that are consistent with whatever is actually passed into the routine. That lets you make an event handler's parameters either more or less specific than those that Visual Studio creates by default, and that can sometimes simplify the code.

For example, consider the following empty `Button Click` event handler generated by Visual Basic:

```
Private Sub btnApply_Click(ByVal sender As System.Object, _
 ByVal e As System.Windows.RoutedEventArgs)

End Sub
```

Now suppose you know that this event handler is only attached to `Button Click` events. Then you know that the `sender` parameter will always be a `Button`, so you can change its data type to `Button` in the event handler's declaration.

Meanwhile, you may not need to use the parameter `e`. That parameter contains information about the event such as the control that originated it, a `RoutedEvent` object representing the event, and a `Handled` flag that you can set to `True` to stop the event from further processing. Often event handlers have no need for this information. In that case, you can change the parameter's data type to the more general and simpler type `Object` instead of the more cumbersome `System.Windows.RoutedEventArgs`.

The following code shows the modified event handler. The new line of code inside the event handler displays the `Button`'s content converted into a string. If the `Button` displays a simple text caption, the code displays the caption.

```
Private Sub btnApply_Click(ByVal sender As Button, _
 ByVal e As Object)
  MessageBox.Show(sender.Content.ToString())
End Sub
```

When `sender` is declared as a `System.Object`, there's not a lot you can do with it. To access its properties, you first need to convert it into a `Control`, `Button`, or some other more specific class.

Declaring the `sender` parameter to be of type `Button` allows the code to skip that conversion and treat it as a `Button` right away. In this example, that means it can look at the control's `Content` property.

Declaring the parameter `e` as an `Object` just makes the code a little simpler.

Relaxed Delegates also let you omit the parameter list entirely if you don't need to use the parameters. The following code shows the simplest form of the `btnApply_Click` event handler:

```
Private Sub btnApply_Click()

End Sub
```

Of course, now you cannot refer to the parameters `sender` and `e` because they no longer exist; but that's often not a problem. In many applications, buttons, menu items, toolbar buttons, and other controls are attached to their own private event handlers that no other control shares. Only the `btnApply Button` sends its `Click` event to the `btnApply_Click` event handler so this routine can assume the user clicked this button. In that case, you can use Relaxed Delegates to omit the parameters and make the code a little easier to read.

The EventNameAttributesRelaxed example program uses the Visual Basic code in the following fragment to handle the events raised by its red scrollbar, red textbox, and Apply button. The code works the same way as the earlier C# version but in Visual Basic and without event handler parameters.

Available for download on Wrox.com

```
' A ScrollBar has been changed. Update the corresponding TextBox.
Private Sub scrRed_ValueChanged()
  If txtRed Is Nothing Then Exit Sub
  txtRed.Text = CInt(scrRed.Value).ToString()
  ShowSample()
End Sub

' A TextBox has been changed. Update the corresponding ScrollBar.
Private Sub txtRed_TextChanged()
  ' Keep the value within bounds.
  Dim value As Integer
  Try
    value = CInt(txtRed.Text)
  Catch ex As Exception
    value = 0
  End Try

  If value < 0 Then value = 0
  If value > 255 Then value = 255
  txtRed.Text = value.ToString()

  If scrRed Is Nothing Then Exit Sub
  scrRed.Value = value
  ShowSample()
End Sub

' Display a sample of the color.
Private Sub ShowSample()
  If borSample Is Nothing Then Exit Sub

  Dim r, g, b As Byte
  Try
    r = CByte(txtRed.Text)
  Catch ex As Exception
    r = 0
  End Try
  Try
    g = CByte(txtGreen.Text)
  Catch ex As Exception
    g = 0
  End Try
  Try
```

```
      b = CByte(txtBlue.Text)
    Catch ex As Exception
      b = 0
    End Try
    borSample.Background = New SolidColorBrush(Color.FromRgb(r, g, b))
  End Sub
```

EventNameAttributesRelaxed

EVENT HANDLERS AT RUN TIME

The preceding sections attach events to code-behind by placing the event handler's name in a XAML attribute such as `Click`.

You can also use code-behind to attach event handlers to controls at run time.

To use this technique, you don't need to add any reference to the event handler in the XAML code. Then, in the code-behind, you add code to attach the event handlers.

The following C# code shows how the RuntimeEventHandlers example program attaches event handlers to its control's events when it starts:

Available for download on Wrox.com

```
public Window1()
{
  this.InitializeComponent();

  // Insert code required on object creation below this point.
  // Attach the event handlers.
  scrRed.ValueChanged += scrRed_ValueChanged;
  scrGreen.ValueChanged += scrGreen_ValueChanged;
  scrBlue.ValueChanged += scrBlue_ValueChanged;
  txtRed.TextChanged += txtRed_TextChanged;
  txtGreen.TextChanged += txtGreen_TextChanged;
  txtBlue.TextChanged += txtBlue_TextChanged;
  btnApply.Click += btnApply_Click;
}
```

RuntimeEventHandlers

(In Visual Basic, attach the event handlers with the `AddHandler` statement.)

One nice feature of this technique is that the UI designer can work on the XAML file without ever needing to know the names of the event handlers in the code-behind. In fact, the interface designer doesn't even need to know what events are handled by the code. The only requirement is that the interface designer must give names to any controls with events that will be caught so the code-behind can refer to them.

EXTRA EVENTS

Don't use both techniques and name an event handler in the XAML code and also attach the same event handler at run time. If you do, then the event handler will execute twice each time the event occurs.

THE HANDLES CLAUSE

If you're using C#, you can skip this section, but if you're using Visual Basic, you have one more option for attaching event handlers to controls — the `Handles` clause.

Like the previous technique that attaches event handlers at run time, this technique does not add event declarations to the XAML file. Instead of attaching the event handlers at run time, the code uses `Handles` clauses to indicate which events the event handlers should catch.

The following code fragment shows how the HandlesClause example program declares event handlers for the red scrollbar and textbox and the Apply button. To save space, the code uses Relaxed Delegates and omits the bodies of the event handlers.

Available for download on Wrox.com

```
' A ScrollBar has been changed. Update the corresponding TextBox.
Private Sub scrRed_ValueChanged() Handles scrRed.ValueChanged
    ...
End Sub

' A TextBox has been changed. Update the corresponding ScrollBar.
Private Sub txtRed_TextChanged() Handles txtRed.TextChanged
    ...
End Sub

' Set the form's background to the sample color.
Private Sub btnApply_Click() Handles btnApply.Click
    ...
End Sub
```

HandlesClause

The `Handles` clauses make Visual Basic automatically attach the event handlers as needed.

Like the previous technique, this method separates the user interface design and the code-behind so the XAML code doesn't need to know anything about the event handlers.

EXTRA EVENTS, REDUX

As with the previous technique, you should be careful not to declare event handlers in the XAML and use the `Handles` clause. If you do both, then the event handler will execute twice each time the event occurs.

SUMMARY

Most applications provide code behind the user interface. The code responds to events generated by the controls, performs calculations, and uses the controls to display results.

WPF provides several methods for attaching code to the user interface including using event name attributes in the XAML code, attaching event handlers to events at run time, and (if you're using Visual Basic) the `Handles` clause. These techniques give you a trade-off between ease of use (event name attributes are easy) and separation between interface design and code-behind (attaching event handlers at run time keeps them more separate).

When you're writing application code in C#, Visual Basic, or some other language, you can reduce duplication by putting shared code in functions and then calling those functions. If the program needs to look up a customer in a database in ten different places, you can write a function to find the customer and then call it from the ten places.

Similarly, XAML allows WPF controls to share values such as colors, brushes, strings, and numbers. These values are stored in resources that are used in other parts of the XAML code.

The following chapter explains how you can use resources to remove duplicated code, make different elements more consistent, and centralize key values so they are easy to manage.

12

Resources

One of the most important concepts in any kind of programming is code reuse. Subroutines, functions, scripts, classes, inheritance, loops, and many other programming constructs let you reuse code in one way or another. For example, if your program needs to perform the same task in many places, you can create a subroutine that performs the task and then call it from those places.

This kind of code reuse has several advantages, including:

- You only need to write and debug the code once.
- If you need to modify the code later to make changes or fix a bug, you only need to do it in one place.
- When you fix bugs or make other changes to the code, you don't need to worry about keeping the changes in multiple parts of the code in synch.
- If you need to do something similar in another program, you may be able to copy the routine that performs the action.
- You need to write less code, which makes the program easier to read and more reliable. That in turn reduces cost.

Just as routines and functions let you reuse code in languages such as C# and Visual Basic, *resources* let you reuse XAML code. They let you define values that you can then use from many places in a XAML file.

Because XAML files are graphical, resources tend also to be graphical. They define objects and values that represent control properties that you can then apply to many controls to give them similar appearance or behavior.

This chapter describes resources and explains how you can define them and refer to them in your XAML code.

REUSE ABUSE

Code reuse can have a few disadvantages, particularly when done incorrectly.

One situation where too much code reuse can cause problems is when a developer tries to make a subroutine do too much. In that case the routine may become overly complicated and confusing, and you might be better off using several simpler routines instead.

A less common problem occurs when a routine is called too many times. The extra overhead of calling a subroutine slows a program down slightly. If the code calls the subroutine an enormous number of times (think billions not thousands), the routine may hurt performance.

In most cases, code reuse is a good thing. When in doubt, ask yourself whether the change will make the code more or less confusing, and whether it will be called so many times that it might hurt performance.

Similarly you can abuse resources in XAML. A resource can make it easy to give several controls a similar appearance, but it makes little sense to create a resource for every single control property whether it's shared or not.

DEFINING RESOURCES

Creating and using a simple resource is fairly easy. (In fact, it's easier to understand from an example than from a description, so, if you have trouble understanding the following explanation, look at the example text and then read the explanation again.)

To define a resource, add a `Resources` property element to an object such as a `Window`, `Grid`, or other container. Inside that element, place the resources that you will want to use later. Each resource is an element such as an object (e.g., a `LinearGradientBrush`, `Thickness`, or `Label`) or a simple value (e.g., a string or an integer).

You must give each resource a unique `x:Key` attribute value to identify it.

KEYS REQUIRED

The `Resources` element defines a *resource dictionary*, a list that allows objects to search for items based on their keys. A resource dictionary requires that every item have a key (except in some rare circumstances that aren't covered here), so you must give every resource a key.

For example, the following XAML code defines a Resources element that contains RadialGradientBrush named brButton and a BitmapEffectGroup named bmeButton. This Resources element is contained in the file's Window element (hence the tag Window.Resources).

Available for download on Wrox.com

```
<Window.Resources>
  <LinearGradientBrush x:Key="brButton" StartPoint="0,0" EndPoint="0,1">
    <GradientStop Color="Red" Offset="0"/>
    <GradientStop Color="White" Offset="0.5"/>
    <GradientStop Color="Blue" Offset="1"/>
  </LinearGradientBrush>
  <BitmapEffectGroup x:Key="bmeButton">
    <DropShadowBitmapEffect/>
  </BitmapEffectGroup>
</Window.Resources>
```

ButtonResources

Any object contained (directly or indirectly) in the object that holds the Resources element can use the resources. In the previous code, the Window contains the Resources element, so any object in the Window can use its resources.

After you have defined a simple property resource such as this one, you can use it with property attribute syntax. The value you give the attribute should have the format {StaticResource *resource_name*}, where you replace *resource_name* with the name of the resource.

For example, the following XAML code defines a Button. It sets the Button's Background property to the brButton resource and its BitmapEffect property to the bmeButton resource.

Available for download on Wrox.com

```
<Button Margin="4" Width="120"
 Background="{StaticResource brButton}"
 BitmapEffect="{StaticResource bmeButton}">
  <TextBlock TextAlignment="Center" Margin="4">
    Traveling<LineBreak/>Salesperson<LineBreak/>Problem
  </TextBlock>
</Button>
```

ButtonResources

The ButtonResources example program shown in Figure 12-1 displays several buttons that use similar code to define their backgrounds and bitmap effects. The only difference between the buttons is the contents of their TextBlocks.

FIGURE 12-1

Using resources has several advantages. Because the buttons shown in Figure 12-1 use the same resources, they are guaranteed to have a consistent appearance (at least as far as the Background and BitmapEffect properties are concerned).

Using resources simplifies the code by converting relatively complex property elements into simpler attribute elements. For example, the following code shows how you could define a similar `Button` without resources. This version is more than twice as long as the previous one. If the application contained several dozen `Buttons`, all of the extra code would quickly add up and make the code more cluttered and harder to understand.

Available for download on Wrox.com

```
<Button Margin="4" Width="120">
  <Button.Background>
    <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
      <GradientStop Color="Red" Offset="0"/>
      <GradientStop Color="White" Offset="0.5"/>
      <GradientStop Color="Blue" Offset="1"/>
    </LinearGradientBrush>
  </Button.Background>
  <Button.BitmapEffect>
    <DropShadowBitmapEffect/>
  </Button.BitmapEffect>
  <TextBlock TextAlignment="Center" Margin="4">
    Traveling<LineBreak/>Salesperson<LineBreak/>Problem
  </TextBlock>
</Button>
```

ButtonResources

The resources also allow you to easily change the appearance of all of the buttons at once. For example, the following XAML code defines resources with the same keys as before but with different values. When you make these changes to the `Windows.Resources` section of the code, the buttons immediately change their appearance.

Available for download on Wrox.com

```
<Window.Resources>
  <RadialGradientBrush x:Key="brButton">
    <GradientStop Color="White" Offset="0"/>
    <GradientStop Color="Blue" Offset="1"/>
  </RadialGradientBrush>
  <BitmapEffectGroup x:Key="bmeButton">
    <BevelBitmapEffect/>
  </BitmapEffectGroup>
</Window.Resources>
```

ButtonResources

Figure 12-2 shows the application's new appearance.

RESOURCE TYPES

A resource can have any data type that the XAML code can understand. Three particularly interesting categories of data types that a resource can have are "normal" property values, controls, and simple data types.

Normal Property Values

FIGURE 12-2

As the previous examples show, you can make resources that are `LinearGradientBrushes`, `RadialGradientBrushes`, and `BitmapEffectGroups`. You can also make resources that are `SolidColorBrushes`, `Thicknesses` (for `Margins`), `DashStyles`, `Colors` (although that's less useful than you might think — you probably want a `Brush` instead), `FontFamilies`, `FontStyles`, and just about any other property type you might want to use.

These "normal" property values are fairly easy to understand and make good intuitive sense. For example, it seems reasonable that you might want several controls to share the same `Background`, `Margin`, or `DashStyle` values.

The examples in the previous section demonstrated this kind of simple property type.

Controls

Resources often contain "normal" property values but they can also hold controls. For example, the following XAML code defines a `TextBlock` control that contains some text in two different styles:

Available for download on Wrox.com

```
<TextBlock x:Key="txtButton" TextAlignment="Center">
  Click Me<LineBreak/>
  <Run
   TextBlock.Foreground="Red"
   TextBlock.FontSize="20"
   TextBlock.FontStyle="Italic"
   TextBlock.FontWeight="Bold">
    PLEASE!
  </Run>
</TextBlock>
```

ContentResource

The ContentResource example program uses the following code to set a `Button`'s `Content` property to the `TextBlock` resource so it displays the text:

```
<Button Width="125" Height="75" Content="{StaticResource txtButton}"/>
```

Figure 12-3 shows the result.

In this example, making the `TextBlock` a resource doesn't save you much trouble. It requires about the same amount of code that you would use if the `Button` defined its own content and you are unlikely to want lots of buttons with the exact same content, so you probably won't use the resource to save duplication.

FIGURE 12-3

This technique is much more useful on the few occasions upon which you *do* want multiple copies of the exact same control. For example, suppose an application displays a series of pictures of people and you want each to provide the same context menu choices: Details, Email, Phone, and Delete. The following code defines a `ContextMenu` that includes those commands and a `BitmapEffectGroup` that displays drop shadows:

Available for download on Wrox.com

```
<Window.Resources>
  <ContextMenu x:Key="ctxPerson">
    <MenuItem Header="Details" Click="mnuDetails_Click"/>
    <MenuItem Header="Email" Click="mnuEmail_Click"/>
    <MenuItem Header="Phone" Click="mnuPhone_Click"/>
    <Separator/>
    <MenuItem Header="Delete" Click="mnuDelete_Click"/>
  </ContextMenu>
  <BitmapEffectGroup x:Key="bmePerson">
    <DropShadowBitmapEffect/>
  </BitmapEffectGroup>
</Window.Resources>
```

ContextMenuResource

The following code shows how an `Image` can use the resources:

Available for download on Wrox.com

```
<Image Name="imgClaude" Width="100" Height="100" Stretch="Uniform"
 MouseDown="img_MouseDown"
 Source="Claude.jpg" Tag="Claude"
 BitmapEffect="{StaticResource bmePerson}"
 ContextMenu="{StaticResource ctxPerson}"/>
```

ContextMenuResource

A series of `Buttons` can use similar code to display the same `ContextMenu`.

The program uses code-behind to respond when the user selects a context menu's `MenuItem`. Unfortunately the `Click` event handler raised by the `MenuItem` doesn't know which `Image` control the user clicked. The `MenuItem`'s parameters tell which `MenuItem` was clicked but not which `Image` displayed the `ContextMenu`.

One way to figure out which `Image` displayed the menu is to save a reference to that `Image` when the menu is displayed. The ContextMenuResource example program assigns the following event handler to all of its `Image` controls' `MouseDown` event handlers. If the user is pressing the right button over the `Image`, then the code saves a reference to the `Image` control for use later when the user selects a menu item.

Available for download on Wrox.com

```
// Remember the image clicked.
private Image m_ImageClicked = null;
private void img_MouseDown(object sender, MouseButtonEventArgs e)
{
  if (e.RightButton == MouseButtonState.Pressed)
  {
    m_ImageClicked = (Image)sender;
  }
}
```

ContextMenuResource

The following code shows how the program responds when the user selects the `ContextMenu`'s `Details` command. The code gets the `Image` control that was previously saved by the `MouseDown` event handler and uses its `Tag` property to display a message about the person clicked. (A real program would probably use the `Tag` property's value to look the person up in a database.)

Available for download on Wrox.com

```
// Display details for the person clicked.
private void mnuDetails_Click(object sender, RoutedEventArgs e)
{
  if (m_ImageClicked != null)
    MessageBox.Show("Display details for " + m_ImageClicked.Tag.ToString());
}
```

ContextMenuResource

Figure 12-4 shows the ContextMenuResource example program displaying its shared `ContextMenu`.

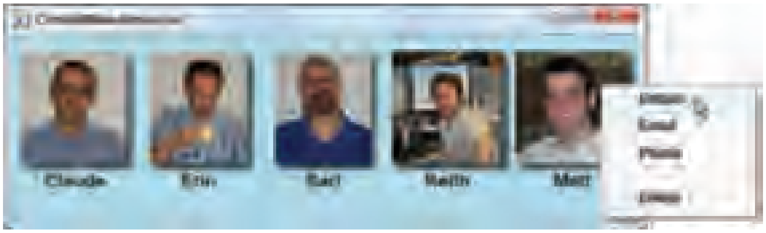


FIGURE 12-4

Simple Data Types

Ironically, some of the simplest data types (such as strings, integers, and doubles) are the most confusing to store in resources, at least until you know the trick. The problem is that a XAML file knows only about XAML data types such as `SolidColorBrush` and `TextBox`, not system data types such as `Double` and `Int32`.

To make the file correctly understand these data types, you need to give it a new namespace attribute telling it what namespace to look in for these data types.

For example, the following XAML snippet opens a `Window` element. The third namespace attribute indicates that the namespace prefix `sys` refers to data types defined in the mscorlib library.

Available for download on Wrox.com

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:sys="clr-namespace:System;assembly=mscorlib"
  x:Class="Window1"
  x:Name="Window"
  Title="ResourceHierarchy"
  Width="350" Height="180"
  FontSize="16" FontWeight="Bold">
```

ResourceHierarchy

Now the following code defines two resources with the `Double` data type. Notice how the code uses the `sys` namespace prefix to indicate where the `Double` data type is defined.

Available for download on Wrox.com

```
<Window.Resources>
  <sys:Double x:Key="dblRadX">5</sys:Double>
  <sys:Double x:Key="dblRadY">20</sys:Double>
</Window.Resources>
```

ResourceDictionaries

The following code shows how a `Rectangle` might use these resources to set its `RadiusX` and `RadiusY` properties:

```
<Rectangle Margin="5" Width="175" Height="60"
 Fill="Red" Stroke="Black" StrokeThickness="5"
 RadiusX="{StaticResource dblRadX}"
 RadiusY="{StaticResource dblRadY}"
/>
```

These simple data types let you place very specific values in resources. For example, you can build a resource dictionary containing all of the strings that your application will use for labels, buttons, menu items, and other controls. While you probably won't want to use the same text for several different buttons (although you might want a button and a menu item that have the same purpose to use the same text), placing these values in resources makes it easier to change them in a central location.

The SysDataTypes example program shown in Figure 12-5 displays sample resources in each of the simple data types defined in the mscorlib library.

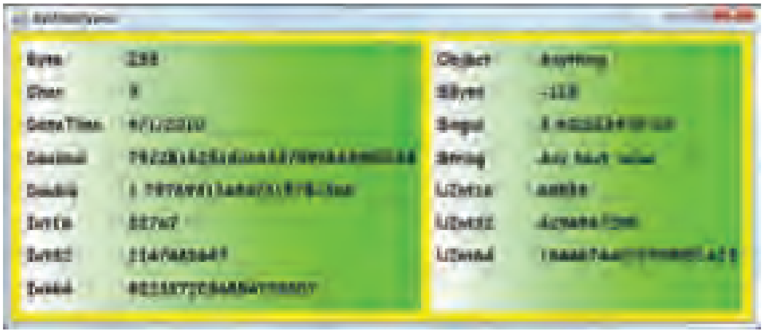


FIGURE 12-5

DISCOVERING DATA TYPES

In Visual Studio, you can use the Object Browser to see what data types are available in various libraries such as mscorlib.

You can also make more than one resource dictionary containing different values for the same resources and then easily switch between them. The section “Skins” in Chapter 16 has more to say about this technique.

RESOURCE HIERARCHIES

In a WPF logical tree, many objects might have resource dictionaries. When a control uses a resource, WPF looks upward through the tree until it finds an object that has defined a resource with the given name and uses it.

For example, suppose a `Window` contains a `StackPanel` that holds a `Label` and the `Label` uses a resource named `LabelValue`. When it builds the `Label`, WPF first checks the `Label`’s resource dictionary (if it exists) for a resource named `LabelValue`. If it doesn’t find one, WPF moves up the logical tree to the `StackPanel` and searches its resources. If it still doesn’t find a `LabelValue` resource, WPF continues up the tree to the `Window` and checks its resources.

If it still hasn’t found the `LabelValue` resource in the `Window` sitting at the top of the logical tree, WPF checks the application’s XAML file App.xaml to see if it defines the resource.

If after all this WPF can’t find the resource, then the application fails in one of several ways. For example, the Expression Blend Window Designer doesn’t render the control that needs the resource, and if you try to run the application, it fails. In contrast, Visual Studio’s Window Designer displays an error message and doesn’t draw any of the controls.

MATCH TYPES

The resource that WPF finds for a reference must have the correct data type. For example, suppose you set a `Button`’s `Width` property to the value in the `btnWidth` resource. Now suppose the first resource WPF finds named `btnWidth` is a `String` with the value “`Big`.” WPF tries to assign the `Button`’s `Width` property the `String` value `Big` and has a tantrum.

In fact, the data type of the resource must exactly match the data type expected by the property. In this example, if the `btnWidth` resource is a `String` containing the value 100, the program still fails because a `String` isn’t the `Double` data type that was expected by the `Width` property.

For an even sneakier problem, suppose that the `btnWidth` resource is of type `Int32` (32-bit integer). Any normal person (or even programming language) should be able to automatically convert an `Int32` value into a `Double`, but WPF won’t even try. If a XAML file tries to put an `Int32` value in a `Double` property, the program won’t run. (Note that most numeric properties take `Double` values.)

Of course, some other conversions *do* work. For example, if you set a `Label`’s `Content` property to a `Double` or `Boolean` resource, WPF will display the resource’s value.

The ResourceHierarchy example program demonstrates the search for resources. The following code shows the program's App.xaml file. This file defines `String` resources named `String1`, `String2`, `String3`, and `String4`.

Available for download on Wrox.com

```
<Application
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:sys="clr-namespace:System;assembly=mscorlib"
  x:Class="App"
  StartupUri="Window1.xaml">
  <Application.Resources>
    <!--  Resources scoped at the Application level
          should be defined here.  -->
    <sys:String x:Key="String1">String1 in App.Resources.</sys:String>
    <sys:String x:Key="String2">String2 in App.Resources.</sys:String>
    <sys:String x:Key="String3">String3 in App.Resources.</sys:String>
    <sys:String x:Key="String4">String4 in App.Resources.</sys:String>
  </Application.Resources>
</Application>
```

ResourceHierarchy

The following code shows the program's main XAML file:

Available for download on Wrox.com

```
<Window
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:sys="clr-namespace:System;assembly=mscorlib"
  x:Class="Window1"
  x:Name="Window"
  Title="ResourceHierarchy"
  Width="350" Height="180"
  FontSize="16" FontWeight="Bold">
  <Window.Background>
    <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
      <GradientStop Color="#FFFFFFFF" Offset="0"/>
      <GradientStop Color="#FFA6009F" Offset="1"/>
    </LinearGradientBrush>
  </Window.Background>

  <Window.Resources>
    <sys:String x:Key="String2">String2 in Window.Resources.</sys:String>
    <sys:String x:Key="String3">String3 in Window.Resources.</sys:String>
    <sys:String x:Key="String4">String4 in Window.Resources.</sys:String>
  </Window.Resources>

  <StackPanel Margin="5">
    <StackPanel.Resources>
      <sys:String x:Key="String3">
        String3 in StackPanel.Resources.
      </sys:String>
      <sys:String x:Key="String4">
        String4 in StackPanel.Resources.
      </sys:String>
```

```
        </StackPanel.Resources>

        <Label Content="{StaticResource String1}"/>
        <Label Content="{StaticResource String2}"/>
        <Label Content="{StaticResource String3}"/>
        <Label>
          <Label.Resources>
            <sys:String x:Key="String4">
              String4 in Label.Resources.
            </sys:String>
          </Label.Resources>
          <StaticResource ResourceKey="String4"/>
        </Label>
      </StackPanel>
    </Window>
```

ResourceHierarchy

In this code, the `Window` defines the resources `String2`, `String3`, and `String4`; the `StackPanel` defines the resources `String3` and `String4`; and the final `Label` defines the resource `String4`.

The first `Label` uses the resource `String1`. WPF searches up the logical tree and doesn't find a resource named `String1` in the `StackPanel` or `Window` that contains it. It eventually finds the resource in the application resource dictionary in Appl.xaml.

Similarly, the second `Label` finds its resource `String2` in the `Window`'s resources, and the third `Label` finds its resource `String3` in the `StackPanel`'s resources.

The fourth `Label` is a bit different because it defines its own `String4` resource. Its content is set to a `StaticResource` object that gives the resource's key `String4`.

ABSENT ATTRIBUTES

It's tempting to use an attribute as in the following code instead of the `StaticResource` object to define the fourth `Label`'s content:

```
<Label Content="{StaticResource String4}">
  <Label.Resources>
    <sys:String x:Key="String4">
      String4 in Label.Resources.
    </sys:String>
  </Label.Resources>
</Label>
```

Unfortunately, the `Label`'s resource dictionary isn't quite ready by the time WPF reads the `Label`'s `Content` attribute. Expression Blend's Window Designer seems to find the value correctly, but at run time and in the Visual Studio Window Designer, the `Label` doesn't find its own resource and moves up the logical tree to the `StackPanel`'s version of `String4`.

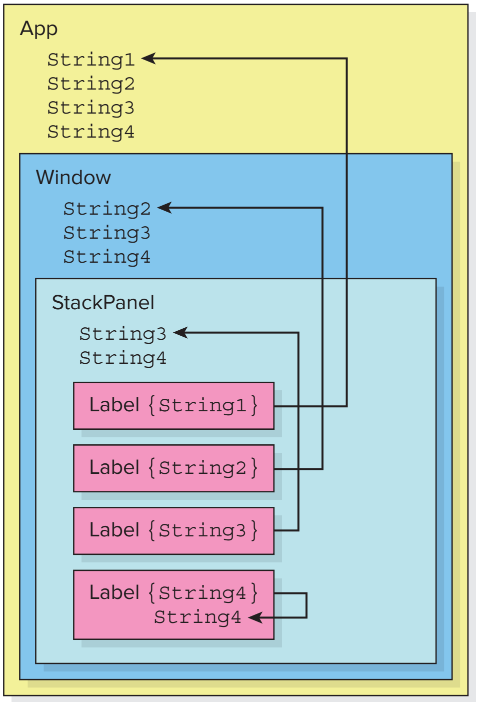


FIGURE 12-6

Figure 12-6 shows the ResourceHierarchy program's search for resources graphically.

Figure 12-7 shows the ResourceHierarchy program in action.

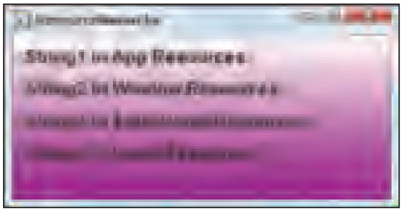


FIGURE 12-7

MERGED RESOURCE DICTIONARIES

A resource dictionary lets several controls share the same values. *Merged resource dictionaries* let several windows or even applications share the same resource values.

To merge a resource dictionary, create a XAML file that has a `ResourceDictionary` object as its root element. Give the element namespace attributes as you would a `Window`, and place resources inside the dictionary.

READY RESOURCES

To save typing, you can copy and paste a `Window`'s namespace declarations into a resource dictionary.

Visual Studio and Expression Blend can also add resource dictionaries with default namespace declarations. In Visual Studio, open the Project menu, select the "Add New Item" command, select the "Resource Dictionary" command, and click Add. In Expression Blend, open the File menu, select the "New Item" command, select the "Resource Dictionary" item, and click OK.

The following code shows part of a resource dictionary file named *RectangleResources.xaml*. This file defines resources that the MultiWindowResources example program uses to define `Rectangle` properties. (To save space, I've omitted some of the resources.)

Available for download on Wrox.com

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
    xmlns:sys="clr-namespace:System;assembly=mscorlib">
    <!--  Resource dictionary entries should be defined here.  -->
    <sys:Double x:Key="rectWidth">140</sys:Double>
    <sys:Double x:Key="rectHeight">50</sys:Double>
    <sys:Double x:Key="rectRadX">5</sys:Double>
    <sys:Double x:Key="rectRadY">20</sys:Double>
    ... More resources omitted ...
</ResourceDictionary>
```

MultiWindowResources

The following code shows how the MultiWindowResources program uses this resource dictionary. The Window1 XAML file uses a `ResourceDictionary.MergedDictionaries` element that contains a reference to the dictionary file.

Available for download on Wrox.com

```
<Window.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="RectangleResources.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Window.Resources>
```

MultiWindowResources

The following code shows how the `Window` uses these resources. This code creates a `Rectangle` that refers to the resources defined in the external dictionary.

Available for download on Wrox.com

```
<Grid MouseDown="rectAddUser_MouseDown">
  <Rectangle
    Margin="{StaticResource rectMargin}"
    Width="{StaticResource rectWidth}"
    Height="{StaticResource rectHeight}"
    RadiusX="{StaticResource rectRadX}"
    RadiusY="{StaticResource rectRadY}"
    Fill="{StaticResource rectFill}"
    Stroke="{StaticResource rectStroke}"
    StrokeThickness="{StaticResource rectStrokeThickness}"
    BitmapEffect="{StaticResource rectBitmapEffect}"
  />
  <Label HorizontalAlignment="Center" VerticalAlignment="Center"
    Content="Add User"
    FontSize="{StaticResource rectFontSize}"
    FontWeight="{StaticResource rectFontWeight}"
  />
</Grid>
```

MultiWindowResources

The MultiWindowResources example program uses similar code to include the resource dictionary and refer to its resources in all four of its `Windows`. Because every form refers to the same resources, they all have a common appearance. If you decide to change the appearance, you can simply modify the resource dictionary, and all of the forms will pick up the changes automatically.

Figure 12-8 shows the MultiWindowResources program displaying several forms that have similar button-like rectangles.

FIGURE 12-8

In addition to allowing multiple forms or applications to use the same resources, merged dictionaries allow a single window to load more than one resource dictionary. This can be handy if you want to use separate dictionaries to hold different groups of resources. For example, you might have separate dictionaries to hold button resources, label resources, and list resources.

RESOURCES OVERWRITTEN

When a program loads a resource dictionary, its values overwrite any resources with the same names that were loaded from other resource dictionaries, so whatever values are loaded last are the ones used by the controls.

You can also use multiple resource dictionaries in the same application to switch easily between different appearances. For example, the ResourceDictionaries example program uses the following code to merge two resource dictionaries, ResBlue.xaml and ResRed.xaml:

Available for download on Wrox.com

```
<Window.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="ResBlue.xaml"/>
      <ResourceDictionary Source="ResRed.xaml"/>
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Window.Resources>
```

ResourceDictionaries

Because the file ResRed.xaml is loaded second, the resource values in that file take precedence. Figure 12-9 shows the ResourceDictionaries program using the values in this resource dictionary.

FIGURE 12-9

By simply switching the order of these two resource dictionaries, you can completely change the application's appearance. Figure 12-10 shows the ResourceDictionaries program using the values in the ResBlue.xaml resource dictionary.

FIGURE 12-10

The section "Skins" in Chapter 16 explains how to switch an application's appearance in a similar manner at run time.

DYNAMIC RESOURCES

Up to this point, this chapter has used only static resources. If WPF encounters a static resource as it reads a XAML file, it looks up the resource's value and assigns it to whatever property it is currently reading. Once the property's value is set, WPF doesn't look at it again.

This is why the following code described in the section "Resource Hierarchies" earlier in this chapter doesn't work. Because the attribute `Content="{StaticResource String4}"` comes before the `Label`'s resources, the resource isn't ready when the resource is needed.

```
<Label Content="{StaticResource String4}">
  <Label.Resources>
    <sys:String x:Key="String4">
      String4 in Label.Resources.
    </sys:String>
  </Label.Resources>
</Label>
```

A *dynamic resource* is similar to a static resource except it is not simply read once when it is first encountered. Instead of setting the property's value right away, WPF waits and looks up the property's value when it is actually needed.

For example, the following code (used by the DynamicLabelResource example program) works correctly. The code records the fact that the `Content` attribute uses the resource value `String4`, but it doesn't look up that value until it is time to render the `Label`. At that time the resource exists.

Available for download on Wrox.com

```
<Label Content="{DynamicResource String4}">
  <Label.Resources>
    <sys:String x:Key="String4">
      String4 in Label.Resources.
```

```
    </sys:String>
  </Label.Resources>
</Label>
```

DynamicLabelResource

> ### RESOURCE-INTENSIVE RESOURCES
>
> Dynamic resources are a lot more work for WPF to implement than static resources; in general, therefore, it's better to use static resources when you can. In the previous example, you could probably store the `String4` resource in the `Label`'s container instead of in the `Label`. Then you can use a static resource instead of a dynamic resource.

In addition to deferring lookup, WPF prepares dynamic resources so that it can detect changes to their values. For example, if the resource value `String4` in the previous example changed while the program was running, WPF would detect the change and update the `Label` to display the new value.

This ability to detect changes is most often used with system-defined resources such as system colors. If the user opens the system's configuration applications and changes the system colors, the program automatically reloads any modified resources.

The SimpleClock example program demonstrates two kinds of changing resources — resources changed by the program and changing system colors.

The program uses the following resource declaration to create a simple `string` resource named `TimeNow`:

```
<Window.Resources>
  <sys:String x:Key="TimeNow">1:00:00 AM</sys:String>
</Window.Resources>
```

It then uses the following code to define three `Labels`:

Available for download on Wrox.com

```
<Label Margin="5" HorizontalAlignment="Stretch"
 Content="{StaticResource TimeNow}"
 Background="Green"
 Foreground="YellowGreen"/>

<Label Margin="5" HorizontalAlignment="Stretch"
 Content="{DynamicResource TimeNow}"
 Background="{DynamicResource lblBackground}"
 Foreground="{DynamicResource lblForeground}"
>
  <Label.Resources>
    <SolidColorBrush x:Key="lblBackground" Color="Green"/>
    <SolidColorBrush x:Key="lblForeground" Color="YellowGreen"/>
  </Label.Resources>
</Label>

<Label Margin="5" HorizontalAlignment="Stretch"
 Content="{DynamicResource TimeNow}"
```

```
 Foreground="{DynamicResource {x:Static SystemColors.ActiveCaptionBrushKey}}"
>
  <Label.Background>
    <SolidColorBrush
     Color="{DynamicResource {x:Static SystemColors.DesktopColorKey}}"/>
  </Label.Background>
</Label>
```

SimpleClock

The first `Label` displays the value of the `TimeNow` resource as a static resource using hard-coded brushes.

The second `Label` displays the `TimeNow` resource dynamically. It also uses the `DynamicResource` keyword to set its `Foreground` and `Background` properties from its own resources.

The third `Label` also displays the `TimeNow` resource dynamically. In addition, it uses the `DynamicResource` keyword to get its `Background` and `Foreground` `Brushes` from the system resources.

The `Label` gets its `Foreground` `Brush` from the system's active caption color. The static resource `ActiveCaptionBrushKey` gives the name of the `Brush` in the system resource dictionary. The code uses that value in a `DynamicResource` statement to look up the corresponding value.

The `Label` builds its `Background` `Brush` so it matches the system's desktop color. Because the system resources do not define a desktop color brush, the code builds a `SolidColorBrush` that uses the desktop color.

The SimpleClock program uses the following code-behind to update the `TimeNow` resource every second:

Available for download on Wrox.com

```
public Window1()
{
  this.InitializeComponent();

  // Insert code required on object creation below this point.
  tmrClock = new System.Timers.Timer(1000.0);
  tmrClock.Elapsed += new System.Timers.ElapsedEventHandler(tmrClock_Elapsed);
  tmrClock.Enabled = true;
  SetTimeResource();
}

private System.Timers.Timer tmrClock;

// Update the TimeNow resource every second.
private void tmrClock_Elapsed(Object sender, System.Timers.ElapsedEventArgs e)
{
  this.Dispatcher.Invoke(new Action(SetTimeResource));
}

// Update the TimeNow resource.
private void SetTimeResource()
{
  this.Resources.Remove("TimeNow");
  this.Resources.Add("TimeNow", DateTime.Now.ToString("T"));
}
```

SimpleClock

The `Window`'s constructor creates a new `Timer` that fires every 1,000 milliseconds (i.e., every second). It registers the event handler `tmrClock_Elapsed` to handle the `Timer`'s `Elapsed` events and enables the `Timer`. It finishes by calling `SetTimeResource` to display the initial time.

When the `Timer` raises its `Elapsed` event, the event handler springs into action. Unfortunately, the event handler's code runs in a different thread of execution from the code that manages the user interface (UI) and its controls. Windows doesn't allow code to directly access UI objects from a separate thread, so the program must use the `Invoke` method to make the `Dispatcher` object running the UI thread do the work. In this example, the code uses `Invoke` to execute the `SetTimeResource` function on the UI thread.

INDIRECT INVOCATION

Contrast this call to `Invoke` with the constructor's initial direct call to `SetTimeResource`. The constructor is running in the same thread as the UI objects, so it can call `SetTimeResource` directly. Later when the `Timer` fires its event, it is running on a separate thread, so it must use `Invoke`.

The function `SetTimeResource` removes the `TimeNow` resource from the `Window`'s resource dictionary and then re-adds it with a new value showing the current time.

Figure 12-11 shows the SimpleClock program on top of the Windows 7 Desktop Background applet. (Right-click on the desktop, select Personalize, and click on the "Desktop Background" link.) You may want to refer back to the XAML code as you review this figure.

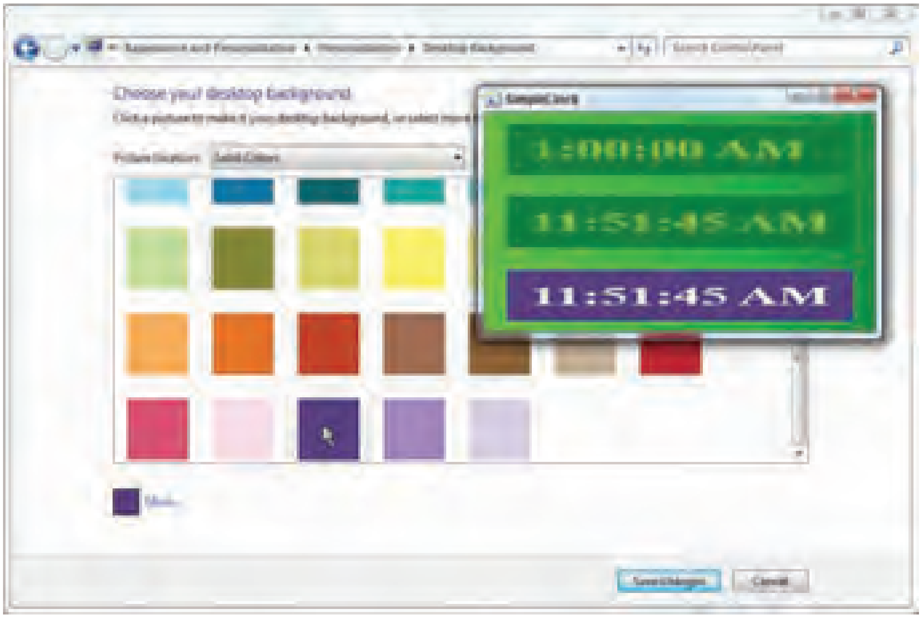


FIGURE 12-11

The first Label displays TimeNow as a StaticResource, so it is not updated every second and displays the resource's initial value, *1:00:00 AM*.

The second Label displays TimeNow as a DynamicResource, so it shows the current time. It also uses DynamicResource statements to load its colors from its own resources.

Like the second Label, the third Label displays TimeNow as a DynamicResource, and thus it shows the current time. It uses DynamicResource statements to load its colors from the system resources. In Figure 12-11, I changed the desktop background color to a dark purple, and the Label's background updated to match.

In summary, since dynamic resources place a bigger load on the system, you should use StaticResource whenever possible. If you expect a resource value to change while the programming is running, however, use DynamicResource so your program can see the change.

SUMMARY

XAML resources provide several benefits including: giving many controls a consistent appearance, providing a central location for resource definitions, and making it easy to change the appearance of related controls so they remain consistent.

By using multiple resource dictionaries, you can change the entire application's appearance quickly and easily.

One big drawback to resources is that they are fairly verbose and relatively hard to read. For example, the following code defines two Rectangles. The first uses hard-coded property values and is relatively easy to read. The second uses more flexible resource values but is longer and harder to understand.

```
<Rectangle Grid.Row="0" Grid.Column="0" Margin="5"
  RadiusX="10" RadiusY="10"
  Fill="Yellow" Stroke="Orange" StrokeThickness="1"
/>

<Rectangle Grid.Row="0" Grid.Column="0"
  Margin="{DynamicResource thkMargin}"
  RadiusX="{DynamicResource dblRadX}"
  RadiusY="{DynamicResource dblRadY}"
  Fill="{DynamicResource brRectFill}"
  Stroke="{DynamicResource brRectStroke}"
  StrokeThickness="{DynamicResource dblRectStrokeThickness}"
/>
```

If an application has many similar controls that share the same property values, resources can make the code much harder to read. For example, if a window contains a dozen or so Rectangles with the same Margin, RadiusX, RadiusY, Fill, Stroke, and StrokeThickness values, then using this kind of resource code would take up a huge amount of space.

The next chapter explains one way to practically eliminate this problem: *styles*. By using styles, you can package sets of properties for use by controls that should have a similar overall appearance. The chapter also explains property triggers, objects that styles can use to change a control's properties when certain conditions arise.

13

Styles and Property Triggers

Chapter 12 explains *resources* — named property values that you can assign to controls to give them a common appearance and behavior.

Unfortunately, resources often lead to long and complicated code. If a group of controls shares many property values, then converting the properties into resources requires a lot of repetition using the verbose `StaticResource` or `DynamicResource` keywords.

A *Style* is a special kind of resource that lets you extract this redundancy and centralize it, much as other resources let you centralize property values. `Styles` let you define packages of property values that you can apply to a control all at once instead of applying individual resource values to the control one at a time.

This chapter explains `Styles` and shows how to use them to give controls similar appearance and behavior.

SIMPLIFYING PROPERTIES

Resources make it easier to give a group of controls a consistent appearance. The ButtonValues example program shown in Figure 13-1 displays five button-like rectangles containing text. It sets each button's properties individually, so making changes to the code is repetitive and a bit tricky. If you don't change every button's code in the same way, they will no longer have a common appearance.

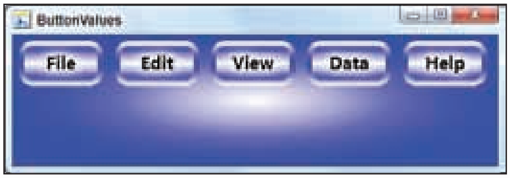


FIGURE 13-1

The ButtonResources example program uses resources to display a similar set of buttons. The resources make it easy to change the appearance of all of the buttons at once, but the resulting code is long and confusing.

The following code shows how the program defines its button resources:

Available for download on Wrox.com

```
<Window.Resources>
  <LinearGradientBrush x:Key="btnFill" StartPoint="0,0" EndPoint="0,1">
    <GradientStop Color="Blue" Offset="0"/>
    <GradientStop Color="White" Offset="0.5"/>
    <GradientStop Color="Blue" Offset="1"/>
  </LinearGradientBrush>
  <LinearGradientBrush x:Key="btnStroke" StartPoint="0,0" EndPoint="0,1">
    <GradientStop Color="White" Offset="0"/>
    <GradientStop Color="Blue" Offset="0.5"/>
    <GradientStop Color="White" Offset="1"/>
  </LinearGradientBrush>
  <sys:Double x:Key="btnStrokeThickness">5</sys:Double>
  <sys:Double x:Key="btnWidth">75</sys:Double>
  <sys:Double x:Key="btnHeight">40</sys:Double>
  <sys:Double x:Key="btnRadiusX">10</sys:Double>
  <sys:Double x:Key="btnRadiusY">10</sys:Double>
  <Thickness x:Key="btnMargin">5</Thickness>
  <FontFamily x:Key="btnFontFamily">Segoe</FontFamily>
  <FontWeight x:Key="btnFontWeight">Bold</FontWeight>
  <sys:Double x:Key="btnFontSize">16</sys:Double>
  <BitmapEffectGroup x:Key="btnTextBitmapEffect">
    <OuterGlowBitmapEffect GlowColor="White" GlowSize="10"/>
  </BitmapEffectGroup>
  <sys:String x:Key="btnText0">File</sys:String>
  <sys:String x:Key="btnText1">Edit</sys:String>
  <sys:String x:Key="btnText2">View</sys:String>
  <sys:String x:Key="btnText3">Data</sys:String>
  <sys:String x:Key="btnText4">Help</sys:String>
</Window.Resources>
```

ButtonResources

The following code shows how the ButtonResources program defines its File button. The other buttons are similar:

Available for download on Wrox.com

```
<!--  File  -->
<Grid
   Width="{StaticResource btnWidth}"
   Height="{StaticResource btnHeight}"
   Margin="{StaticResource btnMargin}"
>
  <Rectangle
   Fill="{StaticResource btnFill}"
   Stroke="{StaticResource btnStroke}"
   StrokeThickness="{StaticResource btnStrokeThickness}"
   RadiusX="{StaticResource btnRadiusX}"
   RadiusY="{StaticResource btnRadiusY}"
  />
  <TextBlock HorizontalAlignment="Center" VerticalAlignment="Center"
   FontFamily="{StaticResource btnFontFamily}"
```

```
      FontSize="{StaticResource btnFontSize}"
      FontWeight="{StaticResource btnFontWeight}"
      BitmapEffect="{StaticResource btnTextBitmapEffect}"
      Text="{StaticResource btnText0}"
    />
  </Grid>
```

ButtonResources

Although resources make it easier to modify all of the buttons' appearance, they make the code long and confusing. Because all of the buttons have the same basic appearance, the code contains a lot of redundancy, with each `Button`'s code repeating the same `Width`, `Height`, `Margin`, `Fill`, and other properties.

A style packages property values that should be set as a group. It begins with a `Style` element contained in a resource dictionary. You can place the `Style` in any resource dictionary depending on how widely you want it to be available. For example, if you want a `Style` to be visible to the entire project, place it in the `Application.Resources` section in the App.xaml file; if you want the `Style` visible to every control on a window, place it in the `Window.Resources` section; and if you want it to be visible only to controls inside a `StackPanel`, place it inside the `StackPanel.Resources` section.

STYLE SCOPE

Suppose you have a `Grid` that contains a group of `Labels`. You could use a style defined in `Window.Resources` to give them a similar appearance but that might cause problems if you want to give other `Labels` in a different `Grid` a different appearance. In that case, you would have to define multiple named styles (named and unnamed styles are described shortly), which will complicate the code.

It's usually better to give `Styles` the most limited scope possible while still getting the job done. That keeps the styles near where they are used and lets you use unnamed styles to simplify your code.

A style can have an `x:Key` attribute to give it a name and a `TargetType` attribute to indicate the kind of control to which it should apply. (The next section says more about these attributes.)

Inside the `Style` element, `Setter` and `EventSetter` elements define the style's property values and event handlers, respectively.

A `Setter` sets a property's value. It takes two attributes — `Property` and `Value` — that give the property to be set and the value it should take. The value can be a simple attribute value like Red or 7, or it can be an element attribute specifying something more complex like a brush. (`EventSetters` are described later in this chapter.)

The RedRectangles example program uses the following code to define a `Style` named `RedRectStyle` that applies to `Rectangles`. The `Stroke`, `StrokeThickness`, `Width`, `Height`, and `Margin` setters use property attribute syntax to provide simple values, while the `Fill setter` uses property element syntax to set its value to a `RadialGradientBrush` object.

Available for download on Wrox.com

```
<Style x:Key="RedRectStyle" TargetType="Rectangle">
  <Setter Property="Stroke" Value="Red"/>
  <Setter Property="StrokeThickness" Value="5"/>
  <Setter Property="Width" Value="100"/>
  <Setter Property="Height" Value="50"/>
  <Setter Property="Margin" Value="5"/>
  <Setter Property="Fill">
    <Setter.Value>
      <RadialGradientBrush>
        <GradientStop Color="Red" Offset="0"/>
        <GradientStop Color="White" Offset="1"/>
      </RadialGradientBrush>
    </Setter.Value>
  </Setter>
</Style>
```

RedRectangles

You can use a style just as you use any other resource. Simply set the "Style" property to the style's key.

The RedRectangles example program uses the following code to display three `recRectangles` that use the `RedRectStyle Style`:

Available for download on Wrox.com

```
<StackPanel Orientation="Horizontal">
  <Rectangle Style="{StaticResource RedRectStyle}"/>
  <Rectangle Style="{StaticResource RedRectStyle}"/>
  <Rectangle Style="{StaticResource RedRectStyle}"/>
</StackPanel>
```

RedRectangles

Notice how much the style simplifies this code. Without the the style, each `Rectangle` would need to have its own `Stroke`, `StrokeThickness`, `Width`, `Height`, `Margin`, and `Fill` properties.

ALL TYPES OF TYPES

The `x:Type` markup extension allows you to pass a type into an attribute such as `TargetType`. For example, the following two `Style` tags do the same thing:

```
<Style x:Key="RedRectStyle" TargetType="Rectangle">

<Style x:Key="RedRectStyle" TargetType="{x:Type Rectangle}">
```

Sometimes you may need to use the `x:Type` syntax to pass a type name into an attribute that cannot convert the type's name from a simple string into a type object, but so far I've been able to use the simpler syntax for styles.

For some reason, possibly because early examples were written that way, the more verbose `x:Type` format is common in the documentation and on the Web, so you'll often see that format.

Figure 13-2 shows the RedRectangles program in action.

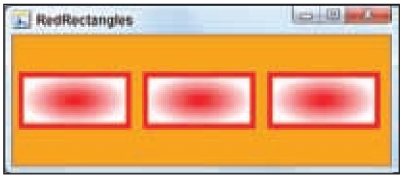


FIGURE 13-2

The ButtonStyles example program displays button-like rectangles similar to those displayed by the ButtonValues and ButtonResources programs (see Figure 13-1), but it uses styles to simplify its code. The program uses the following XAML code to define three styles for its `Grid`, `Rectangle`, and `TextBlock` controls:

Available for download on Wrox.com

```
<Window.Resources>
  … Property resource definitions omitted …

  <Style x:Key="btnGridStyle" TargetType="Grid">
    <Setter Property="Width"
     Value="{StaticResource btnWidth}"/>
    <Setter Property="Height"
     Value="{StaticResource btnHeight}"/>
    <Setter Property="Margin"
     Value="{StaticResource btnMargin}"/>
  </Style>

  <Style x:Key="btnRectStyle" TargetType="Rectangle">
    <Setter Property="Fill"
     Value="{StaticResource btnFill}"/>
    <Setter Property="Stroke"
     Value="{StaticResource btnStroke}"/>
    <Setter Property="StrokeThickness"
     Value="{StaticResource btnStrokeThickness}"/>
    <Setter Property="RadiusX"
     Value="{StaticResource btnRadiusX}"/>
    <Setter Property="RadiusY"
     Value="{StaticResource btnRadiusY}"/>
  </Style>

  <Style x:Key="btnTextBlockStyle" TargetType="TextBlock">
    <Setter Property="FontFamily"
     Value="{StaticResource btnFontFamily}"/>
    <Setter Property="FontSize"
     Value="{StaticResource btnFontSize}"/>
    <Setter Property="FontWeight"
     Value="{StaticResource btnFontWeight}"/>
    <Setter Property="HorizontalAlignment"
     Value="Center"/>
    <Setter Property="VerticalAlignment"
     Value="Center"/>
    <Setter Property="BitmapEffect"
     Value="{StaticResource btnTextBitmapEffect}"/>
  </Style>
</Window.Resources>
```

ButtonStyles

The style definitions are contained in the `Window.Resources` element with the other resource definitions. This program includes resource definitions similar to those used by the ButtonResources program (`Width`, `Height`, `RadiusX`, `FontFamily`, etc.), but they are not shown here to save space.

The following code shows how the ButtonStyles program uses these styles to display its File button:

Available for download on Wrox.com

```
<!--  File  -->
<Grid Style="{StaticResource btnGridStyle}">
  <Rectangle Style="{StaticResource btnRectStyle}"/>
  <TextBlock Style="{StaticResource btnTextBlockStyle}"
   Text="{StaticResource btnText0}"
  />
</Grid>
```

ButtonStyles

This version is much shorter than the previous one, using only 6 lines of code compared to 21 lines used by the ButtonResources program (not counting comments).

This version is also much easier to understand. For example, the `Grid`'s code uses the style `btnGridStyle`, which defines whatever properties are appropriate for the `Grid`. When you are reading this piece of code, you don't really need to know what those values are, just that they make sense for the `Grid`. That lets you focus on the arrangement of the `Grid`, `Rectangle`, and `TextBlock` without being distracted by the property values.

The ButtonStyles program displays five buttons, so the savings in lines of code and complexity adds up. The button-definition code takes only 30 lines compared to the 105 lines used by the ButtonResources program.

A PROGRAMMING DIGRESSION

When you write a Windows Forms program with C# or Visual Basic, Visual Studio allows you to set control properties at design time, so your code only needs to deal with properties that must change at run time.

For example, you can set a `Label`'s font, colors, size, and position properties in the Form Designer at design time. Then at run time, the program's code might only need to change the `Label`'s text. Visual Studio lets you ignore the other properties by hiding them in a separate designer-generated code module that you need to look at rarely if ever.

In XAML code, however, you cannot move some of the unchanging property definitions into a separate file in quite the same way. If you need to set a `Label`'s font, color, and position, then those properties are set in the same file as the control's declaration. That can make the code horribly cluttered and hard to read.

Styles let you move the unchanging properties out of the control's declaration, making it much easier to read the control's code and understand the program's structure. You can define the styles in a resource dictionary provided by a container, window, or at the application level depending on the scope you want the style to have.

Simplifying the code is an important function of styles, so don't be afraid to use them to make the code simpler. Sometimes it's worth creating a style that you will only use for a single control if it makes that control's declaration easier to read.

KEYS AND TARGET TYPES

The `Style` element's `x:Key` and `TargetType` attributes help identify when and where you can use a `Style`. There are at least three particularly interesting and useful ways in which you can use these attributes: non-specific target types, multiple target types, and unnamed styles.

Non-Specific Target Types

The `TargetType` attribute determines the kinds of control that can use the `Style`. While you need to provide this information somehow, you don't necessarily need to be overly specific.

For example, since the following `Style`'s `TargetType` is set to `Button`, only `Buttons` can use this `Style`:

Available for download on Wrox.com

```
<Style x:Key="OrangeButtonStyle" TargetType="Button">
  <Setter Property="Width" Value="150" />
  <Setter Property="Height" Value="40" />
  <Setter Property="Background" Value="Orange" />
  <Setter Property="Foreground" Value="Yellow" />
  <Setter Property="Margin" Value="5"/>
</Style>
```

ControlStyle

But suppose you want to give `Labels` a similar style. Since this style only works with `Buttons`, you can't simply use it for `Labels`.

You can make a second style with `TargetType` set to the `Label` class, but that would require duplicated code. If you later decided to change the program's theme from orange to purple, for example, you would need to update both styles separately. This would be even more difficult if the style were more complex, perhaps using gradient background and foreground brushes.

Another approach is to set the `Style`'s `TargetType` to a less specific class that includes both `Button` and `Label`. Both the `Button` and `Label` classes inherit from the `Control` class, so, if you set the `Style`'s `TargetType` to `Control`, the `Style` can apply to either `Buttons` or `Labels`.

The following code shows the new style:

```
<Style x:Key="OrangeControlStyle" TargetType="Control">
  <Setter Property="Width" Value="150" />
  <Setter Property="Height" Value="40" />
  <Setter Property="Background" Value="Orange" />
  <Setter Property="Foreground" Value="Yellow" />
  <Setter Property="Margin" Value="5"/>
</Style>
```

So far so good — but what if you want to specify property values for a control type that doesn't apply to every subclass of the `Control` class? For example, the `Label` control has `BorderBrush` and `BorderThickness` properties that determine the appearance of its border. A `Button` doesn't have those properties, so how do you add them to this kind of style?

Fortunately, a control ignores any properties set by the style that it doesn't understand. That means that you can simply add the `BorderBrush` and `BorderThickness` properties to the style, and any `Button` controls that use it simply ignore them.

The following code shows the new style. This version includes the previous `Setters` plus new ones for the `Label`'s `BorderBrush` and `BorderThickness` properties.

Available for download on Wrox.com

```
<Style x:Key="OrangeControlStyle" TargetType="Control">
  <Setter Property="Width" Value="150" />
  <Setter Property="Height" Value="40" />
  <Setter Property="Background" Value="Orange" />
  <Setter Property="Foreground" Value="Yellow" />
  <Setter Property="Margin" Value="5"/>
  <Setter Property="BorderBrush" Value="Yellow" />
  <Setter Property="BorderThickness" Value="2" />
</Style>
```

ControlStyle

The ControlStyle example program shown in Figure 13-3 uses the following code to display two `Buttons` and a `Label`. The upper `Button` uses `OrangeButtonStyle`; the lower `Button` and the `Label` both use `OrangeControlStyle`.

Available for download on Wrox.com

```
<StackPanel VerticalAlignment="Center">
  <Button Style="{StaticResource OrangeButtonStyle}" Content="Button"/>
  <Button Style="{StaticResource OrangeControlStyle}" Content="Button"/>
  <Label Style="{StaticResource OrangeControlStyle}" Content="Label"/>
</StackPanel>
```

ControlStyle

Multiple Target Types

The previous section explains how to build a style that can affect two control types (`Button` and `Label`) by setting the `Style`'s `TargetType` to a class that is an ancestor of both control types. (`Button` and `Label` both inherit from `Control`.)

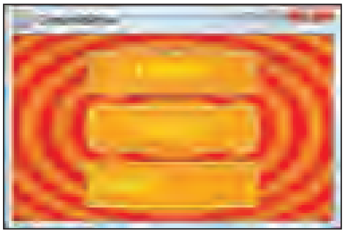


FIGURE 13-3

But what if you also want the style to affect rectangles? `Button` and `Label` have the `Control` class as a common ancestor, but `Rectangle` inherits from the `Shape` class instead of `Control`. If you climb a bit higher in the inheritance hierarchy, however, you'll find that `Control` and `Shape` both inherit from `FrameworkElement`, so you might think you're home free. You could simply create a style with `TargetType FrameworkElement` and build `Setters` as usual.

But, unfortunately, `FrameworkElement` doesn't support all of these properties. For example, if you try to change the previous `Style`'s `TargetType` to `FrameworkElement`, Expression Blend flags errors on the `Background`, `Foreground`, `BorderBrush`, and `BorderThickness` properties.

The problem is that the first common ancestor of `Button`, `Label`, and `Rectangle` doesn't support all of the properties that you'd like to use. Fortunately, there is another way to specify the objects that provide the style's properties.

Instead of indicating the class with the `Style` element's `TargetType` attribute, indicate it in the `Setter`'s `Property` attribute.

The following style is similar to the previous one, but it uses this new technique to add support for the Rectangle control (and it changes some property values for variety):

Available for download on Wrox.com

```
<Style x:Key="VioletStyle">
  <Setter Property="FrameworkElement.Width" Value="150" />
  <Setter Property="FrameworkElement.Height" Value="40" />
  <Setter Property="Control.Background" Value="Violet" />
  <Setter Property="Control.Foreground" Value="Green" />
  <Setter Property="Control.Margin" Value="5"/>
  <Setter Property="Label.BorderBrush" Value="Green" />
  <Setter Property="Label.BorderThickness" Value="2" />
  <Setter Property="Rectangle.Fill" Value="Violet" />
  <Setter Property="Rectangle.Stroke" Value="Green" />
  <Setter Property="Rectangle.StrokeThickness" Value="2" />
</Style>
```

LabelAndRectStyle

The FrameworkElement class supports the Width and Height properties, so the corresponding Setters refer to that class. Since the Control class supports Background, Foreground, and Margin, their Setters refer to the Control class. Finally, the Setters for Label and Rectangle properties refer to those classes.

The LabelAndRectStyle example program shown in Figure 13-4 uses the following code to display a button, a label, and a rectangle that all use this common style:

FIGURE 13-4

Available for download on Wrox.com

```
<StackPanel Orientation="Horizontal" Background="LightBlue"
 HorizontalAlignment="Center" VerticalAlignment="Center">
  <Button Style="{StaticResource VioletStyle}" Content="Button"/>
  <Label Style="{StaticResource VioletStyle}" Content="Label"/>
  <Rectangle Style="{StaticResource VioletStyle}"/>
</StackPanel>
```

LabelAndRectStyle

Unnamed Styles

In all of the styles described in the previous sections, the Style element included an x:Key attribute to give the style a name. Then controls used the StaticResource keyword to refer to the name.

In contrast, if you omit the attribute and include a TargetType, then the resulting *unnamed style* applies to all controls of the TargetType within the style's scope. For example, the following Style applies to every Button within the style's scope, so every Button appears with a yellow background and a red foreground:

```
<Style TargetType="Button">
  <Setter Property="Background" Value="Yellow" />
  <Setter Property="Foreground" Value="Red" />
</Style>
```

CLEVER CLASSES

The classes used in the Setters' property attributes must support their properties, but the style doesn't actually require that the controls using them be from that class.

For example, the previous style sets the FrameworkElement.Width and FrameworkElement.Height properties. The FrameworkElement class has the Width and Height properties, so WPF can use its definitions to understand how to set their values. If you changed the Width property to Red, WPF would complain because FrameworkElement.Width cannot be a color's name.

While the class defines the properties, the style doesn't require that the controls using it have those classes. In this example, if you were to change FrameworkElement to Button, the style would still work. The Button class also has Width and Height properties, so it can define them, but the Label and Rectangle can still use those property values even though they are not descendants of Button.

In fact, if you look back at the style's definition and the result in Figure 13-4, you'll see that the Rectangle uses the same Margin value as the Button and Rectangle even though that property is specified with the Control class and Rectangle isn't a descendant of Control.

The moral of the story is that it doesn't matter too much which classes you use in the Setters as long as they can define the properties.

The UnnamedStyles example program shown in Figure 13-5 defines four unnamed styles. The following list describes the properties that those styles set for their classes:

- Image — Width = 100, Height = 100, Stretch = Uniform, BitmapEffect = drop shadow
- Label — FontFamily = Comic Sans MS, FontSize = 18, FontWeight = bold, HorizontalAlignment = center, BitmapEffect = light blue outer glow
- Button — FontFamily = Arial, FontSize = 12, Height = 20, Background = blue and white linear gradient
- StackPanel — Margin = 5

The following code shows how the program defines its Style for Images. You can download the example program from the book's web site to see how the other Styles are defined.

Available for download on Wrox.com

```
<Style TargetType="Image">
  <Setter Property="Width" Value="100" />
  <Setter Property="Height" Value="100" />
  <Setter Property="Stretch" Value="Uniform" />
  <Setter Property="BitmapEffect">
    <Setter.Value>
      <DropShadowBitmapEffect/>
    </Setter.Value>
  </Setter>
</Style>
```

UnnamedStyles

FIGURE 13-5

The styles are all defined in the XAML code's Window.Resources section so they apply to every control on the window.

Having defined its `Styles`, the program simply creates `Images`, `Buttons`, `Labels`, and `StackPanels` and lets the `Styles` take effect.

The following code snippet shows how the program creates its first two sets of controls. The others are defined similarly.

Available for download on Wrox.com

```
<StackPanel>
  <Image Source="Mammal.jpg"/>
  <Label Content="Mammal"/>
  <Button Content="More Info"/>
</StackPanel>
<StackPanel>
  <Image Source="Bird.jpg"/>
  <Label Content="Bird"/>
  <Button Content="More Info"/>
</StackPanel>
```

UnnamedStyles

STYLE WITH SCOPE

An unnamed style applies to all objects of its `TargetType` but only within its scope. If you want to modify every `Label` in the entire application, define the style in the `Application.Resources` section of App.xaml. If you want to modify the `Labels` in a `Window`, define the style in `Window.Resources`. If you want to modify the `Labels` in a `Grid`, `StackPanel`, or other container, define the style in an appropriate Resources section (`Grid.Resources`, `StackPanel.Resources`, etc.).

PROPERTY VALUE PRECEDENCE

Most of the examples in this book set property values in straightforward, unambiguous ways. For example, the UnnamedStyles example program sets property values using unnamed styles. It sets a few other simple properties (e.g., a `Button`'s or `Label`'s `Content` property), but it doesn't set the same property in both a `Style` and an `element` attribute.

In fact, there are many ways in which you could set values for the same properties. If you do set a property's value in more than one way, WPF uses *property precedence* rules to decide which value is actually used. For example, if you set a control's property in a style and also in the control's attributes, then the value in the attribute takes precedence.

The following list shows a partial ordering of how property values might be applied to a control. You can think of the values as being applied in order so that those that are applied last have a higher precedence and overwrite those that are applied earlier.

1. **Container Inheritance** — A control may inherit a property value from its container. For example, if a `Window` defines a `FontFamily`, then `Labels` inside the `Window` inherit that `FontFamily`. (Note that some containers block some property inheritance.)
2. **Default Style** — Default styles determine the "natural" appearance of controls. For example, `Buttons` have a typical default appearance unless you override it by setting other properties.
3. **Unnamed Styles** — Unnamed styles can have precedence over default styles. If multiple unnamed styles might apply to a control, then the one with the most limited scope has precedence. For example, suppose a `Window` contains a `Grid` and both define unnamed `Label` styles. Then a `Label` contained inside the `Grid` would use the `Grid`'s version of the `Style`.

> **STYLES DON'T COMPROMISE**
>
> Two styles do not *share* a control's properties even if they set values for different properties. One style or the other will set the control's properties, and the other will have no say in the matter.
>
> If two unnamed styles might apply to a control, the one with more limited scope has sole control.

4. **Named Styles** — Named styles have precedence over unnamed styles. As is the case with unnamed styles, if more than one named style might apply to a control, then the one with the most limited scope has precedence.
5. **Local Values** — A property value set explicitly either with property attribute or property element syntax takes precedence over the previous kinds of values.
6. **Active Animations** — If an animation is running, it may modify current property values. (Chapter 14 describes animation in detail.)

This list does not cover every possible way in which property values might be changed, but it covers the most intuitive and useful ways. The others are beyond the scope of this chapter. For more information on property value precedence, see Microsoft's "Dependency Property Value Precedence" web page at `msdn.microsoft.com/ms743230.aspx`.

NO STYLE

Suppose you have defined an unnamed style but don't want the style to apply to a particular control of that type. Then you can set the control's `Style` to the special value `{x:Null}` to indicate that it should have no style. For example, the following code creates a `Label` that has no `Style`:

```
<Label Style="{x:Null}" Content="Test" />
```

STYLE INHERITANCE

Resources let you apply the same property values to multiple controls. Styles let you group related resources into packages to make it even easier to give controls a common appearance.

Style inheritance lets you build one style based on another. That makes it easier still to give controls a common look and feel even if they don't share the same final styles.

The *child style* inherits all of the `Setters` and other values defined by the *parent style*. You can then override some of those values if you like.

To make one style inherit from another, add a `BasedOn` attribute to its starting element, and set the attribute's value to the parent `Style`. Note that the `Style` is an object defined in a resource dictionary, so you should set the value with a `StaticResource` statement.

For example, the following code snippet defines two `Brush` resources and four `Styles`:

Available for download on Wrox.com

```
<LinearGradientBrush x:Key="GreenBrush" StartPoint="0,0" EndPoint="1,0">
  <GradientStop Color="Green" Offset="0"/>
  <GradientStop Color="Lime" Offset="0.3"/>
  <GradientStop Color="Transparent" Offset="0.8"/>
</LinearGradientBrush>

<LinearGradientBrush x:Key="BlueBrush" StartPoint="0,0" EndPoint="1,0">
  <GradientStop Color="Blue" Offset="0"/>
  <GradientStop Color="White" Offset="1"/>
</LinearGradientBrush>

<Style x:Key="SizeStyle" TargetType="FrameworkElement">
  <Setter Property="Width" Value="150" />
  <Setter Property="Height" Value="30" />
  <Setter Property="HorizontalAlignment" Value="Left"/>
  <Setter Property="VerticalAlignment" Value="Top"/>
  <Setter Property="Margin" Value="5"/>
```

```
  </Style>

  <Style TargetType="Rectangle" BasedOn="{StaticResource SizeStyle}">
    <Setter Property="Fill" Value="{StaticResource BlueBrush}"/>
    <Setter Property="RadiusX" Value="10"/>
    <Setter Property="RadiusY" Value="10"/>
  </Style>

  <Style TargetType="Label" BasedOn="{StaticResource SizeStyle}">
    <Setter Property="FontWeight" Value="Bold"/>
  </Style>

  <Style TargetType="Button" BasedOn="{StaticResource SizeStyle}">
    <Setter Property="Background" Value="{StaticResource GreenBrush}"/>
    <Setter Property="Margin" Value="25,5,5,5"/>
    <Setter Property="FontWeight" Value="Bold"/>
    <Setter Property="Width" Value="100"/>
  </Style>
```

InheritedStyles

The code begins by defining a `LinearGradientBrush` named `GreenBrush` that fades from green to lime to transparent. It also creates a `Brush` named `BlueBrush` that fades from blue to white.

The code then creates a style named `SizeStyle` that sets size and alignment properties.

Next the code defines an unnamed style for `Rectangles`. It is based on `SizeStyle` so it inherits that `Style`'s size and alignment properties. The new `Style` adds values for the `Rectangle` properties `Fill` (set using the `BlueBrush` resource), `RadiusX`, and `RadiusY`.

The snippet then defines an unnamed style for `Labels`. This style inherits the `SizeStyle`'s size and alignment properties and adds a new `FontWeight` `Setter`.

Finally, the code creates an unnamed style for `Buttons` that sets the `Background`, `Margin`, and `FontWeight` properties. In this example, the unnamed `Button` `Style` overrides the `Margin` and `Width` values that it inherits from `SizeStyle`.

The InheritedStyles example program shown in Figure 13-6 uses these styles to display `Rectangles` and `Labels` across the top of a `Grid` and `Buttons` beneath. The `Rectangles` are too big to fit in their `Grid` cells, so their rounded edges are truncated on the right. The `Buttons` use the `GreenBrush` `Background`, so they fade from green to transparent, allowing the window's light green background color to show through on their right edges.

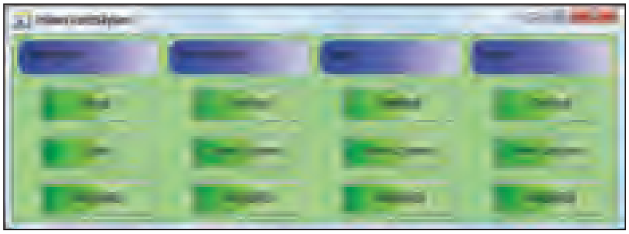


FIGURE 13-6

In this example, the unnamed `Rectangle` and `Label` `Styles` inherit their positioning properties from `SizeStyle`, so they line up properly.

Note that you can only base a style on another style with a target that is a superclass of the new style. For example, in the previous example, the unnamed `Rectangle`, `Label`, and `Button Styles` are based on a `Style` with `TargetType FrameworkElement`. This works because `Rectangle`, `Label`, and `Button` all inherit from `FrameworkElement`.

In contrast, you cannot base a `Button Style` on a `Label Style` because `Button` does not inherit from `Label`.

MIXED ANCESTRY

However, you can still use very general styles that target multiple control types. For example, the following code defines the `Style lblStyle`. This `Style` is really intended for `Labels`, but it doesn't have a `TargetType`, so WPF doesn't enforce that intention. That allows `btnStyle` to inherit from it even though `btnStyle`'s `TargetType` is `Button` and `Button` doesn't inherit from `Label`.

```
<Style x:Key="lblStyle">
  <Setter Property="Label.Background" Value="Yellow"/>
  <Setter Property="Label.FontWeight" Value="Bold"/>
</Style>

<Style TargetType="Button" BasedOn="{StaticResource lblStyle}">
  <Setter Property="Width" Value="100"/>
  <Setter Property="Height" Value="50"/>
</Style>
```

You can use style inheritance to provide consistency across windows or even applications. For example, you could define global styles in a resource dictionary file and then merge it with the application's resources at run time. These styles might define common background brushes, button sizes, and so forth that you want to remain constant for multiple applications.

Then, at the application level, you could define other styles in the application's resource dictionary in App.xaml. These might define the positions of OK and Cancel buttons, more colors or images, label formats, and other values that you want to be consistent across all of the application's windows.

Finally, styles within a window can define common properties for the controls on that window such as margins and positioning within grids or other containers.

TRIGGERS

The styles described so far always apply their property values to their controls. If a style has a setter that makes the `Background` property green, then every control that uses the style gets a green background.

Styles can also define *triggers*, objects that apply setters or start other actions only under certain conditions. For example, a `trigger` might change a `Label`'s color, font, or size when the mouse was over it to provide some visual feedback to the user.

WPF provides several kinds of triggers including property triggers, event triggers, and data triggers. This section describes property triggers. Event triggers are closely tied to animation, so they are described in Chapter 14. Data triggers aren't really covered here, although they are closely related to data binding which is explained in Chapter 18.

A *property trigger* performs its actions when a specific property has a certain value. For example, a property trigger could invoke one or more `Setters` if a `TextBox`'s `Text` property had the value *Test*. (They are called property triggers because it's a property value that triggers the action.)

When the property no longer has the triggering value, the `Trigger` deactivates, and the control's original property value returns. For example, when the user moves the mouse off the `Label`, the `Label` returns to its original color, font, and size.

This ability to respond to changing properties at run time gives the controls new behaviors that make an application feel more responsive to the user.

To make a property `Trigger`, create a `Style` and give it a `Style.Triggers` property element. Inside the `Style.Triggers` section, you can add `Trigger` elements.

Use each `Trigger`'s `Property` and `Value` attributes to indicate the property value that should activate the `Trigger`. Inside the `Trigger` element, add whatever `Setters` you want the `Trigger` to execute.

> **TRIGGER LOCATIONS**
>
> You must place property triggers inside `Styles` — not in a framework element's Triggers section. For example, you cannot place property triggers in a `Window.Triggers` or `Grid.Triggers` section. IntelliSense will pretend that this will work, but, in fact, those sections can only hold event triggers (which are described in the next chapter).
>
> Always put property triggers in a `Style`.

The following sections provide some useful property trigger examples.

Text Triggers

The following XAML code defines two unnamed styles, one for `TextBoxes` and one for `ComboBoxes`:

Available for download on Wrox.com

```
<Style TargetType="TextBox">
  <Setter Property="HorizontalAlignment" Value="Left" />
  <Setter Property="VerticalAlignment" Value="Center" />
  <Setter Property="Width" Value="220" />
  <Style.Triggers>
    <Trigger Property="Text" Value="">
```

```
        <Setter Property="Background" Value="Yellow" />
      </Trigger>
    </Style.Triggers>
  </Style>

  <Style TargetType="ComboBox">
    <Style.Triggers>
      <Trigger Property="Text" Value="">
        <Setter Property="Background" Value="Yellow" />
      </Trigger>
    </Style.Triggers>
  </Style>
```

TextTriggers

Each of these `Styles` defines a `Style.Triggers` section that contains a single `Trigger` that takes action when the control's `Text` property is blank. When that happens, the `Styles`' `Setters` make the controls' `Backgrounds` yellow to indicate that the value is required.

The TextTriggers example program shown in Figure 13-7 uses this code to flag required fields that have blank values. If a required field is blank, its background is yellow. (In Figure 13-7, Mr. Obama hasn't finished filling in his information, so the City, State, ZIP, Phone, and Email fields are yellow to flag them as missing.)

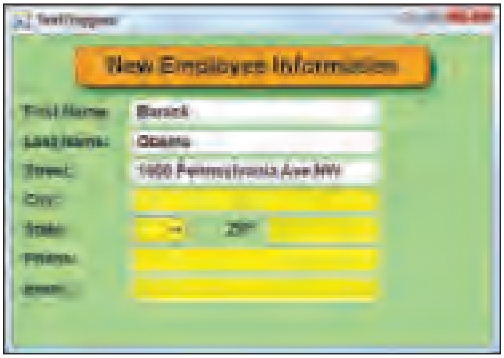


FIGURE 13-7

EXACT MATCHES ONLY

Unfortunately, triggers only detect exact matches for properties and cannot handle ranges, different capitalization, or other more complicated situations. For example, you cannot use a simple trigger to give `TextBoxes` different colors based on the values they contain (e.g., red for negative values, yellow for values between 0 and 10, and green for larger values). XAML just doesn't have the expressive power to represent these sorts of tests.

IsMouseOver Triggers

One common type of property trigger takes action when the user moves the mouse over something. These triggers execute their `Setters` when the `IsMouseOver` property is `True`.

The following XAML code defines an unnamed `Button` `Style`. It sets some basic values and then defines a `Trigger` that executes when the `IsMouseOver` property is `True`. When the mouse moves over the `Button`, the `Trigger` makes the `Button` bigger and makes its font larger and bold.

Available for download on Wrox.com

```
<Style TargetType="Button">
  <Setter Property="VerticalAlignment" Value="Top"/>
  <Setter Property="Margin" Value="10"/>
  <Setter Property="Background" Value="Violet"/>
  <Setter Property="Width" Value="100"/>
  <Setter Property="Height" Value="30"/>
  <Setter Property="FontSize" Value="16"/>
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="Width" Value="105"/>
      <Setter Property="Height" Value="50"/>
      <Setter Property="FontSize" Value="20"/>
      <Setter Property="FontWeight" Value="Bold"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

IsMouseOverTriggers

The IsMouseOverTriggers example program shown in Figure 13-8 uses this code to provide feedback when the mouse is over a button.

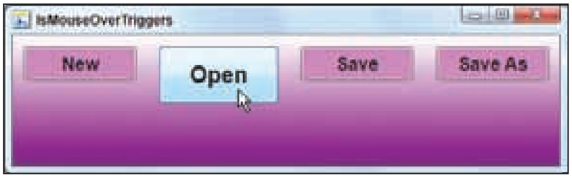


FIGURE 13-8

Setting Transform and BitmapEffect

The previous `Trigger` gives the activated `Button` a new `Width`, `Height`, and `FontSize` to make the button bigger. The MenuMouseOverTriggers example program achieves a similar effect in a different way. Rather than setting these properties individually, it uses a `Setter` that gives the active control a new `LayoutTransform` that scales the control.

The following code shows how the program enlarges a `MenuItem` when the mouse is over it:

Available for download on Wrox.com

```
<Window.Resources>
  <ScaleTransform x:Key="mnuBigScale" ScaleX="1.25" ScaleY="1.25"/>
  <OuterGlowBitmapEffect x:Key="mnuBitmapEffect"/>

  <Style x:Key="mnuStyle" TargetType="MenuItem">
    <Style.Triggers>
      <Trigger Property="IsMouseOver" Value="True">
        <Setter Property="FontWeight" Value="Bold"/>
        <Setter Property="Foreground" Value="Red"/>
        <Setter Property="LayoutTransform"
           Value="{StaticResource mnuBigScale}"/>
        <Setter Property="BitmapEffect"
           Value="{StaticResource mnuBitmapEffect}"/>
      </Trigger>
```

```
      </Style.Triggers>
    </Style>
  </Window.Resources>
```

MenuMouseOverTriggers

First, the program defines a `ScaleTransform` resource and an `OuterGlowBitmapEffect` resource. These are the values that the `Trigger` will use to set the `MenuItem` properties.

The program then defines the `MenuItem Style`. The `Trigger` sets the `MenuItem`'s `FontWeight` and `Foreground` simple properties. It also sets the control's `LayoutTransform` and `BitmapEffect` properties to the resources defined earlier.

Figure 13-9 shows the result.

You can use similar techniques to give controls triggers that modify their `LayoutTransform` and `BitmapEffect` properties.

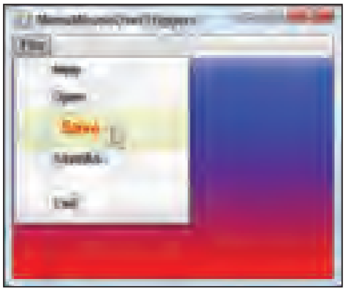


FIGURE 13-9

> **BUTTON BACKGROUNDS**
>
> Don't bother trying to change `Button` or `MenuItem` backgrounds when the mouse is over them. The `Button` and `MenuItem` controls have very definite opinions about what they should look like when the mouse is over them so the user won't see your color or may see it flash briefly before the control's natural behavior takes over and replaces it.
>
> You can change how these controls appear at different times, but that's better done with templates, not triggers. Chapter 15 explains templates in detail.

Setting Opacity

The ImageTriggers example program shown in Figure 13-10 uses `Triggers` to highlight the image under the mouse.

FIGURE 13-10

The program uses several `Styles` to make building the list of images easier. They're rather long, so they aren't shown here. You can download the example program from the book's web site and take a look if you like.

The bottom of the program's window contains a series of `StackPanel` controls, each of which holds an `Image` and a `Label`. The following code shows how the program displays the controls for the Engineering department on the right:

Available for download on Wrox.com

```
<StackPanel Style="{StaticResource DepartmentStackPanel}">
  <Image Source="Engineering.jpg" />
  <Label Style="{StaticResource ImageLabelStyle}" Content="Engineering"/>
</StackPanel>
```

ImageTriggers

The key is the `DepartmentStackPanel` `Style` defined in the following code:

Available for download on Wrox.com

```
<ScaleTransform x:Key="BigScale" ScaleX="1.1" ScaleY="1.1"/>

<Style x:Key="DepartmentStackPanel" TargetType="StackPanel">
  <Setter Property="Margin" Value="10"/>
  <Setter Property="Opacity" Value="0.5"/>
  <Setter Property="Background" Value="Transparent"/>
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="Opacity" Value="1"/>
      <Setter Property="LayoutTransform"
       Value="{StaticResource BigScale}"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

ImageTriggers

Like the previous example, this code first defines a `ScaleTransform` resource.

The code then defines the `DepartmentStackPanel` `Style`. The `Style` begins by setting the `StackPanel`'s `Margin`. It sets the `Opacity` property to 0.5, so the controls in the `StackPanel` are semitransparent, giving them a faded appearance. Next, the `Style` set the `StackPanel`'s `Background` property to Transparent.

TRANSPARENT VERSUS NULL

A Null background gives the same appearance as a transparent background. However, if a control has a Null background, its `IsMouseOver` does not change to `True` when the mouse is over the control. If the control's background is transparent, `IsMouseOver` *does* change when the mouse moves over the control.

If you want to make a `Trigger` using the `IsMouseOver` property, make sure the control's background is not Null.

The DepartmentStackPanel Style then defines its IsMouseOver Trigger. When IsMouseOver is True, the Trigger sets the StackPanel's Opacity to 1, so the controls are fully opaque, and sets the LayoutTransform property to the previously defined ScaleTransform resource, so the StackPanel is enlarged.

IsActive and IsFocused Triggers

Several of the previous trigger examples take action when the IsMouseOver property is True. The IsMouseOver property is a natural choice for examples because it provides nice, graphical feedback that's easy to see, but you can write triggers to watch for values in just about any property.

The IsActiveTrigger example program shown in Figure 13-11 demonstrates two more triggers that watch the Window's IsActive property and the TextBox's IsFocused property.

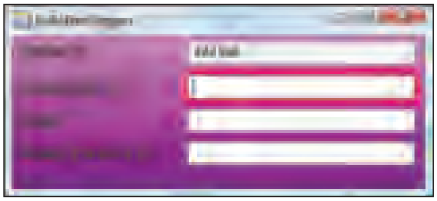


FIGURE 13-11

These triggers provide two effects:

1. First, when the program is inactive because another application has the input focus, the program changes its Title to Inactive.
2. Second, the TextBox that contains the insertion cursor displays a red glow effect.

After the Window's opening <Window> tag, the Window.Style element defines the Window's Style. This example defines the Style directly rather than in a resource that it later references partly for demonstration purposes and partly because there will always be only one Window object in the file.

The following code shows the Window's Style element that changes the Window's Title to Inactive when some other program has the focus:

Available for download on Wrox.com

```
<Window.Style>
  <Style TargetType="Window">
    <Setter Property="Title" Value="IsActiveTrigger"/>
    <Style.Triggers>
      <Trigger Property="IsActive" Value="False">
        <Setter Property="Title" Value="Inactive"/>
      </Trigger>
    </Style.Triggers>
  </Style>
</Window.Style>
```

IsActiveTrigger

The only trick to this Style is the way it sets the Window's Title property. When the form's IsActive property is False, the Trigger tries to set the Title property to Inactive. If the Window element explicitly sets a value for Title in an attribute, then the Trigger won't be able to override that value.

To allow the Trigger to do its job, the Window element must not have a Title attribute. To display a title when it is active, the Style sets the Window's initial Title in its first Setter.

The following code shows the resources that the program uses to display a red glow around the text-box that has the focus:

Available for download on Wrox.com

```
<OuterGlowBitmapEffect x:Key="txtGlow" GlowColor="Red" GlowSize="10"/>

<Style TargetType="TextBox">
  <Setter Property="Margin" Value="5"/>
  <Setter Property="Width" Value="200"/>
  <Style.Triggers>
    <Trigger Property="IsFocused" Value="True">
      <Setter Property="BitmapEffect" Value="{StaticResource txtGlow}"/>
    </Trigger>
  </Style.Triggers>
</Style>
```

IsActiveTrigger

The code first creates an `OuterGlowBitmapEffect` resource. It then defines a `TextBox` `Style` with a `Trigger` that activates when the `IsFocused` property is `True`. The `Trigger` simply sets the `TextBox`'s `BitmapEffect` property to the previously defined glow effect.

SUMMARY

Styles let you define packages of resource values that many controls can share. Shared styles give controls more consistent appearance and behavior, make it easier to change property values without making mistakes, and simplify XAML code so that it's easier to focus on the interesting parts of the window's structure without being distracted by duplicated property values.

You can give a `Style` an `x:Key` value if you want to apply it to individual controls, or you can omit the value to make it apply to every control of the `Style`'s target type.

Property triggers let you define simple behaviors that a control can perform when a property takes on a certain value. For example, you can use property triggers to provide extra feedback, perhaps changing a button's font style, size, or color when the user moves the mouse over it.

While property triggers let a control take action, they are still fairly localized and are intended to act only on the control that defines them. For example, if a `Style` applies to a `Button`, then its property triggers can affect the `Button`'s properties, but they are not intended to modify the `Label` next to the `Button`, the `Grid` containing the `Button`, or the `Window` that holds the whole thing.

Event triggers, which are explained in the next chapter, provide one way around this restriction. While you can define event triggers inside a `Style`, you can also define them at a more global level, where it's easier to work with multiple controls. For example, you could define an event trigger on a `Button` that changes the color of another `Button`, the `Grid` containing the `Buttons`, or the `Window` itself.

14

Event Triggers and Animation

The property triggers described in the previous chapter let you detect specific property values and change some property's value in response. For example, a `Button` might detect when `IsMouseOver` is `True` and change its scale to make the `Button` grow when the mouse is over it.

Event triggers and animation add a new dimension to this type of responsiveness. An event trigger lets you detect when an event occurs. Animation lets you change property values in a series of smoothly varying steps. Together, for example, you can make a `Button` detect the `MouseOver` event and use animation to enlarge the `Button`, providing graphical feedback similar to the experience provided by the previous property event.

So, what's the difference? First, WPF controls provide many events that are not provided by watching property values. For example, while a `Button` has an `IsMouseOver` property to tell you when the mouse is over it, there are no properties that directly correspond to the events `MouseEnter`, `MouseLeave`, `MouseDown`, `MouseUp`, `MouseMove`, and `Click`.

Second, a setter used by property triggers immediately changes a property's value. In contrast, an animation makes a property value gradually change over time. The end result is similar, but the user can see the change in the value as it happens. For example, instead of making a button suddenly pop to a new size, an animation makes it gradually grow. Often the difference isn't critical — the button ends up with the same final size either way — but letting the user watch the change provides a more engaging experience.

This chapter explains event triggers and animations. The following sections explain how you can write event triggers to detect events and execute storyboards. The next sections then explain how storyboards work and the kinds of animations they can execute.

EVENT TRIGGERS

A *property trigger* takes action when a control's property has a certain value. For example, the following style makes a button increase its width and height when the mouse is over it:

```
<Style x:Key="btnPropertyGrowStyle" TargetType="Button">
  <Setter Property="Width" Value="100"/>
  <Setter Property="Height" Value="50"/>
  <Style.Triggers>
```

```
        <Trigger Property="IsMouseOver" Value="True">
          <Setter Property="Width" Value="150"/>
          <Setter Property="Height" Value="75"/>
        </Trigger>
      </Style.Triggers>
    </Style>
```

In contrast, an *event trigger* takes action when a control event occurs. Rather than acting when the IsMouseOver property is True, an event trigger might take action when the MouseEnter event fires, indicating that the mouse moved over the control. Another event could take some other action when the MouseLeave event fires, which occurs when the mouse moves off the control.

To make an event trigger, make a Triggers element, either in a control's definition or in a Style. Inside the Triggers section, add EventTrigger elements.

Each EventTrigger should have a RoutedEvent attribute that indicates the event that executes the trigger. The RoutedEvent should include the name of the class raising the event (Button, Label, Grid, etc.), followed by a dot and the name of the event. For example, the RoutedEvent name for a Button's Click event is *Button.Click*.

Inside the trigger, place an Actions section to hold the actions that you want the trigger to perform. Inside the Actions element, you can place code to run animations by executing storyboards.

For example, the following code shows a Button that takes action when it raises a MouseEnter event (although it doesn't show what that action might be):

```
<Button Width="100" Height="50" Content="Click Me">
  <Button.Triggers>
    <EventTrigger RoutedEvent="Button.MouseEnter">
      <EventTrigger.Actions>
        ... Do something here ...
      </EventTrigger.Actions>
    </EventTrigger>
  </Button.Triggers>
</Button>
```

Normally an event trigger's actions use a BeginStoryboard element to invoke a Storyboard object. Later sections describe storyboards in detail, but for now, assume that you have defined a Storyboard named sbBigScale in a window's Resources section. Then the following code shows how the previous Button could invoke the Storyboard:

Available for download on Wrox.com

```
<Button Width="100" Height="50" Content="Click Me">
  <Button.Triggers>
    <EventTrigger RoutedEvent="Button.MouseEnter">
      <EventTrigger.Actions>
        <BeginStoryboard
         Storyboard="{StaticResource sbBigScale}"/>
      </EventTrigger.Actions>
    </EventTrigger>
  </Button.Triggers>
</Button>
```

GrowingButtons

The `BeginStoryboard` object's `Storyboard` attribute gives the `Storyboard` that it should execute. This example uses the `StaticResource` to run the `Storyboard` defined in the window's resource dictionary.

> **WHAT DOES BEGINSTORYBOARD MEAN?**
>
> Every time I see a `BeginStoryboard` element, I think, "This is the beginning of a storyboard definition," and I expect the subsequent lines of code to define the `Storyboard`. Unfortunately, that's not what `BeginStoryboard` means. As the previous example shows, the `BeginStoryboard` element doesn't need to contain anything more than a reference to a `Storyboard` object.
>
> The *begin* in *BeginStoryboard* really means "execute" or "run," so when you see this element, you should think, "Run this storyboard." Perhaps it would have been less confusing (for me, at least) if Microsoft had named this element `RunStoryboard`.

Event Trigger Locations

The most obvious place to put an event trigger is inside the control that raises the event. For example, the following code defines a `Button` that runs the `sbSpin` `Storyboard` when it is clicked:

Available for download on Wrox.com

```
<Button Content="Spin Me!">
  <Button.Triggers>
    <EventTrigger RoutedEvent="Button.Click">
      <EventTrigger.Actions>
        <BeginStoryboard Storyboard="{StaticResource sbSpin}"/>
      </EventTrigger.Actions>
    </EventTrigger>
  </Button.Triggers>
</Button>
```

SpinButton

The `sbSpin` `Storyboard` rotates the `Button` 360 degrees. Figure 14-1 shows the SpinButton example program when the `Storyboard` has just started.

FIGURE 14-1

The preceding code already uses a few techniques to make the `Button`'s code easier to read. An unnamed button style sets the `Button`'s `Width` and `Height`. It also sets the `Button`'s `RenderTransform` to a `RotateTransform` object so that the `Storyboard` can later rotate the `Button`.

TRANSFORMATION TIP

A `Storyboard` can change a transformation's properties, but it cannot create a new transformation from scratch. In the rotating button example, the `Storyboard` can change the `RenderTransform`'s angle of rotation, but it cannot make the `RenderTransform` out of nothing, so the `Button`'s `Style` makes an initial `RenderTransform` that rotates the button by 0 degrees.

In general, if you want to rotate, stretch, or otherwise change a control's transformation, you must give it an initial transformation for the `Storyboard` to modify.

The `sbSpin` `Storyboard` is also defined as a resource to further simplify the `Button`'s code.

You could even define the `EventTrigger` in the `Button`'s `Style` to make the `Button`'s code really simple.

In some cases, it may make sense to define triggers more centrally. For example, consider the CentralizedTriggers example program shown in Figure 14-2. This program displays a series of thumbnail images on the left. When you click on one, the image expands and moves to fill the area on the right.

FIGURE 14-2

In this example, all of the thumbnail images do more or less the same thing: They expand their images. In programs like this one that display a series of controls that serve similar purposes, it may make the program's logic easier to understand if their triggers are all placed together. This might make sense for a series of thumbnails, buttons, menu items, radio buttons, or any other set of controls that perform roughly the same actions.

When a routed event is triggered, it moves through the hierarchy of controls that contains the one that started it. The CentralizedTriggers example program displays its controls on a `Canvas`, so, when the user clicks on one of the thumbnails (an `Image` control), a `MouseLeftButtonDown` event starts at the thumbnail, moves up to the `Canvas`, and then moves up again to the `Window`. The program's code can catch the event at any of those levels.

The following code fragment shows the structure of the CentralizedTriggers program's `Canvas` control:

Available for download on Wrox.com

```
<Canvas Margin="5">
  <Canvas.Resources>
    <!--  Define an unnamed Image Style.  -->
    <Style TargetType="Image">
      <Setter Property="Opacity" Value="0.5"/>
      <Setter Property="Width" Value="100"/>
      <Setter Property="Height" Value="100"/>
      <Setter Property="Stretch" Value="Uniform"/>
      <Setter Property="BitmapEffect">
        <Setter.Value>
          <DropShadowBitmapEffect/>
        </Setter.Value>
      </Setter>
    </Style>

    <!--  Define a Storyboard for each Image.  -->
    <Storyboard x:Key="sbImg1">
      ... Code omitted ...
    </Storyboard>
    ... Define the other Storyboards ...
  </Canvas.Resources>

  <!--  Handle Image MouseLeftButtonDown events.  -->
  <Canvas.Triggers>
    <EventTrigger SourceName="img1" RoutedEvent="Image.MouseLeftButtonDown">
      <EventTrigger.Actions>
        <RemoveStoryboard BeginStoryboardName="beginSbImg1"/>
        <RemoveStoryboard BeginStoryboardName="beginSbImg2"/>
        <RemoveStoryboard BeginStoryboardName="beginSbImg3"/>
        <RemoveStoryboard BeginStoryboardName="beginSbImg4"/>

        <BeginStoryboard Name="beginSbImg1"
         Storyboard="{StaticResource sbImg1}"/>
      </EventTrigger.Actions>
    </EventTrigger>
    ... Define EventTriggers for the other Images ...
  </Canvas.Triggers>

  <!--  Build the controls.  -->
  <Image Canvas.Left="0" Canvas.Top="0" Name="img1" Source="Canyon01.jpg"/>
  <Image Canvas.Left="0" Canvas.Top="100" Name="img2" Source="Canyon02.jpg"/>
  <Image Canvas.Left="0" Canvas.Top="200" Name="img3" Source="Canyon03.jpg"/>
  <Image Canvas.Left="0" Canvas.Top="300" Name="img4" Source="Canyon04.jpg"/>
</Canvas>
```

CentralizedFigures

The Canvas.Resources section defines an unnamed `Image` `Style` (`Width`, `Height`, `Stretch`, etc.) and the storyboards that enlarge and move the thumbnails to fill the viewing area.

The Canvas.Triggers section defines the `Images`' `EventTriggers`. The previous code shows only the first trigger in detail. Its `SourceName` and `RoutedEvent` attributes indicate that the trigger catches the `MouseLeftButtonDown` event raised by an `Image` control named `img1`.

The Trigger.Actions section uses `RemoveStoryboard` objects to cancel any previously running `Storyboards`, restoring the thumbnails to their original positions. (The section "Controlling Storyboards" later in this chapter says more about objects like `RemoveStoryboard`.) It then uses a `BeginStoryboard` object to run the `sbImg1` `Storyboard`, which moves the `Image` control `img1` to the viewing area.

Having defined the resources and triggers, the program's code that defines the `Image` controls is quite simple and easy to understand.

Often it makes sense to keep triggers with the controls that raise the events, but sometimes it makes the code easier to read if you group similar pieces of code together. That makes the individual sections of code (styles, storyboards, triggers, controls) easier to understand.

Storyboards in Property Elements

Rather than referring to a storyboard defined in a resource dictionary, the code can include a `Storyboard` definition directly inside the `BeginStoryboard` element as an element attribute. The following code shows a `Button` that includes a `Storyboard` defined as an element attribute:

```
<Button Width="100" Height="50" Content="Click Me">
  <Button.Triggers>
    <EventTrigger RoutedEvent="Button.MouseEnter">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard>
            ... Place storyboard animations here ...
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Button.Triggers>
</Button>
```

Storyboards in Styles

Building a storyboard right into a control makes its definition quite long and requires some pretty deeply nested elements, so you may want to define storyboards in resources to simplify the code. If possible, you might want to put the triggers in a style to simplify the `Button`'s code. You can then make the `Style` simpler by placing the `Storyboard` in its own resource.

For example, the following code defines a `Button` `Style` that includes event triggers. When the `Button` raises its `MouseEnter` event, the code runs the `sbBigSize` `Storyboard`. When the `Button` raises its `MouseLeave` event, the code runs the `sbSmallSize` `Storyboard`.

Available for download on Wrox.com

```
<Style x:Key="btnEventSbSize" TargetType="Button">
  <Setter Property="Width" Value="100"/>
  <Setter Property="Height" Value="50"/>
  <Style.Triggers>
    <EventTrigger RoutedEvent="Button.MouseEnter">
      <EventTrigger.Actions>
        <BeginStoryboard Storyboard="{StaticResource sbBigSize}"/>
      </EventTrigger.Actions>
    </EventTrigger>
    <EventTrigger RoutedEvent="Button.MouseLeave">
      <EventTrigger.Actions>
        <BeginStoryboard Storyboard="{StaticResource sbSmallSize}"/>
      </EventTrigger.Actions>
    </EventTrigger>
  </Style.Triggers>
</Style>
```

GrowingButtons

A program could use the following code to define a button that uses this style:

```
<Button Style="{StaticResource btnEventSbSize}"
 Content="New Size"/>
```

PROPERTY TRIGGER ANIMATIONS

Normally a property trigger uses setters to change property values immediately, but you can also use property triggers to run storyboards. How a property trigger applies a storyboard differs from the way it applies a setter because the two have different typical purposes.

A setter changes a property value as long as some other property has a certain value. For example, suppose a `Button` has a property trigger that takes action when the `IsMouseOver` property is `True`. As long as that property is `True`, a `Setter` changes the `Button`'s `Width` to 150. When `IsMouseOver` is no longer `True`, the `Setter` deactivates, and the `Button` returns to its normal size, say 100. Here the `Setter` makes the `Width` property 150 as long as `IsMouseOver` is `True`.

In contrast, an animation makes a property change smoothly from one value to another over time. For example, an animation might make a `Button`'s `Width` vary from its original value of 100 to 150 over a period of 1 second. In that case, it doesn't make sense to have the animation run as long as the `IsMouseOver` property is `True`. The animation will finish in 1 second even if `IsMouseOver` is `True` for 10 minutes. You could make the animation replay again and again, but that would probably be a bit strange in this example.

In order to sensibly apply animations, a property trigger provides two extra sections that can execute animations in addition to ordinary setters: EnterActions and ExitActions.

As before, setters take immediate action.

The animations inside the EnterActions section run when the property's value changes to the target value. In the previous example, they would execute when the `Button`'s `IsMouseOver` property value becomes `True`.

The animations inside the ExitActions section run when the property's value changes from the target value to another value. In the previous example, they would execute when the `Button`'s `IsMouseOver` property changes from `True` to `False`.

The following code shows a `Button Style` that uses a property trigger to invoke both `Setters` and `Storyboards`:

Available for download on Wrox.com

```
<Style x:Key="btnStyle" TargetType="Button">
  <Setter Property="Margin" Value="10"/>
  <Setter Property="Width" Value="100"/>
  <Setter Property="Height" Value="50"/>
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="FontStyle" Value="Italic"/>
      <Trigger.EnterActions>
        <BeginStoryboard Storyboard="{StaticResource sbBigSize}"/>
      </Trigger.EnterActions>
      <Trigger.ExitActions>
        <BeginStoryboard Storyboard="{StaticResource sbSmallSize}"/>
      </Trigger.ExitActions>
    </Trigger>
  </Style.Triggers>
</Style>
```

PropertyTriggerButton

When the `Button`'s `IsMouseOver` property becomes `True`, the `Setter` immediately changes the `Button`'s `FontStyle` to Italic.

When the `IsMouseOver` property changes to `True`, the `EnterActions` execute, launching the `Storyboard` resource named `sbBigSize`. That `Storyboard` animates the `Button`'s `Width` and `Height`, changing to 150 and 75, respectively, over a period of 0.2 second.

When the `IsMouseOver` property changes to `False`, the `ExitActions` execute, launching the `Storyboard` resource named `sbSmallSize`. That `Storyboard` animates the `Button`'s `Width` and `Height`, changing to 100 and 50, respectively, over a period of 0.2 second.

The PropertyTriggerButton example program shown in Figure 14-3 uses this code to make its buttons grow and shrink as you move the mouse over them. In Figure 14-3, the mouse is over the middle button, so that button has italic text. The `sbBigSize` `Storyboard` has finished running, so the button is at its larger size. (You'll just have to imagine the button growing and shrinking until Wrox figures out how to print moving images in its books.)

FIGURE 14-3

The GrowingButtons example program shown in Figure 14-4 demonstrates the three combinations of trigger types and trigger actions: property triggers with setters, property triggers with storyboards, and event triggers with storyboards.

The buttons in the top row change their sizes, so they grow larger but their text stays the same size. The buttons on the bottom row change the scale transformations, so their text grows with them.

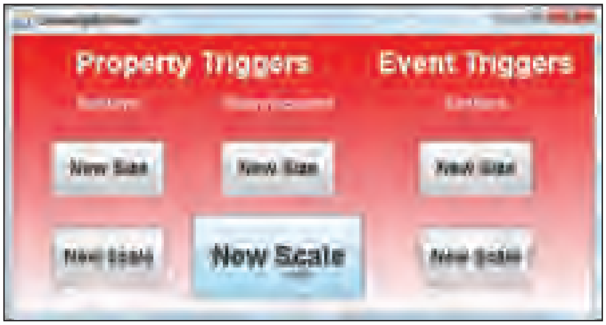


FIGURE 14-4

STORYBOARDS

Now that you understand how to start a storyboard, it's time to learn how to build one. A *storyboard* is an object that defines a timeline for one or more animations.

An *animation* manages the transition of a property from one value to another over time. An animation can include a start time indicating when the transition should begin and a duration telling how long the animation should last.

Animation classes make animation easy by handling a lot of very tricky details. For example, suppose you want to change a control's `Width` property from 50 to 200 over a 1-second interval. To do this yourself, you would need to figure out how many different values you can effectively display in 1 second. The number of *frames* you can display depends on the speed of your computer, the system load, the complexity of the window (enlarging the control might force other controls to move, e.g., if they are in a `StackPanel`), and other factors that are outside your control. With enough time and effort, you could probably work this all out and come up with something reasonable; but, unless your application does nothing more than display animations, you would probably be better off concentrating on other parts of the program.

The animation classes let you specify *what* you want done and then ignore *how* it is done. You say you want a property changed to a certain value over the next 3 seconds, and the animation classes figure out how to do it.

The classes still don't provide absolutely perfect timing. After all, they are subject to the same system issues that you would be if you were to perform the animation yourself. But they do a reasonable job with much less effort on your part.

ANIMATION PERFORMANCE

To get the best performance out of your animations, try to minimize their side effects. For example, suppose you use an animation to change the size of a `Button`. If that `Button` is inside a `StackPanel`, then changes to its size affect the locations of the other controls in the `StackPanel`. If the `Button` contains controls rather

continues

(continued)

than simple text, then changing its size may force WPF to rearrange all of those controls. Depending on the complexity of the program, these kinds of side effects may slow the animation considerably.

When you design an animation, particularly one that changes control positions and sizes, think about the consequences of those changes, and try to minimize the effects on other controls. You can fix this example by moving the `Button` into a `Grid` so that changes to its size don't affect other controls.

Specific kinds of animations handle changes to properties of different data types such as `Double`, `Color`, `String`, `Thickness`, and many others.

The following code shows a simple `Storyboard` containing five `DoubleAnimation` objects. This is the `Storyboard` that the CentralizedTriggers program shown in Figure 14-2 uses to move the first thumbnail into the viewing area.

Available for download on Wrox.com

```
<Storyboard x:Key="sbImg1">
  <DoubleAnimation Duration="0:0:0.5" To="120"
         Storyboard.TargetName="img1"
         Storyboard.TargetProperty="(Canvas.Left)"/>
  <DoubleAnimation Duration="0:0:0.5" To="0"
         Storyboard.TargetName="img1"
         Storyboard.TargetProperty="(Canvas.Top)"/>
  <DoubleAnimation Duration="0:0:0.5" To="485"
         Storyboard.TargetName="img1"
         Storyboard.TargetProperty="Width"/>
  <DoubleAnimation Duration="0:0:0.5" To="400"
         Storyboard.TargetName="img1"
         Storyboard.TargetProperty="Height"/>
  <DoubleAnimation Duration="0:0:0.5" To="1"
         Storyboard.TargetName="img1"
         Storyboard.TargetProperty="Opacity"/>
</Storyboard>
```

CentralizedTriggers

The first animation makes the `Canvas.Left` property of the control `img1` change from its current value (whatever it is) to 120 over a 0.5-second interval. The second animation similarly changes the control's `Canvas.Top` property to 0 over the same 0.5 second.

The third and fourth animations change the `img1` control's `Width` and `Height` properties to 485 and 400, respectively. The final animation changes the control's `Opacity` to 1 so it is completely opaque.

The following section describes the most useful storyboard properties. The next sections then provide more detail about various animation classes.

Storyboard and Animation Properties

The animation classes have several properties that determine the animation's behavior. For example, `Duration` determines how long the animation should take to finish changing the property to its new value.

The `Storyboard` class provides many of the same properties, so the animations it contains can inherit their values. For example, `Storyboard` has a `Duration` property. If you set a `Storyboard`'s `Duration` attribute to `0:0:2` and you do not give the animations their own `Duration` attributes, then the animations each run in 2 seconds.

The following list describes the most useful animation and storyboard properties:

- `AccelerationRatio` — Determines the fraction of the total time that is spent accelerating the value before reaching its peak rate of change.
- `AutoReverse` — If this is `True`, then the animation automatically reverses itself when it is done.
- `BeginTime` — The time at which the animation should start after the `Storyboard` has started running. This has the form *hours:minutes:seconds*.

> **GLACIAL ANIMATION**
>
> If your animation seems to be moving incredibly slowly or not at all, check your `BeginTime` and `Duration` properties and make sure they have the right format. For example, if you accidentally set `Duration` to `0:1` instead of `0:0:1`, then the animation will take 1 minute rather than the 1 second you intended.

- `DecelerationRatio` — Determines the fraction of the total time that is spent decelerating the value from its peak rate of change before stopping at its final value.
- `Duration` — The time the animation should take to move the property from its start value to its final value. This has the form *hours:minutes:seconds*.
- `FillBehavior` — Determines what the animation does when it finishes. This can be `HoldEnd` (the property keeps its final value) or `Stop` (the property immediately snaps back to its "default" value).
- `RepeatBehavior` — Determines how often the animation is repeated. This property can take one of three forms: a repeat count (`2x` means repeat twice, `5x` means repeat five times), a time (`0:0:10` means repeat as needed until 10 seconds have elapsed), or the keyword `Forever` (keep repeating).
- `SpeedRatio` — Determines the speed with which time passes relative to the animation's parent. For example, if `SpeedRatio` is 2, then time passes twice as quickly for the animation as it does for its parent. If the `Storyboard` contains several animations of the same `Duration`, then this one would finish twice as quickly as the others.

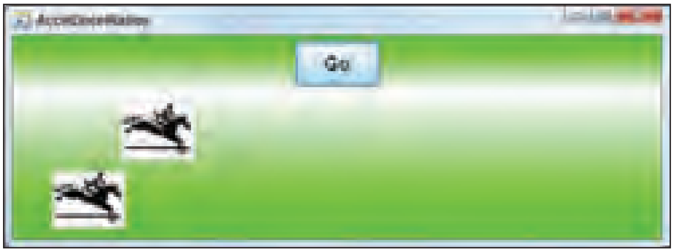


FIGURE 14-5

The AccelDecelRatios example program shown in Figure 14-5 demonstrates the `AccelerationRatio` and `DecelerationRatio` properties. When you click on the Go button, the program starts a storyboard that moves the two `Image` controls across the window. The first `Image` moves at a constant speed throughout the animation. The second `Image` has `AccelerationRatio = 0.4` and `DecelerationRatio = 0.2`, so it takes a while to get up to speed and then slows down as it approaches its final destination. (I guess that horse is a slow starter and pulls up lame at the end.)

The following code shows the storyboard used by the AccelDecelRatios program:

Available for download on Wrox.com

```
<Storyboard>
  <DoubleAnimation Duration="0:0:2" From="5" To="500"
   Storyboard.TargetName="img1"
   Storyboard.TargetProperty="(Canvas.Left)"/>
  <DoubleAnimation Duration="0:0:2" From="5" To="500"
   AccelerationRatio="0.4"
   DecelerationRatio="0.2"
   Storyboard.TargetName="img2"
   Storyboard.TargetProperty="(Canvas.Left)"/>
</Storyboard>
```

AccelDecelRatios

The animation classes add a few extra properties that they do not inherit from the `Storyboard` class. The most important of these are `From`, `To`, and `By`:

- `From` — Indicates the value at which the animation should begin. In the previous code, the `DoubleAnimations` have `From = 5`, so the `Canvas.Left` properties they animate start at the value 5.
- `To` — Indicates the animation's final property value. In the previous example, the `Canvas.Left` property ranges from 5 to 500.
- `By` — The animation ends at the value equal to this value plus the property's original value. For example, if `From` is 10 and `By` is 100, then the final value is 10 + 100 = 110.

If you don't specify a value for the `From` property, the animation begins with the property's current value. That value may be the control's original property value, or it may be the result of a previous animation.

In fact, if an animation is under way but doesn't finish when the new animation starts, the current value may be somewhere in the middle of the previous animation.

For example, suppose you have two buttons that move a control right or left. If you click on the "Move Right" button and then click on the "Move Left" button before the first animation has finished, the control will immediately begin moving back to the left.

> **FROM WHERE?**
>
> Omitting the `From` property is a useful technique because it lets the animation do something sensible even if the previous animation isn't finished. If you explicitly set `From`, then many animations jump to that value before starting their transitions.
>
> For example, suppose you have the "Move Right" and "Move Left" buttons again. If you click on the "Move Right" button and then before it finishes you click on the "Move Left" button, the second button will make the control jump to its final right location before it starts moving back to the left.

Animation Types

WPF provides many classes for animating properties with different data types. The following table lists data types that have corresponding animation classes. For example, you can use the `Int32Animation` class to animate a 32-bit integer property.

`Boolean`	`Int16`	`Point3D`	`String`
`Byte`	`Int32`	`Quaternion`	`Thickness`
`Char`	`Int64`	`Rect`	`Vector`
`Color`	`Matrix`	`Rotation3D`	`Vector3D`
`Decimal`	`Object`	`Single`	
`Double`	`Point`	`Size`	

Most of these data types have two kinds of animation classes. The first has the suffix *Animation* and performs a simple linear interpolation of the property's values so that the property's value is set proportionally between the start and finish values based on the elapsed time. When the `Duration` is 1/4 over, the value will be one-fourth of the way between the start and finish values.

The second type of animation class has the suffix *AnimationUsingKeyFrames*. This type of animation changes a property from one value to another while visiting specific key values in between. You can use several different kinds of objects to specify key frame values, and the type of key frame class determines how the animation moves to the frame's value.

The most common types of key frame objects move to their key values linearly, using a spline, or discretely. A few other key frame animations can make their values follow a path.

The following sections describe linear animations and the main kinds of key frame animations: linear, spline, discrete, and path.

Simple Linear Animations

Linear animations change a property's value in a simple way from one value to another.

For example, the AccelDecelRatios example program shown in Figure 14-5 uses the following `Storyboard` to move its two `Image` controls:

Available for download on Wrox.com

```
<Storyboard>
  <DoubleAnimation Duration="0:0:2" From="5" To="500"
   Storyboard.TargetName="img1"
   Storyboard.TargetProperty="(Canvas.Left)"/>
  <DoubleAnimation Duration="0:0:2" From="5" To="500"
   AccelerationRatio="0.4"
   DecelerationRatio="0.2"
   Storyboard.TargetName="img2"
   Storyboard.TargetProperty="(Canvas.Left)"/>
</Storyboard>
```

AccelDecelRations

Both of the `DoubleAnimation` objects in this code make a `Canvas.Left` property vary from 5 to 500 over a 2-second period. The second animation also demonstrates the `AccelerationRatio` and `DecelerationRatio` properties.

FIGURE 14-6

The RovingButton example program shown in Figure 14-6 provides a more complicated example. It makes a `Button` follow a rectangular path, so its upper-left corner follows the black rectangle shown in the figure.

The following code shows the `Storyboard` that the RovingButton program uses to move its `Button`:

Available for download on Wrox.com

```
<Storyboard x:Key="sbMoveButton" RepeatBehavior="Forever">
  <DoubleAnimation Duration="0:0:1.5" To="370"
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Left)"/>
  <DoubleAnimation BeginTime="0:0:1.5" Duration="0:0:1" To="160"
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Top)"/>
  <DoubleAnimation BeginTime="0:0:2.5" Duration="0:0:1.5" To="10"
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Left)"/>
  <DoubleAnimation BeginTime="0:0:4" Duration="0:0:1" To="10"
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Top)"/>
</Storyboard>
```

RovingButtons

The `sbMoveButton` `Storyboard` uses four `DoubleAnimations` to change the `Button`'s `Left` and `Top` properties along the `Rectangle`'s edges. Each animation has `BeginTime` set to start the animation when the previous one finishes, so the `Button` never stops moving.

Linear Key Frames

A key frame animation that uses linear key frames is similar to a series of simple linear animations. The difference is mainly in whether you want to think of the animation as a series of smaller separate trips or as a single trip with waypoints.

The RovingButtonWithKeyFrames example program uses the following code to make its `Button` run around the same rectangle as the RovingButton program:

Available for download on Wrox.com

```
<Storyboard x:Key="sbMoveButton" RepeatBehavior="Forever">
  <DoubleAnimationUsingKeyFrames
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Left)">
    <LinearDoubleKeyFrame Value="370" KeyTime="0:0:1.5"/>
    <LinearDoubleKeyFrame Value="370" KeyTime="0:0:2.5"/>
    <LinearDoubleKeyFrame Value="10" KeyTime="0:0:4"/>
    <LinearDoubleKeyFrame Value="10" KeyTime="0:0:5"/>
  </DoubleAnimationUsingKeyFrames>

  <DoubleAnimationUsingKeyFrames
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Top)">
    <LinearDoubleKeyFrame Value="10" KeyTime="0:0:1.5"/>
    <LinearDoubleKeyFrame Value="160" KeyTime="0:0:2.5"/>
    <LinearDoubleKeyFrame Value="160" KeyTime="0:0:4"/>
    <LinearDoubleKeyFrame Value="10" KeyTime="0:0:5"/>
  </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

RovingButtonWithKeyFrames

This code uses two `DoubleAnimationUsingKeyFrame` objects, one for the `Button`'s `Canvas.Left` property and one for its `Canvas.Top` property.

The `LinearDoubleKeyFrame` objects give the values that the properties should have at various times during the animation. The program interpolates linearly between the values to make the `Button` move much as the previous code did.

Spline Key Frames

A *spline* is a curve with a shape determined by control points. Figure 14-7 shows a spline in orange. The white squares show the curve's control points. If the curve is being drawn from left to right, then the lower control point is the first control point, and the upper control point is the second control point. These points determine the direction in which the curve leaves the first point (red) and enters the second point (blue).

If you use a linear key frame, the value of its property moves uniformly from the start value to the end value following the green curve in Figure 14-7.

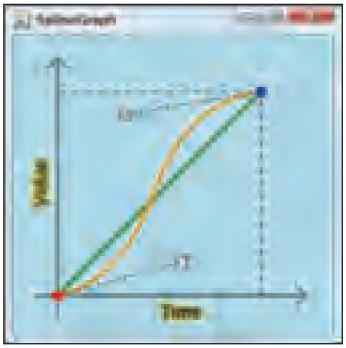


FIGURE 14-7

If you use a spline key frame, the property follows a path similar to the orange spline shown in Figure 14-7. In this example, on the left side of the graph, the value starts by changing relatively little as time increases; in the middle of the graph, the value changes quickly; and near the right side of the graph, the change in the value decreases again. The result is that the value starts moving slowly, picks up the pace, and then slows down again before reaching its final value.

You use the `KeySpline` attribute to specify the control points for a key frame with splines. This attribute should contain two points that give the control point coordinates in a normalized coordinate system (so $0 \le X \le 1$, and $0 \le Y \le 1$).

The RovingButtonWithSplines example program uses the following code to make its `Button` follow the same rectangular path followed by the RovingButton and RovingButtonWithKeyFrames programs. The paths are the same, but the speed with which the `Button` moves is different. In the earlier programs, the `Button` moves at a constant speed across each of the `Rectangle`'s sides. In the RovingButtonWithSplines program, the `Button` starts slowly, accelerates, and then decelerates as it approaches its next value. (It's actually a pretty nice effect. Download the program from the book's web site and see for yourself.)

Available for download on Wrox.com

```
<Storyboard x:Key="sbMoveButton" RepeatBehavior="Forever">
  <DoubleAnimationUsingKeyFrames
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Left)">
     <SplineDoubleKeyFrame Value="370"
     KeyTime="0:0:1.5" KeySpline="0.5,0 0.5,1"/>
    <SplineDoubleKeyFrame Value="370" KeyTime="0:0:2.5"
     KeySpline="0.5,0 0.5,1"/>
    <SplineDoubleKeyFrame Value="10" KeyTime="0:0:4"
     KeySpline="0.5,0 0.5,1"/>
    <SplineDoubleKeyFrame Value="10" KeyTime="0:0:5"
     KeySpline="0.5,0 0.5,1"/>
  </DoubleAnimationUsingKeyFrames>
  <DoubleAnimationUsingKeyFrames
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Top)">
    <SplineDoubleKeyFrame Value="10" KeyTime="0:0:1.5"
     KeySpline="0.5,0 0.5,1"/>
    <SplineDoubleKeyFrame Value="160" KeyTime="0:0:2.5"
     KeySpline="0.5,0 0.5,1"/>
    <SplineDoubleKeyFrame Value="160" KeyTime="0:0:4"
     KeySpline="0.5,0 0.5,1"/>
    <SplineDoubleKeyFrame Value="10" KeyTime="0:0:5"
     KeySpline="0.5,0 0.5,1"/>
  </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

RovingButtonWithSplines

Discrete Key Frames

Linear and spline key frames move a property continuously from one value to another. A *discrete key frame* makes a property suddenly jump from one value to another.

The RovingButtonDiscrete example program uses the following code to animate its `Button`:

Available for download on Wrox.com

```
<Storyboard x:Key="sbMoveButton" RepeatBehavior="Forever">
  <DoubleAnimationUsingKeyFrames
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Left)">
    <DiscreteDoubleKeyFrame Value="130" KeyTime="0:0:0.5"/>
    <DiscreteDoubleKeyFrame Value="250" KeyTime="0:0:1.0"/>
    <DiscreteDoubleKeyFrame Value="370" KeyTime="0:0:1.5"/>
    <DiscreteDoubleKeyFrame Value="370" KeyTime="0:0:2.5"/>
    <DiscreteDoubleKeyFrame Value="250" KeyTime="0:0:3"/>
    <DiscreteDoubleKeyFrame Value="130" KeyTime="0:0:3.5"/>
    <DiscreteDoubleKeyFrame Value="10" KeyTime="0:0:4"/>
  </DoubleAnimationUsingKeyFrames>
  <DoubleAnimationUsingKeyFrames
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Top)">
    <DiscreteDoubleKeyFrame Value="85" KeyTime="0:0:2"/>
    <DiscreteDoubleKeyFrame Value="160" KeyTime="0:0:2.5"/>
    <DiscreteDoubleKeyFrame Value="85" KeyTime="0:0:4.5"/>
    <DiscreteDoubleKeyFrame Value="10" KeyTime="0:0:5"/>
  </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

RovingButtonDiscrete

The example program moves its `Button` in a rectangular path much as the previous examples do, but the `Button` moves in discrete jumps rather than smooth movements.

Because discrete key frames don't generate any intermediate values, your code must list every value that you want the property to become.

Path Animations

Path animations let property values follow the coordinates or angle of a path. For example, you can use a path animation to move a control along the border of an ellipse or a series of curves.

The RovingButtonWithPath example program shown in Figure 14-8 moves a `Button` along the path shown in black. As it moves, the `Button` rotates to match the curve's angle at that point.

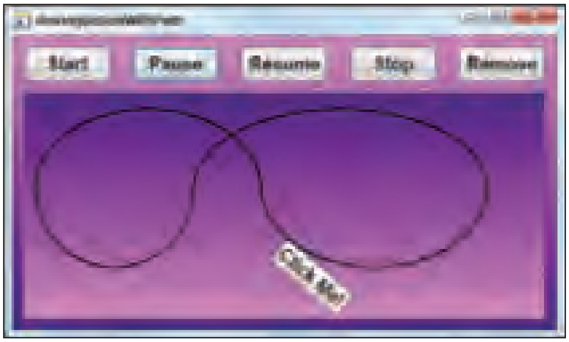


FIGURE 14-8

The RovingButtonWithPath program uses the following code to animate its `Button`:

Available for download on Wrox.com

```
<PathGeometry x:Key="pathMove"
 Figures="M 10,85
  A 100,70 0 1 1 210,85
  A 100,70 0 1 0 410,85
  A 130,70 0 1 0 150,85
```

```
    A 70,70 0 1 1 10,85"/>
  <Storyboard x:Key="sbMoveButton" RepeatBehavior="Forever">
    <DoubleAnimationUsingPath Duration="0:0:4"
     Storyboard.TargetName="btnMover"
     Storyboard.TargetProperty="(Canvas.Left)"
     Source="X"
     PathGeometry="{StaticResource pathMove}"/>
    <DoubleAnimationUsingPath Duration="0:0:4"
     Storyboard.TargetName="btnMover"
     Storyboard.TargetProperty="(Canvas.Top)"
     Source="Y"
     PathGeometry="{StaticResource pathMove}"/>
    <DoubleAnimationUsingPath Duration="0:0:4"
     Storyboard.TargetName="btnMover"
     Storyboard.TargetProperty="RenderTransform.Angle"
     Source="Angle"
     PathGeometry="{StaticResource pathMove}"/>
  </Storyboard>
```

RovingButtonWithPath

The code first creates a PathGeometry object to define the path. It then uses that object in the sbMoveButton Storyboard.

The Storyboard's first DoubleAnimationUsingPath object changes the Button's Canvas.Left property. The Source attribute indicates that the property should be set to the path's X coordinate as the animation progresses over the path.

Similarly, the second DoubleAnimationUsingPath object makes the Button's Canvas.Top property match the path's Y coordinate as the animation progresses.

The Button's definition includes a RenderTransform object that initially rotates the Button by –90 degrees. The final animation object makes the Button's Transform.Angle property match the path's Angle during the animation.

Mix and Match Key Frames

Each of the previous key frame examples uses a single type of key frame, but you can mix and match them if you like. For example, a single storyboard might include linear key frames and discrete key frames. The storyboard can even include path animations.

The RovingButtonMixedKeyFrames example program shown in Figure 14-9 demonstrates a mix of linear and discrete key frames, and path animation. When you click on the Start button, the Button follows the route drawn in black.

FIGURE 14-9

The RovingButtonMixedKeyFrames program uses the following code to animate its `Button`:

Available for download on Wrox.com

```
<PathGeometry x:Key="pathMove"
 Figures="M 350,10 A 50,50 0 1 1 350,170"/>
<Storyboard x:Key="sbMoveButton" RepeatBehavior="Forever">
  <!--  Move right  -->
  <DoubleAnimationUsingKeyFrames BeginTime="0:0:0"
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Left)">
    <LinearDoubleKeyFrame KeyTime="0:0:1" Value="190"/>
    <DiscreteDoubleKeyFrame KeyTime="0:0:1" Value="350"/>
  </DoubleAnimationUsingKeyFrames>

  <!--  Move down  -->
  <DoubleAnimationUsingPath BeginTime="0:0:1" Duration="0:0:1"
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Left)"
   Source="X"
   PathGeometry="{StaticResource pathMove}"/>
  <DoubleAnimationUsingPath BeginTime="0:0:1" Duration="0:0:1"
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Top)"
   Source="Y"
   PathGeometry="{StaticResource pathMove}"/>
  <DoubleAnimationUsingPath BeginTime="0:0:1" Duration="0:0:1"
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="RenderTransform.Angle"
   Source="Angle"
   PathGeometry="{StaticResource pathMove}"/>

  <!--  Move Left  -->
  <DoubleAnimationUsingKeyFrames BeginTime="0:0:2"
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Left)">
    <DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="190"/>
    <LinearDoubleKeyFrame KeyTime="0:0:1" Value="10"/>
  </DoubleAnimationUsingKeyFrames>

  <!--  Rotate right-side up  -->
  <DoubleAnimation BeginTime="0:0:3" Duration="0:0:1" To="0"
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="RenderTransform.Angle"/>

  <!--  Move Up  -->
  <DoubleAnimationUsingKeyFrames BeginTime="0:0:4"
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Top)">
    <LinearDoubleKeyFrame KeyTime="0:0:1" Value="10"/>
  </DoubleAnimationUsingKeyFrames>
</Storyboard>
```

RovingButtonMixedKeyFrames

The Button starts in the upper left, moves horizontally along the solid line, jumps discretely across the dashed line, and follows the circular path, rotating to match the curve's angle. The path leaves the Button upside down.

The Button then jumps across the bottom dashed line and moves horizontally to the left until it reaches the left edge of the form. It then rotates in place until it is right-side up, and finally moves up to its original position.

Special Cases

WPF provides the following animation types for most data types:

- Simple (linear) animation
- Key frame animation with linear key frames
- Key frame animation with spline key frames
- Key frame animation with discrete key frames

It also provides path animation for the data types Double, Matrix, and Point.

A few data types don't provide the standard animation types, however, generally because these animation types don't make sense. For example, it doesn't really make sense to animate a String using linear interpolation or a spline. How would you define a String value that is halfway between "Time" and "banana?"

For the String, Boolean, and Char data types, WPF provides only discrete animation classes.

The AnimatedText example program shown in Figure 14-10 demonstrates string animation. When you click on the Quote button, the program displays the quotation text a couple of letters at a time. After the quotation is visible, Winston Churchill's name gradually fades in at the bottom.

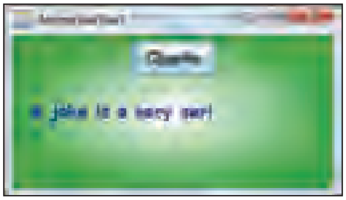


FIGURE 14-10

The following code shows the Storyboard that the AnimatedText program uses to display its quote and attribution:

Available for download on Wrox.com

```
<Storyboard x:Key="sbShowQuote">
  <!--  Display the quote  -->
  <StringAnimationUsingKeyFrames
   Storyboard.TargetName="lblQuote"
   Storyboard.TargetProperty="Content">
    <DiscreteStringKeyFrame KeyTime="0:0:0.1" Value="A j"/>
    <DiscreteStringKeyFrame KeyTime="0:0:0.2" Value="A jok"/>
    <DiscreteStringKeyFrame KeyTime="0:0:0.3" Value="A joke i"/>
    <DiscreteStringKeyFrame KeyTime="0:0:0.4" Value="A joke is a"/>
    <DiscreteStringKeyFrame KeyTime="0:0:0.5" Value="A joke is a ve"/>
    <DiscreteStringKeyFrame KeyTime="0:0:0.6" Value="A joke is a very"/>

    ... Other key frames omitted ...

    <DiscreteStringKeyFrame KeyTime="0:0:1.2"
     Value="A joke is a very serious thing."/>
```

```
    </StringAnimationUsingKeyFrames>

    <!--  Display the attribution  -->
    <DoubleAnimation BeginTime="0:0:2" Duration="0:0:1" To="1"
     Storyboard.TargetName="lblBy"
     Storyboard.TargetProperty="Opacity"/>
  </Storyboard>
```

AnimatedText

The DiscreteStringKeyFrame class is easy to use, although it's rather tedious because you need to specify every change to the String individually.

CONTROLLING STORYBOARDS

In addition to the BeginStoryboard class, which starts a Storyboard running, WPF provides several other objects for controlling Storyboards. The most useful of these classes are:

- PauseStoryboard — Pauses a Storyboard.
- ResumeStoryboard — Resumes a paused Storyboard.
- SeekStoryboard — Moves the Storyboard to a specific position in its timeline.
- StopStoryboard — Stops a Storyboard. This resets properties to their original values.
- RemoveStoryboard — Stops a Storyboard and frees its resources.

Many of the previous examples in this chapter demonstrate most of these classes. For example, the RovingButtonMixedKeyFrames example program shown in Figure 14-9 provides Start, Pause, Resume, Stop, and Remove buttons to let you control its Storyboard.

The following code shows how those example programs manage their Buttons:

Available for download on Wrox.com

```
<EventTrigger RoutedEvent="Button.Click" SourceName="btnStart">
  <EventTrigger.Actions>
    <BeginStoryboard Name="begSbMoveButton"
     Storyboard="{StaticResource sbMoveButton}"/>
  </EventTrigger.Actions>
</EventTrigger>

<EventTrigger RoutedEvent="Button.Click" SourceName="btnPause">
  <EventTrigger.Actions>
    <PauseStoryboard BeginStoryboardName="begSbMoveButton"/>
  </EventTrigger.Actions>
</EventTrigger>

<EventTrigger RoutedEvent="Button.Click" SourceName="btnResume">
  <EventTrigger.Actions>
    <ResumeStoryboard BeginStoryboardName="begSbMoveButton"/>
  </EventTrigger.Actions>
</EventTrigger>

<EventTrigger RoutedEvent="Button.Click" SourceName="btnStop">
```

```
    <EventTrigger.Actions>
      <StopStoryboard BeginStoryboardName="begSbMoveButton"/>
    </EventTrigger.Actions>
  </EventTrigger>

  <EventTrigger RoutedEvent="Button.Click" SourceName="btnRemove">
    <EventTrigger.Actions>
      <RemoveStoryboard BeginStoryboardName="begSbMoveButton"/>
    </EventTrigger.Actions>
  </EventTrigger>
```

RovingButtonMixedKeyFrames

When you click on the Start button, the program uses a `BeginStoryboard` object to start the `Storyboard sbMoveButton` running. The `BeginStoryboard` object has attribute `Name = begSbMoveButton`. The other objects use this name to control the `Storyboard`.

When you click on the Pause button, the program uses a `PauseStoryboard` object, passing it the `BeginStoryboard`'s `Name` so it knows which instance of the `Storyboard` to pause. The other `Buttons` work similarly.

MEDIA AND TIMELINES

The `BeginStoryboard`, `PauseStoryboard`, and other `Storyboard` control classes are really just *action wrappers* — objects that perform actions rather than representing "physical" objects such as buttons, labels, and grids. Another useful action wrapper is `SoundPlayerAction`, which plays an audio file.

The SoundEvents example program uses `SoundPlayerAction` objects in the following code to play sounds when the user clicks on one button or moves the mouse over another:

Available for download on Wrox.com

```
  <Button Content="Click">
    <Button.Triggers>
      <EventTrigger RoutedEvent="Button.Click">
        <EventTrigger.Actions>
          <SoundPlayerAction Source="tada.wav"/>
        </EventTrigger.Actions>
      </EventTrigger>
    </Button.Triggers>
  </Button>
  <Button Content="Hover">
    <Button.Triggers>
      <EventTrigger RoutedEvent="Button.MouseEnter">
        <EventTrigger.Actions>
          <SoundPlayerAction Source="WindowsRingin.wav"/>
        </EventTrigger.Actions>
      </EventTrigger>
    </Button.Triggers>
  </Button>
```

SoundEvents

You can also play a sound inside a `Storyboard`. To do that, add a `MediaElement` control to the window. Then, inside the `Storyboard`, add a `MediaTimeline` object, setting its `Storyboard.TargetName` property to the `MediaElement` and its `Source` property to the name of the sound file you want to play.

The RepeatingSound example program uses the following `Storyboard` to play a sound every time a button passes a certain point in an animation. The `MediaTimeline` object plays the sound in the file WindowsDefault.wav using the `MediaElement` named *medDing*.

Available for download on Wrox.com

```
<Storyboard x:Key="sbMoveButton" RepeatBehavior="Forever">
  <DoubleAnimationUsingPath Duration="0:0:2"
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Left)"
   Source="X"
   PathGeometry="{StaticResource pathMove}"/>
  <DoubleAnimationUsingPath Duration="0:0:2"
   Storyboard.TargetName="btnMover"
   Storyboard.TargetProperty="(Canvas.Top)"
   Source="Y"
   PathGeometry="{StaticResource pathMove}"/>
  <MediaTimeline BeginTime="0:0:1" Source="WindowsDefault.wav"
   Storyboard.TargetName="medDing"/>
</Storyboard>
```

RepeatingSound

MULTIPLE SOUNDS

I've had bad luck trying to play more than one sound in the same storyboard. Multiple MediaTimelines seem to interfere with each other. Perhaps it's a problem with the media player. If you get that working, e-mail me at `RodStephens@vb-helper.com` and let me know the secret.

Unfortunately, you cannot play media files in a `Storyboard` that has its `AutoReverse` property set to `True`. (Presumably, WPF doesn't want to try to reverse the media file.)

One way around this restriction is to build a storyboard that explicitly reverses the actions it takes so it can reverse property animations while not trying to reverse the media file.

A second approach is to make two storyboards or two timelines within the same storyboard. Multiple timelines let different parts of the storyboard run independently, possibly overlapping each other.

The BouncingBall example program shown in Figure 14-11 uses the following `Storyboard` to make its red ball bounce up and down while playing a sound effect each time the ball hits the "ground."

Available for download on Wrox.com

```
<!-- One Storyboard to rule them all. -->
<Storyboard x:Key="sbBounce" RepeatBehavior="Forever">
  <!-- Play the sound after 1 second. -->
  <ParallelTimeline BeginTime="0:0:0">
    <MediaTimeline BeginTime="0:0:1" Source="boing.wav"
     Storyboard.TargetName="medBoing"/>
```

```
        </ParallelTimeline>

        <!--  Move the ball and its shadow.  -->
        <ParallelTimeline BeginTime="0:0:0" AutoReverse="True">
          <DoubleAnimationUsingKeyFrames
           Storyboard.TargetName="ellBall"
           Storyboard.TargetProperty="(Canvas.Top)">
            <SplineDoubleKeyFrame KeyTime="0:0:1"
             KeySpline="0.5,0 1,1"
             Value="120"/>
          </DoubleAnimationUsingKeyFrames>
          <DoubleAnimationUsingKeyFrames
           Storyboard.TargetName="ellShadow"
           Storyboard.TargetProperty="Opacity">
            <SplineDoubleKeyFrame KeyTime="0:0:1"
             KeySpline="0.5,0 1,1"
             Value="1"/>
          </DoubleAnimationUsingKeyFrames>
        </ParallelTimeline>
      </Storyboard>
```

BouncingBall

The Storyboard contains two ParallelTimeline objects that run independently. The first waits 1 second until the ball is touching the "ground" and then uses a MediaTimeline object to play the sound effect.

FIGURE 14-11

The second ParallelTimeline contains two DoubleAnimationUsingKeyframes objects, one to control the ball's Canvas.Top property and one to control the shadow's Opacity. As the ball's animation moves the ball downward, the shadow's animation makes the shadow more opaque. When the animations end, the ball is touching the "ground," and the shadow is fully opaque.

Both animations use similar spline key frames with control points set so that the controlled property value starts changing slowly and speeds up at the end of the animation.

The ParallelTimeline containing the two animations has its AutoReverse property set to True, so after the ball reaches the "ground," it bounces back up to its starting position.

BAD BOUNCES

Recall that an animation that doesn't set its From property starts animating its property from its current value, whatever that may be. This can lead to some unexpected and amusing effects.

continues

(continued)

For example, run the BouncingBall program and click on the Start button. When the ball is almost to the "ground," click on the Pause button to freeze the animations. If you now click on the Resume button, the ball continues falling as usual. However, if you click on the Start button, the animations start running from the ball's current position. Because the animations don't set their `From` values, the animations make the ball bounce up and down from whatever height it had when you paused it. The result is a series of very slow, small bounces. (You can click on the Stop button to reset the program.)

You should always check your animations to see what happens when they are interrupted and to decide whether the results are acceptable.

ANIMATION WITHOUT STORYBOARDS

XAML code can only run animations with storyboards, but code-behind can use animations without a storyboard. The basic idea is to create an animation object and set its properties or give it key frames as appropriate. Then using the control that you want to animate, call the control's `BeginAnimation` method, passing it the property that you want to animate and the animation object.

The AnimationWithoutStoryboards example program shown in Figure 14-12 uses animations built in code-behind to display a ball bouncing off walls. Click on the ball (or press [Alt]+F) to close the program.

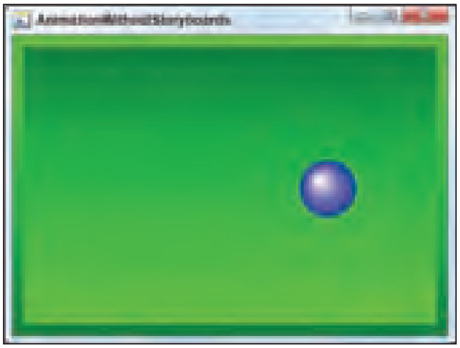


FIGURE 14-12

The AnimationWithoutStoryboards program uses the following C# code to build and start its animations. (You can download this code or a Visual Basic version on the book's web site.)

Available for download on Wrox.com

```
// Start the animation.
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  // Get geometry info.
  double ball_wid = ellBall.ActualWidth;
  double ball_hgt = ellBall.ActualHeight;
  int max_x = (int)(canTable.ActualWidth - ball_wid);
  int max_y = (int)(canTable.ActualHeight - ball_hgt);

  // Set the ball's initial position.
  Random rand = new Random();
  double x = rand.Next(0, max_x);
  double y = rand.Next(0, max_y);
  Canvas.SetLeft(ellBall, x);
  Canvas.SetTop(ellBall, y);

  // Make animations.
```

```
    const double TRANSIT_TIME = 1;
    DoubleAnimationUsingKeyFrames x_animation =
      new DoubleAnimationUsingKeyFrames();
    x_animation.RepeatBehavior = RepeatBehavior.Forever;
    double l_time = TRANSIT_TIME * x / max_x;
    double r_time = TRANSIT_TIME - l_time;
    // To right edge.
    x_animation.KeyFrames.Add(
      new LinearDoubleKeyFrame(
        max_x,
        KeyTime.FromTimeSpan(
          TimeSpan.FromSeconds(r_time))));
    // Back to left edge.
    x_animation.KeyFrames.Add(
      new LinearDoubleKeyFrame(
        0,
        KeyTime.FromTimeSpan(
          TimeSpan.FromSeconds(r_time + TRANSIT_TIME))));
    // Back to start.
    x_animation.KeyFrames.Add(
      new LinearDoubleKeyFrame(
        x,
        KeyTime.FromTimeSpan(
          TimeSpan.FromSeconds(r_time + TRANSIT_TIME + l_time))));

    DoubleAnimationUsingKeyFrames y_animation =
      new DoubleAnimationUsingKeyFrames();
    y_animation.RepeatBehavior = RepeatBehavior.Forever;
    double t_time = TRANSIT_TIME * y / max_y;
    double b_time = TRANSIT_TIME - t_time;
    // To bottom edge.
    y_animation.KeyFrames.Add(
      new LinearDoubleKeyFrame(
        max_y,
        KeyTime.FromTimeSpan(
          TimeSpan.FromSeconds(b_time))));
    // Back to top edge.
    y_animation.KeyFrames.Add(
      new LinearDoubleKeyFrame(
        0,
        KeyTime.FromTimeSpan(
          TimeSpan.FromSeconds(b_time + TRANSIT_TIME))));
    // Back to start.
    y_animation.KeyFrames.Add(
      new LinearDoubleKeyFrame(
        y,
        KeyTime.FromTimeSpan(
          TimeSpan.FromSeconds(b_time + TRANSIT_TIME + t_time))));

    // Apply the animations.
    ellBall.BeginAnimation(Canvas.LeftProperty, x_animation);
    ellBall.BeginAnimation(Canvas.TopProperty, y_animation);
}
```

AnimationWithoutStoryboards

The code first gathers some geometry information. It then gives the ball a random position on the Canvas. (Notice how the program uses the Canvas class's SetLeft and SetTop methods to set the values of the ball's Canvas.Left and Canvas.Top attached properties.)

Next the program creates a Double AnimationUsingKeyframes object to manage the ball's X coordinate and sets its RepeatBehavior to Forever. It calculates the time the ball should spend moving to the left and right of its initial position. The code then creates three LinearDoubleKeyframe objects to make the ball move to the right edge, back to the left edge, and then back to its starting position. (Note that the key times are cumulative so, e.g., the key times might be 0:0:0.25, 0:0:1.25, and 0:0:2.)

The code then repeats these steps to make an animation for the ball's Y coordinate.

Finally, the code calls the ball Ellipse's BeginAnimation method, passing it the properties it should animate and the corresponding animation objects. Notice how the code uses the Canvas class's LeftProperty and TopProperty values to identify the Ellipse's attached Canvas.Left and Canvas.Top properties.

Using animation, storyboard, key frame, and other classes, your code-behind can build any animation that you can make using XAML.

EASY ANIMATIONS

Building animations can be a tedious business. The code is long and deeply nested for even simple animations. For example, the JumpingButton example program uses the following code to build one of the simplest animations possible, modifying a single Button property to make the Button move up and then back to its starting position.

Available for download on Wrox.com

```
<Button Content="Jump!" Width="100" Height="40"
 Canvas.Top="65" Canvas.Left="95">
  <Button.Triggers>
    <EventTrigger RoutedEvent="Button.Click">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard>
            <DoubleAnimation To="10" Duration="0:0:0.25"
             AutoReverse="True"
             Storyboard.TargetProperty="(Canvas.Top)"/>
          </Storyboard>
        </BeginStoryboard>
      </EventTrigger.Actions>
    </EventTrigger>
  </Button.Triggers>
</Button>
```

JumpingButton

If you've read this chapter, you should be able to understand this code without too much trouble, but it is 16 lines long and uses 6 levels of nested code. More complicated animations can be quite involved and confusing.

One of the most useful features of Expression Blend is its ability to build animations for you. Simply open a `Storyboard` and change control properties at various times. Expression Blend keeps track of the property values at different times and builds a complete `Storyboard` for you.

This technique is particularly useful for animating complex properties like the colors used in a gradient brush. You may want to edit the result manually, but at least Expression Blend can get you started.

For more information on using Expression Blend to build `Storyboards`, see the section "Storyboards" in Chapter 3. You can also consult the online help, particularly the Expression Blend web pages for animation at `msdn.microsoft.com/cc294924.aspx` and the page "Create, Modify, or Delete a Storyboard" at `msdn.microsoft.com/cc295300.aspx`.

SUMMARY

This chapter explains event triggers. It shows how you can use event triggers to execute storyboards. It explains how you can use property triggers to execute storyboards before and after a property attains a certain value.

This chapter also explains storyboards and the animations that they perform. It shows how to build storyboards in a control's property elements, in a style, or in resources.

Finally, this chapter provided several examples demonstrating various animation techniques such as using key frames, using spline key frames, building path-following animations, and playing sounds.

All of these techniques give new features to existing controls. They allow a control to perform actions that are not always part of its repertoire, for example, allowing a `Button` to change its color, a `Rectangle` to change its size, or one control to modify the properties of another.

The *templates* described in the next chapter allow you to change a control's behavior at a more fundamental level. Whereas the triggers and animations described in this chapter let you add new actions to a control, templates let you redefine the control's basic behavior, for example, letting you change the way a `Button` responds when the mouse moves over it.

15

Templates

Properties and styles determine a control's appearance and behavior. For example, a `Slider` control's `TickFrequency`, `TickPlacement`, `Background`, and `Width` properties help determine its appearance, while its `Minimum`, `Maximum`, `LargeChange`, and `IsEnabled` properties help determine its behavior.

In contrast, *templates* determine a control's structure. They determine what components make up the control and how those components interact to provide the control's features.

This chapter describes templates in general terms and shows how you can build templates of your own to change the way existing controls work.

TEMPLATE OVERVIEW

If you look closely at Figure 15-1, you can see that a `Slider` control has a bunch of parts including:

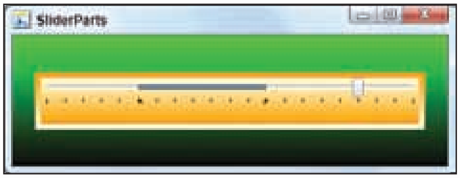


FIGURE 15-1

- A border
- Tick marks
- A background
- Clickable areas on the background (basically anywhere between the top of the control and its tick marks vertically) that change the current value
- A Thumb indicating the current value that you can drag back and forth
- Selection indicators (the little black triangles) that indicate a selected range

These features are provided by the pieces that make up the `Slider`. By default, a `Slider` is made up of a multitude of `Border`, `Grid`, `TickBar`, `Track`, `RepeatButton`, `Rectangle`, `Thumb`, `Canvas`, and `Path` controls, together with many `brushes`, `transformations`, `styles`, and `triggers`.

A *template* determines what the pieces are that make up a control. It determines the control's components together with their styles, triggers, and everything else that is needed by the control. As an analogy, consider a car. Its *properties* are easily changed — things like its color, upholstery, and vanity plate (e.g., *WPF FAN*). Its *template* defines the things it is made of — for example, its chassis, number of doors, and engine.

No matter what combination of properties and components you pick, however, it has certain standard car-like features such as turning on, accelerating, decelerating, turning off, and costing way too much to insure. As you drive down the street, you will see hundreds of combinations, but they are all easily recognizable as cars.

[OK, there may be a few that are hard to recognize such as the MULE robotic logistics vehicle (`www.botmag.com/articles/mule.shtml`), the Terrafugia Transition flyable-car/roadable-plane (`www.theregister.co.uk/2008/07/29/terrafugia_transition_on_show_oshkosh`), or the Toyota PM (`www.toyota.com/concept-vehicles/pm.html`), which looks more like a Star Wars pod racer than a car, but I have yet to see any of these on the road.]

Note that the components influence the behavior of the car. A hybrid has great fuel efficiency but slow acceleration, while a 12-cylinder sports car has great acceleration but poor mileage. Similarly, the components that make up a control can change the way it behaves.

Because the controls in the template determine the control's appearance, WPF controls are sometimes called *lookless*. By creating your own template for an existing control such as a `Button` or `CheckBox`, you can give the control a new appearance and behavior.

WORK WARNING

Building a template can be a lot of work. When you build a template, you take responsibility for most of the control's behavior. You cannot make a `Button` use a diamond-shaped polygon for its surface and expect it to automatically do everything that a normal `Button` does. If you decide to use a template to make a diamond-shaped `Button`, then you need to build most of the `Button`'s behaviors yourself.

The `Button` still provides some very basic features such as raising a `Click` event when the user clicks it, but you need to handle things such as changing the `Button`'s appearance when the mouse is over it, when the user presses the mouse, when the mouse moves off it, and so forth.

CONTENTPRESENTER

If you're assembling a new control from components of your own choosing, how do you handle the essential features of the control? For example, if you're building a template for `Label` controls, perhaps displaying the text inside a `Border` with a beveled edge, how do you know what text to display?

The answer is the `ContentProvider`. The `ContentProvider` is an object that WPF provides to display whatever it is that the control should display. You can place the `ContentProvider` in whatever control hierarchy you build for the template, and it will display the content.

For example, the following code shows an extremely simple `Label` template:

Available for download on Wrox.com

```
<Window.Resources>
    <ControlTemplate x:Key="temSimpleLabel" TargetType="Label">
        <Border BorderBrush="Red" BorderThickness="1">
            <ContentPresenter/>
        </Border>
    </ControlTemplate>
</Window.Resources>
```

SimpleLabelTemplate

The template's name is `temSimpleLabel`, and it applies to `Label` controls. The template contains a `Border` control that displays a red border and that holds the `ContentPresenter`.

The following code shows how the program might use this template. This code creates a `Label`. Its last attribute sets the control's `Template` property to the previously created template.

Available for download on Wrox.com

```
<Label Margin="5" Content="No Template"
 HorizontalContentAlignment="Right"
 VerticalContentAlignment="Center"
 BorderBrush="Yellow" BorderThickness="2"
 Foreground="{StaticResource brForeground}"
 Background="{StaticResource brBackground}"
 Template="{StaticResource temBorderLabel}"
/>
```

SimpleLabelTemplate

The SimpleLabelTemplate example program shown in Figure 15-2 displays two `Labels`. The one on the left uses no template, while the one on the right uses the `temSimpleLabel` template.

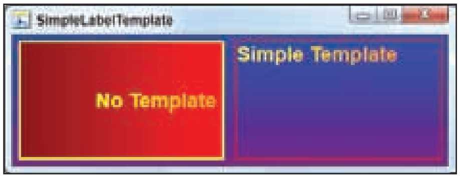


FIGURE 15-2

TEMPLATE BINDING

If you compare the two `Labels` in Figure 15-2, you'll see that even this simple example has some potential problems. Because the template's `Border` control includes explicit values for its `BorderBrush` and `BorderThickness` properties, it overrides any values set in the code that creates the `Label`. The `Border` control also doesn't specify a `Background`, so it uses its default transparent background.

This means the templated control doesn't display the correct background or border. It also doesn't honor the requested `HorizontalContentAlignment` and `VerticalContentAlignment` values.

Fortunately, a template can learn about some of the properties set on the client control by using a *template binding*. For example, the following code fragment sets the `Background` property for a piece of the template to the value set for the control's `Background` property:

```
Background="{TemplateBinding Background}"
```

Template bindings let the template honor values set for the control where appropriate while overriding other values to achieve the appearance you desire.

The following code shows a better version of the Label template that honors several of the control's background and foreground properties:

Available for download on Wrox.com

```
<ControlTemplate x:Key="temBetterLabel" TargetType="Label">
    <Border
     Background="{TemplateBinding Background}"
     BorderBrush="{TemplateBinding BorderBrush}"
     BorderThickness="{TemplateBinding BorderThickness}">
        <ContentPresenter Margin="4"
         HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
         VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>
    </Border>
</ControlTemplate>
```

BetterLabelTemplate

In this template, the `Border` control mimics the client control's `Background`, `BorderBrush`, and `BorderThickness` properties. The `ContentPresenter` sets its `HorizontalAlignment` and `VerticalAlignment` properties to the client control's `HorizontalContentAlignment` and `VerticalContentAlignment` values so that the result is properly aligned within the control.

The BetterLabelTemplate example program shown in Figure 15-3 uses this template to display a `Label` that looks much more like one that has no template.

So now that you can create a template `Label` that looks like a regular `Label`, what's the point? If you just want a `Label` that looks like a `Label`, use a `Label`!

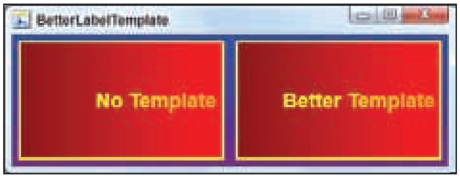


FIGURE 15-3

The point is that you don't have to copy every feature of the original control. You can add bitmap effects, rotate the label, insert an image, and make other changes. For example, the following section describes two `Label` templates that add features not provided by the normal `Label` control.

CHANGING CONTROL APPEARANCE

Of course, you won't always want your template to match exactly the appearance of a control without a template. If you did, you wouldn't bother going to all the trouble of making a template.

Your template will override properties, implement new behaviors, and build the template from controls other than those used by the original control to provide a unique experience.

The InterestingLabelTemplates example program shown in Figure 15-4 demonstrates two more interesting `Label` templates. The first draws a double border around its text if the `Label` specifies `BorderBrush` and `BorderThickness` properties. The second can display text wrapped across multiple lines.

FIGURE 15-4

The following code shows how the InterestingLabelTemplates program displays its `Label` with a double border:

Available for download on Wrox.com

```
<ControlTemplate x:Key="temDoubleBorderLabel" TargetType="Label">
    <Border Background="{TemplateBinding Background}"
     BorderBrush="{TemplateBinding BorderBrush}"
     BorderThickness="{TemplateBinding BorderThickness}">
         <Border Margin="2" Background="Transparent"
          BorderBrush="{TemplateBinding BorderBrush}"
          BorderThickness="{TemplateBinding BorderThickness}">
              <ContentPresenter Margin="2"
               HorizontalAlignment="Center"
               VerticalAlignment="{TemplateBinding VerticalContentAlignment}"/>
        </Border>
    </Border>
</ControlTemplate>
```

InterestingLabelTemplates

This template displays a `Border` control that matches the client control's `Background`, `BorderBrush`, and `BorderThickness` properties.

Inside that is another `Border` control with its `Margin` set to 2, so it sits inside the first `Border`. Its `Background` is set to `Transparent`, so it doesn't cover the background used by the outer `Border`, although the inner `Border` also obeys the client's `BorderBrush` and `BorderThickness` properties.

Finally, inside the inner `Border`, the `ContentPresenter` displays the client's content as before.

The following code shows how the program displays its second `Label` with wrapped text:

Available for download on Wrox.com

```
<ControlTemplate x:Key="temWrappedLabel" TargetType="Label">
    <Grid>
        <Border
         Background="{TemplateBinding Background}"
         BorderBrush="{TemplateBinding BorderBrush}"
         BorderThickness="{TemplateBinding BorderThickness}">
            <TextBlock Name="txtbContent"
             Margin="4"
             TextWrapping="Wrap"
             Text="{TemplateBinding ContentPresenter.Content}"/>
        </Border>
    </Grid>
</ControlTemplate>
```

InterestingLabelTemplates

This template displays a `Border` as before. The `Border` contains a `TextBlock` with `TextWrapping = True`, so it wraps its content if necessary. The `TextBlock`'s `Text` property is set to the `ContentPresenter`'s `Content` property.

Note that this only works if the `ContentPresenter` is trying to display text. For example, if you build the client `Label` control so that it contains a `Button` as shown in the following code, then the `TextBlock` doesn't display anything:

Available for download on Wrox.com

```
<Label Margin="5"
 HorizontalContentAlignment="Right"
 VerticalContentAlignment="Center"
 BorderBrush="Yellow" BorderThickness="2"
 Foreground="{StaticResource brForeground}"
 Background="{StaticResource brBackground}"
 Template="{StaticResource temWrappedLabel}"
>
    <Button Content="Click Me"/>
</Label>
```

InterestingLabelTemplates

TEMPLATE EVENTS

The `Label` control used in the previous example is one of the simplest controls. It mostly just sits there looking pretty without bothering to interact with the user.

But more complicated controls like `Button`, `CheckBox`, and `Slider` must perform all sorts of stunts as the mouse moves, presses, drags, and releases over them.

To make a template control respond to events, you can add property and event triggers to the template much as you added them to styles in Chapters 13 and 14.

In addition to events caused by user actions such as moving or pressing the mouse, controls must respond to changes in state. For example, although a `Label` mostly just sits around doing nothing, it should also change its appearance when it is disabled. If you don't need to display complex animations, then it can simply respond with `Setters` in a property `Trigger` that runs when the `IsEnabled` property is `False`.

The DisabledLabelTemplate example program shown in Figure 15-5 uses a template that gives a disabled `Label` a distinctive appearance.

FIGURE 15-5

The following code shows the template that gives the disabled `Label` its appearance:

Available for download on Wrox.com

```
<ControlTemplate x:Key="temWrappedLabel" TargetType="Label">
    <Grid>
        <Border Name="brdMain"
         Background="{TemplateBinding Background}"
         BorderBrush="{TemplateBinding BorderBrush}"
         BorderThickness="{TemplateBinding BorderThickness}">
            <TextBlock Name="txtbContent"
             Margin="4"
             TextWrapping="Wrap"
             Text="{TemplateBinding ContentPresenter.Content}"/>
        </Border>
        <Canvas Name="canDisabled" Opacity="0">
```

```
                <Canvas.Background>
                    <LinearGradientBrush StartPoint="0,0" EndPoint="3,3"
                     MappingMode="Absolute"
                     SpreadMethod="Repeat">
                        <GradientStop Color="LightGray" Offset="0"/>
                        <GradientStop Color="Black" Offset="1"/>
                    </LinearGradientBrush>
                </Canvas.Background>
            </Canvas>
        </Grid>
        <ControlTemplate.Triggers>
            <Trigger Property="IsEnabled" Value="False">
                <Setter TargetName="canDisabled"
                 Property="Opacity" Value="0.5"/>
                <Setter TargetName="txtbContent"
                 Property="Foreground" Value="Gray"/>
            </Trigger>
        </ControlTemplate.Triggers>
    </ControlTemplate>
```

DisabledLabelTemplate

This version of the `Template` starts with a `Grid` control that contains a `Border` and a `Canvas`. The `Border` holds a `TextBlock` that displays the control's `ContentPresenter` as before. The `Canvas` covers the `Border`, is filled with a linear gradient brush, and initially has `Opacity = 0` so it is invisible.

The template's Triggers section contains a property trigger that activates when the control's `IsEnabled` property is `False`. When that happens, the trigger sets the `Canvas`'s `Opacity` property to 0.5 so it partially obscures the control's content. It also changes the `TextBlock`'s `Foreground` to Gray.

TEMPLATE TRICKS 1

Using a control with `Opacity = 0` is a common and particularly useful template trick. The template can use it to display something new, cover something old, or, as in this example, partially obscure whatever lies behind it.

You can use a translucent white control to wash out whatever is behind, a translucent black control to darken whatever is behind, or an opaque control to cover the background controls completely.

TEMPLATE TRICKS 2

It's easier for triggers to manipulate the template's controls and other objects if you give those objects names. In this example, the `TextBlock` is named `txtbContent` and the translucent `Canvas` is named `canDisabled`, so it's easy for the triggers to control them. If you'll need to animate it, give it a name.

The following sections describe some much more complex templates that change the way `Buttons` work.

GLASS BUTTON

The GlassButton example program shown in Figure 15-6 uses a template to give its buttons a glassy appearance.

FIGURE 15-6

The disabled button on the left looks washed-out and doesn't respond to the user.

The second button labeled *Default* has a different border from that of the other buttons. If no other button has the focus when the user presses the [Enter] key, that `Button` fires its `Click` event. In Figure 15-6, the `TextBox` has the focus, so pressing the [Enter] key will fire the default `Button`.

DESIGNATED DEFAULT

Just because a `Button`'s `IsDefault` property is `True`, that doesn't mean that the `Button` always fires when the user presses [Enter]. If the focus is on another `Button`, then the [Enter] key fires that `Button` instead of the default.

Also, when the default `Button` has the focus, it behaves like any other `Button` with the focus, so it is not acting as the default `Button` at that time.

When a `Button` is acting as the default, it is said to be *defaulted*. You (or, more importantly, your triggers) can see whether a `Button` is defaulted by checking its `IsDefaulted` property.

Figure 15-7 shows the program when the mouse is over Button 3. The button under the mouse becomes less transparent. Notice that the focus is still in the `TextBox` (you can see the caret in Figure 15-7), so the default button still shows its distinctive border.

FIGURE 15-7

Figure 15-8 shows the program when the user presses the mouse down on Button 3.

FIGURE 15-8

At this point, the pressed button is opaque. Pressing the button also moves focus to that `button`. Because focus is now on Button 3, the default button is no longer defaulted. In fact, no button is defaulted right now. Button 3 has the focus so it will fire if the user presses [Enter] but it is not defaulted so it won't display the default border even after the user releases the mouse.

If you drag the mouse off the button while it is still pressed, the button returns to its focused "mouse over" appearance. If you then release the mouse, no mouse click occurs.The following three sections describe the glass button's `Template`. The first describes the `Template` at a high level, explaining the controls the `Template` uses and how they fit together. The two sections that follow describe the `Template`'s `Styles` and `Triggers`.

> *This program is fairly long so the complete code isn't shown in these sections. You can download the example program from the book's web site to see the details.*

Glass Button Template Overview

The following code snippet shows the `Template`'s main sections and the controls that it uses.

Available for download on Wrox.com

```
<ControlTemplate x:Key="temGlassButton" TargetType="Button">
    <ControlTemplate.Resources>
        ... Template Styles omitted here...
    </ControlTemplate.Resources>

    <Grid Name="grdMain" ClipToBounds="True" Opacity="0.5"
     Width="{TemplateBinding Width}"
     Height="{TemplateBinding Height}">
        <Rectangle Name="rectMain"/>

        <ContentPresenter
         VerticalAlignment="Center"
         HorizontalAlignment="Center"/>
    </Grid>

    <!-- Behaviors. -->
```

```
        <ControlTemplate.Triggers>
            ... Template triggers omitted here...
        </ControlTemplate.Triggers>
    </ControlTemplate>
```

GlassButton

The `Template`'s controls are relatively simple. The `Template` contains a `Grid` that holds a `Rectangle` and the `ContentPresenter`. The `ContentPresenter`'s attributes center it on the `Button`, but all of the other interesting properties are set in the `Template`'s `Styles`.

Glass Button Styles

The following code shows the `Styles` defined in the template's Resources section. The `Button` looks differently when it is in the three states (normal, defaulted, and disabled). To make the code easier to understand, the template uses three different `Styles` for those states. The code also includes a base `Style` from which the others inherit.

Available for download on Wrox.com

```
    <!--  Base style that sets corner radii and stroke thickness.  -->
    <Style x:Key="styBase" TargetType="Rectangle">
        <Setter Property="RadiusX" Value="20"/>
        <Setter Property="RadiusY" Value="20"/>
        <Setter Property="StrokeThickness" Value="5"/>
    </Style>

    <!--  Style for "normal" status.  -->
    <Style TargetType="Rectangle"
     BasedOn="{StaticResource styBase}">
        <Setter Property="Fill">
            <Setter.Value>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                    <GradientStop Color="DarkGreen" Offset="0"/>
                    <GradientStop Color="LightGreen" Offset="1"/>
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
        <Setter Property="Stroke">
            <Setter.Value>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                    <GradientStop Color="DarkGreen" Offset="1"/>
                    <GradientStop Color="LightGreen" Offset="0"/>
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>

    <!--  Style when IsDefaulted.  -->
    <Style x:Key="styIsDefaulted" TargetType="Rectangle"
     BasedOn="{StaticResource styBase}">
        <Setter Property="Fill">
            <Setter.Value>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                    <GradientStop Color="DarkGreen" Offset="0"/>
```

```
                    <GradientStop Color="LightGreen" Offset="1"/>
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
        <Setter Property="Stroke">
            <Setter.Value>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                    <GradientStop Color="DarkGreen" Offset="1"/>
                    <GradientStop Color="Black" Offset="0"/>
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>

    <!--  Style when disabled.  -->
    <Style x:Key="styDisabled" TargetType="Rectangle"
     BasedOn="{StaticResource styBase}">
        <Setter Property="Opacity" Value="0.75"/>
        <Setter Property="Fill">
            <Setter.Value>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                    <GradientStop Color="Gray" Offset="0"/>
                    <GradientStop Color="White" Offset="1"/>
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
        <Setter Property="Stroke">
            <Setter.Value>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                    <GradientStop Color="Gray" Offset="1"/>
                    <GradientStop Color="White" Offset="0"/>
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>
```

GlassButton

The first `Style` is a base `Style` that sets the `Rectangle`'s `RadiusX`, `RadiusY`, and `StrokeThickness` properties. These properties are the same for all of the `Button`'s states.

Since the `Button`'s "normal" `Style` is an unnamed `Rectangle Style`, it always applies unless some other `Style` overrides it. It sets the control's `Fill` property to a `LinearGradientBrush` that shades from dark green to light green. The `Style` then sets the `Stroke` property to a brush that does the opposite: It shades from light green to dark green. (This remarkably simple technique gives the `Button` an easy 3D appearance.)

The "defaulted" `Style` uses the same `Fill` property but makes the `Stroke` brush shade from dark green to black. (The difference is actually fairly subtle. You might want to experiment with larger changes, perhaps adding a completely black outline.)

The "disabled" `Style` sets the `Rectangle`'s `Opacity` property to 0.75, so it is translucent. It also changes the `Rectangle`'s `Fill` and `Stroke` properties to shade between gray and white.

Glass Button Triggers

The following code shows the `Template`'s `Triggers`. In response to events and changes in the control's properties, the `Triggers` set new property values and apply the `Styles`.

Available for download on Wrox.com

```
<!--  Mouse over.  -->
<Trigger Property="IsMouseOver" Value="True">
    <Setter TargetName="grdMain" Property="Opacity"
     Value="0.75"/>
</Trigger>

<!--  Focus.  -->
<Trigger Property="IsFocused" Value="True">
    <Setter TargetName="grdMain" Property="Opacity"
     Value="0.75"/>
</Trigger>

<!--  Defaulted.  -->
<Trigger Property="IsDefaulted" Value="True">
    <Setter TargetName="rectMain" Property="Style"
     Value="{StaticResource styIsDefaulted}"/>
</Trigger>

<!--  Pressed. This comes after Focus so it gets precedence.  -->
<Trigger Property="IsPressed" Value="True">
    <Setter TargetName="grdMain" Property="Opacity"
     Value="1"/>
</Trigger>

<!--  Disabled. This comes last so it gets ultimate precedence.  -->
<Trigger Property="IsEnabled" Value="False">
    <Setter TargetName="rectMain" Property="Style"
     Value="{StaticResource styDisabled}"/>
</Trigger>
```

GlassButton

When the control's `IsMouseOver` property is `True`, the first trigger sets the `Grid`'s `Opacity` property to 0.75. This is more opaque than the original value of 0.5, so the control becomes more solid.

When the control receives the focus, the second trigger also sets the `Grid`'s `Opacity` property to 0.75.

When the control's `IsDefaulted` property is `True`, the next trigger sets the `Rectangle`'s `Style` to the "defaulted" `Style`.

When the `IsPressed` property is `True`, the following trigger sets the `Grid`'s `Opacity` to 1, making it fully opaque.

Finally, when the control's `IsEnabled` property is `False`, the last trigger sets the `Rectangle`'s `Style` to the "disabled" `Style`. The `Button` control automatically stops interacting with the user, so you don't need to worry about that.

IMPORTANT ORDER

Notice that the order of the template's triggers is important. Triggers that are defined later are applied later — if two triggers are active at the same time, the second trigger overrides the first.

In this example, the `IsPressed` trigger must come after the `IsMouseOver` trigger. Otherwise, when the user pressed the mouse on the `Button`, the `IsPressed` trigger would occur first. But at that point, since the mouse would be over the `Button`, the `IsMouseOver` trigger would also apply and would override the `IsPressed` trigger so that the user would never see the `Button` look pressed.

Similarly the `IsEnabled` trigger comes last so it overrides all other triggers. If the button is disabled, it should never display any of the other appearances.

ELLIPSE BUTTON

The EllipseButton example program shown in Figure 15-9 uses a template to make an elliptical button that is very different from the glass button described in the preceding section.

FIGURE 15-9

The disabled button on the left is paler than the others and doesn't respond to the user.

The defaulted button is brighter than the others and displays a yellow highlight along its border.

Figure 15-10 shows the program when the mouse is over Button 3. The button under the mouse is even brighter than the defaulted button and displays a yellow glow under its text.

FIGURE 15-10

It's difficult to see, but the button under the mouse in Figure 15-10 also displays an extra white highlight in its border roughly above the number 3. Every second, that highlight makes a trip around the button's circumference to draw the user's attention to the button. It's a small highlight, so the effect is fairly subtle.

ANIMATION ADVICE

Motion is one of the most attention-grabbing effects you can add to a program, but it can also be the most distracting and annoying. A button that flashes bright colors or continually changes size while the mouse is over it would really annoy users. The moving highlight that the EllipseButton program displays is subtle so the effect isn't too bad, but be careful. Keep animations like this one subtle or make them play only once — for example, when the mouse first enters the button, so you don't drive your users crazy. In extreme cases, rapidly flashing lights can even induce seizures in some people so don't use areas that flash brightly, particularly at frequencies between 2 and 55 Hz. Better still, give users a way to disable these sorts of animations.

Also note that users with special needs such as color vision deficiency or visual impairment may not see subtle animations very well or at all. Don't rely solely on subtle animations to give the user information.

Figure 15-11 shows the program when the user presses the mouse down on a button.

FIGURE 15-11

When you press a button, its background shifts slightly, and it displays a larger glow under its text.

If you drag the mouse off the button while it is still pressed, the button returns to its focused "mouse over" appearance. If you then release the mouse, no mouse click occurs.

The following two sections describe the ellipse button's `Template`. The first section describes the `Template` at a high level, explaining the controls the `Template` uses and how they fit together. The next section describes the `Template`'s `Triggers`.

> *This program is fairly long so the complete code isn't shown in these sections. You can download the example program from the book's web site to see the details.*

Ellipse Button Controls

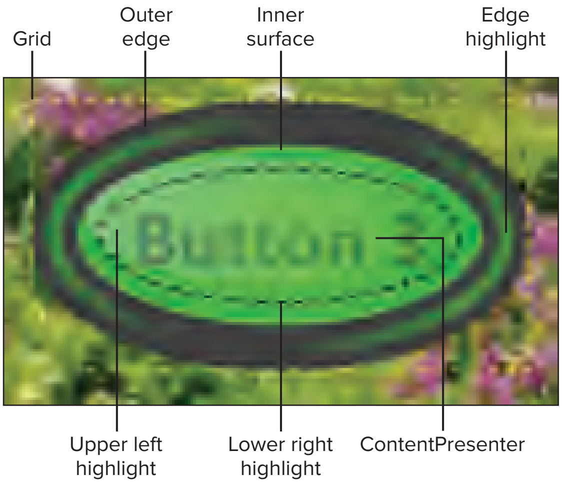


FIGURE 15-12

Figure 15-12 shows the template's control structure. The controls' labels are shown in left-to-right and top-to-bottom order, so you can tell which controls are defined by the XAML code before the others. For example, the "Inner surface" is defined in the XAML code before the "Outer edge."

A `Grid` (shown as a yellow dashed box) contains all of the other controls, most of which are `Ellipses`.

The inner surface is an ellipse filled with a brush that shades from lime to green. It lies beneath all of the other visible controls.

The outer edge is an ellipse with a transparent center and a black edge. Because its `StrokeThickness` is 10, it forms a wide band around the control.

The edge highlight is an ellipse with `Margin = 4` and `StrokeThickness = 4`, forming a band within the outer edge. It has a transparent center. Its `Stroke` property is a gradient brush that shades from lime on the left, to transparent in the middle, to lime again on the right; thus this ellipse makes two highlights on the edge. Its `BitmapEffect` property is set to a `BlurBitmapEffect` object, so it's fuzzy, giving the edge a rounded, 3D appearance.

The sparkle highlight isn't shown in Figure 15-12. It is similar to the green edge highlight except that it's white and only has one visible piece instead of two. It is only visible when the mouse is over the control and it is animated.

The upper-left highlight is an ellipse filled with a brush that shades from white to transparent. Since its `Margin` property is set to `12,12,20,20`, it is offset a bit toward the upper left.

The lower-right highlight is shown in Figure 15-12 as a dashed ellipse because it has `Opacity = 0`, making in invisible. It is displayed when the user presses the button. If you look closely, you can see it in Figure 15-11. Like the upper-left highlight, this ellipse shades from white to transparent but is centered with a `Margin` value of 15.

The `ContentPresenter` is centered in the `Template`'s `Grid`.

The final `Template` control is a light gray `Ellipse` that covers everything. Its `Opacity` is 0.3 and thus it tones down the colors of all of the other controls.

Many of the controls have `Opacity` less than 1, so they are semitransparent. When the control changes state — for example, when the mouse is over the button or the user presses the button — the template's `Triggers` change the `Opacity` of the controls to give the `Button` a different appearance.

Ellipse Button Triggers

When events occur, the template's triggers make appropriate changes to the control's appearance. Mostly these changes involve changing `Opacity` values to make some controls more visible while hiding others.

The `IsMouseOver` property trigger shown in the following code is the most interesting of the template's triggers:

Available for download on Wrox.com

```
<Trigger Property="IsMouseOver" Value="True">
    <Setter TargetName="ellUpperLeftHighlight" Property="Opacity" Value="1"/>
    <Setter TargetName="ellCover" Property="Opacity" Value="0"/>
    <Setter TargetName="cpContent" Property="Opacity" Value="1"/>
    <Setter TargetName="cpContent" Property="BitmapEffect"
     Value="{StaticResource bmeMouseOver}"/>
    <Setter TargetName="ellSparkle" Property="Opacity" Value="0.75"/>

    <!--  Start the sparkle animation.  -->
    <Trigger.EnterActions>
        <BeginStoryboard Name="begSparkle">
            <Storyboard BeginTime="0:0:1" RepeatBehavior="Forever" >
                <DoubleAnimationUsingKeyFrames
                 Duration="0:0:2"
                 Storyboard.TargetName="transSparkle"
                 Storyboard.TargetProperty="Angle">
                    <LinearDoubleKeyFrame
                     Value="0" KeyTime="0:0:0"/>
                    <LinearDoubleKeyFrame
                     Value="360" KeyTime="0:0:1"/>
                    <LinearDoubleKeyFrame
                     Value="360" KeyTime="0:0:2"/>
                </DoubleAnimationUsingKeyFrames>
            </Storyboard>
        </BeginStoryboard>
    </Trigger.EnterActions>

    <!--  Stop the sparkle animation.  -->
    <Trigger.ExitActions>
        <StopStoryboard BeginStoryboardName="begSparkle"/>
    </Trigger.ExitActions>
</Trigger>
```

EllipseButton

This code uses simple setters to do the following immediately when it starts:

- Make the upper-left highlight fully opaque instead of translucent.
- Make the light gray cover that tones down the other controls transparent so that all of the other controls have their full brightness.
- Make the `ContentPresenter` fully opaque instead of translucent.

- Give the `ContentPresenter` an `OuterGlowBitmapEffect` (defined in the template's Resources section).
- Make the white sparkle highlight visible with `Opacity = 0.75`. This makes the left edge highlight brighter than the right edge highlight and prepares the sparkle for the animation described next.

The `Trigger`'s `EnterActions` occur when the `IsMouseOver` property becomes `True`. This code begins a `Storyboard` that uses a `DoubleAnimationUsingKeyFrames` object to animate the sparkle highlight's brush. The brush has a `RotateTransform` named `transSparkle` that initially has `Angle = 0`, so the brush is not rotated. The animation makes `Angle` sweep from 0 to 360 degrees over a 1-second period. It holds `Angle` at 360 degrees for another second to make the animation pause. The `Storyboard` then repeats indefinitely.

The `Trigger`'s `ExitActions` occur when the `IsMouseOver` property becomes no longer `True` (in other words, becomes `False`). When that happens, the code stops the `Storyboard` that animates the sparkle brush.

The `Template`'s other triggers are much simpler. For example, the following code shows the `IsPressed` property trigger that executes when the user presses the button:

Available for download on Wrox.com

```
<Trigger Property="IsPressed" Value="True">
    <Setter TargetName="ellUpperLeftHighlight" Property="Opacity" Value="0"/>
    <Setter TargetName="ellLowerRightHighlight" Property="Opacity" Value="0.75"/>
    <Setter TargetName="cpContent" Property="BitmapEffect"
     Value="{StaticResource bmePressed}"/>
</Trigger>
```

EllipseButton

This code uses simple setters to do the following:

- Hide the upper-left highlight by setting `Opacity = 0`.
- Display the lower-right highlight by setting `Opacity = 0.75`.
- Give the `ContentPresenter` the `OuterGlowBitmapEffect` named `bmePressed`. This effect, which is defined in the template's Resources section, uses a larger `GlowSize` than the `bmeMouseOver` effect, so the glow behind the `ContentPresenter` is larger. You can see the difference if you carefully compare Figures 15-10 and 15-11.

The template's other triggers shown in the following code work similarly:

Available for download on Wrox.com

```
<!-- Defaulted. -->
<Trigger Property="IsDefaulted" Value="True">
    <Setter TargetName="ellCover" Property="Opacity" Value="0.15"/>
    <Setter TargetName="ellEdgeHighlight" Property="Stroke"
     Value="{StaticResource brDefaultedEdgeHighlight}"/>
</Trigger>

<!-- Not defaulted. -->
```

```
<Trigger Property="IsDefaulted" Value="False">
    <Setter TargetName="ellEdgeHighlight" Property="Stroke"
     Value="{StaticResource brEdgeHighlight}"/>
</Trigger>

<!--  Disabled. This comes last so it gets ultimate precedence.  -->
<Trigger Property="IsEnabled" Value="False">
    <Setter TargetName="ellCover" Property="Opacity" Value="0.6"/>
</Trigger>
```

EllipseButton

These triggers are fairly straightforward, giving controls new `Stroke` values and shuffling around `Opacity` values.

TEMPLATE TRICKS 3

The EllipseButton example demonstrates several useful template tricks including:

- Using `BlurBitmapEffect` to make an object appear three-dimensional (the outer edge with the blurred edge highlight on it)
- Using `Opacity` <1 to make translucent highlights
- Using brushes that blend from a color to transparent to make highlights fade away
- Using brushes that blend from a color to transparent and back to a color to make objects that are visible in multiple places
- Using different brushes for different control states (the normal vs. defaulted highlights)
- Using different `BitmapEffects` for different control states (the smaller glow for `IsMouseOver` and the larger glow for `IsPressed`)
- Animating changes to a brush's transformation

RESEARCHING CONTROL TEMPLATES

To effectively build templates, you need to learn what behaviors the control provides for you and what behaviors you need to provide for it. You also need to determine what events the control provides so that you know when you have a chance to make the control take action.

For example, WPF provides a confusing assortment of mouse events including `Mouse.MouseEnter`, `IsMouseOver`, `MouseLeftButtonDown`, `Pressed`, and `Click`. If you're trying to write a `Button` template, which mouse events can you use to change the `Button`'s appearance? Which properties and behaviors does the `Button` provide for you, and which do you need to implement?

The `Button` templates described in the previous sections use the `Button`'s `IsMouseOver`, `IsPressed`, `IsEnabled`, `IsFocused`, and `IsDefaulted` properties. The `Button` class provides these no matter what controls you add to the template to provide basic `Button` behavior.

As described in the "Template Binding" section earlier in this chapter, templates can also read some of the property values provided by the underlying control. For example, the `Button` class provides `Background`, `BorderBrush`, and `BorderThickness` properties that a template can read by using template bindings. `Button` also inherits properties such as `Width` and `Height` that you can also read with template bindings.

So, how do you learn what properties and template bindings are available?

One good source of information is Microsoft's "Control Styles and Templates" web page at `msdn.microsoft.com/cc278075(VS.95).aspx`. That page provides links to other pages that describe the features available to different control templates.

For example, the "Button Styles and Templates" page lists the Button's states and properties and tells where it gets them. For instance, the `Pressed` state (which you can read with the `IsPressed` property) tells when the button is pressed.

These web pages also show the default templates used by the controls. The `Button` control's default template is 84 lines long and fairly complicated. Some are much longer and much more complex.

In addition to using Microsoft's web pages, you can make a control tell you about its template. The ShowTemplate example program shown in Figure 15-13 displays the default template for a control. When you click on the "Show Template" button, the program displays the default template for the control named `Target`. In Figure 15-13, that control is the `Slider` in the upper-left corner. To see the template used by a different kind of control, replace the `Slider` with a different control, name it `Target`, and run the program.

FIGURE 15-13

The ShowTemplate program uses the following code to display the `Target` control's template:

Available for download on Wrox.com

```
private void btnShowTemplate_Click(object sender, RoutedEventArgs e)
{
    XmlWriterSettings writer_settings = new XmlWriterSettings();
    writer_settings.Indent = true;
    writer_settings.IndentChars = "    ";
    writer_settings.NewLineOnAttributes = true;

    StringBuilder sb = new StringBuilder();
    XmlWriter xml_writer = XmlWriter.Create(sb, writer_settings);

    XamlWriter.Save(Target.Template, xml_writer);

    txtResult.Text = sb.ToString();
}
```

ShowTemplate

The key to this code is the `XamlWriter` class, which includes methods that extract XAML from a WPF object such as a control or template.

The code starts by initializing an `XmlWriterSettings` object to make the writer produce nicely formatted output. If you don't do this, the code comes out in one long line of XML without carriage returns or indentation.

The program then creates a `StringBuilder` to hold the result text. It uses the writer settings to create an `XmlWriter` attached to the `StringBuilder`.

The code then calls the `XamlWriter` class's static `Save` method to write a XAML representation of the `Target` control's `Template` property into the `StringBuilder`.

The program finishes by displaying the result in its `TextBox txtResult`.

After you find a control's default template, you can modify it to make your own template that changes the control's appearance. That lets you ensure that your template behaves the same way the default template does except in those places where you want changes.

SUMMARY

Properties and styles let you change a control's appearance in superficial ways. Templates let you change a control more fundamentally, altering the pieces that make up the control and changing the way it responds to events and changes in property values. By using templates, you can give your applications a distinctive look and feel.

The next chapter explains two topics closely related to styles and templates: themes and skins. Skins let an application change its entire appearance, sometimes radically. They let you change the application to suit your immediate need or even your mood.

Themes provide a unifying appearance across controls, windows, and even separate applications. By providing a common look and feel, themes can make even unrelated programs seem to fit together in a single system.

16

Themes and Skins

WPF lets you build applications that have engaging, distinctive appearances. By using relatively simple techniques such as drop shadows, partial transparency, and transparency masks, you can make an eye-catching interface that adds interest and excitement to even the most routine application.

Properties let you change the appearance of controls. They let you change visual characteristics such as a control's colors, size, location, and contents. Resources and styles let you package those changes so that you can easily apply them to many controls simultaneously.

Templates let you alter the way controls behave by changing the pieces that make up the controls. They let you change the appearance and behavior of `Buttons`, `ListBoxes`, `Menus`, and other controls in fundamental ways while still allowing them to perform their essential functions.

Themes and skins bring all of these ideas together to let you easily change the appearance and behavior of an entire application to suit the users' needs and moods.

THEMES

Some developers use the terms *theme* and *skin* interchangeably, but I make the distinction that a *skin* applies to a single application (or part of an application), and a *theme* applies to more than one application.

More precisely, a *theme* is a unifying plan that helps determine the appearance and behavior of more than one application. The most common themes are those provided by Windows. For example, Windows 7 provides the themes:

- Aero
- Classic
- Luna (Homestead, Metallic, and Normal versions)
- Royale

Using the System Theme

The ShowThemes example program shown in Figure 16-1 displays controls that use each of the themes that come with Windows 7. The differences are fairly subtle, so you'll need to look closely to see the changes in each theme.

FIGURE 16-1

If you don't use properties, styles, or templates to change a control's appearance or behaviors, it uses the values defined by the system's current theme. In Figure 16-1, you can see that the controls in the "default" group have the same appearance as those that use the Aero theme. This is because the system had the Aero theme selected when I ran the program.

To change the system's theme in Windows 7, open the Control Panel. Under "Appearance and Personalization" click "Change the theme." On the dialog shown in Figure 16-2, click the theme that you want to use.

FIGURE 16-2

After taking the screenshot shown in Figure 16-1, I followed these steps to change the system's theme to Windows Classic without closing the ShowThemes program. The program automatically detected the change in the system theme and redrew itself appropriately.

Figure 16-3 shows the result. If you look closely, you'll see that the controls in the default group now match those that use the Classic theme.

FIGURE 16-3

If you compare Figures 16-1 and 16-3 very closely, you'll also see that the controls in the ShowThemes program look a little different. In Figure 16-3, the window's upper corners are not as rounded, the title bar is darker, the border is no longer shaded with a light blue pattern, and the system icons in the form's upper-left and upper-right corners are different.

Using a Specific Theme

Normally you don't need to even think about themes. If you leave a control's appearance alone, it will automatically use the system's current theme and even change its appearance if you change the theme. If you really want to, however, you can select a specific theme.

> **CHANGING THEMES**
>
> This technique is probably more useful for testing an application to see what it looks like in a particular theme than it is for actually forcing the theme on the users.
>
> Some users may select a particular theme for a good reason. For example, a visually impaired user may select a high-contrast theme to make programs easier to see. If you change the theme, that user may be unable to use your application.
>
> If you don't really *need* a specific theme, you should let your program use the default.

To use a specific system theme in Visual Studio, begin a new WPF project. Open the Project menu and select "Add Reference." On the .NET tab, select the theme(s) that you want to use and click OK. Figure 16-4 shows the Add Reference dialog with the Aero, Classic, Luna, and Royale themes selected.

Next, in a resource dictionary, use a `MergedDictionaries` object to load the theme. The theme will apply to any controls that should be modified by the dictionary.

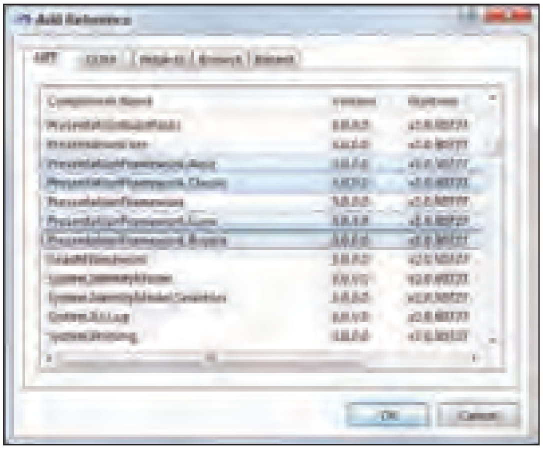


FIGURE 16-4

For example, the following code shows how the ShowThemes program uses the Luna Metallic theme. Each group of controls shown in Figures 16-1 and 16-3 is contained in a `StackPanel`. Each `StackPanel` has a `ResourceDictionary` that loads its theme.

Available for download on Wrox.com

```
<StackPanel>
    <StackPanel.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source=
"/PresentationFramework.Luna;component/themes/luna.metallic.xaml" />
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </StackPanel.Resources>
    <Label Style="{StaticResource lblStyle}" Content="luna.metallic"/>
    <Button Margin="10" Content="Click Me"/>
    <CheckBox Margin="10" Content="Check Me"/>
    <RadioButton Margin="10" Content="Press Me"/>
</StackPanel>
```

ShowThemes

Notice the unusual syntax for the `ResourceDictionary`'s `Source` property. The PresentationFramework.Luna piece tells WPF which library contains the theme, and the rest of the `Source` gives the name of the theme within the library.

For more information on Microsoft's standard themes, go to `msdn.microsoft.com/aa358533.aspx`. Links at the bottom of the web page lead to pages about the specific themes Aero, Classic, Luna, and Royale. From those pages, you can download XAML files that show how the themes are defined. You can then modify those files to build your own theme files.

THEME RESTRICTIONS

I have had bad luck getting Expression Blend to use specific themes. It seems to have trouble finding the DLLs for use by the `ResourceDictionary`'s `Source` property. I've also had bad luck getting the compiled executable to run.

Perhaps these issues will be fixed in a later release, but for now I use this technique only to see what the program will look like in different themes in programs built with Visual Studio. If you figure out how to get these working in Expression Blend, e-mail me at `RodStephens@vb-helper.com` and I'll post your solutions on the book's web page.

SKINS

Themes let a program automatically change to match the rest of the system's appearance. Selecting a specific theme lets a program change its appearance deliberately, but that's generally not necessary. The differences between the Luna Metallic and Aero Normal themes are so small that there's little reason to force the user to see one or the other when you could let the program use the system's default theme.

Skins are much more interesting. A *skin* is a packaged set of appearances and behaviors that can give an application (or part of an application) a distinctive appearance while still allowing it to provide its key features.

Skins are somewhat similar to themes in the sense that they define the appearance and behavior of an application, but they generally make much larger changes in the application's appearance than those shown in Figures 16-1 and 16-3. Rather than unifying all of the applications running on a system, the larger changes provided by skins can make an application stand out. A skin differentiates the application and makes it easier for the user to tell applications apart at a glance.

For example, Figures 16-5 and 16-6 show the ResourceDictionaries example program (which is described in Chapter 12) displaying two very different appearances. It's the same program in both figures and it contains the same controls — just rearranged slightly and with different colors, fonts, and so forth.

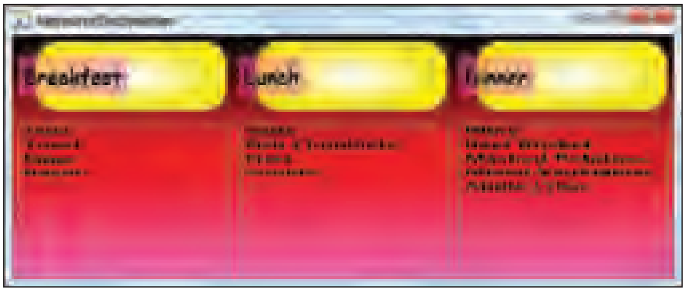


FIGURE 16-5

FIGURE 16-6

The following sections describe skins and explain several ways you can implement them in WPF.

> **HARD WORK WARNING**
>
> Be warned that skinning takes a lot of work! Depending on the technique you use, it may not be very complicated work, but it can be very time-consuming.
>
> WPF provides so many tools for creating attractive user interfaces that it's easy to spend hours fiddling with control properties and arrangements, trying to build the world's most beautiful interface. Now multiply that effort to provide multiple skins, and you could end up spending days on a window instead of "only" hours.

Skin Purposes

Usually skins are mostly decorative, changing the application's colors, button shapes, form designs, background images, and so forth. The skins shown in Figures 16-5 and 16-6 look very different but only superficially. They still use the same controls in roughly the same positions.

Although skins are often decorative and used to increase a program's "coolness factor," multiple skins can have legitimate business purposes.

For example, in the United States, roughly 8 percent of men and 0.4 percent of women have some form of color vision deficiency and thus have trouble distinguishing among certain colors. If your application provides multiple skins, users can change the colors or shapes used by the program so they have less trouble getting the information they need.

In addition, as the general user population ages, applications must be ready to help older users. Larger fonts, menus, buttons, and other components can make understanding and using an application easier for users. Providing multiple skins with different element sizes also allows users with larger screens to take advantage of the space they have available.

One use for skins that is usually overlooked is to make different interfaces so you can use the same application for different purposes. For example, suppose you're writing an order entry system. Different kinds of users would need to see different pieces of an order at different times.

When an order is initially created, the order entry clerk needs to know all about the customer and order, and possibly payment information (depending on your arrangements with the customers). Later, the shipping clerk who packages up the customer's order only needs to know about the items ordered and the customer's shipping address, not payment information or previous order history. The program might automatically send the customer an invoice, but if the customer calls with a question, a billing clerk may need to know about the customer's payment method and possibly past orders.

You can use different skins to satisfy the needs of these different users. The order entry clerk's skin would let the user locate customer data and enter information about a new order. The shipping clerk's skin would display information about the current order and the customer's shipping address while hiding payment information and previous order history.

The OrderTracking example program displays four different interfaces for different kinds of users. Figure 16-7 shows the program's four skins for managers, billing clerks, order entry clerks, and shipping clerks.

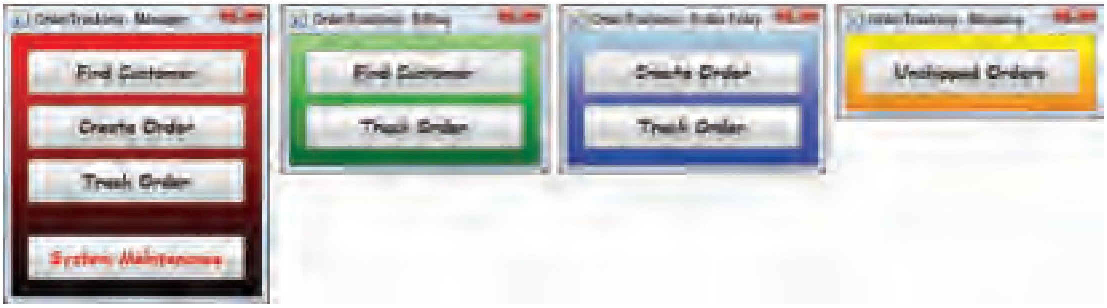


FIGURE 16-7

The following XAML code shows how the OrderTracking program works:

Available for download on Wrox.com

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="Window1"
    x:Name="Window"
    SizeToContent="WidthAndHeight"
    Width="300" Height="360"
    ResizeMode="NoResize">

    <!--  Load skin resources  -->
    <Window.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="resBasics.xaml"/>
                <!--
                <ResourceDictionary Source="resBillingClerk.xaml"/>
                <ResourceDictionary Source="resOrderEntry.xaml"/>
                <ResourceDictionary Source="resShippingClerk.xaml"/>
                 -->
                <ResourceDictionary Source="resManager.xaml"/>
```

```
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Window.Resources>

    <!--  Set window properties from resources  -->
    <Window.Background>
        <StaticResource ResourceKey="brWindow"/>
    </Window.Background>
    <Window.FontFamily>
        <StaticResource ResourceKey="ffWindow"/>
    </Window.FontFamily>
    <Window.FontSize>
        <StaticResource ResourceKey="fsWindow"/>
    </Window.FontSize>
    <Window.FontWeight>
        <StaticResource ResourceKey="fwWindow"/>
    </Window.FontWeight>
    <Window.Title>
        <StaticResource ResourceKey="txtTitle"/>
    </Window.Title>

    <StackPanel Margin="10">
        <Button Content="Unshipped Orders" Click="btnUnshippedOrders_Click"
         Visibility="{StaticResource visUnshippedOrder}"/>
        <Button Content="Find Customer" Click="btnFindCustomer_Click"
         Visibility="{StaticResource visFindCustomer}"/>
        <Button Content="New Order" Click="btnNewOrder_Click"
         Visibility="{StaticResource visCreateOrder}"/>
        <Button Content="Track Order" Click="btnTrackOrder_Click"
         Visibility="{StaticResource visTrackOrder}"/>

        <Label Height="30"
         Visibility="{StaticResource visSystemMaintenance}"/>
        <Button Content="System Maintenance" Click="btnSystemMaintenance_Click"
         Foreground="Red" Height="40"
         Visibility="{StaticResource visSystemMaintenance}"/>
    </StackPanel>
</Window>
```

OrderTracking

The program begins by defining `Window` attributes. Setting the `SizeToContent` attribute to `WidthAndHeight` makes the window automatically resize itself to fit its content so the window is an appropriate size no matter which skin it is using. The code also sets the `ResizeMode` attribute to `NoResize` so the window stays that size.

Next, the code loads its resource dictionaries. The first one, resBasics.xaml, contains values that are the same for every skin. It defines the window's font properties and contains an unnamed `Button` style that sets the `Button` sizes and margins.

After that, the code includes the resource dictionary for the skin it should display. The previous code includes the resource file for managers, resManager.xaml, and the other resource files are commented out.

The code then uses resource properties to set the window's background brush, font, and title. Giving the skins different backgrounds and titles makes it easier to tell the skins apart at a glance.

Next, the code defines a `StackPanel` containing a series of `Buttons`. The `Buttons`' `Visibility` properties are set using resources defined in the skin resource dictionaries. The basic dictionary resBasics.xaml sets `Visibility = Collapsed` for all of the buttons. The other dictionaries override those settings to display the appropriate buttons. For example, in the Order Entry dictionary, resOrderEntry.xaml, the values `visCreateOrder` and `visTrackOrder` are set to `Visible` so the "Create Order" and "Track Order" buttons are shown in the order entry skin.

The following code shows the resBasics.xaml resource dictionary that defines common values for all of the skins. It defines the `Window`'s font characteristics and the `Button` style. It also hides all of the `Buttons`.

Available for download on Wrox.com

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib">

    <!-- Main window -->
    <FontFamily x:Key="ffWindow">Comic Sans MS</FontFamily>
    <FontWeight x:Key="fwWindow">Bold</FontWeight>
    <sys:Double x:Key="fsWindow">18</sys:Double>
    <sys:String x:Key="txtTitle">OrderTracking</sys:String>

    <!-- Buttons style -->
    <Style TargetType="Button">
        <Setter Property="Width" Value="200"/>
        <Setter Property="Height" Value="50"/>
        <Setter Property="Margin" Value="10"/>
    </Style>

    <!-- Button visibilities -->
    <Visibility x:Key="visUnshippedOrders">Collapsed</Visibility>
    <Visibility x:Key="visCreateOrder">Collapsed</Visibility>
    <Visibility x:Key="visFindCustomer">Collapsed</Visibility>
    <Visibility x:Key="visTrackOrder">Collapsed</Visibility>
    <Visibility x:Key="visSystemMaintenance">Collapsed</Visibility>
</ResourceDictionary>
```

OrderTracking

The following code shows the resOrderEntry.xaml skin resource dictionary. It defines the `Window`'s background brush and title, and the `Button` visibilities for the order entry skin.

Available for download on Wrox.com

```
<ResourceDictionary
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:sys="clr-namespace:System;assembly=mscorlib">

    <!-- Main window -->
    <LinearGradientBrush x:Key="brWindow" StartPoint="0,0" EndPoint="0,1">
        <GradientStop Color="LightBlue" Offset="0"/>
        <GradientStop Color="Blue" Offset="1"/>
    </LinearGradientBrush>
```

```
        <sys:String x:Key="txtTitle">OrderTracking - Order Entry</sys:String>

        <!--  Button visibilities  -->
        <Visibility x:Key="visCreateOrder">Visible</Visibility>
        <Visibility x:Key="visTrackOrder">Visible</Visibility>
    </ResourceDictionary>
```

OrderTracking

To use the OrderTracking program, you would compile the program and save the executable program. Then you would change the included resource dictionary to load a different skin, recompile the program, and save the new executable. You would repeat the process until you had created an appropriate executable for each type of user.

Rather than creating separate versions of the program for each type of user, you could load the appropriate skin at run time. The following sections describe three ways you can build skinnable applications in WPF, all of which let the program change its skin at run time.

Resource Skins

The program ResourceDictionaries shown in Figures 16-5 and 16-6 uses two different sets of resources to change its appearance at design time.

The following code shows the `Window`'s resource dictionary. The inner `ResourceDictionary` elements load two different resource dictionaries. Because dictionaries loaded later override those loaded earlier, you can change the application's appearance by changing the order of these two elements. (As shown here, the RedRed.xaml dictionary is loaded second, so the program uses its red interface, shown in Figure 16-5.)

Available for download on Wrox.com

```
    <Window.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="ResBlue.xaml"/>
                <ResourceDictionary Source="ResRed.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Window.Resources>
```

ResourceDictionary

While the program ResourceDictionaries can display two different appearances, you need to modify the program at design time to pick the skin you want. This may be useful for building different interfaces for different kinds of users, but a truly skinnable program should allow the user to change skins at run time.

To turn this into a truly skinnable application, all you need to do is give the program the ability to change skins at run time.

The Skins example program is very similar to the program ResourceDictionaries except that it can change skins at run time. To make that possible, most of its resources are dynamic rather than static. The program also contains two new user interface elements: an `Image` and a `Label`.

> **RESTRICTED SKINS**
>
> If you use different skins for different kinds of users (e.g., the order entry clerk and shipping clerk described in the previous section), then you'll need to restrict the skins that each user can load. For example, you probably wouldn't want the shipping clerk to be able to load the billing clerk's skin and view the customer's credit card information.

When it displays its red interface, this program adds a small `Image` in its upper-right corner. This `Image` has a context menu that displays the choices Red and Blue (shown on the left in Figure 16-8), which let you pick between the red and blue skins.

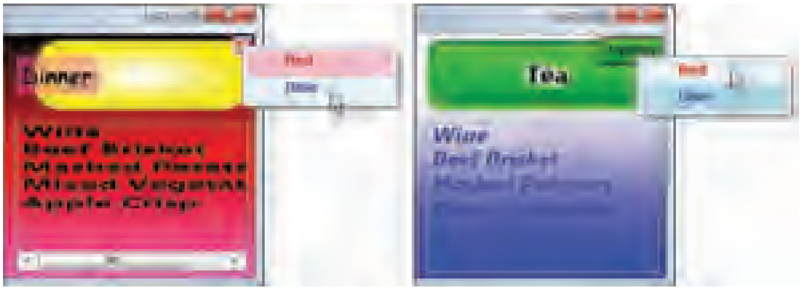


FIGURE 16-8

The program's blue interface displays a label in its upper-right corner (on the right in Figure 16-8) that displays the same context menu.

The following code shows how the program displays its Options textbox on the blue interface:

Available for download on Wrox.com

```
<Label MouseDown="Options_MouseDown"
 Grid.Row="0" Grid.Column="2" Margin="2"
 Content="Options" FontSize="10"
 HorizontalAlignment="Right" VerticalAlignment="Top"
 Foreground="Black" BorderBrush="Black"
 BorderThickness="1"
 Visibility="{DynamicResource visBlue}"
>
    <Label.Background>
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Color="Lime" Offset="0"/>
            <GradientStop Color="Green" Offset="1"/>
        </LinearGradientBrush>
    </Label.Background>
    <Label.ContextMenu>
        <ContextMenu Name="ctxOptions">
            <MenuItem Header="Red" Background="Pink"
             Foreground="Red"
             Click="ctxSkin_Click" Tag="ResRed.xaml"/>
            <MenuItem Header="Blue" Background="LightBlue"
```

```
                Foreground="Blue"
                Click="ctxSkin_Click" Tag="ResBlue.xaml"/>
            </ContextMenu>
        </Label.ContextMenu>
    </Label>
```

Skins

This code contains four real points of interest:

1. First, the `Label`'s `MouseDown` event triggers the `Options_MouseDown` event handler. This routine, which is shown in the following code, displays the context menu by setting the `ContextMenu`'s `IsOpen` property to `True`:

Available for download on Wrox.com

```
// Display the options context menu.
private void Options_MouseDown(object sender, RoutedEventArgs e)
{
    ctxOptions.IsOpen = true;
}
```

Skins

2. Second, the `Label`'s `Visibility` property is set to the value of the `visBlue` static resource. This resource has the value `True` in the Blue resource dictionary and `False` in the Red resource dictionary. This means that the `Label` is visible only in the blue interface. The red interface's skin-changing `Image` uses a similar `visRed` resource that is only `True` in the red interface.
3. Third, the context menu's items have a `Tag` property that names the XAML resource file that they load. For example, the Blue menu item has its `Tag` set to `ResBlue.xaml`. The program uses the `Tag` property to figure out which file to load when the user picks a menu item.
4. Finally, both of the context menu's items fire the `ctxSkin_Click` event handler shown in the following code to load the appropriate skin:

Available for download on Wrox.com

```
// Use the selected skin.
private void ctxSkin_Click(object sender, RoutedEventArgs e)
{
    // Get the context menu item that was clicked.
    MenuItem menu_item = (MenuItem)sender;

    // Create a new resource dictionary, using the
    // menu item's Tag property as the dictionary URI.
    ResourceDictionary dict = new ResourceDictionary();
    dict.Source = new Uri((String)menu_item.Tag, UriKind.Relative);

    // Remove all but the first dictionary.
    while (App.Current.Resources.MergedDictionaries.Count > 1)
    {
        App.Current.Resources.MergedDictionaries.RemoveAt(1);
    }

    // Install the new dictionary.
    App.Current.Resources.MergedDictionaries.Add(dict);
}
```

Skins

This code gets the menu item that triggered the event and looks at the item's `Tag` property to see which resource file to load. It creates a `ResourceDictionary` object loaded from the file, removes old resource dictionaries from the application's MergedDictionaries collection, and adds the new dictionary.

> **REMOVED RESOURCES REDUX**
>
> The program doesn't remove the first resource dictionary so that WPF doesn't get confused about missing resources and issue a flock of warnings. For more information, see the note "Removed Resources" in the "Dynamic Resources" section of Chapter 12.

When the program loads the new resource dictionary, WPF detects the changed values and updates all of the window's dynamic resources.

This technique is what most developers think of as *skinning* in WPF applications: The program loads multiple resource files at run time to provide different skins.

Animated Skins

The skins described in the previous section use separate resource dictionaries to provide different appearances. The program's XAML file sets its control properties to resource values so that when you change the resource values, the interface changes accordingly.

Another way to change property values is to use property animation. Chapter 14 covers property animation in greater detail, but this section explains briefly how to use animation to provide skinning.

XAML files allow you to define triggers that launch storyboards that represent property animations. For example, when the user presses the mouse down over a rectangle, the XAML code can run a storyboard that varies the `Rectangle`'s `Width` property smoothly from 100 to 200 over a 1-second period.

The AnimatedSkins example program uses this technique to provide skinning. Figure 16-9 shows the program displaying its green skin. Figure 16-10 shows its blue skin.

FIGURE 16-9

FIGURE 16-10

When you click the appropriate control, a trigger launches a storyboard that:

- Resizes the main window and changes its `Background` brush.
- Hides and displays the small blue or green ellipses in the upper-right corner that you click to switch skins.
- Moves `Labels`.

- Resizes, moves, and changes the corner radii of the rectangles that act as buttons.
- Changes the `Fill` and `Stroke` brushes for the rectangles pretending to be buttons.
- Changes the text displayed in the `Labels`.
- Moves and resizes the `Image`.
- Changes the background and foreground colors.

Figure 16-11 shows the program a bit less than halfway done switching from the green to the blue skin. In this figure, you can see that the colors are moving from green to blue, the labels are moving, and the rectangle buttons have new positions, sizes, captions, and rounded corners.

In addition to displaying very different appearances, animated skins let the user watch as one interface morphs into another. The effect is extremely cool.

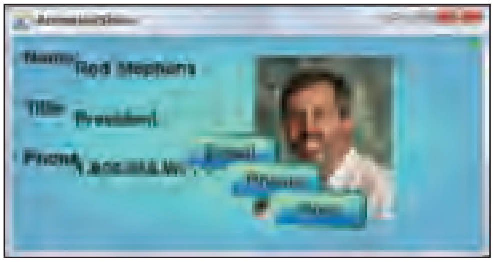


FIGURE 16-11

THE PRICE OF COOLNESS

You might argue that coolness isn't really the focus in many applications, and you would be completely correct, but programmers who write skins aren't usually focused on getting by with the least possible work. It's hard to argue that most skinning serves anything other than an aesthetic purpose, so as long as you're spending extra effort providing skins, it's not completely fair to say that the extra coolness of animated skins isn't worth the effort. By the same token, you could argue that you shouldn't even be using WPF and should stick with Windows Forms programming, which is generally easier.

That being said, however, be warned that animating skins is a *lot* of work. Tweaking the animations to give everything exactly the right position, size, and appearance takes time. The AnimatedSkins example program uses only two storyboards (one for each skin) but more than 100 property animations to get everything right. And the differences between these two skins aren't as great as some I've seen, so you could spend a huge amount of time getting everything just right.

One interesting side effect of this technique is that one animation doesn't need to finish before a new animation can start. For example, suppose you click on the blue circle in Figure 16-9 to switch to the blue skin. After the controls start moving to their new positions, you can click on the green circle shown in Figure 16-10. At that point, the controls immediately start moving back to their positions for the green skin without going all the way to their blue skin positions.

Dynamically Loaded Skins

One of the drawbacks of the previous two skinning techniques is that they only modify existing objects. They can display an `Ellipse`, `Button`, or `Label` with different properties, but they are still the same `Ellipse`, `Button`, or `Label`. For example, you cannot provide one skin that launches tasks with `Buttons`, another that uses `Menus`, and a third that uses `Labels`.

One common solution to this problem is to include every set of controls in every skin and then hide the ones that you don't need. For example, you would include `Buttons`, `Menus`, and `Labels` in every skin. Then the button-oriented skin would hide the `Menus` and `Labels`, the menu-oriented skin would hide the `Buttons` and `Labels`, and the label-oriented skin would hide the `Buttons` and `Menus`.

Another solution to this problem might be to use separate XAML files that sit on top of the same code-behind. Unfortunately, WPF doesn't handle this situation very well.

WPF provides methods for loading XAML files with or without event handlers attached. The short version of the story is, if you want to load XAML code with event handlers, then you can only have one XAML file associated with each code-behind class. If you load XAML code without event handlers, then you need to wire up the event handlers yourself.

THE LONGER STORY

If you want to load XAML files with event handlers, then you need to associate the XAML with a class defined in your code-behind. Unfortunately, WPF adds its own automatically generated bonus routines to perform some extra chores such as connecting the XAML events with the event handlers provided by your class. If you try to associate two XAML files with the same class, WPF creates multiple copies of those routines with the same signatures and that confuses Visual Studio.

You might try to make multiple classes for the XAML files by having them inherit from a common base class that provides all of the necessary functionality. Sadly, the automatically generated code makes your class inherit from a WPF control type. For example, if your XAML file contains a `Grid` as its root element, then the code makes your class inherit from the `Grid` class. That means that you cannot also make it inherit from your desired base class.

The only direct solution I've found is to make completely separate classes for each XAML file, but that kind of defeats the goal of trying to use common code-behind.

Wiring up events to event handlers isn't hard, although it does reduce the separation between user interface design and writing the code-behind. Now the interface designer and the programmer must agree on the event handlers that the code will use and on the names of the controls that use them.

INTERFACE IRONY

The difficulty of attaching multiple XAML files to the same code-behind seems somewhat ironic given how much emphasis WPF places on separation of user interface and code-behind. You can separate an interface from its code but only as long as you keep them logically associated with each other.

The SkinInterfaces example program displays new skins at run time by loading XAML files and wiring up their event handlers. Figures 16-12 and 16-13 show the program displaying its two skins.

These skins not only provide radically different appearances, but they also use different types of controls that generate different kinds of events. The following table lists the types of controls and events that each skin uses:

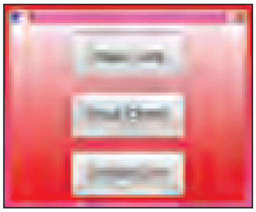


FIGURE 16-12

FIGURE 16-13

PURPOSE	RED SKIN		BLUE SKIN	
	CONTROL	EVENT	CONTROL	EVENT
Switch skin	`Polygon`	`MouseDown`	`Ellipse`	`MouseDown`
Move form	`Rectangle`	`MouseDown`	`Ellipse`	`MouseDown`
Exit	`Grid` (containing a `Rectangle` and a `TextBlock`)	`MouseDown`	`Grid` (containing an `Ellipse` and a `TextBlock`)	`MouseDown`
Repair disk	`Button`	`Click`	`Grid` (containing an `Ellipse` and a `TextBlock`)	`MouseDown`
Virus check	`Button`	`Click`	`Grid` (containing an `Ellipse` and a `TextBlock`)	`MouseDown`
Format disk	`Button`	`Click`	`Grid` (containing an `Ellipse` and a `TextBlock`)	`MouseDown`

When the program loads a XAML file, it looks through the new controls and attaches event handlers to those that need them.

To provide some separation between the XAML files and the code-behind, the program uses a separate group of routines to do the real work. Event handlers catch the control events and call the work routines to do all the interesting stuff.

The following code shows how the blue skin defines its red switch skin circle on the left at the form's top:

Available for download on Wrox.com

```
<Grid
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Tag="Blue"
    >

        Lots of code omitted ...

    <Ellipse Name="ellSkin" Width="10" Height="10"
     Canvas.Left="70" Canvas.Top="13"
     Cursor="Cross" ToolTip="Change Skin"
     Fill="HotPink" Stroke="{StaticResource brRedStroke}"
     StrokeThickness="2" Tag="Red.xaml"
    />

        Lots of code omitted ...

</Grid>
```

SkinInterfaces

The code fragment starts with a `Grid` control as its root element. (You can use other controls for the file's root, but `Grid` is convenient, partly because the Visual Studio WPF Window Designer can understand how to display the file if its root is a `Grid`.) The root element's `Tag` property is set to the name of the skin it represents, in this case, *Blue*.

The code shown here omits all of the other controls except the "switch skin" circle.

The most important pieces of the circle's definition are its name, *ellSkin*, and its `Tag`, `Red.xaml`. The `Tag` property tells the code-behind which XAML skin file to load when the circle is clicked.

Ignoring for the moment how this control is wired up to its event handler, the following code shows the event handler that the circle executes. Since this event handler is shared by this circle and the red skin's "change skin" polygon (the blue triangle in the upper-left corner), it's called `pgnSkin_MouseDown`.

ilable for nload on rox.com

```
private void pgnSkin_MouseDown(object sender, MouseButtonEventArgs e)
{
    FrameworkElement element = (FrameworkElement)sender;
    LoadSkin(element.Tag.ToString());
}
```

SkinInterfaces

This code gets the element that triggered the event (either the blue skin's `Ellipse` or the red skin's `Polygon`), reads that element's `Tag` property to see which XAML file to load, and passes the filename to the function `LoadSkin`.

The function `LoadSkin` uses the following code to load a XAML skin file. To save space, the code only shows a few of the statements that connect controls to their event handlers.

Available for download on Wrox.com

```
// Load the skin file and wire up event handlers.
private void LoadSkin(string skin_file)
{
    // Load the controls.
    FrameworkElement element =
        (FrameworkElement)Application.LoadComponent(
            new Uri(skin_file, UriKind.Relative));
    this.Content = element;

    // Wire up the event handlers.
    Button btn;
    Polygon pgn;
    Rectangle rect;
    Grid grd;
    Ellipse ell;

    switch (element.Tag.ToString())
    {
        case "Red":
            btn = (Button)element.FindName("btnRepairDisk");
            btn.Click += new RoutedEventHandler(btnRepairDisk_Click);

                Code for other controls omitted
            break;

        case "Blue":
                Lots of code omitted

            // Uses the same event handler as rectMove.
            ell = (Ellipse)element.FindName("ellMove");
            ell.MouseDown +=
                new System.Windows.Input.MouseButtonEventHandler(
                    rectMove_MouseDown);

            grd = (Grid)element.FindName("grdExit");
            grd.MouseDown +=
                new System.Windows.Input.MouseButtonEventHandler(
                    grdExit_MouseDown);
            break;
    }
}
```

SkinInterfaces

The code starts by using the WPF `LoadComponent` method to load the desired XAML skin file. It sets the `Window`'s main content element to the root loaded from the file so the new controls are displayed.

Next, the code checks the newly loaded root element's `Tag` property to see whether it is now displaying the red or the blue skin. Depending on which skin is loaded, the code looks for specific controls in the skin and connects their event handlers.

For example, if the red skin is visible, the code uses FindName to locate the btnRepairDisk Button and adds the btnRepairDisk_Click event handler to its Click event.

The previous code omits most of the code connecting controls to event handlers. It does, however, show how the code finds the ellSkin control (the "change skin" circle) and adds the pgnSkin_MouseDown event handler to its MouseDown event.

The code that wires up the controls that are not shown here is similar.

The program's final piece is in the Window's constructor, which is shown in the following code. After the Window is initialized, the code calls LoadSkin to start with the red skin.

Available for download on Wrox.com

```
public Window1()
{
    this.InitializeComponent();

    // Insert code required on object creation below this point.

    // Start with the red skin.
    LoadSkin("Red.xaml");
}
```

SkinInterfaces

That completes the program's circle of life. When the program starts, it calls LoadSkin to load the red skin. LoadSkin loads the controls and wires up their event handlers. In particular, it attaches an event handler to the pgnSkin "change skin" control's MouseDown event. When you click on the polygon, the pgnSkin_MouseDown event handler executes and calls LoadSkin to start the whole process over again.

Despite the extra code-behind that locates specific controls and attaches events to event handlers, this technique is reasonably straightforward. Wiring up the controls can be long, but it's easy to understand.

The skin files can set control properties directly instead of requiring that you use a huge number of dynamic resources, so the code is a lot simpler than it is when you use different resource dictionaries.

This method doesn't provide property animation, which makes it less cool, but it's much easier to implement.

Finally, this technique allows different skins to use different controls for similar purposes. It lets you make skins that launch actions from Buttons, MenuItems, MouseDown events, and pretty much any other event you might want to catch.

SUMMARY

This chapter explains themes and skins. Themes let every application on the user's computer provide a similar look and feel. Normally, you don't need to do anything to take advantage of themes. If you don't override the default appearance of controls, then they automatically match the system's currently selected theme and update themselves as needed when the theme changes.

Skins let you change an application's appearance and behavior, essentially letting you define a "mini-theme" for the application. The examples in this chapter show how to use different skins for different purposes, load skins at design time or run time, build animated skins, and load skins that may use completely different controls.

Chapters 12 through 16 cover topics that control the application's behavior and appearance. They explain how to use resources, styles, templates, triggers, and themes to give an application a distinctive and consistent look-and-feel. These techniques are useful for building any WPF application.

The chapters that follow turn to more specific topics that are not necessarily essential for every application. These chapters explain important techniques that you will find useful in many applications. For example, Chapter 17 explains one of the more basic needs of many applications: printing.

17

Printing

Although some programs never need to produce "hard" output, printing is an important part of many applications. Printing lets you produce a permanent physical record of your work. It lets you make reports to give to your boss, pamphlets and newsletters to give to your neighbors, and doodles to stick on your refrigerator.

> **AMAZING FACT**
>
> According to HP Communities' Print 2.0 Blog, people printed around 45 trillion pages in 2005. Of those, roughly 9 percent, or 4.05 trillion, were printed with digital printers. (I don't know how many were printed with WPF.)

WPF provides some remarkably powerful printing capabilities. One of the most impressive of these is the ability to produce transformed output with little or no loss of resolution. Because WPF's retained-mode graphics strategy uses objects to represent graphical output, those objects can generate output even after they are transformed. That means a printout can display even a small piece of output zoomed to a large scale but still at a high resolution.

Figure 17-1 shows two printouts of the same window at different scales. The one on the left shows the window at its normal scale. The one on the right shows the window greatly enlarged.

It's hard to tell in Figure 17-1, but both printouts display smooth curves and lines even though the one on the right is enlarged. If the one on the right were enlarged even further, perhaps to the point where the word *New* in the title area filled the entire page, it would show smooth lines and curves with none of the grainy or blocky appearance that you would see if you enlarged a bitmap.

This chapter explains printing in WPF. It explains how you can print objects that inherit from the `Visual` class such as the window shown in Figure 17-1. It also tells how you can print output generated by code and complex documents.

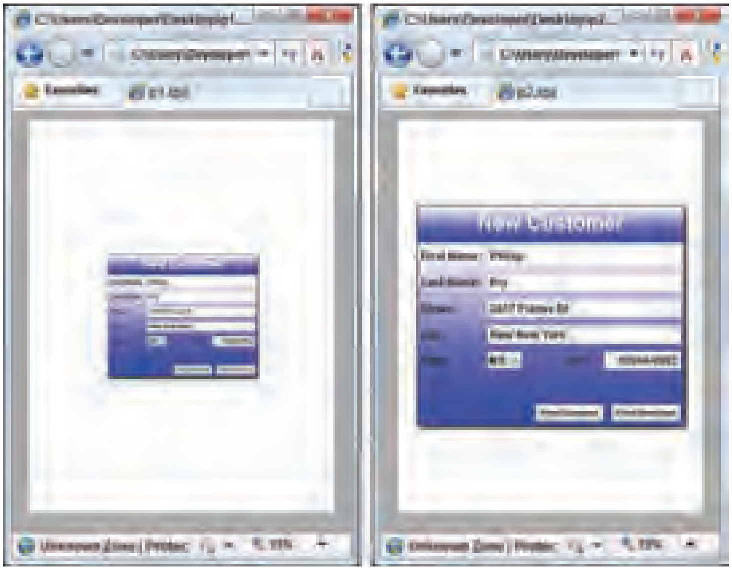


FIGURE 17-1

DOCUMENT PICTURES

To display pictures of documents in this chapter, I selected Microsoft XPS Document Writer as the printer to save the output into an XPS file. I then opened the file in Internet Explorer (as in Figure 17-1) or with XPS Viewer (as in Figure 17-4).

Unfortunately, Internet Explorer and XPS Viewer don't always display the same result produced by the printer. Throughout this chapter, I'll mention when the printout looks significantly different from the figures.

PRINTING VISUAL OBJECTS

One of the central classes for printing in WPF is `PrintDialog`. This class provides methods for letting the user pick a printer, interacting with print queues, and sending output to the printer.

DUST OFF YOUR PROGRAMMING SKILLS

WPF does not include a `PrintDialog` control that you can put on your windows, and XAML code cannot manipulate it. For printer-oriented tasks, therefore, you're going to have to write some code-behind.

Normally there are three steps to printing:

1. Create a new `PrintDialog` object and use its `ShowDialog` method to display it to the user.
2. Check `ShowDialog`'s return result to see if the user clicked OK.
3. Use the `PrintDialog`'s methods to generate a printout.

Figure 17-2 shows the dialog displayed by the `PrintDialog`'s `ShowDialog` method. Select a printer from the list at the top. Click on the Preferences button to set advanced options for the printer such as the paper tray, draft or normal resolution, and portrait or landscape orientation.

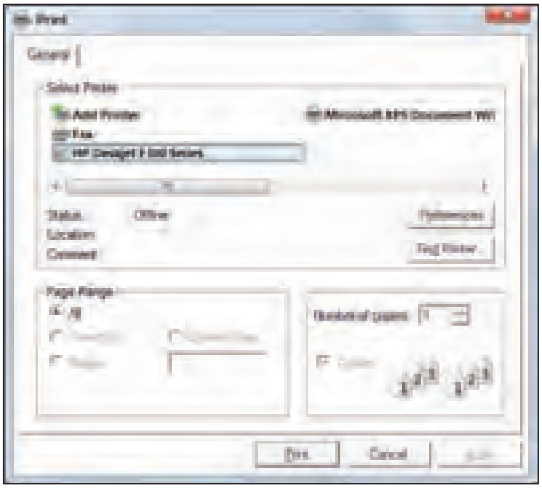


FIGURE 17-2

The following two sections explain two ways to print `Visual` objects. The first is simple but produces a mediocre result. The second is more work but produces much better output.

Simple Printing with PrintVisual

The SimplePrintWindow example program uses the following code to perform these three steps:

Available for download on Wrox.com

```
// Display the print dialog.
PrintDialog pd = new PrintDialog();

// See if the user clicked OK.
if (pd.ShowDialog() == true)
{
    // Print.
    pd.PrintVisual(this, "New Customer");
}
```

SimplePrintWindow

After displaying the dialog and verifying that the user pressed OK, the code calls the dialog's `PrintVisual` method, passing it the program's window. The `PrintDialog` object knows about the printer selected by the user and sends the visual object (in this case, the window) to the printer's queue.

PRINTING PRONTO

If you call `PrintVisual` without displaying the dialog, then the printout is immediately sent to the default printer.

This is remarkably simple, but it has an unfortunate drawback: WPF tries to draw the visual in the upper-left corner of the paper. Since printers generally cannot print all the way to the edges of the paper, the result is chopped off.

BETTER THAN PRINTING

If you select the Microsoft XPS Document Writer as your printer, the result is different from what you see coming out of a printer. The page is sized to fit the visual and, because the Document Writer can print all the way to the edges of its logical paper, the result isn't chopped off.

The `PrintVisual` method is extremely simple but doesn't produce a very nice result. It's chopped off at the top and left edges, and cannot rotate, scale, or center the result.

The following section describes a much more flexible method of printing.

Advanced Printing with PrintVisual

The `PrintVisual` method sends a `Visual` object to the printer, but it doesn't scale, rotate, or center the object. Fortunately, these are things that your program can do.

Rather than passing the program's window into the `PrintVisual` method, you can make a hierarchy of controls that contains an image of the window. That hierarchy can use `Grids`, `Viewboxes`, `Images`, `Rectangles`, and any other control that you want to produce the results. It can even include extra controls to produce such items as a page header and footer.

The PrintWindow example program shown in Figure 17-3 uses this approach to print its window centered at either normal or enlarged scale. Click on the "Print Centered" button to print the window at its normal scale. Click on the "Print Stretched" button to print the window as large as possible on the page.

FIGURE 17-3

The PrintWindow program requires references to the ReachFramework and System.Printing libraries. In Visual Studio, open the Project menu and select "Add Reference." Select these two libraries and click OK.

To make working with the libraries easier, the program includes the following `using` statements:

```
using System.Printing;
using System.Windows.Media.Effects;
```

The program's Visual Basic version includes the following `Imports` statements:

```
Imports System.Windows.Shapes
Imports System.Windows.Media.Effects
```

When you click its buttons, the PrintWindow program uses the following code to start the printing process. Both buttons display the `PrintDialog` and, if the user selects a printer and clicks OK, call `PrintWindowCentered` to do all of the interesting work.

Available for download on Wrox.com

```
// Print the window centered.
private void btnPrintCentered_Click(object sender, RoutedEventArgs e)
{
    PrintDialog pd = new PrintDialog();
    if (pd.ShowDialog() == true)
    {
        PrintWindowCentered(pd, this, "New Customer", null);
    }
}

// Print the window stretched to fit.
private void btnPrintStretched_Click(object sender, RoutedEventArgs e)
{
    PrintDialog pd = new PrintDialog();
    if (pd.ShowDialog() == true)
    {
        PrintWindowCentered(pd, this, "New Customer", new Thickness(50));
    }
}
```

PrintWindow

The following code shows the `PrintWindowCentered` function:

Available for download on Wrox.com

```
// Print a Window centered on the printer.
private void PrintWindowCentered(PrintDialog pd, Window win,
 String title, Thickness? margin)
{
    // Make a Grid to hold the contents.
    Grid drawing_area = new Grid();
    drawing_area.Width = pd.PrintableAreaWidth;
    drawing_area.Height = pd.PrintableAreaHeight;

    // Make a Viewbox to stretch the result if necessary.
    Viewbox view_box = new Viewbox();
    drawing_area.Children.Add(view_box);
    view_box.HorizontalAlignment = HorizontalAlignment.Center;
    view_box.VerticalAlignment = VerticalAlignment.Center;

    if (margin == null)
    {
        // Center without resizing.
        view_box.Stretch = Stretch.None;
```

```
        }
        else
        {
            // Resize to fit the margin.
            view_box.Margin = margin.Value;
            view_box.Stretch = Stretch.Uniform;
        }

        // Make a VisualBrush holding an image of the Window's contents.
        VisualBrush vis_br = new VisualBrush(win);

        // Make a rectangle the size of the Window.
        Rectangle win_rect = new Rectangle();
        view_box.Child = win_rect;
        win_rect.Width = win.Width;
        win_rect.Height = win.Height;
        win_rect.Fill = vis_br;
        win_rect.Stroke = Brushes.Black;
        win_rect.BitmapEffect = new DropShadowBitmapEffect();

        // Arrange to produce output.
        Rect rect = new Rect(0, 0,
            pd.PrintableAreaWidth, pd.PrintableAreaHeight);
        drawing_area.Arrange(rect);

        // Print it.
        pd.PrintVisual(drawing_area, title);
    }
```

PrintWindow

The function uses `Grid`, `Viewbox`, and `Rectangle` controls to display a window centered on the printed page. It takes four parameters:

- The `PrintDialog` to use. This object holds information about the printer that the user selected.
- The window to print
- A title for the print job
- A `Thickness` value to use as a margin around the printed window. The question mark in the parameter's declaration means this is a *nullable* type, so the program can pass the value `null` for this parameter. If this value is `null`, the routine prints the window at its normal size centered on the printout. If this is not null, then the routine enlarges the window as much as possible while still allowing this margin around it.

The code starts by creating a `Grid` control. It uses the `PrintDialog`'s `PrintableAreaWidth` and `PrintableAreaHeight` properties to see how big the printer's page is, and it makes the `Grid` fill the page.

Next, the code places a `Viewbox` centered inside the `Grid`. Recall that a `Viewbox`'s purpose is to stretch its single child in various ways.

If the routine's margin parameter is `null`, the code should display the window at its normal scale, so it sets the `Viewbox`'s `Stretch` property to `None`.

If the routine's margin parameter is not null, the code should enlarge the window. It sets the `Viewbox`'s `Margin` property to the `margin` parameter so the `Viewbox` expands to fill the `Grid` except for the `margin`. It also sets the `Viewbox`'s `Stretch` property to `Uniform` so it enlarges its child control as much as possible without changing its aspect ratio (the ratio of width to height).

Next, the code makes a `VisualBrush` that contains an image of the window that it should print.

It then makes a `Rectangle` with the same size as the window, fills the `Rectangle` with the `VisualBrush`, and places the `Rectangle` inside the `Viewbox`. The `Viewbox` stretches the `Rectangle` if appropriate.

Next, the code calls the `Grid`'s `Arrange` method. This makes the `Grid` recursively arrange its children and renders them. If you omit this call, the code runs, but the printout is blank.

Finally, the code calls the `PrintDialog`'s `PrintVisual` method to print the `Grid` and its contents.

If the user clicks on the print dialog's Preferences button and changes the printer's orientation to landscape, the `PrintDialog`'s `PrintableAreaWidth` and `PrintableAreaHeight` properties automatically switch so the controls are arranged appropriately.

Figure 17-4 shows the result when the program's window is stretched and printed using the Microsoft XPS Document Writer with preferences set to landscape orientation and displayed in XPS Viewer.

FIGURE 17-4

PRINTING CODE-GENERATED OUTPUT

The `PrintDialog` object's `PrintVisual` method is easy to use and, with a little extra work, can produce nicely scaled and centered results. It still assumes that you are only printing one page at a time, however. If you want to print a longer document, you'll need to call `PrintVisual` once for each page, and it will produce one print job for each page. This isn't an ideal solution. Fortunately there's a better way to produce multi-page printouts.

The `PrintDialog` object's `PrintDocument` method takes a `DocumentPaginator` object as a parameter. That object generates the pages of a printout, and the `PrintDocument` places them in a single print job.

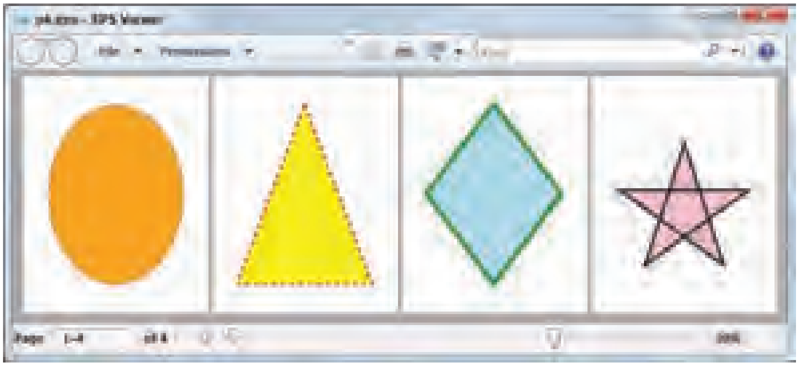


FIGURE 17-5

`DocumentPaginator` is an abstract class that defines methods for you to implement in a subclass. Those methods give the `PrintDocument` method information such as the printout's total number of pages and the objects to print.

The PrintShapes example program uses a `ShapesPaginator` class that inherits from `DocumentPaginator` to print four pages of shapes. Figure 17-5 shows the output saved by the Microsoft XPS Document Writer printer and displayed in XPS Viewer.

The following code shows how the program responds when you click on its "Print Shapes" button. Like the previous code that uses `PrintVisual`, it displays a `PrintDialog` and sees whether the user selected a printer and clicked OK. It then calls the `PrintDocument` method, passing it a paginator object and a print job title.

Available for download on Wrox.com

```
// Print the shapes.
private void btnPrintShapes_Click(object sender, RoutedEventArgs e)
{
    PrintDialog pd = new PrintDialog();
    if (pd.ShowDialog() == true)
    {
        pd.PrintDocument(
            new ShapesPaginator(
                new Size(pd.PrintableAreaWidth, pd.PrintableAreaHeight)),
            "Shapes");
    }
}
```

PrintShapes

PRINTING PRONTO, PART 2

If you call `PrintDocument` without displaying the dialog, then the printout is immediately sent to the default printer.

The following code fragment shows the key pieces of the `ShapesPaginator` class:

Available for download on Wrox.com

```
class ShapesPaginator : DocumentPaginator
{
    private Size m_PageSize;

    // Save the page size.
    public ShapesPaginator(Size page_size)
    {
        m_PageSize = page_size;
    }

    // Return the needed page.
    public override DocumentPage GetPage(int pageNumber)
    {
        const double WID = 600;
        const double HGT = 800;

        Grid drawing_grid = new Grid();
        drawing_grid.Width = m_PageSize.Width;
        drawing_grid.Height = m_PageSize.Height;

        switch (pageNumber)
        {
            case 0: // Ellipse
                Ellipse ell = new Ellipse();
                ell.Fill = Brushes.Orange;
                ell.Stroke = Brushes.Blue;
                ell.StrokeThickness = 1;
                ell.HorizontalAlignment = HorizontalAlignment.Center;
                ell.VerticalAlignment = VerticalAlignment.Center;
                ell.Width = WID;
                ell.Height = HGT;
                drawing_grid.Children.Add(ell);
                break;

            ... Code for other pages omitted ...
        }

        // Arrange to make the controls draw themselves.
        Rect rect = new Rect(new Point(0, 0), m_PageSize);
        drawing_grid.Arrange(rect);

        // Return a DocumentPage wrapping the grid.
        return new DocumentPage(drawing_grid);
    }

    // If pagination is in progress and PageCount is not final, return False.
    // If pagination is complete and PageCount is final, return True.
    // In this example, there is no pagination to do.
    public override bool IsPageCountValid
    {
        get { return true; }
    }

    // The number of pages paginated so far.
```

```
        // This example has exactly 4 pages.
        public override int PageCount
        {
            get { return 4; }
        }

        // The suggested page size.
        public override Size PageSize
        {
            get { return m_PageSize; }
            set { m_PageSize = value; }
        }

        // The element currently being paginated.
        public override IDocumentPaginatorSource Source
        {
            get { return null; }
        }
    }
```

PrintShapes

The class's constructor takes as a parameter the size of the page on which it will draw and saves that size for later use.

The most interesting part of the class is the `GetPage` method, which returns objects to print. It starts by creating a `Grid` control that fills the printed page.

Next, depending on the page number, the code places a shape on the `Grid`. The previous code shows how the program adds an `Ellipse` to the `Grid` on the first page. The code for the other pages is similar, so it has been omitted to save space.

The `GetPage` method then makes the `Grid` control arrange itself to produce output. It finishes by returning a new `DocumentPage` object initialized to display the `Grid` and its contents.

PRINTING DOCUMENTS

By using the `PrintDialog`'s `PrintDocument` method, you can print many pages of relatively simple output. For example, you could draw a series of pictures as the PrintShapes program does. You could also print tables spread across several pages or simple blocks of text.

To produce really complicated output, however, you would need to write a complicated `DocumentPaginator` class. For example, suppose you want to print a multi-page newsletter that has paragraphs flowing around embedded pictures and tables. Building a paginator that could handle these complex layout tasks would be quite a chore.

However, displaying complex output like this is much easier using the `FlowDocument` and `FixedDocument` controls. A `FlowDocument` arranges its paragraphs, tables, lists, and other contents to make best use of the available space much as a web browser arranges the contents of a web page. A `FixedDocument` displays items at specific positions that never change, much as Adobe Reader displays the items in a PDF file in fixed positions.

The following sections explain how to print `FlowDocuments` and `FixedDocuments`.

Printing FlowDocuments

Instead of building a complicated paginator object to produce a complex printout, you can build a FlowDocument object and place paragraphs, tables, lists, figures, and other elements inside it. The FlowDocument automatically arranges its contents to fill the available space. (Chapter 5 includes several example programs that demonstrate FlowDocuments.)

Unfortunately, the FlowDocument class does not inherit from the Visual class, so you cannot simply pass it into the PrintDialog's PrintVisual method. You could pass the Window containing the FlowDocument into PrintVisual, but then the printout will only include an image of whatever is visible on the screen at the time — not the FlowDocument's entire contents (which may not all be currently visible).

Printing a FlowDocument correctly is simple but fairly confusing. The basic idea is to create an XpsDocumentWriter object associated with a print queue. That object's Write method can write a DocumentPaginator into the queue to start a print job. To make it all work, you just need to figure out how to build a DocumentPaginator for the FlowDocument.

Fortunately, the FlowDocument class implements the IDocumentPaginatorSource interface, and that interface defines a DocumentPaginator property that returns a paginator for the document.

It's a little convoluted, but the necessary code is quite short.

The following code shows how the PrintFlowDocument example program prints its FlowDocument when you open its File menu and select Print:

Available for download on Wrox.com

```
// Print the FlowDocument.
private void mnuFilePrint_Click(object sender, RoutedEventArgs e)
{
    PrintDialog pd = new PrintDialog();
    if (pd.ShowDialog() == true)
    {
        // Make an XPS document writer for the print queue.
        XpsDocumentWriter xps_writer =
            PrintQueue.CreateXpsDocumentWriter(pd.PrintQueue);

        // Turn the FlowDocument into an IDocumentPaginatorSource.
        IDocumentPaginatorSource paginator_source =
            (IDocumentPaginatorSource)fdContents;

        // Use the IDocumentPaginatorSource's
        // property to get a paginator.
        xps_writer.Write(paginator_source.DocumentPaginator);
    }
}
```

PrintFlowDocument

As usual, the code starts by displaying a PrintDialog. If the user selects a printer and clicks OK, the code makes an XpsDocumentWriter associated with the selected printer's queue.

Next, the code creates a reference to the FlowDocument control named fdContents using its IDocumentPaginatorSource interface.

Finally, the code uses the `XpsDocumentWriter` object's `Write` method to write the document's paginator into the print queue.

Figure 17-6 shows the PrintFlowDocument program running in TwoPage mode, so it displays two pages of the `FlowDocument`'s contents.

FIGURE 17-6

The program displays its `FlowDocument` inside a `FlowDocumentReader`. Note that the format of the program's printout depends on the reader's mode when you make the printout. Figures 17-7 through 17-9 show the results when the reader is in Page mode, TwoPage mode, and Scroll mode, respectively.

Figures 17-7 and 17-8 look a lot like their printed versions, although the printed results are much smaller than they appear in the figures. The figures make it look like each page is fairly full, but when printed, the results occupy only about the upper third of the printed page.

FIGURE 17-7

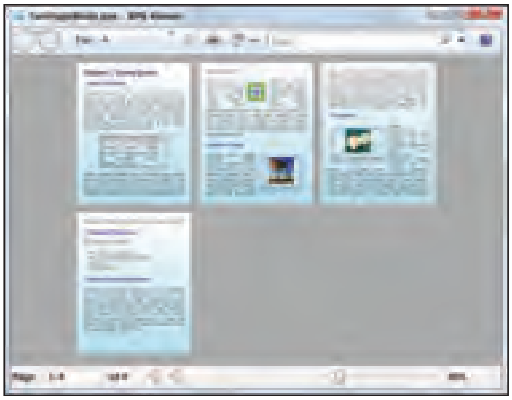


FIGURE 17-8

Figure 17-9 looks almost exactly the same as the printed version.

As is the case with the previous examples, since the `PrintDocument` method tries to print all the way to the paper's upper-left corner, the top and left edges of the output are clipped unless you include margins within the `FlowDocument`.

You can get a more consistent result by setting the `FlowDocument`'s `PageHeight` and `PageWidth` properties. Unfortunately, setting these properties appropriately for the printer makes the result on the screen quite small.

One way to work around this problem is to set the control's properties just before printing and then restore their original values afterward.

FIGURE 17-9

> **BUG ALERT**
>
> Because of an apparent bug in the FlowDocumentReader, the control does not redraw properly after printing in Scroll mode. If you try to scroll the control at that point, it crashes.
>
> If you change the control's zoom level or viewing mode, it refreshes itself, and all is well.

Printing FixedDocuments

The PrintFixedDocument example program shown in Figure 17-10 displays a `FixedDocument` inside a `DocumentViewer` control.

If the user clicks on the printer icon on the upper left, the viewer automatically launches a print dialog, and if the user selects a printer and clicks OK, it handles the printing for you.

If you want to do the work yourself, it's not hard to print a `FixedDocument` programmatically using the same code you would use to print a `FlowDocument`. Simply replace the

FIGURE 17-10

`FlowDocument` with a `FixedDocument`. See the previous section for more information about printing `FlowDocuments`.

SUMMARY

WPF's `PrintDialog` class provides easy access to the printing system. Its `PrintVisual` method lets you print a single page displaying an image of a visual control almost effortlessly. Its `PrintDocument` method lets you generate more complex printouts such as multi-page documents, `FlowDocuments`, and `FixedDocuments`.

Earlier chapters in this book explain how to set a control's properties to a value defined in a resource, style, or template. For example, if you define a string resource, you can later set a Window's Title property to the value of that resource.

This is a handy technique for centralizing and reusing values but it ignores other potentially useful data sources such as the properties of other controls and objects defined by code-behind.

The following chapter describes WPF data binding methods that let you attach a control's properties to objects other than resources including arrays of values, other controls' properties, and the properties of other kinds of objects.

18

Data Binding

WPF data binding lets you bind a target to a data source so the target automatically displays the value in the data source. For example, this lets you:

- Make a `ListBox` display an array of values defined in XAML code.
- Make a `ListBox` display a list of objects created in code-behind.
- Make a `TreeView` build a hierarchical display of objects created in code-behind.
- Make `TextBoxes`, `Labels`, and other controls display additional detail about the currently selected item in a `ListBox`, `ComboBox`, or `TreeView`.

Additional WPF data-binding features let you sort, filter, and group data; let the user modify a control to update a data source; and validate changes to data.

This chapter provides an introduction to data binding in WPF. It explains how to use bindings to associate objects and how to use bindings to let a control use values supplied by other controls, XAML resources, and objects created in code-behind.

BINDING BASICS

Data bindings have these four basic pieces:

- **Target** — The object that will use the result of the binding
- **Target Property** — The target object's property that will use the result
- **Source** — The object that provides a value for the target object to use
- **Path** — A path that locates the value within the source object

As a trivial example, suppose you want to bind a `Label` control's `Content` property so that the `Label` displays whatever you type in a `TextBox`. If the `Label` control is named `lblResult` and the `TextBox` is named `txtTypeHere`, then you would need to create a binding where:

- The target is `lblResult` (the control where you want the binding's result to go).

- The target property is `Content` (you want the `Label` to display the result in its `Content` property).
- The source is `txtTypeHere` (the object providing the value).
- The path is `Text` (the path to the data in the source object, in this case, the `TextBox`'s `Text` property).

FIGURE 18-1

The TextBoxToLabel example program shown in Figure 18-1 demonstrates this kind of simple binding. When you type in the textbox, the two labels below it echo whatever you type.

The following XAML code shows how the program works:

Available for download on Wrox.com

```
<StackPanel Margin="5">
    <TextBox Name="txtTypeHere" Margin="5" Height="30"
     VerticalAlignment="Top" Text="Type here!"/>

    <Label Margin="5" BorderBrush="Yellow" BorderThickness="1">
        <Binding ElementName="txtTypeHere" Path="Text"/>
    </Label>

    <Label Margin="5" BorderBrush="Yellow" BorderThickness="1"
     Content="{Binding ElementName=txtTypeHere, Path=Text}"/>
</StackPanel>
```

TextBoxToLabel

The code starts by defining a `TextBox` named `txtTypeHere`.

Next, the code defines a `Label`. The control's start and end tags enclose its content, in this case, a `Binding` object. The object's `ElementName` property defines the source control — the `TextBox`. Its `Path` property determines the location of the data value in the source control — the `Text` property.

In this binding, the four key pieces are:

- **Target** — The `Label` that contains the binding
- **Target Property** — The `Content` property that is set equal to the binding
- **Source** — Given by the `ElementName` attribute
- **Path** — The name of the source control's property

Finally, the code defines a second `Label` that uses a binding similar to the first one, except this time it defines the binding with the ungainly string `{Binding ElementName=txtTypeHere, Path=Text}`.

The following sections describe the four pieces of the binding in greater detail and explain how to set their values.

Binding Target and Target Property

The target and target property are the easiest parts of the binding to understand. The *target* is the control that contains the binding, and the *target property* is the property that is set to the binding's value. In the following code, the `Label` control is the target, and its `Content` property is the target property:

```
<Label Margin="5" BorderBrush="Yellow" BorderThickness="1"
 Content="{Binding ElementName=txtTypeHere, Path=Text}"/>
```

IT ALL DEPENDS

A binding's target property must be a dependency property. Fortunately, most of the properties that you are most likely to want to bind are dependency properties.

The situation is somewhat more confusing when the target uses several pieces of the binding's data.

For example, suppose you define a `Planet` class to track information about planets. Its properties include `Name`, `Picture`, `Stats`, and so forth.

If you make a collection of `Planet` objects, you can bind it to a `ListBox` that displays some or all of these values. This case is a bit more complicated than binding a `Label`'s `Content` property to a `TextBox`'s `Text` property.

The short explanation for this example is that the target property is the `ListBox`'s `ItemsSource` property. The section "Binding Collections" later in this chapter explains how a `ListBox` can convert a collection bound to its `ItemsSource` property into a display of the collection's fields.

Binding Source

The binding's source is the object from which it takes a value. This object can be a WPF control, XML data, or an object defined in the program's code-behind.

You can specify the source object by setting any of the binding's `ElementName`, `Source`, or `RelativeSource` properties.

PICK ONE

You can only specify one of the `ElementName`, `Source`, and `RelativeSource` properties at a time.

ElementName

The binding's `ElementName` property gives the name of a control. The TextBoxToLabel example program described earlier uses `ElementName` to identify the `TextBox` that a binding should use as its source.

Source

The binding's `Source` property identifies an object that should be used as a binding's source. Typically, a program uses a `StaticResource` to define this object.

The PersonSource example program shown in Figure 18-2 uses a binding with its `Source` property set to display information about a `Person` object.

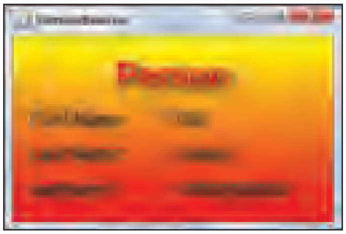


FIGURE 18-2

The PersonSource example program defines a `Person` class with the properties `FirstName`, `LastName`, and `NetWorth`. The program's XAML code includes the following namespace definition so it can use the class:

```
xmlns:local="clr-namespace:PersonSource"
```

The following code shows how the program's XAML code defines a static resource named `a_person` that is a `Person` object:

Available for download on Wrox.com

```
<Window.Resources>
    <local:Person x:Key="a_person" FirstName="Bill"
     LastName="Gates" NetWorth="40000000000"/>
</Window.Resources>
```

PersonSource

The following code shows how the program uses bindings to display the `Person` object's `FirstName`, `LastName`, and `NetWorth` values. It uses binding `Source` properties to identify the `Person` object that provides the values.

Available for download on Wrox.com

```
<Label Grid.Row="1" Grid.Column="0" Content="First Name:"/>
<Label Grid.Row="1" Grid.Column="1"
 Content="{Binding Source={StaticResource a_person}, Path=FirstName}"/>

<Label Grid.Row="2" Grid.Column="0" Content="Last Name:"/>
<Label Grid.Row="2" Grid.Column="1"
 Content="{Binding Source={StaticResource a_person}, Path=LastName}"/>

<Label Grid.Row="3" Grid.Column="0" Content="NetWorth:"/>
<Label Grid.Row="3" Grid.Column="1"
 Content="{Binding Source={StaticResource a_person}, Path=NetWorth}"/>
```

PersonSource

RelativeSource

The binding's `RelativeSource` property lets you specify a source object by its relationship to the target control. One situation in which this is useful is when you want to bind two properties on the same control.

The TypeAColor example program shown in Figure 18-3 binds the `TextBox`'s `Background` property to its `Text` property, so when you type the name of a color, the control uses that color as its background.

FIGURE 18-3

> **WHAT ERROR?**
>
> Bindings don't report errors at run time. If a binding's value doesn't make any sense — for example, if you set a `Width` equal to a `Color` — nothing happens. In the program TypeAColor, if you type some text that isn't a color's name, nothing happens and the program keeps running just fine.

The following code shows how the TypeAColor program binds the `TextBox`'s `Background` property to its `Text` property:

Available for download on Wrox.com

```
<TextBox Margin="10" Height="30" VerticalAlignment="Top"
 Background="{Binding RelativeSource={RelativeSource Self}, Path=Text}"/>
```

TypeAColor

The text `RelativeSource={RelativeSource Self}` sets the binding's `RelativeSource` property to a new `RelativeSource` object built from the string `Self`.

You can also build a `RelativeSource` object with the strings `TemplatedParent`, `FindAncestor`, or `PreviousData`.

The `TemplatedParent` setting uses a reference to the object to which a template is being applied. The TemplatedParent example program shown in Figure 18-4 displays a `Label` with a template that displays the `Label`'s margins along its edges. This `Label`'s `Margin` property was set to `10,20,30,40`.

FIGURE 18-4

The following code shows the part of the `Label`'s template that displays the `Label`'s top margin. The binding uses its `RelativeSource` property to find the `Label` that is using the template. The `Path` value `Margin.Top` makes the binding get that `Label`'s `Margin` and use its `Top` value.

Available for download on Wrox.com

```
<Label VerticalAlignment="Top" HorizontalAlignment="Center"
 FontSize="10" Content=
 "{Binding RelativeSource={RelativeSource TemplatedParent}, Path=Margin.Top}"/>
```

TemplateParent

The `RelativeSource` value `FindAncestor` lets you find one of the target control's ancestors of a certain type. The ColumnWidths example program shown in Figure 18-5 uses this kind of binding to make `Labels` display the widths of the `Grid` columns that contain them.

FIGURE 18-5

The following code defines the third `Label`. Its binding searches for the `Label`'s ancestor of type `Grid`. It then gets its value from the `Grid`'s `ColumnDefinitions[2].Width` value.

Available for download on Wrox.com

```
<Label Grid.Column="2"
 Content="{Binding RelativeSource={RelativeSource FindAncestor,
 AncestorType={x:Type Grid}},
 Path=ColumnDefinitions[2].Width}"/>
```

ColumnWidths

The `RelativeSource` value `PreviousData` is useful in lists and other controls that display multiple values. It lets the code that displays one value refer to the previous value.

The PreviousData example program shown in Figure 18-6 demonstrates the `PreviousData` value. Each `ListBox` entry displays its value and the value of the item before it. The first item has no previous data, so it displays a blank value. (Fixing this is a bit outside the scope of the current discussion.)

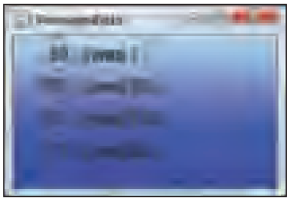


FIGURE 18-6

The following code shows how the program defines its `ListBox`. The `ItemsSource` property refers to a static resource named `numbers` that contains an array of numbers.

Available for download on Wrox.com

```
<ListBox ItemsSource="{StaticResource numbers}"
 Background="Transparent">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <Label Content="{Binding}" Width="50"
                 HorizontalContentAlignment="Right"/>
                <Label Content=" (was"/>
                <Label Content="{Binding
                 RelativeSource={RelativeSource PreviousData}}"
                 Margin="-5,0,0,0"/>
                <Label Content=")" Margin="-10,0,0,0"/>
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

PreviousData

Don't worry about the code's details. Focus on the bindings and ignore the `ItemTemplate` and `DataTemplate` elements for now. The section "Binding Collections" later in this chapter has more to say about binding `ListBoxes` to arrays and other lists of data.

DataContext

There is one additional way to specify a binding's source: by setting a control's `DataContext` property. Since controls inherit the `DataContext` property of their ancestors, this method provides a convenient way for several controls to share the same data source.

For example, the PersonSource example program described earlier in this chapter and shown in Figure 18-2 uses several `Label` controls to display a `Person` object's property values. The following code shows how one of the `Labels` is defined:

```
<Label Grid.Row="1" Grid.Column="1"
 Content="{Binding Source={StaticResource a_person}, Path=FirstName}"/>
```

Each of the `Labels` uses a similar binding that includes its own source.

The PersonSource2 example program is similar except it uses `DataContext` properties to simplify the `Labels`' bindings. The following code shows how the program sets the `DataContext` property for the `Grid` that contains the `Labels`. The code binds the `Grid`'s `DataContent` property to the `a_person` object.

```
<Grid Margin="10" DataContext="{Binding Source={StaticResource a_person}}">
```

The following code shows the new `Label` definition with its simplified binding:

```
<Label Grid.Row="1" Grid.Column="1" Content="{Binding Path=FirstName}"/>
```

Because the `Label` inherits the `Grid`'s `DataContext` property, it already has a binding source defined. The new code only needs to specify the binding's path.

Using `DataContext` properties in this way can simplify the code. It also makes the data source definition more centralized so it's easier to change if you need to use a different data source later.

Binding Path

Usually a binding's path is easy enough to understand. In many cases, you can set it by setting the `Path` attribute to the name of the source property that you want to use. For example, the following code sets the binding's path to the `txtTypeHere` `TextBox`'s `Text` property:

```
<Label Margin="5" BorderBrush="Yellow" BorderThickness="1"
 Content="{Binding ElementName=txtTypeHere, Path=Text}"/>
```

The following list summarizes the rules for specifying a binding's path:

- To use a property of the binding's source, set the path equal to the property's name, as in `Path = Property`.
- If a property is an object, you can use the object's properties, as in `Path = Object.Property`.
- If the path includes an attached property, enclose the property in parentheses. For example, the GridColumns program uses the following code to make a `Label` control display its own `Grid.Column` value:

```
<Label Grid.Column="0"
 Content="{Binding RelativeSource={RelativeSource Self}, Path=(Grid.Column)}"/>
```

ANOTHER EXAMPLE

The DockPanelValues example program uses a `Style` with `TargetType = Label` to make `Labels` contained in a `DockPanel` display their `DockPanel.Dock` attached property values.

- If a value provides an indexer, include the index in square brackets. For instance, the ColumnWidths example program described earlier in this chapter uses paths of the form `ColumnDefinitions[2].Width` to display a `Grid`'s column widths.
- If the binding's source is a collection, you can refer to the collection's current item with the slash character (/).

The ColorList example program shown in Figure 18-7 demonstrates bindings with `Source = /`. When you select a color from the `ListBox` on the left, the `Label` on the right displays the name of the color you selected and changes its `Background` to match.

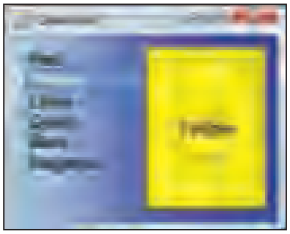


FIGURE 18-7

The ColorList program uses the following code to define its `ListBox` and `Label`:

Available for download on Wrox.com

```
<Grid Margin="10">
    <Grid.DataContext>
        <x:Array Type="sys:String">
            <sys:String>Red</sys:String>
            <sys:String>Yellow</sys:String>
            <sys:String>Lime</sys:String>
            <sys:String>Cyan</sys:String>
            <sys:String>Blue</sys:String>
            <sys:String>Magenta</sys:String>
        </x:Array>
    </Grid.DataContext>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>

    <ListBox Name="lstColors" Grid.Column="0" Background="Transparent"
     IsSynchronizedWithCurrentItem="True"
     ItemsSource="{Binding}"/>

    <Label Grid.Column="1" BorderBrush="Black" BorderThickness="1"
     HorizontalContentAlignment="Center" VerticalContentAlignment="Center"
     Content="{Binding Path=/}"
     Background="{Binding Path=/}"/>
</Grid>
```

ColorList

The code starts with a `Grid` control. It sets this control's `DataContext` to an array of strings containing color names.

After defining two grid columns, the code creates a `ListBox`. Its `ItemsSource` property is set to the unusual value `{Binding}` to indicate that this property should be set to the control's inherited `DataContext` value.

The `Label`'s `Content` and `Background` properties are both set to `{Binding Path=/}`, so their binding source is the control's inherited `DataContext`, and their binding path is the array's currently selected value.

BINDING COLLECTIONS

Some controls, such as `Button`, `TextBox`, and `Label`, are considered *content controls*. They are intended to display one piece of data such as some text or a grid containing other controls.

Other controls, such as `ListBox`, `ComboBox`, and `TreeView`, are considered *item controls*. These display a repeating sequence of values stored in collections, arrays, trees, and other repetitive data structures. The values can be simple things such as numbers or strings, or they can be more complex objects such as instances of the `Order` or `Employee` classes.

Binding content controls is relatively straightforward. Set the binding's target, target property, source, and path; and you're done.

On the surface, binding item controls is just as easy. Just set the control's `ItemsSource` property to some sort of collection that holds repeating values. If the values are simple numbers or strings, this works.

FIGURE 18-8

The NumberList example program shown in Figure 18-8 works this way.

The NumberList example program defines an array of integers. To define the numbers using the `Int32` data type, the program includes the following namespace declaration:

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

The program then defines its array in its Windows.Resources section:

Available for download on Wrox.com

```
<Window.Resources>
    <x:Array x:Key="numbers" Type="sys:Int32">
        <sys:Int32>100</sys:Int32>
        <sys:Int32>200</sys:Int32>
        <sys:Int32>50</sys:Int32>
        <sys:Int32>800</sys:Int32>
        <sys:Int32>175</sys:Int32>
    </x:Array>
</Window.Resources>
```

NumberedList

Finally, it uses the following code to bind the array to a `ListBox`:

```
<ListBox ItemsSource="{StaticResource numbers}" Background="Transparent"/>
```

As you can see in Figure 18-8, the `ListBox` automatically displays the values in the array.

Unfortunately, the situation is more complicated if you bind a `ListBox` to more complex objects. To determine what to display for an item, the control calls the item's `ToString` method. For simple things like numbers and strings, `ToString` returns the item's textual representation.

For a more complex item like a `Person` object, `ToString` returns the name of the class. For example, in the PersonList example program, the `Person` class's `ToString` method returns `PersonList.Person` for every item, so a `ListBox` displays `PersonList.Person` for every item.

One way around this problem is to make the class override its `ToString` method to return something more useful.

The PersonList example program shown in Figure 18-9 displays `ListBoxes` holding items from two classes: `Person` and `Person2`. The `Person` class uses its default `ToString` method, whereas the `Person2` class overrides its `ToString` method.

FIGURE 18-9

The `Person2` class uses the following code to override its `ToString` method:

```
public override string ToString()
{
    return FirstName + " " + LastName;
}
```

CLASSES IN CODE

XAML code can define objects using the classes you have defined in your code-behind. To make this possible, first include a namespace declaration identifying your program's namespace like this:

```
xmlns:local="clr-namespace:PersonList"
```

Now the XAML code can use the class as a data type. For example, the PersonList program uses the following code to make its array of `Person` objects:

```
<Window.Resources>
    <x:Array x:Key="people" Type="local:Person">
        <local:Person FirstName="James" LastName="Kirk"/>
        <local:Person FirstName="Spock"/>
        ...
    </x:Array>
</Window.Resources>
```

Note that the class must have an empty constructor (one that takes no parameters) to be used by XAML code.

The overridden `ToString` method makes the results shown in a `ListBox` much more useful, but WPF's `ListBox` class can do much more than display simple strings. Item controls like `ListBox` provide templates that you can use to determine how each item is displayed.

ListBox and ComboBox Templates

The `ListBox` and `ComboBox` controls have an `ItemTemplate` property that determines how each item is displayed. These templates are somewhat similar to the templates described in Chapter 15.

The template can hold whatever controls you like to represent the control's items. Usually many of these controls are bound to the properties of whatever item the control is currently displaying.

The Planets example program shown in Figure 18-10 displays an array of `Planet` objects in a `ListBox` on the left and in a `ComboBox` on the right.

FIGURE 18-10

The following code shows the `ItemTemplate` used by the program's `ListBox`:

Available for download on Wrox.com

```
<ListBox.ItemTemplate>
    <DataTemplate>
        <DataTemplate.Resources>
            <!--  The ListBoxItem style must be set
                  in ListBox.Resources.  -->
            <Style TargetType="TextBlock">
                <Setter Property="Margin" Value="3"/>
                <Setter Property="HorizontalAlignment" Value="Left"/>
                <Setter Property="VerticalAlignment" Value="Center"/>
                <Setter Property="FontSize" Value="20"/>
                <Setter Property="FontWeight" Value="Bold"/>
                <Setter Property="Foreground" Value="Blue"/>
            </Style>
            <Style TargetType="Image">
```

```
                <Setter Property="Height" Value="50"/>
                <Setter Property="Margin" Value="3"/>
                <Setter Property="Stretch" Value="Uniform"/>
                <Setter Property="HorizontalAlignment" Value="Right"/>
            </Style>
            <Style TargetType="TextBox">
                <Setter Property="Margin" Value="3"/>
                <Setter Property="Width" Value="250"/>
                <Setter Property="Background" Value="SkyBlue"/>
                <Setter Property="TextWrapping" Value="Wrap"/>
                <Setter Property="IsReadOnly" Value="True"/>
            </Style>
        </DataTemplate.Resources>
        <StackPanel>
            <Grid>
                <TextBlock Text="{Binding Name}"/>
                <Image Source="{Binding Picture}"
                 Height="50"/>
            </Grid>
            <TextBox Text="{Binding Stats}"/>
        </StackPanel>
    </DataTemplate>
</ListBox.ItemTemplate>
```

Planets

Most of this code uses `Styles` to define properties for the controls that will display the item's data. The `StackPanel` at the end contains those controls. The controls that actually display the data are:

- A `TextBlock` bound to the `Planet` item's `Name` property
- An `Image` bound to the `Planet`'s `Picture` property
- A `TextBox` bound to the `Planet`'s `Stats` property

In addition to the `ItemTemplate` property, the `ListBox` and `ComboBox` controls provide an `ItemsPanel` property that determines the panel control used to lay out the items. By default, the `ListBox` and `ComboBox` use a vertically oriented `StackPanel` for their `ItemsPanel` so that the items appear in a single column, but you can change this.

The PlanetsPanel example program uses the following code to make its `ListBox` arrange items in a `WrapPanel` Instead of a vertical `StackPanel`:

Available for download on Wrox.com

```
<ListBox.ItemsPanel>
    <ItemsPanelTemplate>
        <WrapPanel Orientation="Horizontal" HorizontalAlignment="Stretch" />
    </ItemsPanelTemplate>
</ListBox.ItemsPanel>
```

PlanetsPanel

Figure 18-11 shows the PlanetsPanel program in action. If you widen the `Window` at run time, the `ListBox` widens, and so does the `DockPanel` that it uses to arrange its items. If you make the `Window` wide enough, the `DockPanel` will have room to place another item on each row, so it will display the items in three columns instead of the two shown in the figure.

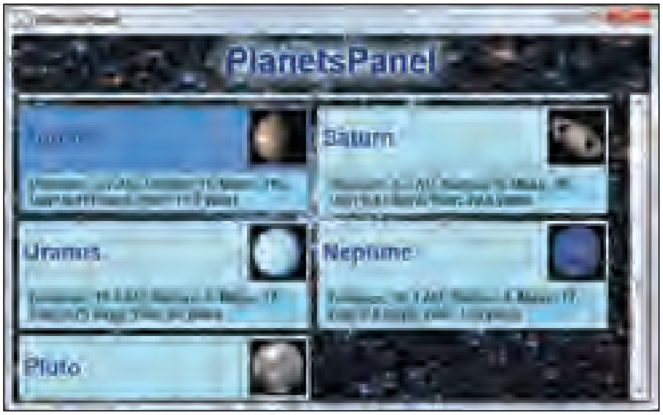


FIGURE 18-11

TreeView Templates

The `TreeView` control works a bit differently than the `ListBox` and `ComboBox` because it displays hierarchical data instead of items in a simple list.

In a `TreeView`, you must specify two things at each node in the tree — the data to display for that node and the path to follow deeper into the tree.

For example, consider an organizational chart that lists corporate regions that contain departments, which, in turn, contain employees. When the `TreeView` reaches a department node, your program must tell it what to display (e.g., the department's name) and how to move deeper into the tree (by listing the department's employees).

The OrgChartTreeView example program shown in Figure 18-12 displays a company organizational chart with departments displayed either by managers or by projects. To make understanding the items easier, region names are red, department names are blue and end in the word *department*, manager names are shown in blue boxes, project names are shown in goldenrod boxes, and employee names are plain black text.

The `TreeView` on the left shows regions, each region's departments, each department's managers, and each manager's direct report employees. The `TreeView` on the right shows regions, each region's departments, each department's projects, and each project's team members.

The following table describes the classes defined in the program's code-behind and the properties those classes have:

CLASS	PROPERTY	DATA TYPE
Region	RegionName	String
	Departments	List of Department
Department	Name	String
	Managers	List of Manager
	Projects	List of Project
Manager	Title	String
	Reports	List of Employee
Project	Name	String
	Description	String
	TeamLead	Employee
	TeamMembers	List of Employee
Employee	FirstName	String
	LastName	String
	Extension	String

FIGURE 18-12

The Manager class is derived from the Employee class, so it also inherits the Employee class's FirstName, LastName, and Extension properties.

The TreeView controls that produce these two displays include HierarchicalDataTemplate objects in their Resources sections to describe what the control should do at each node. For example, the following code shows how the controls handle Region objects:

Available for download on Wrox.com

```
<HierarchicalDataTemplate
 DataType="{x:Type local:Region}"
 ItemsSource="{Binding Path=Departments}">
    <TextBlock Text="{Binding Path=RegionName}" Foreground="Red">
        <TextBlock.BitmapEffect>
            <OuterGlowBitmapEffect/>
        </TextBlock.BitmapEffect>
    </TextBlock>
</HierarchicalDataTemplate>
```

OrgChartTreeView

The HierarchicalDataTemplate has three key jobs:

1. First, its DataType attribute tells what object this template handles. The preceding template springs into action when the TreeView encounters a Region object in its data.
2. Second, the template's ItemsSource attribute tells where the TreeView should look for children of this item. The previous code indicates that a Region object has children that are Departments. You can use the ItemsSource attribute to make the TreeView follow different paths through the data hierarchy.
3. Third, the HierarchicalDataTemplate provides the template that the TreeView should use to display the data at this node. This example displays the Region's RegionName property in a TextBlock.

The TreeView on the left in Figure 18-12 uses the following HierarchicalDataTemplate to display data for a Department object:

Available for download on Wrox.com

```
<HierarchicalDataTemplate
 DataType="{x:Type local:Department}"
 ItemsSource="{Binding Path=Managers}">
    <TextBlock Text="{Binding Path=Name}" Foreground="Blue"/>
</HierarchicalDataTemplate>
```

OrgChartTreeView

This template's ItemsSource attribute indicates that the children of a Department node are given by the object's Managers property, so the TreeView follows the Managers path out of Department objects.

Contrast that code with the following code used by the TreeView on the right in Figure 18-12:

Available for download on Wrox.com

```
<HierarchicalDataTemplate
 DataType="{x:Type local:Department}"
 ItemsSource="{Binding Path=Projects}">
    <TextBlock Text="{Binding Path=Name}" Foreground="Blue"/>
</HierarchicalDataTemplate>
```

OrgChartTreeView

This template's `ItemsSource` attribute makes the `TreeView` use the `Department`'s `Projects` property to build child nodes.

BINDING MASTER-DETAIL DATA

The OrgChartTreeView example program shown in Figure 18-12 displays hierarchical data in a tree-like format, but there are other ways to display this kind of data.

For example, the OrgChartMasterDetail example program shown in Figure 18-13 displays a list of regions in its upper-left `ListBox`. When you click on one, the program displays a list of departments in that region. When you select a department, the program displays a list of managers in that department. Finally, when you select a manager, the program displays a list of employees who report to that manager. The `ListBoxes` on the right display the same data, except they follow department projects instead of managers.

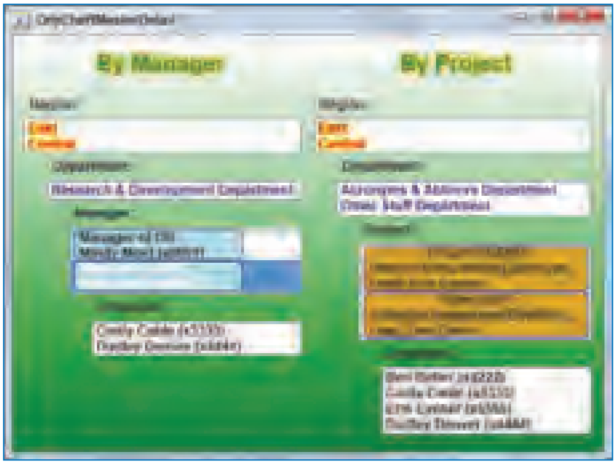


FIGURE 18-13

When the OrgChartMasterDetail program starts, it builds the same `Region`, `Department`, `Manager`, `Project`, and `Employee` objects used by the OrgChartTreeView program. It then uses the following code to make two `StackPanel` controls use the list of `Regions` as their `DataContexts`. The controls they contain inherit those `DataContext` values.

```
spByManager.DataContext = regions;
spByProject.DataContext = regions;
```

The following code shows how the program defines its first `Region` `ListBox`:

Available for download on Wrox.com

```
<ListBox Name="lstByManagerRegion" ItemsSource="{Binding}">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding Path=RegionName}" Foreground="Red">
                <TextBlock.BitmapEffect>
                    <OuterGlowBitmapEffect/>
```

```
                </TextBlock.BitmapEffect>
            </TextBlock>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

OrgChartMasterDetail

The only surprise here is the `ItemsSource` value, `{Binding}`. That value makes the control build its list of items from whatever object is stored in its `DataContext` property. It inherits that value from the `StackPanel` that holds it, and that value is the list of `Region` objects built by the code-behind.

The following code shows how the first `Department ListBox` displays the departments that correspond to the selected region:

Available for download on Wrox.com

```
<ListBox Name="lstByManagerDepartment" Margin="20,0,0,0"
 ItemsSource="{Binding ElementName=lstByManagerRegion,
              Path=SelectedItem.Departments}">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <TextBlock Text="{Binding Path=Name}" Foreground="DarkBlue"/>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

OrgChartMasterDetail

This control's `ItemsSource` property is bound to the `Departments` property of the `lstByManagerRegion ListBox`'s currently selected item. When you select a region from the `lstByManagerRegion ListBox`, this `ListBox` fills itself with the objects stored in that item's Departments list.

The bindings for the other master-detail relationships (department/manager, manager/employee, department/project, project/employee) work similarly. The `ListBoxes` have an `ItemsSource` property that uses a binding with `ElementName` set to the master `ListBox` and `Path` set to the appropriate list property of the currently selected item.

BINDING XAML

The OrgChartTreeView program shown in Figure 18-12 and the OrgChartMasterDetail program shown in Figure 18-13 build their data in code. The programs' `Window_Loaded` event handlers create the objects that represent the organizational chart. The following code shows a tiny part of code shared by those routines:

Available for download on Wrox.com

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Define Employees.
    Employee emp_a = new Employee()
        {FirstName = "Alice", LastName = "Able", Extension = "1111"};
    Employee emp_b = new Employee()
        {FirstName = "Ben", LastName = "Better", Extension = "2222"};
    ...
```

```
        // Define Managers.
        Manager mgr_a = new Manager()
            {FirstName = "Mindy", LastName = "Most",
             Extension = "0001", Title = "Manager of OR"};
        ...
        mgr_a.Reports.Add(emp_a);

        ... Lots of code omitted ...

        // Make a list of the regions.
        List<Region> regions = new List<Region>();
        regions.Add(div_work);
        regions.Add(div_acro);

        // Make the TreeView controls display this list.
        trvByManager.ItemsSource = regions;
        trvByProject.ItemsSource = regions;
    }
```

OrgChartTreeView **and** *OrgChartMasterDetail*

The code starts by creating the objects that the `TreeView` will display. For example, the previous code shows how the programs create `Employee` and `Manager` objects, and add an `Employee` to a `Manager`'s `Reports` list. Similarly, the code creates `Project`, `Department`, and `Region` objects and connects all of the objects by adding them to each other's list properties.

The code finishes by making a list containing the two `Region` objects that it has created and setting the `TreeView` controls' `ItemsSource` properties to that list.

The rest of the code is long but fairly straightforward. Download the example program from the book's web site to see the details.

Creating complex data structures like this one in code-behind is common, but it is not the only possible approach. One alternative is to build the objects in XAML code.

To do this, start by including a namespace declaration similar to the following one to define your project's namespace for use in XAML. This statement means that you can use the prefix *local* to refer to classes defined in your code-behind.

```
xmlns:local="clr-namespace:OrgChartTreeView"
```

Now the XAML code can use the code-behind classes as data types to define the data you need. The OrgChartXaml example program demonstrates this approach. For example, the following XAML code shows how the program defines a `Manager` object:

Available for download on Wrox.com

```
<local:Manager FirstName="Mindy" LastName="Most" Extension="0001"
 Title="Manager of OR">
    <local:Manager.Reports>
        <local:Employee FirstName="Alice" LastName="Able" Extension="1111"/>
        <local:Employee FirstName="Ben" LastName="Better" Extension="2222"/>
    </local:Manager.Reports>
</local:Manager>
```

OrgChartXaml

The code starts with a `local:Manager` element that uses attributes to set the object's `FirstName`, `LastName`, `Extension`, and `Title` properties.

The `Manager` class has a `Reports` property that holds an array of `Employee` objects. (XAML doesn't support the `List<Employee>` data type, so I converted the code-behind class definitions to use arrays for this example.) The XAML code sets the `local:Manager.Reports` property to an array of `Employee` objects.

The OrgChartXaml program uses its Windows.Resources section to create an array named `regions` containing two `local:Region` objects. Later it's easy to use this array as a data source for the program's `TreeView` controls as shown in the following code:

Available for download on Wrox.com

```
<TreeView Name="trvByManager" Background="Transparent"
 Grid.Row="1" Grid.Column="0" ItemsSource="{StaticResource regions}">
    ...
</TreeView>
```

OrgChartXaml

The rest of the program's code is long but reasonably straightforward. The program's `HierarchicalDataTemplates` are exactly the same as those used by the OrgChartTreeView program. You can download the project from the book's web site to see the remaining details.

NOT QUITE THE SAME

Although the OrgChartTreeView and OrgChartXaml programs look like they produce the same result, the actual data is slightly different. The OrgChartTreeView program reuses the same objects several times. For example, the employee Alice Able is in two Managers' Reports collections, is TeamLead for project ACROBAT, and is a TeamMember for that project. The OrgChartTreeView program creates one `Employee` object to represent Alice and then adds it in all of those locations.

In the OrgChartXaml program, I was unable to use the same object in multiple places in this manner. The obvious approach would be to create an `Employee` resource representing Alice and then use it in the appropriate places. Unfortunately, if you try this, XAML complains that you are trying to add a `StaticResourceExtension` object to an array that should contain Employees. To avoid this problem, the program uses multiple Employee objects that contain the values for Alice.

If you figure out how to share objects in this way, let me know at `RodStephens@vb-helper.com`.

BINDING XML

The previous two sections explain how to build objects from classes defined in the code-behind and bind them to controls such as `TreeView`. A third approach builds the objects as XML data stored completely in the XAML code.

When you use this approach, you can simply define the data by typing in elements and their attributes, making up their names as you go. For example, the following code defines a `Manager` object containing two `Employee` objects:

Available for download on Wrox.com

```
<Manager FirstName="Mindy" LastName="Most" Extension="0001" Title="Manager of OR">
    <Employee FirstName="Alice" LastName="Able" Extension="1111"/>
    <Employee FirstName="Ben" LastName="Better" Extension="2222"/>
</Manager>
```

OrgChartXml

Neither the XAML code nor the code-behind needs to include definitions for the `Manager` and `Employee` classes. In fact, WPF doesn't actually create `Manager` and `Employee` objects. Instead, it creates `XmlElement` objects with `Name` properties set to `Manager` and `Employee`.

TERRIBLE TYPOS

Be very careful when you type XAML data. If you misspell an element's name, WPF merrily creates a new object with the misspelled name. For example, if you accidentally make an `Empoyee` element, WPF will create an `XmlElement` named `Empoyee`. Later, when a `HierarchicalDataTemplate` searches for `Employee` objects (described next), it will miss this one.

To use this XML data, you need to wrap it in an `XmlDataProvider` that you can then bind to a `TreeView` control. The `XmlDataProvider` should contain an `x:XData` element that holds the actual data.

The following code shows the high-level XML elements used by the OrgTreeXml program:

Available for download on Wrox.com

```
<XmlDataProvider x:Key="regions" XPath="Regions">
    <x:XData>
        <Regions xmlns="">
            <Region RegionName="East">
                ...
            </Region>
            <Region RegionName="Central">
                ...
            </Region>
        </Regions>
    </x:XData>
</XmlDataProvider>
```

OrgTreeXml

The `XmlDataProvider`'s `XPath` property defines the `XPath` within the data that should be used as the root of the provider's data source. Usually `XPath` is set to the root element of the XML data, which sits just inside the `XData` element. In this example, that element is named `Regions`.

If you wanted to use only some of the data, however, you could change `XPath`. For example, the value `Regions/Region[2]` would make the provider start gathering data at the second `Region` element inside the `Regions` element.

Note that indexing in XML elements like this starts at 1 not 0, so `Region[2]` *is the second* `Region` *element, not the third.*

The XML items inside the `x:XData` element can redefine XML namespaces, but usually it's easiest to reset the namespace for all of the elements to an empty string so that you don't need to worry about using them later.

EXTERNAL XML

You can also use an `XmlDataProvider` to refer to a separate file that contains the data as in:

```
<XmlDataProvider x:Key="OrgChartData" Source="orgchart.xml"
 XPath="Regions"/>
```

When you bind `TreeViews` and other controls to XML data in this way, you must make a couple of important changes to the way templates refer to the data.

First, because the XML code defines its own data types, you don't need to use a namespace prefix to refer to classes defined in the code-behind.

Second, because the XML objects are `XmlElements` and not classes defined in the code-behind, you should use `XPath` instead of `Path` syntax.

For example, the following code shows the `HierarchicalDataTemplate` that the OrgChartXml program uses for `Region` elements:

Available for download on Wrox.com

```
<HierarchicalDataTemplate
 DataType="Region"
 ItemsSource="{Binding XPath=*}">
    <TextBlock Text="{Binding XPath=@RegionName}" Foreground="Red">
        <TextBlock.BitmapEffect>
            <OuterGlowBitmapEffect/>
        </TextBlock.BitmapEffect>
    </TextBlock>
</HierarchicalDataTemplate>
```

OrgChartXml

In this template, the `DataType` is the simple type `Region` rather than the more complex `local:Region`.

`ItemsSource` uses the `XPath` value `*`, indicating that the node's children should include all of the children of the current node.

Finally, the `TextBlock`'s `Text` property is bound to `@RegionName`. When the previous programs referred to a `Region` object, that object's region name was stored in its `RegionName` property.

Because the XML version stores this value in an attribute, the `XPath` syntax must use the @ symbol to access attribute values.

Compare the preceding code to the following `HierarchicalDataTemplate`, which is used by the earlier programs in this chapter:

Available for download on Wrox.com

```
<HierarchicalDataTemplate
 DataType="{x:Type local:Region}"
 ItemsSource="{Binding Path=Departments}">
    <TextBlock Text="{Binding Path=RegionName}" Foreground="Red">
        <TextBlock.BitmapEffect>
            <OuterGlowBitmapEffect/>
        </TextBlock.BitmapEffect>
    </TextBlock>
</HierarchicalDataTemplate>
```

OrgChartXml

BINDING DATABASE OBJECTS

The sections earlier in this chapter explain how to bind controls to objects created in code-behind, objects defined in XAML code, and objects defined by XML code. These scenarios are all useful, but many applications store their data in relational databases.

ADO.NET provides objects that interact with relational databases. Because they are objects, you can bind them to controls much as you can bind the objects described earlier in this chapter. Because these objects are more complicated than those you would typically build yourself, binding them is a little more complicated.

The basic strategy is to open the database in code-behind as you would for any other ADO.NET application. Load data into `DataSet`, `DataTable`, and other ADO.NET objects, and then bind them to the application's controls.

You can think of a `DataTable` as similar to a collection of `DataRow` objects and a `DataSet` as similar to a collection of `DataTables`, so the binding syntax isn't totally unfamiliar.

FIGURE 18-14

The StudentData example program shown in Figure 18-14 demonstrates ADO.NET binding. This program loads its data from the StudentData .mdb Microsoft Access database.

The `ListBox` on the left lists the students in the `Students` table. When you click on a student, the controls in the middle display the student's address and phone number, and the `ListBox` on the right displays the corresponding test score data from the `TestScores` table.

Note that you don't need to have Access installed on your computer to use an Access database in your program. You only need a database engine that can open the database, such as the Jet database engine.

If you modify any of the student values in the middle, the controls automatically update the corresponding ADO.NET objects. If you use the Data menu's "Save Changes" command, the program writes those changes back into the database. (This example doesn't provide a way to add or delete students, or to add or modify a student's test scores.)

The database has a third table named `States` that lists the U.S. states (plus Washington, DC) and their abbreviations. The program uses this table as a lookup table. The `Students` table contains each student's state as an abbreviation. When it displays that value, the `ComboBox` in the middle of the window uses the `States` table to convert the abbreviation into the state's full name.

The following sections explain how the program performs its major tasks.

Loading Data

The following code shows how the StudentData program loads its data from the database:

Available for download on Wrox.com

```
OleDbDataAdapter m_daUsers;
DataSet m_dsStudentData;

private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Connect to the database.
    OleDbConnection conn = new OleDbConnection(
        "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=StudentData.mdb");

    try
    {
        // Open the connection.
        conn.Open();

        // The DataSet to hold all of the data.
        m_dsStudentData = new DataSet();

        // Load Students table.
        m_daUsers = new OleDbDataAdapter();
        m_daUsers.SelectCommand =
            new OleDbCommand(
                "SELECT * FROM Students ORDER BY FirstName, LastName", conn);
        m_daUsers.Fill(m_dsStudentData, "Students");

        // Load TestScores table.
        OleDbDataAdapter daTestScores = new OleDbDataAdapter();
        daTestScores.SelectCommand =
            new OleDbCommand("SELECT * FROM TestScores", conn);
        daTestScores.Fill(m_dsStudentData, "TestScores");
```

```
            // Load States table.
            OleDbDataAdapter daStates = new OleDbDataAdapter();
            daStates.SelectCommand =
                new OleDbCommand("SELECT * FROM States ORDER BY StateName", conn);
            daStates.Fill(m_dsStudentData, "States");

            // Relation: States.State = Students.State.
            m_dsStudentData.Relations.Add(
                "relStates_Students",
                m_dsStudentData.Tables["States"].Columns["State"],
                m_dsStudentData.Tables["Students"].Columns["State"]);

            // Relation: Students.StudentId = TestScores.StudentId.
            m_dsStudentData.Relations.Add(
                "relStudents_TestScores",
                m_dsStudentData.Tables["Students"].Columns["StudentId"],
                m_dsStudentData.Tables["TestScores"].Columns["StudentId"]);

            // Set the Window's DataContext to the DataSet.
            this.DataContext = m_dsStudentData;
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
        finally
        {
            // Close the connection.
            conn.Close();
        }
    }
```

StudentData

USING USING

The code also uses the following `Using` statements to make using the ADO.NET objects easier. (The Visual Basic version uses `Imports` statements.)

```
using System.Data;
using System.Data.OleDb;
```

Depending on the kind of database you use, you may need to use other namespaces and may need to add references to the project.

The code defines an `OleDbDataAdapter` and `DataSet` at the module level so they will be easier to use later to save changes to the data.

When the program's window loads, the `Window_Loaded` event handler creates a new connection to the database. It opens the database and instantiates the `DataSet`.

Next, for each of the three tables (Students, TestScores, and States), the program creates a data adapter and uses it to load the data from the database table into the DataSet. The second parameter to the Fill method is the name of the DataTable to be filled in the DataSet.

The program then defines data relations between the States and Students tables and between the Students and TestScores tables. The first parameter used to create the relations gives the relations' names and will be important later in data binding.

The first relation means that each Students.State value must appear somewhere in a States.State value. For example, this prevents you from changing a Students record's State value to XY because the States table doesn't hold that value.

The second relation means that each TestScores.StudentId value must appear in a Students.StudentId value. This prevents you from adding a TestScores record without a corresponding Students record. It also provides the link for use by the master-detail relationship between Students and TestScores records.

DISTANT RELATIONS

The database also defines these relations between its tables so it can enforce these restrictions itself. If your program tried to violate the relations, the database would throw a tantrum.

After it finishes loading the data and defining the relations, the code sets the window's DataContext to the DataSet and closes the database connection.

Saving Changes

The following code shows how the program saves changes to the Students table:

Available for download on Wrox.com

```
// Save the changes to the Students table.
private void mnuDataSaveChanges_Click(object sender, RoutedEventArgs e)
{
    // Shameless kludge to move focus off of the current control
    // so any pending changes are written into the DataSet.
    IInputElement has_focus = FocusManager.GetFocusedElement(this);
    lstStudents.Focus();
    has_focus.Focus();

    // Make a command builder to generate the UPDATE command.
    OleDbCommandBuilder cb = new OleDbCommandBuilder(m_daUsers);

    // Update the table.
    m_daUsers.Update(m_dsStudentData, "Students");

    MessageBox.Show("Changes saved.", "Changes Saved",
        MessageBoxButton.OK, MessageBoxImage.Information);
}
```

StudentData

The code starts with a hack to save any edit that is in progress.

Suppose you are typing a new `Street` value in its `TextBox` when you invoke the Data menu's "Save Changes" command. At that point WPF has not yet notified the `DataSet` of your edit in progress, so that change isn't saved into the database. To force WPF to make that notification, the program saves a reference to the control that currently has focus, moves focus off that control, and then moves it back.

Next, the code creates a command builder to generate any commands that the data adapter might need to update the database. Finally, the code calls the data adapter's `Update` method to save the changes.

That's it for the code-behind. The rest of the program is in XAML code. Naturally, the most interesting pieces are the bindings.

Binding the Student Name ListBox

The following code shows how the program binds the student name `ListBox` on the left in Figure 18-14 to the `Students` data:

Available for download on Wrox.com

```
<ListBox Grid.Row="0" Grid.Column="0" Margin="10" Name="lstStudents"
 IsSynchronizedWithCurrentItem="True"
 DataContext="{Binding Tables[Students]}"
 ItemsSource="{Binding}">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <Grid>
                <StackPanel Orientation="Horizontal" Margin="3,0,3,0">
                    <TextBlock Text="{Binding Path=FirstName}" />
                    <TextBlock Text=" " />
                    <TextBlock Text="{Binding Path=LastName}" />
                </StackPanel>
            </Grid>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

StudentData

The code sets the control's `DataContext` attribute to the `Tables` entry named *Students*. Remember that at this point the window's `DataContext` property points to the `DataSet`. The `DataSet` has a `Tables` property that contains references to the `DataTable` objects it contains. The syntax `Tables[Students]` shown here returns a reference to the `DataTable` named *Students*.

WHAT'S IN A NAME?

The name *Students* used here is the name of the `DataTable` in the `DataSet`, not the name of the table in the database. To make things easier to understand, database code typically gives a `DataTable` the same name as the table it represents, but that's not required. The `DataTables`' names are set in the calls to the data adapters' `Fill` methods.

The code then sets the ListBox's ItemsSource attribute to {Binding}. This rather odd syntax makes the control bind to whatever is in its DataContext property. In this case, that means the Students DataTable.

The ListBox's ItemTemplate property element defines a template to use when displaying the items in the DataTable. In this example, it simply displays the FirstName and LastName fields separated by a space. (Note that the binding sources are simply the names of the database fields. There's plenty of hard stuff in WPF, but this part is easy.)

Displaying Student Details

When you click on an entry in the student list, the program displays the corresponding address and phone data in the middle controls.

Displaying the data in the TextBoxes is easy. Recall that the TextBoxes inherit the window's DataContext, so their DataContexts also point to the DataSet. All a TextBox needs to do is bind to the appropriate field in the Students DataTable.

The following code shows how the program makes its Street TextBox. The other TextBoxes are defined similarly.

```
<TextBox Grid.Row="0" Grid.Column="1" Margin="3"
 Text="{Binding Path=Tables[Students]/Street}"/>
```

When you click on a student in the list, the Tables[Students] DataTable's current item changes, and the Street TextBox displays the newly selected record's Street value.

The State ComboBox is trickier than the TextBoxes because it uses both the Students table and the States lookup table. The following code shows how the program makes its State ComboBox:

Available for download on Wrox.com

```
<ComboBox Grid.Row="2" Grid.Column="1" Margin="3"
 IsSynchronizedWithCurrentItem="True"
 ItemsSource="{Binding Path=Tables[States]}"
 DisplayMemberPath="StateName"
 SelectedValuePath="State"
 SelectedValue="{Binding Path=Tables[Students]/State}"
 />
```

StudentData

This control uses four attributes to determine the values that it represents and displays. The following list describes these properties:

- ItemsSource — This gives the source that the control uses to build its list of choices. In this example, the control displays items from the States DataTable.
- DisplayMemberPath — This gives the path in the ItemsSource to the value that is displayed by the control. In this program, the ComboBox displays the state names, not their abbreviations, so DisplayMemberPath is StateName, the name of the field in the States DataTable that holds the state names.

- SelectedValuePath — This gives the path in the ItemsSource to the value that is stored and manipulated by the control. In this program, the Students DataTable stores each student's state abbreviation, not its full name, so SelectedValuePath is the field that gives the abbreviation: State.
- SelectedValue — This gives the value that the control displays and modifies. In this case, the ComboBox displays the value for the current Students DataTable's State value.

To better understand how this works, suppose you select a student who has State = AZ. The ComboBox's SelectedValue binding gets the value of Tables[Student].State, which is AZ. The control then looks in its ItemsSource (which is Tables[States]) for the field referred to by SelectedValuePath (which is State). It finds the value that matches AZ, looks up the corresponding DisplayMemberPath value (Arizona), and displays it.

If you select a new value from the ComboBox, the process reverses. The control gets the value you selected and looks it up in the ItemsSource's DisplayMemberPath field. It gets the corresponding SelectedValuePath value and saves it as the SelectedValue.

Binding the Scores ListBox

The final part of the StudentData program binds the ListBox on the right so it displays the TestScores values for the currently selected student. The following code shows how the program defines this ListBox:

Available for download on Wrox.com

```
<ListBox Name="lstTestScores" Grid.Row="0" Grid.Column="2" Grid.RowSpan="100"
 Margin="10,3,0,0" IsSynchronizedWithCurrentItem="True"
 DataContext="{Binding Path=Tables[Students]}"
 ItemsSource="{Binding Path=relStudents_TestScores}"
 >
    <ListBox.ItemTemplate>
        <DataTemplate>
            <Grid>
                <StackPanel Orientation="Horizontal" Margin="3,0,3,0">
                    <TextBlock Text="Test: " />
                    <TextBlock Text="{Binding Path=TestNumber}" />
                    <TextBlock Text=", Score: " />
                    <TextBlock Text="{Binding Path=Score}" />
                </StackPanel>
            </Grid>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

StudentData

Like the student names ListBox, this control's DataContext is set to the Students DataTable.

The code sets the control's ItemsSource property to the relation defined in the code-behind between the Students and TestScores DataTables. When you click on a student in the students list, the Students DataTable gets a new current record. The control uses the relation to find the corresponding entries in the TestScores DataTable and displays them.

The ListBox's ItemTemplate property determines how the control displays its TestScores values.

SUMMARY

WPF provides a very powerful model for data binding. It lets you bind a control's properties to:

- Another control's properties
- Objects defined in code-behind
- Objects built in XAML code
- Objects defined by XML data
- Database objects such as ADO.NET `DataSet` and `DataTable` objects

Bindings let you display many items stored in arrays, collections, and other data structures in lists, hierarchical displays, and master-detail displays.

Chapter 19 explains a different kind of binding. It explains how you can bind commands to program behaviors. For example, it shows how to make the program respond when the user performs a "Save" action whether the user invokes that action from a menu command, a shortcut key, a toolbar button, or by any other method.

19

Commanding

Many applications provide several ways to perform the same function. For example, in many applications you can copy the text selected in a textbox by any of the following actions:

- Press [Ctrl]+C.
- Right-click on the textbox and select the context menu's Copy command.
- Open the Edit menu and select Copy.
- Click the Copy button on the toolbar.

In all of these cases, the program performs the same action and copies the selected text.

Rather than using different pieces of code to perform the same action for each of these input scenarios, a well-designed application calls the same code for each. That reduces repeated code, so it means less code to write, debug, and maintain.

WPF formalizes this idea of providing centralized actions with *commanding*. Commanding separates the logical intent of an action (such as copying text) from the code that implements it.

Figure 19-1 shows three different approaches to handling text copying. In the bad design on the left, different input actions invoke different pieces of code. This design is bad because it contains several copies of the same code, making it harder to debug and maintain the code.

In the design in the middle, all of the input methods invoke the same piece of code. This design is better because all of the text copying code is stored in a central routine where it's easy to debug, modify, and maintain.

The design on the right illustrates commanding. All of the input methods invoke the same command object, which, in turn, calls the appropriate code-behind. This provides roughly the same features as the middle design with an extra layer added. In addition to encouraging centralized processing, commanding provides a few extra features. For example, a command can automatically display standard keyboard shortcuts for menu items and for managing enabling and disabling all of the input actions that activate the command.

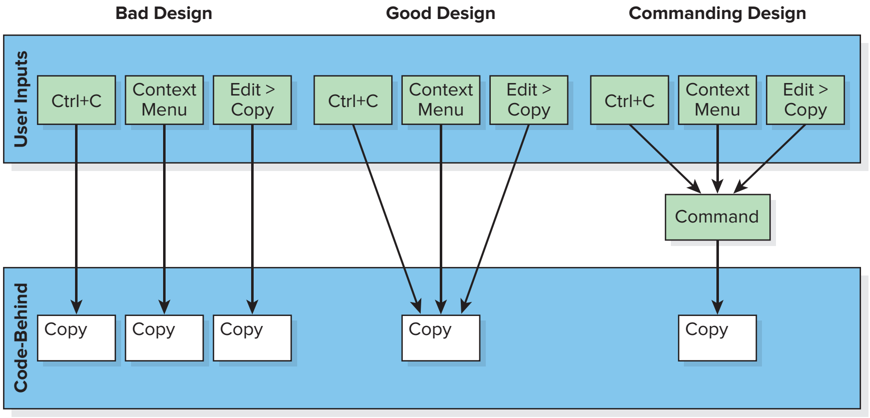


FIGURE 19-1

This chapter describes commanding. It explains how to use standard predefined commands that are useful in many applications, and it shows how to create custom commands for actions that are unique to your application. It also shows how to bind commands to user inputs such as menu selection, button clicks, and keyboard sequences.

COMMANDING CONCEPTS

Commanding is based on four key concepts:

- **Command** — This object represents the logical action that should take place. The action is something conceptual such as copying text, opening a document, or printing.
- **Command Source** — This is an object that invokes the command. This could be a menu item, button, or key sequence.
- **Command Target** — This is the object upon which the command is executed. For example, a `Copy` command might be executed on a `TextBox`.
- **Command Binding** — This is the object that maps code to the command.

For example, suppose your program allows the user to start a new document (text file, drawing, or whatever) by selecting the File menu's New command. In this example:

- The *command* is the concept of starting a new document.
- The *command source* is the File menu's New menu item.
- The *command target* is the object that creates the new document, possibly the window holding the menu.
- The *command binding* maps the menu item to the code in the command target's class that creates the new document.

Not only does commanding make it easy for the code to handle an operation in a single centralized place, but the command object also provides a central point for enabling and disabling the controls that trigger it. For example, consider a copy command. If no text is selected that could be copied, then the command object can disable the [Ctrl]+C, context menu, Edit ➪ Copy, and any other actions that could trigger a copy.

WPF predefines many command objects that you can bind to your controls to define actions. Some even have a special relationship with other controls so they can automatically perform actions for you. Finally, you may need to define new command objects to handle other actions that were not anticipated by WPF.

The following sections describe these three groups of command objects: those predefined by WPF that can take automatic action, those predefined by WPF that cannot take automatic action, and custom commands that you define in your code.

PREDEFINED COMMANDS WITH ACTIONS

Some control classes automatically implement command bindings for you. For example, the `TextBox` control knows how to provide copy, cut, paste, undo, and redo actions. If the program executes the predefined `ApplicationCommands.Copy` command while a `TextBox` has focus, then the `TextBox` copies its selected text to the clipboard.

Furthermore, all you need to do is tell a menu item or button that its command is `ApplicationCommands.Copy`, and the control will automatically trigger the appropriate action as needed. You don't need to write a single line of code-behind!

The TextBoxCommands example program shown in Figure 19-2 demonstrates predefined commands that are supported by the `TextBox` control. Partly hidden by the program's Edit menu, a toolbar contains Copy, Cut, Paste, Undo, and Redo commands similar to those shown in the menu.

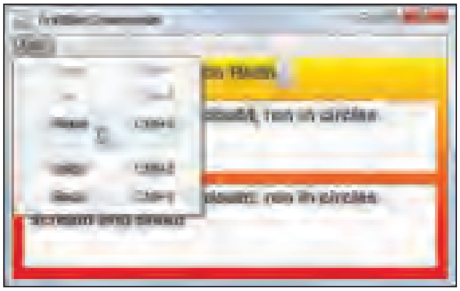


FIGURE 19-2

The Copy and Cut menu commands and toolbar buttons are disabled because the textbox that has the focus (you can see the focus caret just after the word *doubt* in the upper textbox) has no text selected so those commands don't make sense. The Paste command is enabled because there was text in the clipboard when that screenshot was taken so you could paste it into the `TextBox`. The Undo and Redo commands are always enabled. (It would be better if those command objects enabled and disabled themselves appropriately depending on whether the `TextBox` had changes that could be undone or redone but currently they don't.)

The following code shows how the program creates its menu:

Available for download on Wrox.com

```
<Menu DockPanel.Dock="Top">
    <MenuItem Header="Edit">
        <!--  Menu items that invoke default commands.  -->
        <MenuItem Command="ApplicationCommands.Copy"/>
```

```
            <MenuItem Command="ApplicationCommands.Cut"/>
            <MenuItem Command="ApplicationCommands.Paste"/>
            <Separator/>
            <MenuItem Command="ApplicationCommands.Undo"/>
            <MenuItem Command="ApplicationCommands.Redo"/>
        </MenuItem>
    </Menu>
```

TextBoxCommands

The code sets each `MenuItem`'s `Command` property to the name of a predefined application command.

SHORTER NAMES

The text *ApplicationCommands.* before the command name is optional, so, for example, you could call the Copy command *Copy* instead of *ApplicationCommands.Copy*. Writing the full name makes it more obvious that these are application commands but you can use whichever style you think is easier to read.

The `MenuItems` do the rest automatically. They automatically display the appropriate text and shortcut key combinations. When you invoke one of the menu items, it activates the corresponding command object and that object makes the `TextBox` with the focus perform the appropriate action.

COMMANDING THE EASY WAY

This is the easiest commanding scenario. It uses a predefined command, a control that has built-in support for the command (`TextBox`), and a control that has built-in support for representing and invoking commands (`MenuItem`).

The following code shows how the program makes its toolbar:

Available for download on Wrox.com

```
<ToolBarTray DockPanel.Dock="Top" Background="Transparent">
    <ToolBar Band="1" BandIndex="1" Background="Transparent">
        <Button Command="Copy"  Content=
         "{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
        <Button Command="Cut"   Content=
         "{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
        <Button Command="Paste" Content=
         "{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
        <Separator/>
        <Button Command="Undo"  Content=
         "{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
        <Button Command="Redo"  Content=
         "{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
     </ToolBar>
</ToolBarTray>
```

TextBoxCommands

Like the `MenuItems`, the toolbar's `Buttons` have `Command` properties set to the names of the commands they execute. Unlike `MenuItems`, however, `Buttons` do not automatically display the commands' text and shortcut keys, so the program includes XAML code that sets the `Buttons`' text.

Each `Button`'s `Content` property is set to a binding that makes WPF look at the `Button`, get its `Command` property, and get that object's `Text` property. That makes the `Button` display the appropriate text for its command.

SETTING BUTTON TEXT

You could set each `Button`'s text explicitly to Copy, Cut, and so forth, but it would add an extra chance for you to introduce bugs with the text not matching the command's name.

Left to its own devices, commands like `ApplicationCommands.Copy` enable their sources when any appropriate control makes the operation possible. For example, the `Copy` command activates its controls when the textbox that has the focus has text selected.

If you want, you can set a command's target to tie it to a specific control. For example, you can use a textbox as a button's command target. Then the command enables or disables its menu item, button, and other controls depending on whether text is selected in that particular textbox.

The CommandTarget example program shown in Figure 19-3 demonstrates command targets.

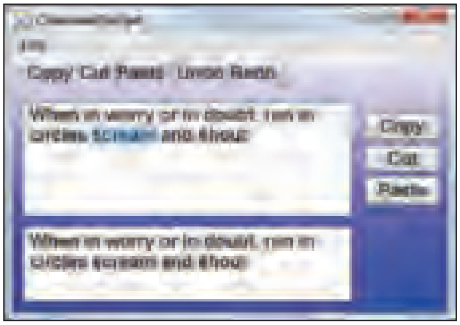


FIGURE 19-3

Many of the program's controls are similar to those used by the TextBoxCommands program. The program's Edit menu and toolbar contain Copy, Cut, Paste, Undo, and Redo items that use the appropriate predefined `ApplicationCommands`.

The program also provides Copy, Cut, and Paste buttons that have the upper textbox in Figure 19-3 as their command targets. Their commands enable and disable these controls depending on the state of that textbox.

The following code shows how the CommandTarget program defines its new buttons. You can see that the buttons' `CommandTarget` properties are set to the upper textbox `txtTop`.

Available for download on Wrox.com

```
<StackPanel Grid.Row="0" Grid.Column="1" Margin="0,0,5,0"
 VerticalAlignment="Center">
    <!--  Buttons that invoke commands on txtTop.  -->
    <Button Grid.Row="0" Grid.Column="1" Margin="2"
     Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"
     Command="ApplicationCommands.Copy"
     CommandTarget="{Binding ElementName=txtTop}"/>
    <Button Grid.Row="0" Grid.Column="1" Margin="2"
     Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"
     Command="ApplicationCommands.Cut"
```

```
        CommandTarget="{Binding ElementName=txtTop}"/>
       <Button Grid.Row="0" Grid.Column="2" Margin="2"
        Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"
        Command="ApplicationCommands.Paste"
        CommandTarget="{Binding ElementName=txtTop}"/>
   </StackPanel>
```

CommandTarget

In Figure 19-3, the lower textbox has the input focus and has no text selected, so the Copy and Cut toolbar buttons are disabled (as are the corresponding menu items). However, there is text selected in the upper textbox, so the Copy and Cut buttons on the right are enabled. (That textbox doesn't have the focus so you can't tell that it has text selected. You'll have to trust me on this.) If you clicked on the Paste button, whatever text is currently in the clipboard would be pasted into the upper textbox. If you clicked on the Cut button, the selected text in the upper textbox would disappear.

BAFFLING BINDINGS

Binding commands to particular controls in this way can be very confusing. In Figure 19-3, there's no indication that the buttons only work with the upper textbox except for the fact that they are sitting next to it. There's no indication that the toolbar buttons and menu items apply to either textbox.

To avoid confusion, think carefully before you bind commands to some controls but not to other, similar controls. Try to make the interface as consistent as possible. For example, give similar buttons to each textbox, disable the buttons when their textbox doesn't have focus, or forget the whole thing and stick with the normal Copy, Cut, and Paste menu items and toolbar buttons that apply to every textbox.

PREDEFINED COMMANDS WITHOUT ACTIONS

Although a few command objects can automatically take action when they are invoked, many cannot. Those objects define a logical action, but your code must determine what the action means to your application.

For example, the Open, New, Save, Save As, and Close commands all make sense for a document-oriented application, but the specific actions they perform will differ depending on whether your program is a text editor, drawing tool, or report generator.

To attach actions to one of these predefined commands, a program can create a *command binding*. A command binding tells WPF what code to run to determine whether the command should be allowed and what code to run when the command executes.

The DocumentCommands example program shown in Figure 19-4 demonstrates several document-oriented commands that are predefined without automatic actions. The program lets you click-and-drag to draw rectangles with random colors.

The program's New, Open, Save, Save As, and Close commands work more or less as you would expect. For example, if you open a document and draw a new rectangle on it, then the Save and Save As commands are enabled, so you can save your changes.

The drawing shown in Figure 19-4 was a new one that had not yet been saved. Because it did not yet have a filename, the Save command is disabled and the Save As command is enabled.

FIGURE 19-4

One feature of this program that is not common among document editors is that it will not let you use the New or Open commands if there are unsaved changes to the current document. Many other programs let you invoke New or Open and then warn you if the current document has unsaved changes. The DocumentCommands program makes you save any changes or close the document before using New or Open. (I did it this way to make the commands more interesting.)

The following XAML code shows how the program binds the New, Open, Save, and Save As commands to code-behind. The code-behind creates the binding for the Close command to show how it's done in code.

Available for download on Wrox.com

```
<Window.CommandBindings>
    <CommandBinding Command="New"
     CanExecute="DocumentNewAllowed" Executed="DocumentNew"/>
    <CommandBinding Command="Open"
     CanExecute="DocumentOpenAllowed" Executed="DocumentOpen"/>
    <CommandBinding Command="Save"
     CanExecute="DocumentSaveAllowed" Executed="DocumentSave"/>
    <CommandBinding Command="SaveAs"
     CanExecute="DocumentSaveAsAllowed" Executed="DocumentSaveAs"/>
    <!--  The Close binding is defined in code
          just to show how it's done.
    <CommandBinding Command="Close"
     CanExecute="DocumentCloseAllowed" Executed="DocumentClose"/>
     -->
</Window.CommandBindings>
```

DocumentCommands

The Window.CommandBindings section contains the bindings. Note that you can put these bindings in a more restricted scope, for example, in a `Grid` or `StackPanel`, but usually it makes the most sense to handle commands at a global level. It can be confusing if some commands are only available from certain parts of the user interface.

Each `CommandBinding` object defines the command that it is binding. The `CanExecute` attribute gives the name of the function that decides whether the command should be enabled. The `Executed` attribute gives the name of the routine to run when the command executes.

The following code shows the code-behind for the New command:

Available for download on Wrox.com

```
// Return True if we can perform the New action.
private void DocumentNewAllowed(Object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = (!m_IsDirty);
```

```
}

// Perform the New action.
private void DocumentNew(Object sender, ExecutedRoutedEventArgs e)
{
    // Remove any existing controls on the Canvas.
    cvsDrawing.Children.Clear();

    // Display the canvas.
    borDrawing.Visibility = Visibility.Visible;
    m_FileName = null;
}
```

DocumentCommands

The `DocumentNewAllowed` routine sets `e.CanExecute` to `True` if the program should allow the New command. When this function sets `e.CanExecute` to `False`, the command object disables the menu item and toolbar button for this command. The function simply returns `True` if the `m_IsDirty` flag is `False`.

DOWN AND DIRTY

Without going into too much detail about how the program works, `m_IsDirty` is set to `True` whenever there are unsaved changes to the drawing. When the user clicks-and-drags to make a new rectangle, the variable is set to `True`. When the user opens an existing file, creates a new file, or saves changes, the program sets `m_IsDirty` to `False`.

The `DocumentNew` routine removes any rectangle children on the drawing canvas `cvsDrawing`. It ensures that the `Border` containing the canvas is visible and sets `m_FileName` to `null` to indicate that the new drawing doesn't have a filename yet.

SAFETY FIRST

Note that this code doesn't need any logic to determine whether it is safe to create a new document. For example, it doesn't check whether there is an existing document with unsaved changes. The `DocumentNewAllowed` routine only enables the New command when it is safe to make a new document, so the `DocumentNew` routine only executes when it is safe.

This is one of the benefits of using WPF commanding. Moving the "is allowed" logic into a separate routine simplifies the code that performs the action.

CUSTOM COMMANDS

WPF provides around 150 predefined commands, so in many cases, you can find an existing command object to suit your needs. There are classes for copy, cut, paste, zoom in, zoom out, print, help, and many other features that are common to many programs. Appendix K summarizes predefined command classes.

Occasionally, however, you may want to define a command that isn't common in other programs and for which WPF doesn't define a command class. In that case, you can define your own command objects. You can take two basic approaches to defining new command objects.

First, you can build a class that implements the `ICommand` interface. This isn't too hard, but it's harder than the other approach so it isn't discussed here.

Second, you can make an instance of the `RoutedCommand` or `RoutedUIEvent` class. These classes are mostly the same except the `RoutedUIEvent` class has a `Text` property that the `RoutedCommand` class lacks. That makes `RoutedUIEvent` a better choice if you want to display the command's text somewhere such as in buttons and menu items.

The basic steps for building a custom `RoutedUICommand` object aren't too complicated, but understanding what's going on can be tricky.

The command object that you define should be static (shared in Visual Basic) so all instances of the window will share the same instance of the command object. Because all instances of the window share the same command object, you may wonder if that object enables and disables all of the commands for every window, for example, if a single object determines whether the "Invert Image" command is enabled or disabled for every window. That would be bad because it would mean that different windows could not be in different states at the same time.

Fortunately, that's not the case. Although the object that represents a command is shared between windows, the `CanExecute` and `Executed` routines are not. WPF queries each window's version of the `CanExecute` routine so that each window can decide whether it should allow its command to execute.

The ImageColors example program shown in Figure 19-5 demonstrates custom command objects. Note that the top window's commands are enabled, but the bottom window's commands are disabled. Click on the New button to make a new window. Check the Enabled box to enable a window's commands.

Note also that the program's buttons and menu items display their accelerator keys (the underlined keys) and that the menu items display their shortcut key sequences (e.g., [Alt]+I for the Invert command). All of that is explained shortly.

So what do you need to do to make a custom command object? First, declare the object as static. WPF won't need to set the value of the object, so if you like, you can also declare it read-only. The following code shows how the ImageColors program declares its Invert command object:

```
public readonly static RoutedUICommand CommandImageInvert;
```

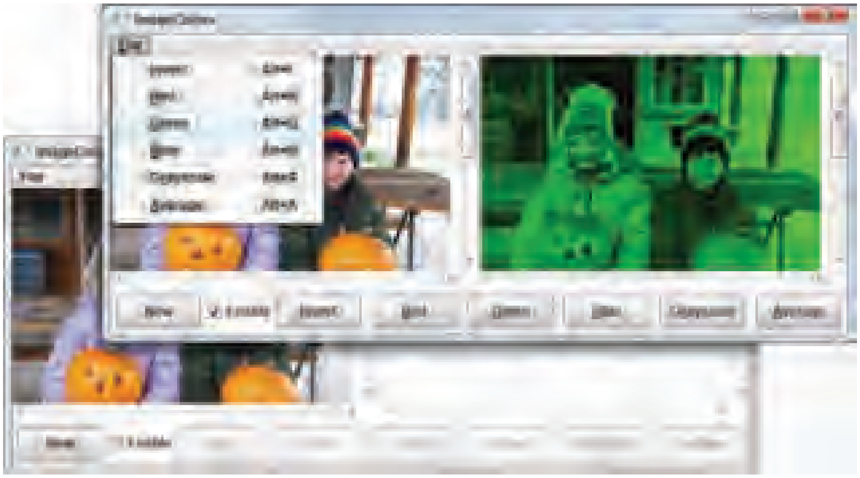


FIGURE 19-5

Next, write code to initialize the object. Because it is declared static, the program can only initialize the object in a static constructor. The following code shows how the ImageColors program initializes its Invert command:

Available for download on Wrox.com

```
static Window1()
{
    CommandImageInvert = new RoutedUICommand("_Invert",
        "Invert Command", typeof(Window1));
    CommandImageInvert.InputGestures.Add(
        new KeyGesture(Key.I, ModifierKeys.Alt));

        Code for other commands omitted
}
```

ImageColors

When the code creates the `RoutedUICommand`, it adds an underscore before the command's accelerator character. If you look closely at Figure 19-5, you'll see that the *I* is underlined in the Invert button and menu item.

This code also adds the [Alt]+I input gesture to the command. The menu item shown in Figure 19-5 automatically displays the gesture to the right.

LOTS OF STATIC

If you try to initialize the command object in a non-static constructor, Visual Studio complains at design time. If you try to initialize the command object in the window's `Loaded` event handler, the program fails at run time.

Now write the appropriate `CanExecute` and `Executed` routines in the code-behind. The ImageColors program uses the following `CanExecute` routine for all of its commands so they are all enabled when the window's checkbox is checked:

Available for download on Wrox.com

```
// Return True if the CheckBox is checked.
private void CanExecuteChecked(Object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = (chkEnable.IsChecked == true);
}
```

ImageColors

The Invert command's `Executed` method is interesting but not relevant to this discussion, so it isn't shown here. Download the example program from the book's web site to see how it works.

Next in the XAML code, add command bindings to tell the program which routines to use for the command's `CanExecute` and `Executed` properties. The following code shows how the ImageColors program binds its Invert command:

Available for download on Wrox.com

```
<Window.CommandBindings>
    <CommandBinding
        Command="{x:Static local:Window1.CommandImageInvert}"
        CanExecute="CanExecuteChecked"
        Executed="ExecuteImageInvert"/>
    <!--  ... Code for other commands omitted      -->
</Window.CommandBindings>
```

ImageColors

Finally, add the XAML code to display the buttons, menu items, and other controls that invoke the command.

The following code shows how the ImageColors program displays its Invert menu item. The `MenuItem` control automatically reads the command object's `Text` property to display its caption. It uses the command's input gesture to display the shortcut text [Alt]+I.

```
<MenuItem Command="{x:Static local:Window1.CommandImageInvert}"/>
```

The following code shows how the program displays its Invert button. Unlike the corresponding menu item, the button does not automatically get its caption from the command object, so the XAML code must set the button's caption explicitly.

Available for download on Wrox.com

```
<Button Name="btnInvert"
    Command="{x:Static local:Window1.CommandImageInvert}"
    Style="{StaticResource ButtonStyle}"
    Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
```

ImageColors

Notice how the code binds the button's `Content` to the command object's `Text` property. Because the code-behind added the underscore to the object's text, the button automatically underlines its accelerator when you press the [Alt] key and responds to the [Alt]+I key sequence.

The process seems complicated and involves many steps, but the steps are relatively simple if taken one at a time. I encourage you to download the ImageColors example program from the book's web site and look at the code to see how it all fits together.

SUMMARY

This chapter describes WPF command objects, which provide several benefits. They let a program easily enable and disable any number of buttons, menu items, and other controls that invoke them; they allow you to attach input gestures that let the user invoke commands with keyboard sequences; and they automatically display those gestures on menu items.

WPF provides around 150 predefined command objects that you can use to represent logical operations in your applications. A few, such as Copy, Cut, and Paste, are even supported by controls such as `TextBox`. See Appendix K for a list of predefined command objects.

If the predefined commands don't' do everything that you need, you can always create your own.

Whether you use command objects is still up to you. Your application can catch events and take direct action without using commands. That may be practical for simple actions, but if you want to provide multiple ways to invoke the same command or support input gestures, then you should give commands a try.

WPF commanding is a general mechanism for controlling logical operations at a fairly abstract level. Many applications have a notion of "creating a new document" although you cannot see the action itself, only the results.

The following chapter describes two much more concrete tools: transformations, which modify the way a program draws graphics; and effects, which add decorations to graphical output.

20

Transformations and Effects

As is mentioned in Chapter 1, WPF uses DirectX as its rendering engine. DirectX provides access to any high-performance graphics hardware available on the system, so it can produce amazing effects relatively quickly and easily.

This chapter looks at two useful techniques that WPF gets as a benefit of using DirectX: transformations and bitmap effects. Judicial use of these can make an application easier to use and more attractive.

TRANSFORMATIONS

A *transformation* alters an object's geometry before it is drawn. Different kinds of transformations stretch, rotate, squash, skew, and move an object.

Internally WPF represents transformations using 3-by-3 matrices and manipulates them with linear algebra. Fortunately, you don't need to understand how transformations work to use them in your XAML code.

WPF provides four basic kinds of transformations represented by the following XAML elements:

- `RotateTransform` — This transformation rotates an object. The `Angle` property determines the number of degrees by which the object is rotated clockwise.
- `ScaleTransform` — This transformation scales the object vertically and horizontally. The `ScaleX` and `ScaleY` properties determine the horizontal and vertical scale factors respectively.
- `SkewTransform` — This transformation skews the object by rotating its X and Y axes through an angle given by the `AngleX` and `AngleY` properties.
- `TranslateTransform` — This transformation moves the object. The `X` and `Y` properties determine how far the object is moved horizontally and vertically.

Normally, rotation, scaling, and skewing take place relative to an object's origin, which is usually the object's upper-left corner. However, you can change the center of transformation by setting their `CenterX` and `CenterY` properties. For example, you can move the center to a control's middle to rotate the control around its middle instead of its upper-left corner.

To apply a transformation to an object, give that object a `LayoutTransform` or `RenderTransform` property element that contains one of the four basic transformations. When you use a `LayoutTransform`, WPF modifies the control before it arranges the controls. When you use a `RenderTransform`, WPF arranges the controls first and then modifies the control. The section "Layout and Render Transforms" later in this chapter says more about the differences between these two elements.

For example, the following code creates a `Label`. The `Label.LayoutTransform` element contains a `RotateTransform` that rotates the `Label` by 45 degrees.

Available for download on Wrox.com

```
<Label Canvas.Left="140" Canvas.Top="50"
 Background="LightGreen" Foreground="Red"
 Content="Rotate 45 degrees">
    <Label.LayoutTransform>
        <RotateTransform Angle="45"/>
    </Label.LayoutTransform>
</Label>
```

Transformations

In addition to the four basic transformations, WPF provides a `MatrixTransform` class that represents an arbitrary combination of rotation, scaling, skewing, and translation as a matrix. Using matrices is trickier than using the four basic transformation classes and doesn't really give you any extra flexibility so it isn't covered here. For more information, see the `MatrixTransform` class's online help at `msdn.microsoft.com/system.windows.media.matrixtransform.aspx`.

The Transformations example program shown in Figure 20-1 demonstrates an assortment of transformations.

FIGURE 20-1

The following list mentions a few notable features in Figure 20-1:

- The big yellow label at the top is scaled by a factor of 5 horizontally and 1 vertically.
- The smaller white label in the middle is scaled by a factor of –1 horizontally.
- The orange translated label sits inside a blue canvas, so its translation is relative to the canvas's upper-left corner.

- The various rotated labels are rotated around their upper left corners.
- The label that says "Matrix" defines its transformation using a matrix.

COMBINING TRANSFORMATIONS

You can transform an object using any combination of rotation, scaling, skewing, and translation transformations. To make a combined transformation, use a `TransformGroup` element instead of a single transformation object. Inside the group, you can place any number of other transformations.

For example, the following code shows a `TransformGroup` for a `Grid` control that contains a translation followed by a rotation:

Available for download on Wrox.com

```
<Grid.RenderTransform>
    <TransformGroup>
        <TranslateTransform X="150" Y="0"/>
        <RotateTransform Angle="60"/>
    </TransformGroup>
</Grid.RenderTransform>
```

CombinedTransformations

Note that the ordering of a set of transformations is important. In general, if you change the order of the transformations, you will get different results.

The CombinedTransformations example program shown in Figure 20-2 displays a `Border` (shown as a large black-edged rectangle) that contains a `Canvas`. The `Canvas` holds several objects drawn with different transformations. Neither the Border nor the Canvas have `ClipToBounds` set to `True` so the objects are visible even where they stick out of the `Canvas` and `Border`.

FIGURE 20-2

Because the objects are contained inside the canvas and the canvas fills the border, the objects' origins are in the border's upper-left corner.

The white object numbered 0 is the untransformed object. The blue object 1 has been translated, and the blue object 2 has been translated and then rotated.

The green object 1 has been rotated, and the green object 2 has been rotated and then translated.

The two objects labeled 2 use the same rotation and translation but in different orders, and they produce very different results.

CENTERED SUGGESTION

Often it's easiest to build the transformation you need if you start by creating an object centered at the origin. Then you can easily scale, rotate, or skew it around its center before translating it to its final destination.

LAYOUT AND RENDER TRANSFORMS

WPF controls have two transformation properties: `LayoutTransform` and `RenderTransform`.

WPF applies a control's `LayoutTransform` before it arranges that control and the others on the window. It then uses the transformed control's new width and height to calculate control placement. For example, if you rotate a square 45 degrees, then the result is a diamond that is wider and taller than the original square. If the square is placed inside a `StackPanel`, the `StackPanel` will allow enough room for the rotated square to fit.

WPF applies a control's `RenderTransform` after the controls have been arranged but before they are drawn. This can produce some strange results in arranging controls such as `StackPanel` and `WrapPanel`, with controls sticking out past the edges of the areas allocated by their containers.

`RenderTransforms` work quite well, however, if a control's container doesn't use control sizes to arrange its children. For example, a `Canvas` doesn't use control sizes to arrange controls, so `RenderTransforms` work well there.

TRANSLATION TRICKINESS

Depending on a control's container, transformations sometimes produce unexpected results or may have no noticeable effect.

For example, suppose you place a `Label` inside a `Grid` and give the `Label` a translation transformation. WPF translates the `Label` but then centers it inside the `Grid` so it appears in the same place it would have been if you hadn't bothered with the translation.

In situations such as this, it may be easier to get the result you want if you use a `RenderTransform` instead of a `LayoutTransform`, or if you move the control into a `Canvas`.

The LayoutVsRender example program shown in Figure 20-3 demonstrates the difference between the `LayoutTransform` and `RenderTransform` properties.

FIGURE 20-3

The `StackPanel` on the left contains three `Labels` rotated with `LayoutTransformations`, so the `StackPanel` allows enough height for each rotated `Label`. After allowing room for the `Labels`, the `StackPanel` centers each of them.

The `StackPanel` on the right contains three `Labels` rotated with `RenderTransformations`. The `StackPanel` only allows enough vertical space for each `Label` to fit before it is rotated and then drops in the `Labels` whether they fit or not.

SIZE WISE

Note how the `Labels` on the right in Figure 20-3 stretch to match the width of the `StackPanel`, while those on the left size themselves to fit their contents.

Normally a `StackPanel` stretches its `Labels` to fit its width. However, the `LayoutTransforms` used by the `Labels` on the left in Figure 20-3 are applied before the `Labels` are stretched so the `Labels` get their desired fit-to-content sizes. In contrast, the `StackPanel` on the right of Figure 20-3 arranges the `Labels` first, stretching them, and only then rotates the results.

EFFECTS

Much as transformations modify an object's geometry, *bitmap effects* modify the way in which an object is drawn.

To add an effect to an object, give it a `BitmapEffect` property element that contains an effect object. For example, the following code defines a `Canvas` that holds an `Ellipse` and a `Path`. The `Canvas.BitmapEffect` element holds a `DropShadowBitmapEffect` object to give the result a drop shadow.

Available for download on Wrox.com

```
<Canvas Width="100" Height="100">
    <Canvas.BitmapEffect>
        <DropShadowBitmapEffect/>
    </Canvas.BitmapEffect>
    <Ellipse Stroke="Blue" Fill="Yellow" Width="100" Height="100"/>
    <Path Data="M 20,50 A 30,30 180,1,0 80,50"
     Stroke="Blue" StrokeThickness="2"/>
</Canvas>
```

Effects

The following list summarizes WPF's bitmap effect classes:

- `BevelBitmapEffect` — Adds beveled edges to the object. This effect has several important properties. `BevelWidth` determines how wide the beveled edges are. `LightAngle` determines the apparent direction that light hits the beveled edges. `Relief` determines how stark the contrast is between the shadowed and lighted edges. `Smoothness` determines how smooth the shadows are. `EdgeProfile` is discussed in the following paragraphs.
- `BlurBitmapEffect` — Blurs the object. `Radius` determines how large the blurring is.
- `DropShadowBitmapEffect` — Adds a drop shadow behind the object. This effect's most useful properties are `Color` (which determines the shadow's color) and `Direction` (which gives the direction in degrees from the object to its shadow). `ShadowDepth` determines how far "behind" the object the shadow appears.

- `EmbossBitmapEffect` — Embosses the object.
- `OuterGlowBitmapEffect` — Adds a glowing aura around the object. `GlowSize` determines the aura's size. `GlowColor` determines the aura's color.

If you want to use more than one effect at the same time, use a `BitmapEffect` element that contains a `BitmapEffectGroup`. Inside the `BitmapEffectGroup`, place the effects that you want to use. For example, the following code makes a `Canvas` use an `EmbossBitmapEffect` followed by a `BevelBitmapEffect`.

Available for download on Wrox.com

```
<Canvas.BitmapEffect>
    <BitmapEffectGroup>
        <EmbossBitmapEffect/>
        <BevelBitmapEffect BevelWidth="10"/>
    </BitmapEffectGroup>
</Canvas.BitmapEffect>
```

Effects

EFFECTIVE EFFECTS

Note that the order of effects is important. In general, applying effects in different orders will produce different results.

The Effects example program shown in Figure 20-4 displays an unmodified object and the same object using the five basic bitmap effects. The final two versions on the lower right show the object using combinations of effects.

FIGURE 20-4

The `BevelBitmapEffect` class's `EdgeProfile` property determines the exact shape of the beveled edges. The BevelEffects example program shown in Figure 20-5 demonstrates each of the property's four possible values.

FIGURE 20-5

All of the bitmap effects act only on pixels that are visible. If an area is unfilled or filled with the color Transparent, then the effect ignores that area.

FILLED WITH NOTHING

To make an area unfilled, you can often just not fill it. For example, if you don't specify an `Ellipse`'s `Fill` property, then the `Ellipse` is unfilled.

To explicitly make an area unfilled, fill it with `{x:Null}` in XAML code, `null` in C#, or `Nothing` in Visual Basic.

The TransparentEffects example program shown in Figure 20-6 draws Smiley Faces similar to those displayed by the Effects program except the main ellipse is transparent instead of yellow.

FIGURE 20-6

The effects shown in Figure 20-6 only use the pixels that are actually drawn. This looks reasonable for the blur, drop shadow, and outer glow effects, but doesn't look as good for the bevel and emboss effects. The bevel and emboss effects need filled areas to do their work, so they don't do much for this example. (You can still see their effects in the eyes and nose, which are filled.)

SUMMARY

Transformations and bitmap effects can add a bit of extra interest and sophistication to an application without forcing you to write a lot of extra code. Most of the examples in this chapter are overly garish because they demonstrate specific techniques rather than produce an aesthetically pleasing result. Often you can and should use effects more subtly.

The Games example program shown in Figure 20-7 produces a more elegant result. It uses labels rotated –90 degrees, drop shadows behind its labels and images, and grids that are beveled to look like buttons. The result is much more pleasant than the collection of strange Smiley Faces shown in Figure 20-4.

FIGURE 20-7

Most of WPF's container controls arrange their children in fairly simple ways. For example, the `StackPanel`, `WrapPanel`, `Grid`, and `UniformGrid` controls arrange their children in non-overlapping rectangular areas.

These controls are easy to use but sometimes you may not want such simple arrangements. Instead you might want to display text and other objects flowing around sections that contain images, tables, or other items. The next chapter explains how you can use documents to do just that.

21

Documents

WPF documents can contain a wide variety of objects such as:

- Paragraphs
- Tables
- Lists
- Floaters
- Figures
- User interface elements such as `Buttons` and `TextBoxes`
- Three-dimensional objects

WPF has two different kinds of documents: fixed documents and flow documents. This chapter describes these two kinds of documents. It explains the objects that each kind of document can hold and how they must be arranged to produce a valid result.

FIXED DOCUMENTS

A *fixed document* displays its contents in exactly the same size and position whenever you view it. If you resize the control that holds the document, parts of the document may not fit, but they won't be moved. In that sense, this kind of document is similar to a PDF (Portable Document Format) file.

The following sections explain how you can build fixed documents and display them in your WPF applications.

Building XPS Documents

When it defined fixed documents, Microsoft also defined *XML Paper Specification* (XPS) files. An *XPS file* is a fixed document saved in a special format that can be read and displayed by certain programs such as recent versions of Internet Explorer.

One of the most flexible ways you can make an XPS document is to use Microsoft Word. Simply create a Word document containing any text, graphics, lists, or other content that you want, and then export it as an XPS document.

Microsoft Word 2007 can export files in XPS (or PDF) format, but you need to install an add-in first. As of this writing, you can install the add-in by following the instructions on Microsoft's "2007 Microsoft Office Add-in: Microsoft Save as PDF or XPS" download page at `r.office.microsoft.com/r/rlidMSAddinPDFXPS`. (If the page has moved so you can't find it, go to Microsoft's Download Center at `www.microsoft.com/downloads` and search for "Save as PDF or XPS.")

After you install the add-in, simply click on the Windows icon in Word's upper-left corner to expand the main file menu. Click on the arrow next to the "Save As" command, and select the "PDF or XPS" item.

> **WORD OF WARNING**
>
> Although Word can export a file in XPS format, it cannot read an XPS file successfully, at least in the version I'm using right now (Word 2007 SP 1). Be sure you save your document as a normal Word document in addition to exporting it so you can reload it later.
>
> Some third party products can also convert XPS documents back into Word documents. For example, see `www.investintech.com/xpscentral/xpstoword/`.

If you don't have Microsoft Word but are running the Microsoft Windows 7 operating system, you can still create XPS documents relatively easily. Use Notepad, WordPad, or some other editor to create the document. Next print it, selecting Microsoft XPS Document Writer as the printer. The writer will prompt you for the file where you want to save the result.

You can even display a web page in a browser such as Firefox or Internet Explorer, print it, and select Microsoft XPS Document Writer as the printer.

In addition to using Word or printing to the XPS Document Writer, you can use the XPS Document Writer as a stand-alone application. You can download and install the writer in the Microsoft XPS Essentials Pack available at `www.microsoft.com/whdc/xps/viewxps.mspx`.

Displaying XPS Documents

After you have created an XPS file, you have several options for displaying it.

The Microsoft XPS Viewer, which is installed by default in Windows 7 and integrated into Internet Explorer 6.0 and higher, can display XPS files. By default, .xps files are associated with Internet Explorer, so you can probably just double-click on the file to view it.

You can also download Microsoft XPS Viewer or Microsoft XPS Essentials Pack (which contains a stand-alone viewer) at `www.microsoft.com/whdc/xps/viewxps.mspx`.

Finally, you can make a WPF application display an XPS file inside a `DocumentViewer` control. To do that, first add a reference to the ReachFramework library.

In Expression Blend, open the Project menu and select “Add Reference.” Browse to the ReachFramework DLL (on my system, it’s at \Program Files\Reference Assemblies\Microsoft\Framework\v3.0\ReachFramework.dll) and click Open.

In Visual Studio, open the Project menu and select “Add Reference.” On the Add Reference dialog, select the .NET tab, select the ReachFramework entry, and click OK.

Now you can write code-behind to load an XPS document into a `DocumentViewer` control. To make using the necessary classes easier, you can add the following `using` statement to the program:

```
using System.Windows.Xps.Packaging;
```

The following code loads the file “Fixed Document.xps” when the program’s window loads:

Available for download on Wrox.com

```
// Load the XPS document.
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Note: Mark the file as "Copy if newer"
    // to make a copy go in the output directory.
    XpsDocument xps_doc = new XpsDocument("Fixed Document.xps",
        System.IO.FileAccess.Read);
    docViewer.Document = xps_doc.GetFixedDocumentSequence();
}
```

ViewXpsDocument

The ViewFixedDocument example program shown in Figure 21-1 uses this code to display an XPS file.

FIGURE 21-1

Building Fixed Documents in XAML

Building an XPS document is easy with Microsoft Word, WordPad, or some other external program, but sometimes it's more convenient to build a fixed document in XAML or code-behind. For example, you might want a program to generate a report at run time and display it as a fixed document.

Building a fixed document in XAML code isn't too complicated, although it is a lot of work because it's a fairly verbose format.

Start by creating a `DocumentViewer` to hold the document. Give it any properties and resources it needs (such as a background), and place a `FixedDocument` object inside it.

The `FixedDocument` object should contain one or more `PageContent` objects that define its pages.

PAINLESS PAGES

If you don't want to build a page within the code, you can set a `PageContent` object's `Source` property to the Uniform Resource Identifier (URI) of the page it should display.

You can set each `PageContent` object's `Width` and `Height` properties to determine the size of the page.

Inside the `PageContent`, place a `FixedPage` object to define the page's contents.

Inside the `FixedPage`, you can place controls such as `StackPanel`, `Grid`, `TextBlock`, and `Border` to create the page's contents.

CONTESTED INHERITANCE

`FixedPage` objects don't seem to inherit unnamed styles, so, for example, you can't set `Label` font properties in an unnamed style defined in the Window.Resources section. You need to either define the style inside the `FixedPage` or use a named style.

The SimpleFixedDocument example program shown in Figure 21-2 draws four fixed pages that contain simple shapes and labels.

The SimpleFixedDocument program uses the following code to draw its four pages:

Available for download on Wrox.com

```
<DocumentViewer Background="Pink">
    <FixedDocument>
        <PageContent Width="850" Height="1100">
            <FixedPage>
                <Grid Margin="100" Width="650" Height="900" Background="LightBlue">
                    <Ellipse Stroke="Green" Fill="Orange" StrokeThickness="10" />
                    <Label Content="Ellipse" Style="{StaticResource styLabel}"/>
                </Grid>
            </FixedPage>
```

```
            </PageContent>
            <PageContent Width="850" Height="1100">
                <FixedPage>
                    <Grid Margin="100" Width="650" Height="900" Background="LightBlue">
                        <Rectangle Stroke="Blue" Fill="Yellow" StrokeThickness="10" />
                        <Label Content="Rectangle" Style="{StaticResource styLabel}"/>
                    </Grid>
                </FixedPage>
            </PageContent>
            <PageContent Width="850" Height="1100">
                <FixedPage>
                    <Grid Margin="100" Width="650" Height="900" Background="Yellow">
                        <Polygon Stroke="Red" Fill="LightBlue" StrokeThickness="10"
                         Points="325,0 650,900 0,900" />
                        <Label Content="Triangle" Style="{StaticResource styLabel}"/>
                    </Grid>
                </FixedPage>
            </PageContent>
            <PageContent Width="850" Height="1100">
                <FixedPage>
                    <Grid Margin="100" Width="650" Height="900" Background="Orange">
                        <Polygon Stroke="Red" Fill="Pink" StrokeThickness="10"
                         Points="325,0 650,450 325,900 0,450" />
                        <Label Content="Diamond" Style="{StaticResource styLabel}"/>
                    </Grid>
                </FixedPage>
            </PageContent>
        </FixedDocument>
    </DocumentViewer>
```

SimpleFixedDocument

You can see that this code is relatively straightforward, containing the nested objects: `DocumentViewer`, `FixedDocument`, `PageContent`, `FixedPage`, and then content objects. It's fairly simple but quite verbose, using 50 lines of code to draw four simple shapes.

FIGURE 21-2

Saving XPS Files

Having gone to the trouble of building a fixed document in XAML or code-behind, you might want to save it as an XPS file.

To save a fixed document into an XPS file, open a project in Visual Studio and add references to the ReachFramework and System.Printing libraries. To make using the libraries easier, you can add the following `using` statements to the program:

```
using System.Windows.Xps;
using System.Windows.Xps.Packaging;
```

Next, create an `XpsDocument` object to write into the file that you want to create. Make an `XpsDocumentWriter` associated with the document object, and use its `Write` method to write the `FixedDocument` into the file.

The SaveFixedDocument example program uses the following code to save its `FixedDocument` object named `fdContents` into a file.

Available for download on Wrox.com

```
// Save as an XPS file.
private void mnuFileSave_Click(System.Object sender, System.Windows.RoutedEventArgs e)
{
    // Get the file name.
    Microsoft.Win32.SaveFileDialog dlg =
        new Microsoft.Win32.SaveFileDialog();
    dlg.FileName = "Shapes";
    dlg.DefaultExt = ".xps";
    dlg.Filter = "XPS Documents (.xps)|*.xps|All Files (*.*)|*.*";
    if (dlg.ShowDialog() == true)
    {
        // Save the document.
        // Make an XPS document.
        XpsDocument xps_doc = new XpsDocument(dlg.FileName,
            System.IO.FileAccess.Write);

        // Make an XPS document writer.
        XpsDocumentWriter doc_writer =
            XpsDocument.CreateXpsDocumentWriter(xps_doc);
        doc_writer.Write(fdContents);
        xps_doc.Close();
    }
}
```

SavedFixedDocument

The program uses a `SaveFileDialog` to ask the user where to save the file. It the user selects a file and clicks OK, the program makes an `XpsDocument` object for the file and creates an associated `XpsDocumentWriter` object. Finally it writes `fdContents` into the file.

FLOW DOCUMENTS

A *flow document* rearranges its contents as necessary to fit the container that is holding it. If you make the viewing control tall and thin, the document reorganizes its contents to fit. In that sense, this kind of document is similar to a simple web page that rearranges its contents when you resize the browser.

A WPF program displays flow documents inside one of three kinds of viewers: FlowDocumentPageViewer, FlowDocumentReader, or FlowDocumentScrollViewer. See the descriptions of these controls in Chapter 5 for more information about them.

A `FlowDocument` object represents the flow document itself. The `FlowDocument`'s children must be objects that are derived from the `Block` class. These include `BlockUIContainer`, `List`, `Paragraph`, `Section`, and `Table`.

The following sections provide a bit more detail about each of these classes. They also show the code used by the ShowFlowDocument example program to demonstrate each of these and include figures showing the results.

BlockUIContainer

The `BlockUIContainer` control can hold a single user interface (UI) control as a child. For example, it can hold a `Button`, `CheckBox`, `StackPanel`, or `Grid`. If you want the control to hold more than one element, place a container such as a `StackPanel` or `Grid` inside the control, and add other controls to the container.

The following XAML code creates a `BlockUIContainer` holding a `GroupBox`, which holds a `StackPanel`, which holds a `TextBlock` and some `CheckBoxes`:

Available for download on Wrox.com

```
<BlockUIContainer>
    <GroupBox Header="BlockUIContainer" Width="250" HorizontalAlignment="Left">
        <StackPanel>
            <TextBlock Margin="30,0,0,10">A FlowDocument can hold:</TextBlock>
            <CheckBox Content="BlockUIContainer" Margin="50,0,0,5"/>
            <CheckBox Content="List" Margin="50,0,0,5"/>
            <CheckBox Content="Paragraph" Margin="50,0,0,5"/>
            <CheckBox Content="Section" Margin="50,0,0,5"/>
            <CheckBox Content="Table" Margin="50,0,0,5"/>
        </StackPanel>
    </GroupBox>
</BlockUIContainer>
```

ShowFlowDocument

Figure 21-3 shows the result.

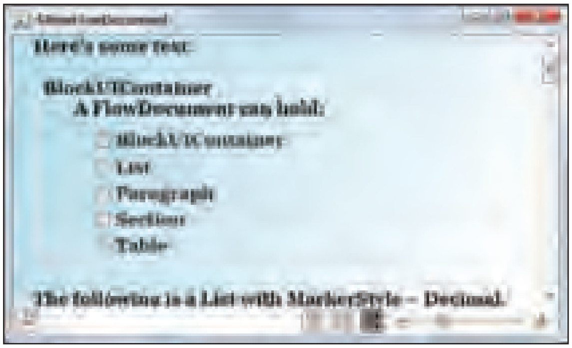


FIGURE 21-3

List

A `List` object contains `ListItem` objects. Each `ListItem` should contain block objects such as a `Paragraph` or `List`.

The ShowFlowDocument example program uses the following code to display the list shown in Figure 21-4:

Available for download on Wrox.com

```
<List MarkerStyle="Decimal">
    <ListItem><Paragraph>Item one</Paragraph></ListItem>
    <ListItem><Paragraph>Item two</Paragraph></ListItem>
    <ListItem>
        <Paragraph>
            Item three. The sub-list has MarkerStyle = LowerLatin.
        </Paragraph>
        <List MarkerStyle="LowerLatin">
            <ListItem><Paragraph>Sub-item 3a</Paragraph></ListItem>
            <ListItem><Paragraph>Sub-item 3a</Paragraph></ListItem>
        </List>
    </ListItem>
</List>
```

ShowFlowDocument

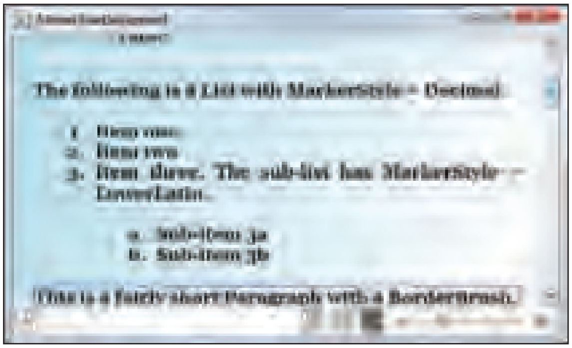


FIGURE 21-4

Paragraph

A `Paragraph` groups its contents into a paragraph. A `Paragraph` adds some vertical space between itself and the previous element.

A `Paragraph` typically contains inline elements including:

- `Text` — Plain old text
- `Bold` — Makes its text bold
- `Figure` — Embeds an area inside the paragraph where you can place other block elements such as a `Paragraph` or `BlockUIContainer`. Anchor and offset properties let you determine where in the paragraph the `Figure` appears.
- `Floater` — Similar to a `Figure` except you cannot control exact positioning. Instead, the `FlowDocument` moves the `Floater` to a position where it fits reasonably.

- `Hyperlink` — Displays a hyperlink. The `NavigateUri` property holds the URI to which the object should navigate. The `Hyperlink` can automatically navigate to that address only if it is contained in a navigation host such as a `NavigationWindow`, `Frame`, or browser.
- `InlineUIContainer` — Can hold UI elements such as `Buttons`, `CheckBoxes`, and `RadioButtons`
- `Italic` — Makes its text italic
- `LineBreak` — Makes the text start a new line. This does not add extra vertical space between the lines.
- `Run` — Contains a run of text, possibly with a different appearance from the rest of the paragraph. You can use the `Run`'s properties to change the text's background color, text color, font size, weight, style, and so forth.
- `Span` — Similar to a `Run` except it can also contain inline elements (e.g., `Bold`, `Italic`) and UI elements (e.g., `Button`, `Ellipse`, `Polygon`)
- `Underline` — Makes its text underlined.

FIGURE 21-5

The following code shows how the ShowFlowDocument program produces the paragraphs shown in Figure 21-5:

Available for download on Wrox.com

```
<Paragraph>
    This paragraph contains a Floater.
    Notice how the Floater is integrated into the paragraph.
    <Floater Width="120" Background="White" HorizontalAlignment="Left">
        <BlockUIContainer>
            <Image Source="Bird.jpg" Stretch="Uniform" Width="100" Height="60"/>
        </BlockUIContainer>
        <Paragraph TextAlignment="Center">Fig 3. Chirp.</Paragraph>
    </Floater>
    The Paragraph also holds a
    <Run Foreground="Blue" Background="Pink">
        Run with blue text on a pink background</Run>.
    <Span>A Span includes an
        <Ellipse Width="30" Height="10" Stroke="Red"/>
        and some text in various styles:
        <Bold>Bold</Bold>, <Italic>Italic</Italic>,
        and <Underline>Underline</Underline>.
     </Span>
     You can use Runs to make equations and formulas like:
     E=MC<Run BaselineAlignment="Superscript">2</Run> and
     C<Run
         BaselineAlignment="Subscript">8</Run>H<Run
         BaselineAlignment="Subscript">10</Run>N<Run
         BaselineAlignment="Subscript">4</Run>O<Run
         BaselineAlignment="Subscript">2</Run>.
</Paragraph>
```

ShowFlowDocument

ODD FORMATTING

The XAML code for the caffeine formula $C_8H_{10}N_4O_2$ is formatted somewhat strangely, with line breaks inside the opening `Run` elements. If the `Runs` are placed on separate lines, WPF adds some space around them, spreading the formula out and making it look odd. This also allows the `FlowDocument` to break the formula in the middle.

The unusual formatting shown here keeps the text on one logical line in the XAML code so the pieces of the formula stay together.

(To make matters even more confusing, if you copy-and-paste the formula's XAML code in Visual Studio, the editor reformats the code to place each `Run` on a separate line, making the result ugly again.)

Section

A `Section` groups other block-level elements such as `Paragraphs`, `Lists`, and `Tables`. A `Section` adds some additional space around its contents so you can use it to separate parts of the document.

For example, you could place two groups of paragraphs in different sections to increase the spacing between the two groups slightly. The effect is somewhat subtle, however, so you may want to include section headers in a distinctive font to make the break more obvious.

A `Section` can also change the appearance of the objects it contains by setting its `Background`, `Foreground`, `FontSize`, `BorderBrush`, and other properties.

Table

A `Table` displays its contents in rows and columns. The `Table.Columns` element attribute determines the widths of the columns. For example, the following code fragment defines five columns. The first has a width of 100 pixels, and the other columns divide the remaining width evenly.

Available for download on Wrox.com

```
<Table CellSpacing="5">
    <Table.Columns>
        <TableColumn Width="100"/>
        <TableColumn/>
        <TableColumn/>
        <TableColumn/>
        <TableColumn/>
    </Table.Columns>
    ...
</Table>
```

ShowFlowDocument

The `Table`'s content includes `TableRowGroup` objects that hold `TableRow` objects. Each `TableRow` contains a series of `TableCells` that hold block elements such as `Paragraphs` and `BlockUIContainers`.

The following code shows how the ShowFlowDocument program defines the content for the `Table` shown in Figure 21-6. Since the code contains lots of repeating elements (rows for each planet and five cells in each row), many of them are omitted to save space.

Available for download on Wrox.com

```
<TableRowGroup>
    <!-- Title -->
    <TableRow Background="#FFA0A0FF">
        <TableCell ColumnSpan="5" TextAlignment="Center">
            <Paragraph FontSize="16" FontWeight="Bold">Planets</Paragraph>
        </TableCell>
    </TableRow>

    <!-- Headers -->
    <TableRow Background="#FFD0D0FF">
        <TableCell>
            <Paragraph FontWeight="Bold">Name</Paragraph>
        </TableCell>
        <TableCell>
            <Paragraph FontWeight="Bold">Picture</Paragraph>
        </TableCell>
        <TableCell>
            <Paragraph FontWeight="Bold">Dist. To Sun</Paragraph>
        </TableCell>
        <TableCell>
            <Paragraph FontWeight="Bold">Year</Paragraph>
        </TableCell>
        <TableCell>
            <Paragraph FontWeight="Bold">Mass</Paragraph>
        </TableCell>
    </TableRow>
</TableRowGroup>

<!-- The planets. -->
<TableRowGroup>
    <TableRowGroup.Background>
        <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
            <GradientStop Color="LightBlue" Offset="0"/>
            <GradientStop Color="White" Offset="1"/>
        </LinearGradientBrush>
    </TableRowGroup.Background>

    <!-- Mercury -->
    <TableRow>
        <TableCell>
            <Paragraph>Mercury</Paragraph>
        </TableCell>
        <TableCell>
            <BlockUIContainer>
                <Image Source="Mercury.jpg"/>
            </BlockUIContainer>
```

```
            </TableCell>

            ... Lots of code omitted ...

        </TableRow>
    </TableRowGroup>
```

ShowFlowDocument

The code starts with a `TableRowGroup` that defines the `Table`'s title row and column headers. `Styles` not shown here set common properties such as text alignment.

The code then ends the first `TableRowGroup` and starts a second one to hold the planet data. This group uses a gradient background that appears behind the data.

The code then makes a `TableRow` that includes the `TableCells` that define the first planet, Mercury. Note how the second cell contains a `BlockUIContainer` that holds an `Image`.

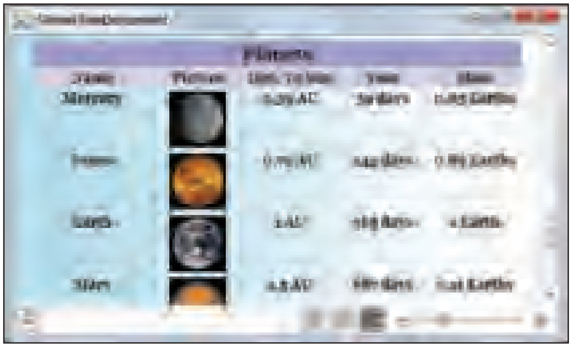


FIGURE 21-6

Mercury's other cells and the rows for the other planets are similar to the code shown here. You can download the example program from the book's web site to see all of the details.

SUMMARY

This chapter explains WPF's two kinds of documents: fixed documents and flow documents. Fixed documents display text, shapes, and other objects at exactly the same position every time they are viewed, much as a PDF file does. Flow documents rearrange their content as necessary when the available space changes, much as a web page rearranges its contents when you resize the browser.

This chapter explains how to build fixed and flow documents using external applications (such as Microsoft Word and WordPad) or with a WPF program. Chapter 17 explains how to print documents.

One of the inline objects that document objects such as `Paragraph` can contain is a `Hyperlink`. A `Hyperlink` can automatically navigate to a URI, but only if it is contained in a navigation host such as a `NavigationWindow`, `Frame`, or browser.

The next chapter describes navigation-based applications in greater detail. It tells how to use the `NavigationWindow` and `Frame` objects to build applications that use a navigation model similar to the one used by web browsers.

22

Navigation-Based Applications

As the Internet and the Web have grown more popular, many desktop applications have adopted features similar to those used by web applications. In particular, many have started providing Back and Next buttons that let you move through the application's navigation history.

This style of navigation is simple and intuitive, particularly for applications where the user visits a series of locations or performs a series of tasks in a particular sequence. It is also useful if the user's locations or tasks are only weakly related, as they are when browsing through web pages. The Back and Next buttons let the user explore safely while always knowing that it will be easy to get back to a previous position if necessary.

For example, recent versions of Windows Explorer, the Control Panel, and other Windows accessories include Back and Next buttons. If you press the Back button in Windows Explorer, the program moves to the directory it visited previously.

NAVIGATION FRUSTRATION

Using web-like navigation does not mean that you can't provide other means for moving around the application. The Back and Next buttons are intended to help the user get back to a place the application was before, but usually an application should provide other, faster means for moving around.

For example, Windows Explorer provides many ways to navigate through a directory hierarchy. You can double-click on a folder to open it, click on a directory in the current path (at the top of the program) to move up the directory hierarchy, click to the right of the path and type in a new path, and open the dropdown next to the path to select from a list of recent locations. The Back and Next buttons are handy but they are some of the weakest methods for navigating.

WPF includes tools and classes that make this style of navigation easy to add to applications. This chapter describes these tools and shows how you can use them to build navigation-based applications of your own.

PAGE

A `Page` object is somewhat similar to a `Window` object in the sense that it is the top-level container that you use to build its style of application. Unlike a `Window`, however, a `Page` must be contained in a navigation host such as a web browser, `NavigationWindow`, or `Frame`. Those hosts add support for the back/next style navigation to the `Page`. Your program moves through `Page` objects, either by using code or by using hyperlinks within the `Pages`, and the navigation host takes care of the rest.

Because it acts as a top-level container for a program, a `Page` includes namespace and class declarations similar to those used by a `Window`.

In some ways, a `Page` is also similar to a `Border` control. Like a `Border`, it can contain a single child.

If you create a `Style` with `TargetType = Border` in the `Page`'s Resources section, that style also modifies the appearance of the `Page`.

The PageBorder example program shown in Figure 22-1 uses a `Style` that makes `Border` controls display a thick, red border with rounded corners.

FIGURE 22-1

The following code shows how the PageBorder program works:

Available for download on Wrox.com

```
<Page
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="Page1"
    x:Name="Page"
    WindowTitle="WindowTitle"
    Title="Title"
    FlowDirection="LeftToRight"
    Width="300" Height="200"
    WindowWidth="400" WindowHeight="300"
    Background="LightBlue">
    <Page.Resources>
        <Style TargetType="Border">
            <Setter Property="BorderBrush" Value="Red"/>
            <Setter Property="BorderThickness" Value="10"/>
            <Setter Property="CornerRadius" Value="20"/>
        </Style>
    </Page.Resources>
    <Label Content="This is a Page"
     FontSize="30" FontWeight="Bold" Foreground="Blue"/>
</Page>
```

PageBorder

Although this code is quite simple, it demonstrates some important points. First, the `Page`'s `WindowTitle` property determines the title displayed in the container holding the `Page`. In contrast, the `Title` property determines the text displayed in the navigation dropdown next to the Back and Next buttons (disabled in

The `Width` and `Height` properties determine how big the `Page` is within its container. The `WindowWidth` and `WindowHeight` properties determine how big the container is initially.

The unnamed `Border Style` modifies the `Page` so that it displays its thick red border.

> **NO NAVIGATIONWINDOW NEEDED**
>
> If you run an application that uses a `Page` as its startup object, the program displays the `Page` in a `NavigationWindow` that provides the Back and Next buttons. Normally, you display a `Page` either in this way or by placing it inside a `Frame` control.

HYPERLINK NAVIGATION

One simple way to navigate in a `Page` is to place a document on it that contains hyperlinks. The hyperlinks can automatically navigate between the application's `Pages` and even to external locations on the Web.

The PageDocument example program shown in Figure 22-2 contains two `Pages`. Each `Page` displays a `FlowDocument` that contains a series of hyperlinks that let you navigate to web sites. The bottom link on each `Page` leads to the other `Page`.

FIGURE 22-2

Figure 22-2 shows the navigation dropdown open and displaying the program's navigation history. When I started the program, it displayed its Writing Links page. I used the hyperlinks to navigate to the Map Links page (displayed as "Current Page" in the dropdown) and then to the Google Maps site. Next, I clicked on the Back button to move back to the Map Links page and opened the dropdown. The mouse is over the Writing Links page so it is highlighted and displays a left arrow in Figure 22-2. ("Future" pages such as Google Maps in this case display a right arrow.)

The following code shows the body of the program's first page. The second page is similar.

Available for download on Wrox.com

```
<FlowDocument>
    <Paragraph FontSize="20" FontWeight="Bold" TextAlignment="Center">
        Writing Links
    </Paragraph>
    <List>
        <ListItem>
            <Paragraph>
                <Hyperlink
  NavigateUri="http://www.rhymezone.com/">RhymeZone</Hyperlink>
            </Paragraph>
        </ListItem>
        <ListItem>
            <Paragraph>
                <Hyperlink
```

```
NavigateUri="http://thesaurus.reference.com/">Thesaurus.com</Hyperlink>
          </Paragraph>
      </ListItem>
      <ListItem>
          <Paragraph>
              <Hyperlink
NavigateUri="http://dictionary.reference.com/">Dictionary.com</Hyperlink>
              <LineBreak/>
          </Paragraph>
      </ListItem>
      <ListItem>
          <Paragraph>
              <Hyperlink
NavigateUri="Page2.xaml">Map Links</Hyperlink>
          </Paragraph>
      </ListItem>
  </List>
</FlowDocument>
```

PageDocument

The `Page` holds a `FlowDocument` object that contains everything else. The document holds a title paragraph followed by a list.

Each list item contains a paragraph that holds a `Hyperlink` object. The `Hyperlink`'s `NavigateUri` attribute tells where the `Page` should go when the user clicks the link.

Most of the links lead to web addresses, but the last one has `NavigateUri` set to "Page2.xaml." When the user clicks on this hyperlink, the program creates a new `Page2.xaml` object and navigates to it.

RECYCLING OLD OBJECTS

It is important to note that navigating to a `Page` creates a new instance of the `Page` rather than reusing any existing `Page` objects. In contrast, if you click on the Back or Next button to move to a previously viewed `Page`, then the program redisplays the object it previously showed.

For example, suppose a program's first `Page` contains a `TextBox`. You enter text into it and click on a hyperlink to move to a second `Page`. Now, if you click the Back button, you will see your text on the first `Page`. However, if you click on another hyperlink to move to the first `Page`, you'll see a new object with a blank textbox.

If you want to allow this sort of navigation but want the hyperlink to lead to the original `Page` object, you'll need to use some code-behind rather than letting the hyperlink navigate for you automatically.

NAVIGATIONSERVICE

In addition to using hyperlinks, an application's code-behind can use a `NavigationService` to navigate. A `NavigationService` object lets an application use code-behind to control how the container navigates.

The PageApp example program shown in Figure 22-3 uses a `NavigationService` to move between its pages.

FIGURE 22-3

When you click on one of the buttons shown in Figure 22-3, the program uses the `NavigationService`'s `Navigate` method to open the new page. The following code shows how the program moves to the "Add Customer" page shown in Figure 22-4:

Available for download on Wrox.com

```
private void btnAddCustomer_Click(object sender, RoutedEventArgs e)
{
    NavigationService.Navigate(new pagAddCustomer());
}
```

PageApp

This code simply calls the `NavigationService`'s `Navigate` method, passing it a new `pagAddCustomer Page` object.

In addition to displaying a `Page` object, the `Navigate` method can display a web page. The following code executes if you click on the Help icon in the lower-right corner of Figure 22-3. This code navigates to the web page `www.vb-helper.com/wpf.htm`.

Available for download on Wrox.com

```
// Go to the help web site.
private void btnHelp_Click(object sender, RoutedEventArgs e)
{
    NavigationService.Navigate(new Uri("http://www.vb-helper.com/wpf.htm"));
}
```

PageApp

FIGURE 22-4

The `NavigationService` object provides other useful properties and methods in addition to `Navigate`. The following code shows how the program PageApp returns from the “Add Customer” page shown in Figure 22-4 when you click on the checkmark image in the lower right:

Available for download on Wrox.com

```
// Pretend we did something and return to the main page.
private void btnOk_Click(object sender, MouseButtonEventArgs e)
{
    MessageBox.Show("Customer created.", "Customer created",
        MessageBoxButton.OK, MessageBoxImage.Information);
    if (NavigationService.CanGoBack)
    {
        NavigationService.GoBack();
    }
    else
    {
        NavigationService.Navigate(new pagMain());
    }
}
```

PageApp

This code first displays a message box. It then checks the `NavigationService`’s `CanGoBack` property to see if the service can go back to a previous page — in this case, the program’s initial page. If `CanGoBack` is `True`, then the code calls the `NavigationService`’s `GoBack` method. If `CanGoBack` is `False`, then the code calls the `NavigationService`’s `Navigate` method, passing it a new `pagMain Page` object.

The following list summarizes the `NavigationService`’s most useful properties and methods:

- `AddBackEntry` — Adds a navigation entry that contains a custom state object.

- `CanGoBack` — `True` if the `NavigationService` can move back to a previous object.
- `CanGoForward` — `True` if the `NavigationService` can move forward to a following object.
- `GoBack` — Goes to the previous page if one exists.
- `GoForward` — Goes to the next page if one exists.
- `Navigate` — Displays a `Page` or URI.
- `RemoveBackEntry` — Removes the most recent navigation history entry.
- `Source` — Gets or sets the URI of the current display source. Normally in code-behind, you would use the `Navigate` method to open a source, but you can set this property in XAML code to make the `Page` open a URI when it is displayed.

FRAME

The previous example programs display `Page` objects. The result is a form containing navigation buttons, the `Page` object, and nothing else.

Sometimes you may want a `Page` to appear as only part of a form that also contains other elements. You can achieve this by placing a `Page` inside a `Frame` control.

You cannot write XAML code that places a `Page` inside a `Frame`, but you can set the `Frame`'s `Source` property to the name of the `Page` class that it should display when it appears.

The FrameApp example program shown in Figure 22-5 uses this approach to display the same `Pages` used by the PageApp program inside a `Frame`. In Figure 22-5, the top and bottom labels and the green background are part of a `Window`. Everything else is inside the `Frame`.

The following XAML fragment shows how the FrameApp program displays its two `Labels` and `Frame`. An unnamed style targeted at `Label` controls aligns the `Labels` and gives them drop shadows.

Available for download on Wrox.com

```
<StackPanel>
    <Label FontSize="30" FontWeight="Bold" Content="FrameApp"/>
    <Frame Name="fraMain" Source="pagMain.xaml" Height="420" Margin="10"/>
    <Label BorderBrush="Black" BorderThickness="1" FontSize="16" Margin="10"
     Content="Use the buttons inside the pages to navigate within the frame."/>
</StackPanel>
```

FrameApp

This code contains two points worth mentioning. First, it sets the `Frame` control's `Source` attribute to pagMain.xaml so the `Frame` displays a `pagMain Page` object when it is loaded.

Second, the code explicitly sets the `Frame`'s height. The `Frame` only displays its Back and Next buttons when the user can navigate backward or forward. Initially, the `Frame` has not visited any other pages, so navigation isn't possible and these buttons are hidden. Then, when you move to a new `Page`, the `Frame` displays its buttons. If you don't set the `Frame`'s height explicitly, adding the buttons makes the `Frame` taller, and that makes the `Window` rearrange its controls.

FIGURE 22-5

If you don't mind this rearrangement, you can omit the Frame's height and let it figure out how big to be based on the Pages it contains. If you don't want other controls such as the Label at the bottom of the Window to move around, give the Frame a fixed height.

SUMMARY

By using Pages, you can add a web-like navigation model to an application. The web browser, NavigationWindow, or Frame that hosts the Pages automatically allows the user to move backward and forward through the navigation history.

This style of navigation doesn't replace other methods of opening new windows such as menus and buttons. It just gives the user a new option for returning to places previously visited.

All of the chapters (and almost all of the examples) so far in this book have worked only in two dimensions. Careful use of graphical techniques such as BevelBitmapEffect and overlaid highlights can give buttons, borders, and other objects a three-dimensional (3D) appearance, but they are still really two-dimensional (2D) objects.

The next chapter explains how to display truly 3D objects. It shows how you can make 3D shapes with colored and textured surfaces and multiple light sources. It even shows how you can place controls such as TextBoxes and MediaElements on 3D objects.

23

Three-Dimensional Drawing

WPF relies on Microsoft's DirectX technologies for high-quality multimedia support. The DirectX libraries include methods to display images, present video, and play audio files that take advantage of the computer's hardware to provide the best performance and highest quality possible.

One of my favorite parts of DirectX is *Direct3D*, the three-dimensional (3D) drawing library. Direct3D lets a program display complex animated 3D scenes with multiple light sources, smooth color shading, and realistic textures — all in real time.

The TexturedBlock example program shown in Figure 23-1 displays a simple cube with sides made of various materials sitting on a brick and grass surface. You can use the sliders to rotate the scene in real time.

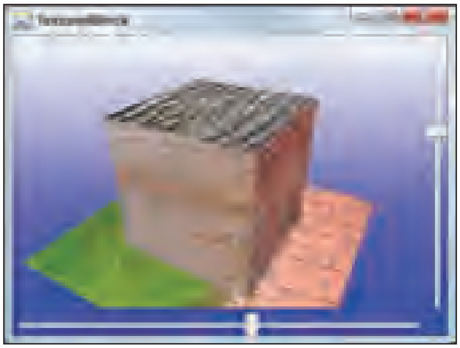


FIGURE 23-1

Unfortunately, writing Direct3D code to display these types of objects is fairly involved. One of the first and most annoying tasks is figuring out what graphics hardware is available on the user's computer. Determining which device objects you need to get the most out of the user's computer can be confusing.

Fortunately, WPF's 3D drawing objects handle this setup for you. Although they don't always give you quite the same performance you could achieve using Direct3D yourself, it's a lot easier to get a program up and running in WPF.

That doesn't mean that the rest of the process is easy. Producing a 3D scene requires a fair amount of work because there are many complicated factors to consider, but at least WPF takes care of initialization and redrawing.

The remaining factors you need to consider include:

- **Geometry** — The geometry determines what kinds of objects are in the scene and where they are located. This is the part that many people think of as 3D graphics. It defines the shapes in 3D space that make up the buildings, Death Star, and aliens that you will display.
- **Camera** — The camera acts just as you would expect in a motion picture, determining the location from which you view the scene and the direction in which you are looking. It also determines the type of projection used to translate from a 3D model onto your 2D computer screen.
- **Lighting** — Without any light, you can't see a thing in the real world or in 3D graphics. Light objects determine the color and type of lighting used in a scene.
- **Materials** — An object's appearance actually depends both on the light that hits it and its material. If you shine a blue light on a white ball, the result is blue, not white.

The following section explains the basic structure and geometry of a simple 3D scene. The sections after that describe cameras, lighting, and materials in greater detail.

BASIC STRUCTURE

The `Viewport3D` control displays a 3D scene. You can think of it as a window leading into 3D space.

You can size, position, and otherwise arrange a `Viewport3D` control just as you can any other control. For example, the `Viewport3D` control in Figure 23-1 is contained in the upper-left cell of a `Grid` control. The sliders lie in `Grid`'s other cells.

Like other controls, the `Viewport3D` can contain a Resource section. This is a handy place to store values that will be used by the 3D scene such as colors, materials, and camera parameters.

The `Viewport3D` object's `Camera` property defines the camera used to view the scene. Often a program sets this property as a `Viewport3D.Camera` property element so it can specify the `Camera`'s properties, but you can also define the `Camera` as a resource and then use the resource as an attribute. The "Cameras" section later in this chapter explains cameras in more detail.

The `Viewport3D` should contain one or more `ModelVisual3D` objects that define the items in the scene. The `ModelVisual3D`'s `Content` property should contain the visual objects.

Typically, the `Content` property holds either a single `GeometryModel3D` object that defines the entire scene or a `Model3DGroup` object that holds a collection of `GeometryModel3D` objects.

Each `GeometryModel3D` object can define any number of triangles, so a single `GeometryModel3D` object can produce a very complicated result. Since the triangles don't even need to be connected to each other, a `GeometryModel3D` could define several separate physical objects.

The catch is that a single `GeometryModel3D` can only have one material, so any separate objects would have a similar appearance. For example, if a `GeometryModel3D` that uses a green material defines three cubes, all three cubes are green.

If you want to give the cubes different colors, use a `Model3DGroup` object holding three `GeometryModel3D` objects that define the cubes. The "Materials" section later in this chapter describes materials in greater detail.

The `GeometryModel3D` object's two most important properties are `Material` (which was just described) and `Geometry`. The `Geometry` property should contain a single `MeshGeometry3D` object that defines the triangles that make up the object.

`MeshGeometry3D` has four key properties that define its triangles: `Positions`, `TriangleIndices`, `Normals`, and `TextureCoordinates`. The following sections describe these properties.

Positions

The `MeshGeometry3D`'s `Positions` property is a list of 3D point coordinate values. You can separate the coordinates in the list by spaces or commas.

> **CLEAR COORDINATES**
>
> To make coordinate lists easier to read, I prefer to separate points with spaces and coordinates with commas as in "1,-1,1 -1,-1,1 1,-1,-1 -1,-1,-1."

For example, the value "`1,0,1 -1,0,1 1,0,-1 -1,0,-1`" defines the four points in the Y = 0 plane where X and Z are 1 or –1. This defines a square two units wide centered at the origin.

TriangleIndices

The `TriangleIndices` property gives a list of indexes into the `Positions` array that give the points that make up the object's triangles. Note that the triangles are free to share the points defined by the `Positions` property.

For example, if `TriangleIndices` is "`0,1,2 0,2,3`" then the first triangle is made up of points 0, 1, and 2; and the second triangle is made up of points 0, 2, and 3. If the `Positions` property has the value "`1,0,1 -1,0,1 1,0,-1 -1,0,-1`" used in the previous section, then these triangles fill the square $-1 \le X \le 1$, $Y = 0$, $-1 \le Z \le 1$.

Note that the orientation of the points that make up a triangle is extremely important. To allow Direct3D to draw a triangle correctly, it must be *outwardly oriented* according to the *right-hand rule*. The next section explains how to give triangles an *outward orientation*.

Outward Orientation

A vector that points perpendicularly to a triangle is called a *normal* or *surface normal* for the triangle (or any other surface for that matter). Sometimes people require that the normal have length 1. Alternatively, they may call a length 1 normal a *unit normal*.

Note that a triangle has two normals that point in opposite directions. If the triangle is lying flat on a table, then one normal points toward the ceiling, and the other points toward the floor.

The right-hand rule lets you use the order of the points that define the triangle to determine which normal is which.

To use the right-hand rule, picture the triangle ABC that you are building in 3D space. Place your right index finger along the triangle's first segment AB as shown in Figure 23-2. Then bend your middle finger inward so it lies along the segment AC. If you've done it right, then your thumb (the blue arrow in Figure 23-2) points toward the triangle's "outside." The *outer normal* is the normal that lies on the same side of the triangle as your thumb.

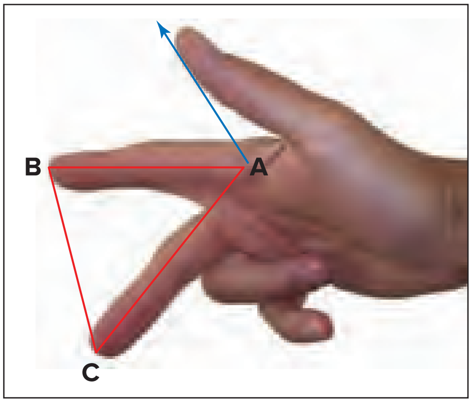


FIGURE 23-2

RIGHT-HAND RULE REDUX

Another way to think about the right-hand rule is to align your right palm along the first segment AB and then curl your fingers toward the third point C. Again, your thumb points to the side of the outward normal.

This makes more sense if you think of the triangle as being one of the faces on a tetrahedron or some other 3D solid. If the triangle is properly oriented, then the right-hand rule makes your thumb point *out* of the solid, not into it.

DO IT RIGHT

There are a couple of ways to mess up the right-hand rule. First, be sure you use your right hand! Second, be sure to bend your middle finger in toward your palm instead of bending your index finger toward your palm. If you make either of these mistakes, then your thumb points toward the inward normal — the opposite of the direction you want.

If you can't figure out how to make your fingers line up, try turning your hand upside down. The triangle is probably oriented in the other direction.

Why should you care about the right-hand rule and outwardly-oriented normals? Direct3D uses the triangle's orientation to decide whether it should draw the triangle. If the outwardly-oriented normal points *toward* the camera's viewing position, then Direct3D draws the triangle. If the outwardly-oriented normal points *away from* the camera's viewing position, then Direct3D doesn't draw the triangle.

To understand why this makes sense, suppose a triangle is part of a 3D object such as a tetrahedron. If the triangle is oriented properly, then its outward normal points out of the tetrahedron. If that normal points toward the camera, then the camera can see the triangle (if nothing else blocks the view).

However, if the outward normal points away from the camera, then the triangle is on the far side of the tetrahedron, so the camera cannot see it. Direct3D doesn't even bother to draw the triangle because it cannot be visible.

BACKFACE REMOVAL

Removing triangles that are oriented away from the camera is called *backface removal*. Sometimes it's also called *culling*, although that can also mean any method for quickly removing triangles from consideration. For example, if you can quickly identify the triangles behind the camera, they can also be culled.

Note that this only really works for closed solids, and open surfaces won't appear properly from both sides. For example, suppose the camera views a shoebox without a lid from the top. In this position, the camera is looking at the inner sides of the triangles, so, if they are outwardly oriented, Direct3D won't draw them. To fix this problem, you can include each triangle in the program's geometry twice, once with each orientation.

Normals

As you'll see in the section "Lighting" later in this chapter, a triangle's normal not only determines whether it's visible, but it also helps determine the triangle's color. That means more accurate normals give more accurate colors.

Left to its own devices, Direct3D finds a triangle's normal by performing some calculations using the points that define the triangle. (It lines up its virtual fingers to apply the right-hand rule.) The resulting normal points perpendicularly away from the triangle.

For objects defined by flat surfaces such as cubes, octahedrons, and other polyhedrons, that works well. For smooth surfaces such as spheres, cylinders, and torii (donuts), it doesn't work as well because the normal at one part of a triangle on the object's surface points in a slightly different direction from the normals at other points. If you make the triangles small enough, the difference isn't too noticeable, but if the triangles are larger, using the same normal across the entire surface of a triangle makes the result appear faceted instead of smooth.

The `MeshGeometry3D` object's `Normals` property lets you tell the drawing engine what normals to use for the points defined by the object's `Positions` property. The engine still uses the calculated normal to decide whether to draw a triangle, but it uses the normals you supply to color the triangle.

For example, suppose you are building a sphere. In that case, each of the points in the `MeshGeometry3D` object is on the surface of the sphere. The surface's normal at the point (x, y, z) on the sphere points from the sphere's center to the point. If the sphere is centered at the origin, then the normal vector is simply <x, y, z>. If you set the normal for each point to the corresponding value, then Direct3D can use it to make the resulting colors smoother.

IS THERE A POINT?

Standard notation surrounds a vector's components with pointy brackets. A vector indicates a direction and not a position. For example, the vector <1, 0, 0> indicates a direction pointing parallel to the positive X axis.

The SpheresWithNormals example program shown in Figure 23-3 demonstrates this technique. All of the spheres were drawn using a separate `MeshGeometry3D` object for each triangle. The spheres on the right are the same as those on the left except they specify normals for each `MeshGeometry3D` object. You can see that the results are much smoother.

FIGURE 23-3

Earlier in this section, I said that Direct3D creates its own normals for a triangle if you don't supply them. Actually, this is a bit of a simplification. If a point is used by more than one triangle in a `MeshGeometry3D` object, then Direct3D sets the normal at that point to an average of the normals at that point in the adjoining triangles. The result is somewhere between the individual triangles' normals, so the result is generally smoother.

For a sphere, adjoining triangles meet symmetrically around each point, so the averaged normal points away from the center of the sphere as it should, and the result is nicely smooth.

The SingleMeshSpheres example program draws spheres where every triangle in a sphere is contained in the same `MeshGeometry3D` object. The result is very smooth despite the fact that the program doesn't explicitly specify normals.

This program is also much faster than the SpheresWithNormals program. It takes Direct3D a lot longer to draw many `MeshGeometry3D` objects each containing one triangle than it takes to draw one `MeshGeometry3D` object containing many triangles. (Try running the two programs and rapidly dragging their sliders back and forth. You'll find that the SingleMeshSpheres program can keep up a lot better than the SpheresWithNormals program.)

SHARING NORMALS

The `Normals` property specifies the normal at a point, not for a particular triangle. This means that if two triangles share the same point, then they use the same normal at that point, so they meet smoothly there.

This also means that two triangles cannot share the same points if they should *not* join smoothly. For example, suppose you build a cube in a single `MeshGeometry3D` object. If two triangles on different sides of the cube share corners, then Direct3D will smooth out the edge between then, so you won't see a nice, crisp edge between the sides. Instead, you'll see something similar to the left sphere in Figure 23-4 with corners around its profile but the middle blurred together.

To avoid this problem, either use a separate `MeshGeometry3D` object for each of the cube's faces or place multiple copies of the same point in the Positions collection so the triangles don't need to share.

TextureCoordinates

The `TextureCoordinates` property is a collection that determines how points are mapped to positions on a material's surface. You specify a point's position on the surface by giving the coordinates of the point on the brush used by the material to draw the surface. (The section "Materials" later in this chapter says more about materials.)

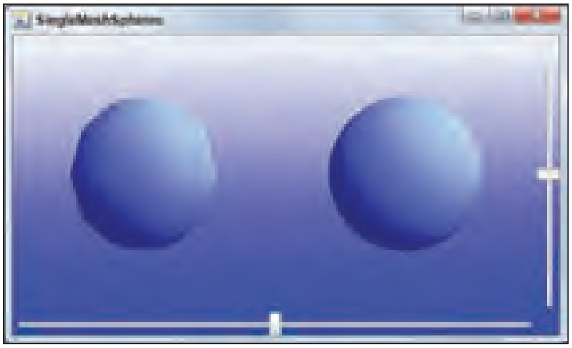


FIGURE 23-4

The coordinates on the brush begin with (0, 0) in the upper left with the first coordinate extending to the right and the second extending downward. This is similar to the way you use X and Y coordinates in a bitmap (although to avoid confusion with 3D space, many graphics programmers call the texture axes U and V rather than X and Y).

Figure 23-5 shows the idea graphically. The material could come from any brush such as a solid color, gradient, or drawing brush (although textures are often easiest to visualize when the brush uses an image as shown in Figure 23-5).

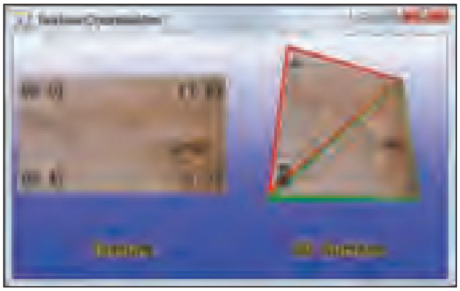


FIGURE 23-5

The picture on the left shows the texture material with its corners labeled in the U–V coordinate system.

The picture on the right shows a `MeshGeometry3D` object that defines four points (labeled A, B, C, and D) and two triangles (outlined in red and green).

The following code shows the `MeshGeometry3D`'s definition. The corresponding `Positions` and `TextureCoordinates` entries map the points A, B, C, and D to the U–V coordinates (0, 0), (0, 1), (1, 0), and (1, 1), respectively. You can also see in Figure 23-5 how the program maps the knot hole from the texture's lower-right corner to the 3D rectangle's lower right corner.

Available for download on Wrox.com

```
<MeshGeometry3D
    Positions="-1,1,1 -1,-1,1 1,1,1 1,-1,1"
    TriangleIndices="0,1,2 2,1,3"
    TextureCoordinates="0,0 0,1 1,0 1,1"
/>
```

TextureCoordinates

The TexturedBlock program shown in Figure 23-1 uses similar code to map textures to points for all of its surfaces.

CAMERAS

The camera determines the location and direction from which a 3D scene is viewed. You can think of the camera as if it were a motion picture camera pointed at a scene. The camera is in some position (possibly on a boom, a crane, or in a cameraperson's hand) pointed toward some part of the scene.

The following camera properties let you specify the camera's location and orientation:

- `Position` — Gives the camera's coordinates in 3D space.
- `LookDirection` — Gives the direction in which the camera should be pointed relative to its current position. For example, if the camera's `Position` is `"1, 1, 1"` and `LookDirection` is `"1, 2, 3"`, then the camera is pointed at the point (1 + 1, 1 + 2, 1 + 3) = (2, 3, 4).
- `UpDirection` — Determines the camera's *roll* or *tilt*. For example, you might tilt the camera sideways or at an angle.

COMMON DIRECTIONS

Often the camera's `LookDirection` is the negative of its `Position` so the camera is looking back toward the origin. For example, if `Position = "1, 2, 3"`, then `LookDirection = "-1, -2, -3"` looks back toward the origin.

It's also common to set `UpDirection` to a vector pointing upward such as <0, 1, 0> so the camera is "right-side up."

The two most useful kinds of cameras in WPF are perspective and orthographic.

In a *perspective view*, parallel lines seem to merge toward a vanishing point and objects farther away from the camera appear smaller. Since this is similar to the way you see things in real life, the result of a parallel camera is more realistic.

In an *orthographic view*, parallel lines remain parallel and objects that have the same size appear to have the same size even if one is farther from the camera than another. While this is less *realistic*, orthographic views can be useful for engineering diagrams and other drawings where you might want to perform measurements.

The CameraTypes example program shown in Figure 23-6 displays images of the same scene with both kinds of cameras.

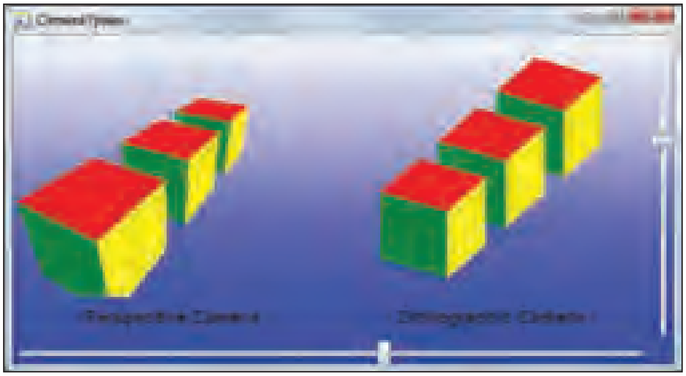


FIGURE 23-6

ORTHOGRAPHIC ILLUSIONS

Because your brain expects far away objects to look smaller than closer objects, orthographic projections can sometimes cause an optical illusion. When you look at the right side of Figure 23-6, your brain may tell you that the box in the back is bigger than those in front because it does not appear smaller as it should.

The following code shows how the CameraTypes program defines its perspective camera. The `FieldOfView` parameter determines the camera's field of view in degrees. A larger value makes the camera see more to the sides so the camera must shrink its results to fit in its available area.

Available for download on Wrox.com

```
<PerspectiveCamera
 Position="0, 0, 8"
 LookDirection="0, 0, -8"
 UpDirection="0, 1, 0"
 FieldOfView="60">
```

CameraTypes

The following code shows how the CameraTypes program defines its orthographic camera. The `Width` parameter determines the width of the area that the camera sees in the scene's coordinate system. This parameter is roughly the orthographic equivalent of the perspective camera's `FieldOfView` parameter.

Available for download on Wrox.com

```
<OrthographicCamera
 Position="0, 0, 8"
 LookDirection="0, 0, -8"
 UpDirection="0, 1, 0"
 Width="8">
```

CameraTypes

LIGHTING

No matter how much effort you put into building a scene, you won't see anything in the dark. You need to add light to a scene before you can see anything.

The color that you see in a scene depends on both the lights in the scene and on the materials used by the objects. The next section discusses materials in detail. This section considers only lights.

A light has a color that helps determine its effects on the scene's objects. For example, if you shine a blue light on a white plane, you get a blue result. Conversely, if you shine a white light on a red ball, you see a red ball.

Other color combinations can be more confusing. For example, if you shine a blue light on a green ball, you see a black result. A green ball reflects only green light, and, because the blue light contains no green light, there's nothing for the ball to reflect.

LOTS OF LIGHTS

If you use a light that is some shade of gray (including white, the brightest shade of gray), then objects of all colors will be illuminated at least to some extent.

WPF provides several kinds of lights that provide different effects. The following list summarizes these kinds of lights:

- **Ambient Light —** This is light that comes from all directions and hits every surface equally. It's the reason you can see what's under your desk even if no light is shining directly there. Most scenes need at least some ambient light.
- **Directional Light —** This light shines in a particular direction as if the light is infinitely far away. Light from the Sun is a close approximation of directional light, at least on a local scale, because the Sun is practically infinitely far away compared to the objects near you.
- **Point Light —** This light originates at a point in space and shines radially on the objects around it. Note that the light itself is invisible, so you won't see a bright spot as you would if you had a real lightbulb in the scene.
- **Spot Light —** This light shines a cone into the scene. Objects directly in front of the cone receive the full light, with objects farther to the sides receiving less.

EFFICIENT ILLUMINATION

Ambient and directional lights are more efficient than point and spot lights.

The Lights example program shown in Figure 23-7 demonstrates the different kinds of lights. Each scene shows a yellow square (the corners are cropped by the edges of the viewport) made up of lots of little triangles.

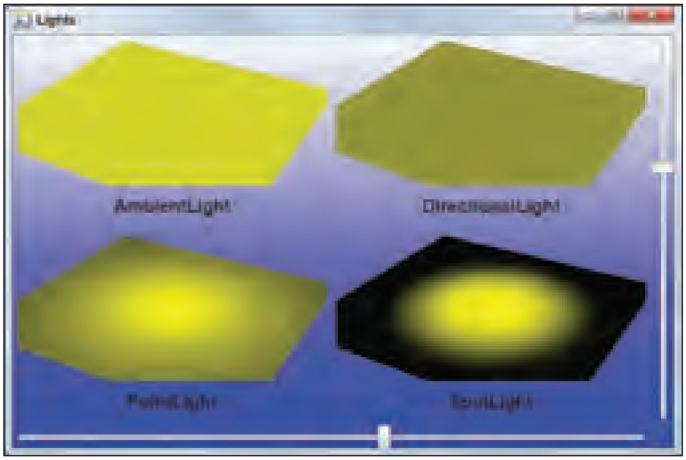


FIGURE 23-7

The ambient light shines on every surface with the same intensity so every triangle appears yellow. This light is light gray because it would produce an overly bright result if it were white. Usually ambient lighting is even darker than this and is only intended to display surfaces that are not illuminated by any other light. The following code shows how the program makes its `AmbientLight`:

```
<AmbientLight Color="LightGray"/>
```

The directional light is shining at an angle to the square. Because all of the square's triangles lie in the same plane, they all make the same angle with respect to the directional light, so they all show the same color. The light shines at an angle to the square, so the square isn't as brightly lit as possible. The following code shows how the program makes its `DirectionalLight`:

```
<DirectionalLight Color="White" Direction="-1,-1,-1"/>
```

The point light is located just above the middle of the square. The light is shining directly on the triangles sitting below it, so they are the brightest. Light shining on triangles farther to the sides hits those triangles at an angle, so those triangles are not lit as brightly. The following code shows how the program makes its `PointLight`:

```
<PointLight Color="White" Position="0,0.5,0"/>
```

If you look closely at Figure 23-7, you'll see that there is an area of roughly uniform brightness directly below the spot light. This area is inside the light's inner cone and is defined by the light's `InnerConeAngle` property. Outside the inner cone, the light's intensity drops off until it reaches the edge of the light's outer cone, which is defined by the `OuterConeAngle` property. The following code shows how the program makes its `SpotLight`:

```
<SpotLight Color="White" Position="0,2,0" Direction="0,-1,0"
 InnerConeAngle="10" OuterConeAngle="40" Range="10"/>
```

Usually a scene includes ambient light so the user can see all of the objects.

Often you will also want to use additional lights, particularly if multiple objects in the scene have the same color. If you don't use multiple lights, then objects with the same color will tend to blend together, so you can't tell where one ends and the next begins.

PLENTY OF PIECES

The squares shown in Figure 23-7 are made up of lots of small triangles for a reason. Direct3D calculates a triangle's color based on the light, material, and the angle at which the light hits the material. While it may blend colors to smooth the edges between one triangle and a neighbor, it does not blend the colors across a single triangle. Instead, each triangle's color is determined once at a single point and then used for the entire triangle (possibly with blending for neighbors).

In this example, that means if the squares were made up of two big triangles, then each triangle would have a single color, and you wouldn't see the drop-off effects provided by the `PointLight` or `SpotLight`.

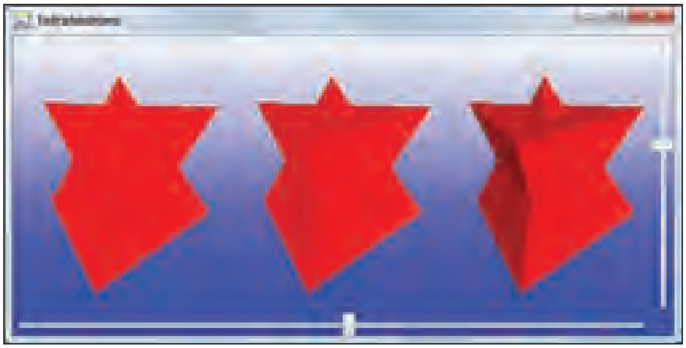


FIGURE 23-8

For example, the Tetrahedrons program shown in Figure 23-8 displays two intersecting tetrahedrons. The picture on the left uses only ambient light, so every red triangle has exactly the same color and you can't tell where one stops and the next begins.

The middle picture uses ambient light and a directional light that shines from right-to-left, so the surfaces facing to the right are brighter than those facing left, up, or down. That allows you to see differences between some adjacent triangles but not others.

The picture on the right uses ambient light plus four directional lights shining left, in toward the back of the scene, down, and up. It takes a lot of lights to make every red surface have a slightly different color.

One alternative to using lots of lights is to give the objects in the scene different colors. For example, the sides of the cubes shown in Figure 23-6 are arranged so that adjacent sides have different colors.

The textured block shown in Figure 23-1 takes a different approach, giving each of the scene's surfaces a completely different texture.

TOO MUCH OF A GOOD THING

Lights are additive, so two lights shining on the same spot can make that spot very bright. If too many lights shine on too much of the scene, then many of the scene's objects will reach saturation and display the brightest possible versions of their colors (red, green, or whatever). If too many objects have the same saturated color, then you have the same problem again with them blending together.

To prevent this problem, you generally need to reduce the brightness of the lights as you add new ones. The more lights you have, the dimmer they must be.

MATERIALS

As previous sections have explained, the exact color given to a triangle in a scene depends on the light and the angle at which the light hits the triangle. The result also depends on the triangle's type of material. WPF provides three kinds of materials: diffuse, specular, and emissive.

A *diffuse* material's brightness depends on the angle at which light hits it, but the brightness does not depend on the angle at which you view it.

For example, hold a white sheet of paper so that it is perpendicular to the brightest light source near you. If the light is white, then the paper should appear bright white. If you move your head around to look at the paper from different angles, it should appear roughly the same color (unless it's shiny paper, in which case it acts as a specular material — more on that in a moment). All of the examples so far in this chapter have used diffuse materials.

A *specular* material is somewhat shiny. In that case, an object's apparent brightness depends on how closely the angle between you, the object, and the light sources matches the object's *mirror angle*. The *mirror angle* is the angle at which most of the light would bounce off the object if it were perfectly shiny.

For example, suppose you have a perfectly shiny stainless steel ball. If you looked closely, you would see a small image of the light reflected in the ball. The place where you see the reflection is where the light's mirror angle points directly at you. On other parts of the ball, you will be off the mirror angle so you won't see the light's reflection there.

If the ball is only somewhat shiny, for example, a billiard ball, then you'll see a bright patch where the mirror angle lines up. Areas near the mirror angle will be slightly less bright and areas far from the mirror angle will be the least bright.

The final type of material, an *emissive* material, glows. An emissive material glows but only on itself. In other words, it makes its own object brighter, but it doesn't contribute to the brightness of other nearby objects as a light would.

Specular and emissive materials are not really intended to be used alone. Most often they are combined with a diffuse material in a `MaterialGroup`.

The Materials example program shown in Figure 23-9 shows four identical spheres made of different materials. From left to right, the materials are diffuse, specular, emissive, and a `MaterialGroup` combining all three types of materials.

FIGURE 23-9

Notice that the specular material adds a bright spot to the final sphere and that the emissive material makes the final sphere brighter overall.

The following code shows how the program creates its spheres:

Available for download on Wrox.com

```
MakeSingleMeshSphere(Sphere00, new DiffuseMaterial(Brushes.Green), 1, 20, 30);
MakeSingleMeshSphere(Sphere01, new SpecularMaterial(Brushes.Green, 50), 1, 30, 30);
MakeSingleMeshSphere(Sphere02, new EmissiveMaterial(Brushes.DarkGreen), 1, 20, 30);

MaterialGroup combined_material = new MaterialGroup();
combined_material.Children.Add(new DiffuseMaterial(Brushes.Green));
```

```
combined_material.Children.Add(new SpecularMaterial(Brushes.Green, 50));
combined_material.Children.Add(new EmissiveMaterial(Brushes.DarkGreen));
MakeSingleMeshSphere(Sphere03, combined_material, 1, 20, 30);
```

Materials

The most noteworthy part of this code is the materials it passes to the `MakeSingleMeshSphere` function that creates the spheres. The `MakeSingleMeshSphere` function simply builds a sphere out of a single `MeshGeometry3D` object. It's interesting but fairly long so it isn't shown here. Download the example program from the book's web page to see how it works.

Note also how the code builds a `MaterialGroup` for the final sphere.

BUILDING COMPLEX SCENES

Throughout this book, I've tried to use XAML code as much as possible. As inconsistent and confusing as XAML sometimes is, it still hides some of the complexity of WPF programming and makes building typical interfaces easier. The designers in Expression Blend and Visual Studio also let you instantly view any changes you make to XAML code so you don't need to run the program to see what you're doing.

XAML code, however, will only get you so far when you're building 3D scenes. XAML is just fine for simple scenes containing a dozen or so triangles that display cubes, tetrahedrons, and other objects with large polygonal faces, but building anything really complex in XAML can be difficult.

For example, the spheres shown in Figure 23-9 each use 1,140 triangles. While in principle you could define all of those triangles in XAML code by hand, in practice that would be extremely difficult and time-consuming. It would also be hard to debug the data if you found that one of the triangles wasn't in exactly the right position or if it was inwardly oriented. It's a lot easier to write a little code-behind to generate spheres and other objects that contain a lot of triangles.

The following sections explain how code can generate some particularly useful 3D objects.

Geometric Shapes

At a low level, it makes sense to build scenes out of triangles because they are simple, it's relatively easy to blend colors across them, and you can easily make more complex shapes out of triangles. At a high level, however, you're more likely to think in terms of rectangles, spheres, cylinders, and other more complex shapes.

Building these shapes out of triangles by hand can be time-consuming and tricky. Even building a rectangle out of two triangles without messing up the orientation and texture coordinates can be harder than you'd think.

To make building complex scenes easier, it's helpful to have a library of code-behind routines that you can use to build these more complex shapes.

The RectanglesAndBoxes example program shown in Figure 23-10 demonstrates routines that draw

truncated cone under the globe), and spheres (the globe).

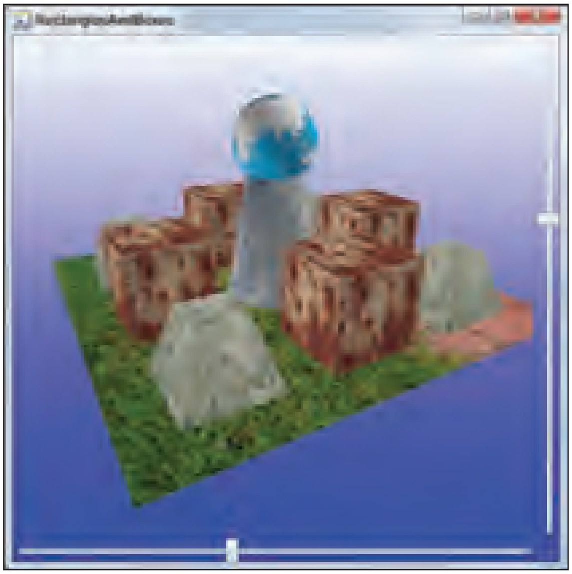


FIGURE 23-10

The following code shows the program's `MakeRectangle` routine, which builds a textured rectangle:

Available for download on Wrox.com

```
// Make a rectangle.
// If rect_mesh is null, make a new one and add it to the model.
// The points p1, p2, p3, p4 should be outwardly oriented.
// The points u1, u2, u3, u4 give the texture coordinates
// for the points p1, p2, p3, p4.
private void MakeRectangle(Model3DGroup rect_model,
    ref MeshGeometry3D rect_mesh, Material rect_material,
    Point3D p1, Point3D p2, Point3D p3, Point3D p4,
    Point u1, Point u2, Point u3, Point u4)
{
    // Make the mesh if we must.
    if (rect_mesh == null)
    {
        rect_mesh = new MeshGeometry3D();
        GeometryModel3D new_model =
            new GeometryModel3D(rect_mesh, rect_material);
        rect_model.Children.Add(new_model);
    }

    // Make the points.
    rect_mesh.Positions.Add(p1);
    rect_mesh.Positions.Add(p2);
    rect_mesh.Positions.Add(p3);
    rect_mesh.Positions.Add(p4);

    // Set the texture coordinates.
    rect_mesh.TextureCoordinates.Add(u1);
    rect_mesh.TextureCoordinates.Add(u2);
```

```
        rect_mesh.TextureCoordinates.Add(u3);
        rect_mesh.TextureCoordinates.Add(u4);

        // Make the triangles.
        int i1 = rect_mesh.Positions.Count - 4;
        rect_mesh.TriangleIndices.Add(i1);
        rect_mesh.TriangleIndices.Add(i1 + 1);
        rect_mesh.TriangleIndices.Add(i1 + 2);

        rect_mesh.TriangleIndices.Add(i1);
        rect_mesh.TriangleIndices.Add(i1 + 2);
        rect_mesh.TriangleIndices.Add(i1 + 3);
    }
```

RectanglesAndBoxes

The code starts by checking whether its `MeshGeometry3D` group is missing. If it is, then the routine makes a new one. Allowing you to pass an existing mesh into the routine lets the program reuse the same mesh for multiple calls to `MakeRectangle` or any of the other object creation routines. (Remember that one mesh that contains many triangles is more efficient than many meshes containing one triangle each, so it's better to use the same mesh if the objects can all use the same material.)

Next, the code adds the rectangle's corner points to the mesh and sets their texture coordinates. Finally, the code adds the two triangles needed to build the rectangle.

The other object-building routines used by the RectanglesAndBoxes program follow the same basic approach: Create a mesh if necessary, create points and set texture coordinates, and create triangles. The big differences are in how the code generates the points and sets their texture coordinates.

Unfortunately, some of these routines are fairly long and complicated (drawing a sphere and mapping a texture onto it takes some work), so they are not shown here. Download the example program from the book's web page to see the details.

BOX BUILDING

The program's `MakeBox` routine calls `MakeRectangle` six times to make the six sides of a box. The version used by this program is relatively simple and uses the same material and texture coordinates for each side. If you look closely at Figure 23-10, you'll see that all of the sides of the metal cubes are the same.

You could modify the routine to use six sets of materials and texture coordinates to give you greater control over box appearances.

Charts and Graphs

Drawing globes and metal boxes is fun but not the sort of thing most businesses need on a daily basis. You can use the same basic tools and techniques, however, to produce more business-oriented results such as 3D charts and graphs.

The BarChart and Graph example programs shown in Figures 23-11 and 23-12, respectively, use the same `MakeRectangle` routine described earlier to draw bar charts and ribbon graphs.

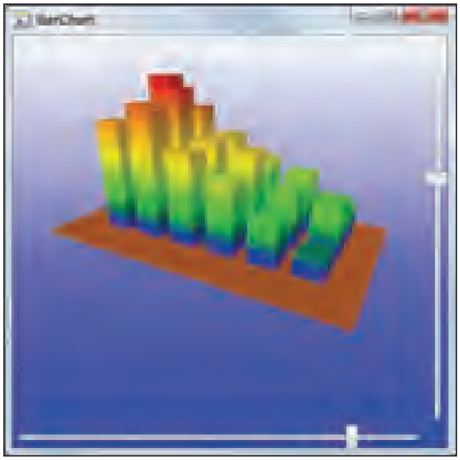


FIGURE 23-11

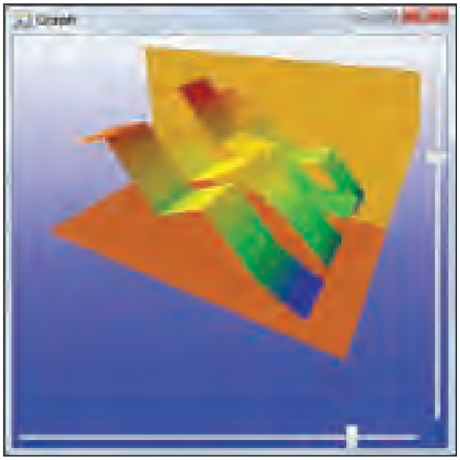


FIGURE 23-12

These two programs only add a few new techniques to those described earlier. Both of these programs color their rectangles with a material that uses a linear gradient brush that shades from blue on the bottom to red on the top. For example, the following code shows how the Graph program builds its material:

Available for download on Wrox.com

```
Dim graph_brush As New LinearGradientBrush()
graph_brush.StartPoint = New Point(0, 0)
graph_brush.EndPoint = New Point(0, 1)
graph_brush.GradientStops.Add(New GradientStop(Colors.Blue, 0.0))
graph_brush.GradientStops.Add(New GradientStop(Colors.Lime, 0.25))
graph_brush.GradientStops.Add(New GradientStop(Colors.Yellow, 0.5))
graph_brush.GradientStops.Add(New GradientStop(Colors.Orange, 0.75))
graph_brush.GradientStops.Add(New GradientStop(Colors.Red, 1.0))
Dim graph_material As New DiffuseMaterial(graph_brush)
```

Graph

The programs then use this material to shade all of the rectangles they draw, setting each point's texture coordinates to a value that depends on its Y coordinate. For example, a point with Y coordinate 0 gets texture coordinates with second component 0. That gives the point the color at the start of the gradient brush — blue.

Other Y coordinates are scaled so that the largest Y values in the data are given texture coordinate values closest to 1; thus, they get colors near the end of the brush — red.

The top and bottom of the bars drawn by the BarChart program have texture coordinates set to the appropriate scaled Y values, so the bottoms are solid blue and the tops depend on the bars' heights.

You could determine directly where to create each triangle in these figures, but the `MakeRectangle` routine makes things a lot easier.

Generated Textures

The chart and graph shown in Figures 23-11 and 23-12 are attractive, but it's hard to see what values they represent without any kind of labels. These programs use gradient textures to give you a sense of which values are large and which are small, but they would be a lot easier to read if the values were labeled.

The LabeledBarChart example program shown in Figure 23-13 displays a bar chart with labels on the rows and columns and a label on the end of each bar, making it much easier to understand the data.

The following code shows the MakeLabel routine that the LabeledBarChart program uses to draw its labels:

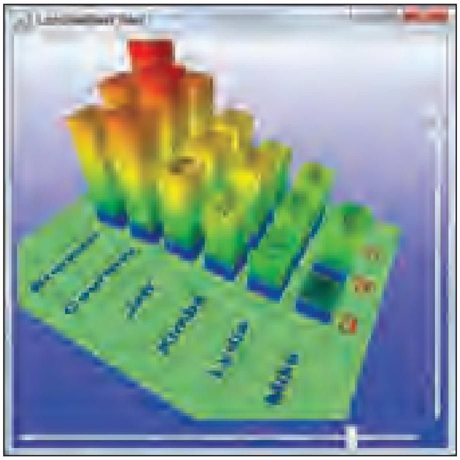


FIGURE 23-13

Available for download on Wrox.com

```
// Make a horizontal label at the specified position.
private void MakeLabel(Model3DGroup chart_model, double wid, double hgt, string txt,
    string font_name, double font_size, FontWeight font_weight, Brush font_brush,
    double y, double x1, double z1, double x2, double z2)
{
    Label lbl = new Label();
    lbl.Width = wid;
    lbl.Height = hgt;
    lbl.Content = txt;
    lbl.HorizontalContentAlignment = HorizontalAlignment.Center;
    lbl.VerticalContentAlignment = VerticalAlignment.Center;
    lbl.Foreground = font_brush;
    lbl.Background = Brushes.Transparent;
    lbl.FontFamily = new FontFamily(font_name);
    lbl.FontSize = font_size;
    lbl.FontWeight = font_weight;
    Brush label_brush = new VisualBrush(lbl);
    Material label_material = new DiffuseMaterial(label_brush);
    MeshGeometry3D label_mesh = null;
    MakeRectangle(chart_model, ref label_mesh, label_material,
        new Point3D(x1, y, z1), new Point3D(x1, y, z2),
        new Point3D(x2, y, z2), new Point3D(x2, y, z1),
        new Point(1, 1), new Point(1, 0), new Point(0, 0), new Point(0, 1));
}
```

LabeledBarChart

This code is actually fairly straightforward. It first creates a `Label`, setting its properties to display the necessary text using the specified font. It uses the `Label` to create a `VisualBrush` and then uses the brush to create a `DiffuseMaterial`. Finally, it creates a rectangle using the new material.

Surfaces

The MakeSurface example program shown in Figure 23-14 is another application that generates a 3D scene from data at run time. This program displays a surface generated by the equation:

$$y = \mathrm{Cos}(x^2 + z^2)/[1 + (x^2 + z^2)/2]$$

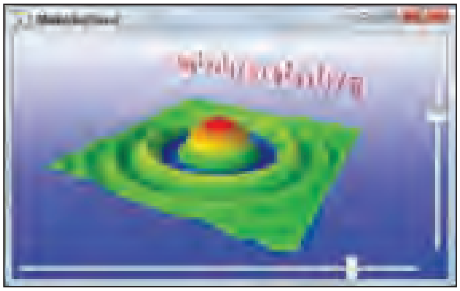


FIGURE 23-14

Like the BarChart, Graph, and LabeledBarChart programs, the MakeSurface program uses a gradient texture that draws larger Y values using warmer colors (yellow and red) and smaller Y values using cooler colors (green and blue).

The MakeSurface program contains no big surprises. It simply loops through X and Z values in an area, calculating the corresponding Y values and building triangles to represent the result. Download the example program from the book's web page to see the details.

SUMMARY

This chapter explains how to produce three-dimensional scenes using WPF. It explains the basic structure of a 3D scene and how to build a scene made of triangles in XAML code. Its examples also demonstrate code-behind techniques for generating more complicated objects such as rectangles, boxes, cylinders, spheres, and surfaces.

For more discussion and examples using WPF's 3D capabilities, see these articles that I wrote for DevX.com:

- `www.devx.com/dotnet/Article/42370`
- `www.devx.com/dotnet/Article/42450`
- `www.devx.com/dotnet/Article/42497`

The second and third articles in particular provide more advanced discussion about building different kinds of brushes and materials, and using 3D transformations.

Check my web site (`www.vb-helper.com`) or e-mail me (`RodStephens@vb-helper.com`) to learn about other examples.

Although WPF's 3D tools have a lot of benefits, they also have several drawbacks including:

- **No Shadows —** While the positions of lights determine an object's appearance, objects don't block light or cast shadows.
- **No Reflections —** You can't have a mirror or chrome ball that reflects what's around it.
- **No Refraction —** Light doesn't bend as it passes through transparent or translucent materials, so you can't get appropriate distortion, for example, when you look through a glass full of water.
- **Suboptimal Performance —** You can get slightly better performance if you use Direct3D directly instead of via WPF.

I suspect that shadow calculations will eventually be provided by graphics hardware and trickle into Direct3D and finally into WPF — but it could take a while. I suspect many of these other issues have low priority for Microsoft and hardware vendors, so I'm not holding my breath.

Even with these drawbacks, WPF's 3D graphics are very impressive. They're easier to use than Direct3D itself and, while they may never compete effectively with high-performance computer games, they are fast enough for many applications.

All of the chapters up to this point have covered WPF. The next chapter describes WPF's cousin, Silverlight. Silverlight is a version of WPF that is intended to bring all of the interactivity and aesthetic appeal of WPF to the browser.

24

Silverlight

Although this is not a Silverlight book, Silverlight is so closely related to WPF that it makes sense to include at least a brief introduction to Silverlight here.

WHAT IS SILVERLIGHT?

Briefly stated, Silverlight is a version of WPF that is intended to run in web browsers. The basic idea is that Silverlight applications should be able to run anywhere on any browser (via a browser plug-in) on any operating system. This idea of running anywhere was emphasized by Silverlight's original name *WPF/E*, where the *E* stood for "everywhere."

To make running Silverlight in a browser possible, it uses a smaller version of the .NET Framework than the version used by WPF. That means some of the features you've been using in WPF are unavailable in Silverlight. The differences between Silverlight and WPF have been changing with each new release of Silverlight and WPF so I'm not even going to try to list them here. Any such list would almost certainly be out-of-date before this book was printed.

A few places you can look to learn about differences between WPF and different versions of Silverlight include:

- **Silverlight Overview** —

  `msdn.microsoft.com/library/bb404700(VS.95).aspx`
- **Silverlight Differences on Windows and the Macintosh** —

  `msdn.microsoft.com/library/cc838247(VS.95).aspx`
- **A short forum discussion about the differences between** WPF and Silverlight —

  `forums.silverlight.net/forums/p/77613/183874.aspx`
- **Programmatic Differences Between Silverlight and WPF** — A Whitepaper

  `realworldsa.dotnetdevelopersjournal.com/`
  `programmaticdifferencesbetweensilverlightwpf.htm`
- **Guidance on Differences Between WPF and Silverlight (another whitepaper)** —

  `wpfslguidance.codeplex.com/Release/ProjectReleases.aspx?ReleaseId=28278`

Because Silverlight applications are intended to run in a user's browser, Silverlight applications do not enjoy the same level of trust as desktop applications. Chances are, someone who uses a Silverlight application on your web site doesn't want to give you access to the system's Registry and files.

While there are some differences between WPF and Silverlight applications, the majority of what you know about WPF applies to Silverlight. In particular, you can still build a user interface with Expression Blend or Visual Studio using familiar controls. Silverlight does not provide all of the controls that WPF does, but it does include the basics such as `Button`, `TextBox`, `StackPanel`, `Grid`, and so on.

A COLOR SELECTION EXAMPLE

As a quick introduction to Silverlight, this section shows how to use Visual Studio to build a simple example program.

Start Visual Studio, open the File menu, and select "New Project" to display the dialog shown in Figure 24-1.

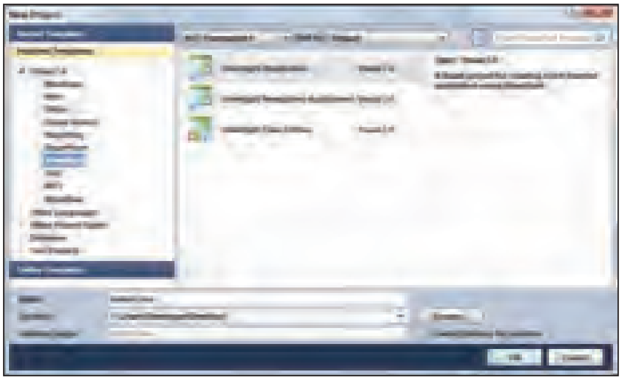


FIGURE 24-1

Expand the template tree on the left to find the Silverlight category for your language (C# or Visual Basic). Select the Silverlight Application template, enter a project name and a location, and click OK to display the dialog shown in Figure 24-2.

Leave the checkbox checked if you want Visual Studio to create a web site for the application. Otherwise, uncheck the checkbox to create an application without a web site.

When you click OK, Visual Studio builds a Silverlight project much as it builds any other kind of project.

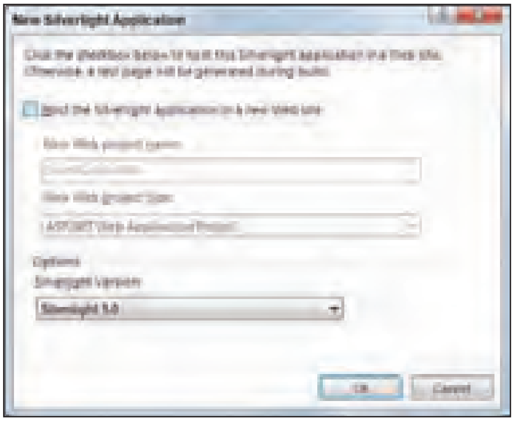


FIGURE 24-2

Figure 24-3 shows the new project. If you look closely, you'll see that Visual Studio created an object named MainPage.xaml.

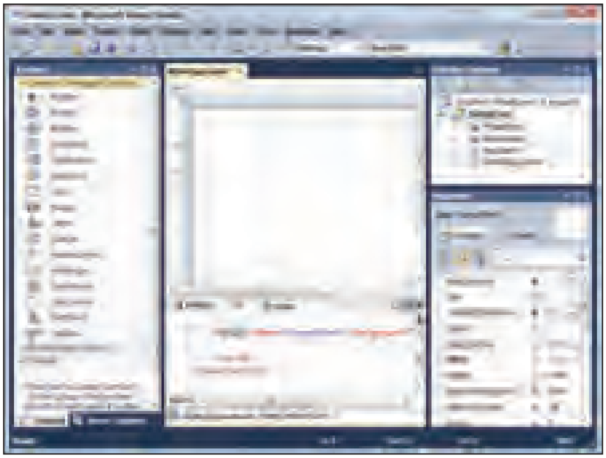


FIGURE 24-3

The middle part of Visual Studio shows a designer surface and XAML Editor similar to those that you use to make a WPF application. The following code shows the XAML code that Visual Studio creates for the new project's MainPage.xaml:

```
<UserControl x:Class="SelectColor.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400">

    <Grid x:Name="LayoutRoot" Background="White">

    </Grid>
</UserControl>
```

This code is similar to the code used by a new WPF project except its top-level object is a UserControl instead of a Window. Both kinds of applications start with a Grid control inside the top-level object to hold any other controls that you may add.

You can use the Toolbox to place controls on the design surface and use the Properties window to set control properties just as you would in a WPF application. One of the main differences is that the WPF Toolbox holds more controls than the Silverlight Toolbox.

This example displays the color selection tool shown in Figure 24-4 running inside a Mozilla Firefox browser. When you change the ScrollBars' values, the program displays their new values in the TextBlocks to the right. It also uses those values as the red, green, and blue color components of a color and displays a sample of that color in the Grid on the far right.

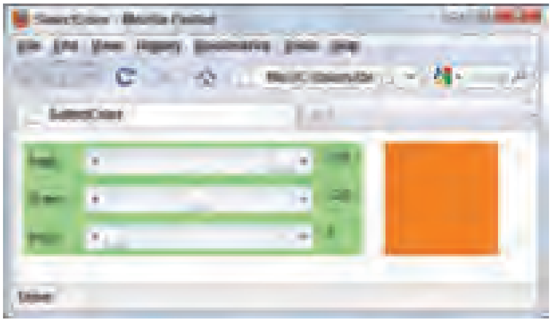


FIGURE 24-4

The following code shows the program's XAML definition:

Available for download on Wrox.com

```
<UserControl x:Class="SelectColor.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="150" d:DesignWidth="450">

    <Grid x:Name="LayoutRoot" Background="White">
        <StackPanel Orientation="Horizontal" VerticalAlignment="Top">
            <Grid Margin="10" Height="100" Width="300"
             Background="LightGreen">
                <Grid.RowDefinitions>
                    <RowDefinition Height="*"/>
                    <RowDefinition Height="*"/>
                    <RowDefinition Height="*"/>
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="Auto"/>
                    <ColumnDefinition Width="0.85*"/>
                    <ColumnDefinition Width="0.15*"/>
                </Grid.ColumnDefinitions>

                <TextBlock Margin="5" Grid.Column="0"
                 Text="Red:" VerticalAlignment="Center"/>
                <TextBlock Margin="5" Grid.Column="0" Grid.Row="1"
                 Text="Green:" VerticalAlignment="Center"/>
                <TextBlock Margin="5" Grid.Column="0" Grid.Row="2"
                 Text="Blue:" VerticalAlignment="Center"/>

                <ScrollBar Margin="5" Grid.Column="1"
                 Orientation="Horizontal" Maximum="255"
                 Name="scrRed" Scroll="scr_Scroll"/>
                <ScrollBar Margin="5" Grid.Column="1"
                 Orientation="Horizontal" Maximum="255"
                 Grid.Row="1" Scroll="scr_Scroll" Name="scrGreen"/>
                <ScrollBar Margin="5" Grid.Column="1"
                 Orientation="Horizontal" Maximum="255"
```

```
                        Grid.Row="2" Scroll="scr_Scroll" Name="scrBlue"/>

                       <TextBlock Margin="5" Grid.Column="2"
                        Text="{Binding ElementName=scrRed,Path=Value}"/>
                       <TextBlock Margin="5" Grid.Column="2" Grid.Row="1"
                        Text="{Binding ElementName=scrGreen,Path=Value}"/>
                       <TextBlock Margin="5" Grid.Column="2" Grid.Row="2"
                        Text="{Binding ElementName=scrBlue,Path=Value}"/>
                   </Grid>
                   <Grid Margin="10" Height="100" Width="100"
                    Background="Black" Name="grdSample"/>
               </StackPanel>
           </Grid>
       </UserControl>
```

SelectColor

If you look at the definitions for the `TextBlocks` on the right, you'll see that their `Text` properties are bound to the corresponding `ScrollBars'` `Value` properties so they automatically update themselves to show the currently selected values.

The program's only remaining task is to display a sample of the selected color. The program does this in the `Scroll` event handler named `scr_Scroll` that is shared by the three `ScrollBars`.

To create this event handler, select one of the `ScrollBars`, click the Properties window's `Events` button, and double-click the `Scroll` event. This opens the Code Editor for the `scr_Scroll` event handler.

The following code shows how the event handler works:

Available for download on Wrox.com

```
// Update the sample.
private void scr_Scroll(object sender,
 System.Windows.Controls.Primitives.ScrollEventArgs e)
{
    // Get the new color.
    Color clr = new Color();
    clr.R = (byte)scrRed.Value;
    clr.G = (byte)scrGreen.Value;
    clr.B = (byte)scrBlue.Value;
    clr.A = 255;

    // Apply the color.
    SolidColorBrush br = new SolidColorBrush(clr);
    grdSample.Background = br;
}
```

SelectColor

This code creates a new `Color` object. It sets the `Color`'s red, green, and blue color components to the values selected by the `ScrollBars`, converted into byte values. It sets the `Color`'s alpha component to 255 so the `Color` is completely opaque.

The code finishes by creating a brush to display the color and setting the sample `Grid` control's `Background` property to that brush.

After you build the XAML and write the code-behind, simply press F5 or open the Debug menu and select "Start Debugging" to launch the application in your system's default browser.

A BOUNCING BALL EXAMPLE

The BouncingBalls example program shown in Figure 24-5 uses a timer to display several balls bouncing around in the browser. This example demonstrates code that more closely ties Silverlight controls with code-behind.

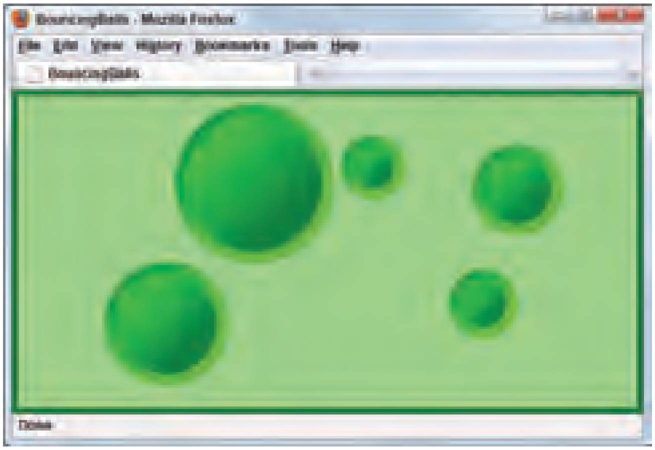


FIGURE 24-5

The following XAML code shows how the program creates its controls. The program simply displays a `Border` containing a `Canvas`.

Available for download on Wrox.com

```
<UserControl x:Class="BouncingBalls.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300" d:DesignWidth="400">

    <Grid x:Name="LayoutRoot">
        <Border BorderBrush="Green" BorderThickness="5">
            <Canvas Name="canField" Background="LightGreen"
             SizeChanged="canField_SizeChanged"/>
        </Border>
    </Grid>
</UserControl>
```

BouncingBalls

There are only two interesting points to the XAML code. First, the `Canvas` has a name so the code-behind can refer to it. Second, the `Canvas` has a `SizeChanged` event handler.

When the `canField` `Canvas` control is resized, the following `SizeChanged` event handler executes. This happens when the control is first created and if the user resizes the control later by resizing the browser.

Available for download on Wrox.com

```
private DateTime LastUpdate;
private List<Ball> Balls;
private DispatcherTimer Clock;

// Get ready to run.
private void canField_SizeChanged(object sender, SizeChangedEventArgs e)
{
    GetReady();
}

// Get ready to run.
private void GetReady()
{
    // Delete any old Balls. Necessary if the user resizes.
    canField.Children.Clear();

    // Make some Balls.
    Balls = new List<Ball>();
    Random rand = new Random();
    Rect bounds = new Rect(0, 0, canField.ActualWidth, canField.ActualHeight);
    for (int i = 1; i <= 5; i++)
    {
        Balls.Add(RandomBall(rand, bounds, canField));
    }

    // Prepare the timer.
    LastUpdate = DateTime.Now;
    Clock = new DispatcherTimer();
    Clock.Interval = TimeSpan.FromSeconds(0.1);
    Clock.Tick += Clock_Tick;
    Clock.Start();
}
```

BouncingBalls

The program declares some class-level variables that hold the time when the balls were last updated, the references to the balls, and a `DispatchTimer` object that provides periodic `Tick` events.

The `SizeChanged` event handler simply calls `GetReady`. That function removes any `Ellipses` that are currently in the `Canvas`. (Try commenting out this step to see an interesting result.)

Next, the code uses the `RandomBall` function to create five random `Ball` objects, storing them in the `Balls` list. The `Ball` class keeps track of the size, position, and velocity of balls moving on the screen and is described shortly.

The `RandomBall` function simply creates a new `Ball` and initializes its properties randomly. It is fairly straightforward so it isn't shown here.

The `GetReady` function finishes by preparing the `Clock` `DispatchTimer`. It creates the timer, sets its `Interval` property so it ticks every tenth of a second, gives it a new `Tick` event handler, and starts it.

The following code shows the timer's `Tick` event handler that fires roughly every tenth of a second:

Available for download on Wrox.com

```
// Update the balls.
private void Clock_Tick(object sender, EventArgs e)
{
    DateTime time_now = DateTime.Now;
    TimeSpan elapsed = time_now.Subtract(LastUpdate);

    foreach (Ball a_ball in Balls)
    {
        a_ball.Update(elapsed);
    }

    LastUpdate = time_now;
}
```

BouncingBalls

The following code shows the program's `Ball` class:

Available for download on Wrox.com

```
public class Ball
{
    public Shape MyShape { get; set; }
    public TranslateTransform Transform { get; set; }
    public double Vx { get; set; }
    public double Vy { get; set; }
    public Rect Bounds { get; set; }

    // Constructor.
    public Ball(Shape new_shape, TranslateTransform initial_transform,
        double initial_vx, double initial_vy, Rect the_bounds)
    {
        MyShape = new_shape;
        Transform = initial_transform;
        Vx = initial_vx;
        Vy = initial_vy;
        Bounds = the_bounds;
    }

    // Update our position for the elapsed time.
    public void Update(TimeSpan elapsed)
    {
        double x = Transform.X + Vx * elapsed.TotalSeconds;
        if ((Vx < 0) && (x < Bounds.Left))
        {
            // We hit the left wall.
            Vx = -Vx;
            x += 2 * (Bounds.Left - x);
        }
        else if ((Vx > 0) && (x + MyShape.Width > Bounds.Right))
        {
            // We hit the right wall.
            Vx = -Vx;
            x -= 2 * (x + MyShape.Width - Bounds.Right);
```

```
            }

            double y = Transform.Y + Vy * elapsed.TotalSeconds;
            if ((Vy < 0) && (y < Bounds.Top))
            {
                // We hit the top wall.
                Vy = -Vy;
                y += 2 * (Bounds.Top - y);
            }
            else if ((Vy > 0) && (y + MyShape.Width > Bounds.Bottom))
            {
                // We hit the bottom wall.
                Vy = -Vy;
                y -= 2 * (y + MyShape.Width - Bounds.Bottom);
            }

            Transform.X = x;
            Transform.Y = y;
        }
    }
```

BouncingBalls

The `MyShape` property holds a reference to the Silverlight `Shape` control (an `Ellipse` in this example) that the `Ball` represents. The `Transform` property holds a reference to a `TranslateTransform` object applied to the `Shape`. To move itself, the `Ball` updates this transform.

The `Vx` and `Vy` properties give the X and Y components of the `Ball`'s velocity in pixels per second. The `Bounds` property is a `Rect` object that determines where the ball is allowed to be.

The most interesting code in this class is its `Update` method. This function adds the ball's velocity times the elapsed time since the last update to its current position. For example, suppose the ball is currently at position (30, 60) moving 100 pixels per second to the right, so `Vx` = 100. Also suppose it has been 0.2 seconds since the last time the ball's position was updated. Then the `Update` method changes the ball's X coordinate to 30 + 100 * 0.2.

After calculating the ball's new position, the code determines whether the ball has hit one of the edges of its `Bounds` rectangle, and, if it has, the code reverses the ball's velocity component appropriately.

Having calculated the ball's new position, the code updates the `Transform` property to move the ball's `Shape` to its new location.

FOR MORE INFORMATION

This chapter is of necessity but rather short. The intent is to give you a small taste of Silverlight so you know how Silverlight relates to WPF. For more information, see a book about Silverlight such as *Silverlight 3 Programmer's Reference* (J. Ambrose Little et al., Wrox, 2009) or *Silverlight 4 Problem–Design–Solution* (Nick Lecrenski, Wrox, 2010).

There are also many places where you can turn for information on the Internet. The following list describes some links that you may find useful while learning Silverlight:

- **Silverlight homepage —**

  `msdn.microsoft.com/library/ee656762(VS.95).aspx`

  This page provides links to pages for specific Silverlight versions. Currently these include Silverlight 4 Beta and Silverlight 3. For example, if the Silverlight 4 Beta link described next goes away, you should be able to find the latest Silverlight 4 information here.

- **Silverlight 4 homepage —**

  `msdn.microsoft.com/library/cc838158(VS.96).aspx`

  Links to complete information about Silverlight 4 Beta. Topics include an overview, controls, input, printing, graphics, animation, communications, performance, deployment, and more.

- **Silverlight 3 homepage —**

  `msdn.microsoft.com/library/cc838158(VS.95).aspx`

  This is similar to the previous page but for Silverlight 3.

- **Get Started Building Silverlight 3 Applications —**

  `silverlight.net/getstarted`

  This is the official Silverlight "getting started" web page. It contains lots of useful links and a step-by-step tutorial by Silverlight program manager Tim Heuer that walks you through an example that uses Twitter's Search Web service.

- **Silverlight 4 Beta —** A Guide to the New Features

  `timheuer.com/blog/archive/2009/11/18/whats-new-in-silverlight-4-complete-guide-new-features.aspx`

  A long blog entry by Tim Heuer describing Silverlight 4 tools, resources, and new features.

SUMMARY

Silverlight is a version of WPF intended to run applications in web browsers. To squeeze Silverlight applications into a format that will run reasonably on a browser, Silverlight has a few restrictions such as fewer controls and more restricted access to the user's system than WPF has, but there are many similarities between the two.

You can still use Visual Studio and Expression Blend to build Silverlight applications that use C# or Visual Basic code-behind. Both Visual Basic and Expression Blend provide feature-rich interactive design tools that let you drop controls on the design surface and set their properties. While you will need to learn some new techniques to get the most out of Silverlight, your knowledge of WPF should give you a big head start.

This chapter concludes the more tutorial part of this book. The appendixes that follow provide a handy reference for WPF and XAML syntax and common usage. If you need an in-depth introduction to a particular topic such as data binding or commanding, look at the corresponding chapters earlier in this book. If you only need to refresh your memory about a topic's syntax, see the corresponding appendix.

A

Common Properties

This appendix summarizes properties that are common to many WPF controls. See the earlier chapters and the following appendixes for more information about specific kinds of controls.

GENERAL PROPERTIES

Table A-1 summarizes general properties that apply to many controls. Note that not all properties apply to all control types.

TABLE A-1: General Properties

PROPERTY	PURPOSE
`AcceptsReturn`	Determines whether a [Return] character is inserted into a `TextBox` or `RichTextBox`, or whether a [Return] character in this control is ignored.
`AcceptsTab`	Determines whether a [Tab] character is inserted into a `TextBox` or `RichTextBox`, or whether a [Tab] character moves focus to the next control.
`Background`	The control's background color
`BorderBrush`	Determines the color of the control's border. Note that you must set `BorderBrush` to a visible color and `BorderThickness` to a value >0 before you can see the border.
`BorderThickness`	Determines the thickness of the control's border. Note that you must set `BorderBrush` to a visible color and `BorderThickness` to a value >0 before you can see the border.

continues

TABLE A-1 *(continued)*

PROPERTY	PURPOSE
`Content`	The content that the control should contain. Some controls (such as `Button` and `Label`) can hold only a single child as content, while others (such as `Grid` and `StackPanel`) can have many children. If a control can display text (such as a `Button` or `Label`), then you can set the `Content` property to a string for the control to display.
`ContextMenu`	The `ContextMenu` that the control should display when the user right-clicks on it
`Cursor`	Determines the cursor displayed by the control. This can be None, No, Arrow, AppStarting, Cross, Help, IBeam, SizeAll, SizeNESW, SizeNS, SizeNWSE, SizeWE, UpArrow, Wait, Hand, Pen, ScrollNS, ScrollWE, ScrollAll, ScrollN, ScrollS, ScrollW, ScrollE, Scroll, ScrollNW, ScrollNE, ScrollSW, ScrollSE, or ArrowCD.
`Foreground`	The control's foreground color, usually used for text
`Header`	Some controls that display two kinds of content use a `Content` property for one kind and a `Header` property for the other. For example, a `GroupBox` displays a caption (`Header`) in addition to content contained inside the `GroupBox`'s border (`Content`). A few other controls such as `TreeViewItem` also use a `Header` property to hold special kinds of content.
`Height`	Determines the control's absolute height. Often interfaces are more flexible if you let the control stretch to fill its parent rather than setting this value explicitly.
`HorizontalAlignment`	Determines whether the control is aligned horizontally to the left, center, or right in its container. The special value `Stretch` makes the control try to fill its container horizontally.
`HorizontalContentAlignment`	Determines how content is aligned horizontally within the control. For example, you can use this property to align the text within a `Label`.
`HorizontalScrollBarVisibility`	Determines whether the control's horizontal scrollbar is visible. This property can take the values `Auto` (displayed when needed), `Disabled` (not displayed), `Hidden` (not displayed), and `Visible` (always displayed).
`IsEnabled`	Determines whether the control will respond to user actions.

PROPERTY	PURPOSE
`IsReadOnly`	Determines whether the user can modify the text in a `TextBox` or `RichTextBox`. Even if `IsReadOnly` is `True`, the user can select text and press [Ctrl]+C to copy it to the clipboard, so this is a useful technique for displaying text that the user might want to copy.
`IsTabStop`	Determines whether the user can tab to a focusable control. Even if `IsTabStop` is `False`, the user can click on the control to give it focus.
`IsUndoEnabled`	Determines whether the user can press [Ctrl]+Z and [Ctrl]+Y to undo and redo changes in a `TextBox` or `RichTextBox`.
`LayoutTransform`	The transformation used to translate, scale, rotate, and skew the control before its container arranges its child controls
`Margin`	Determines how much space is left around the control within its container. This property can include one value (all four margins use the same value), two values (left/right and top/bottom margins), or four values (left, top, right, and bottom values).
`MaxHeight`	The largest height the control will give itself
`MaxLines`	The maximum number of lines that a `TextBox` will display. This is only meaningful if `AcceptsReturn` is `True`.
`MaxWidth`	The largest width the control will give itself
`MinHeight`	The smallest height the control will give itself
`MinLines`	The minimum number of lines that a `TextBox` will display. This is only meaningful if `AcceptsReturn` is `True`.
`MinWidth`	The smallest width the control will give itself
`Name`	The control's name. This is often not needed if no XAML or code-behind refers to the control.
`Opacity`	Determines how opaque the control is. The value 0 means completely transparent and 1 means completely opaque (although Expression Blend displays percentage values between 0 and 100 in the Properties window).
`OpacityMask`	A brush that determines the opacity at various parts of the control. Often this is a gradient brush or is defined by an image. Note that only the alpha components of the brush's colors matter and the red, green, and blue components are ignored.

continues

TABLE A-1 *(continued)*

PROPERTY	PURPOSE
`Padding`	Determines the extra space added inside this control around its contents.
`Parent`	The control's logical parent
`RenderTransform`	The transformation used to translate, scale, rotate, and skew the control after its container arranges its child controls but before it draws this one
`Resources`	Resources defined by this control. Typically this includes resource values and styles for use by controls contained within this one.
`SelectionLength`	Gets or sets the length of the `TextBox`'s current selection.
`SelectionStart`	Gets or sets the starting position of the `TextBox`'s current selection.
`SelectionText`	Gets or sets the text in the `TextBox`'s current selection.
`Stretch`	Determines how a control such as an `Image` or `Viewbox` stretches its contents. This can be `Fill` (stretch contents to fill the control even if that distorts the contents), `None` (contents keep their original size), `Uniform` (stretch contents uniformly as much as possible while still fitting within the control), and `UniformToFill` (stretch contents uniformly until the control is filled even if the contents are partially clipped).
`Style`	Determines the style used by the control to define such things as property values.
`TabIndex`	Determines the control's position in the tab order.
`Tag`	Any arbitrary data you want to attach to the control
`Text`	Determines the text contained in a `TextBox` or `TextBlock`.
`TextAlignment`	Determines how text is arranged in a `TextBox` or `TextBlock`. This can be `Center`, `Justify`, `Left`, or `Right`.
`TextWrapping`	Determines how text is wrapped if it won't fit in a `TextBox` or `TextBlock`. This can be `NoWrap` (lines are truncated), `Wrap` (lines wrap and very long words may be split if they won't fit), or `WrapWithOverflow` (lines wrap and very long words may be truncated if they won't fit).
`ToolTip`	The text that the control should display in a tooltip when the mouse hovers over it

PROPERTY	PURPOSE
`VerticalAlignment`	Determines whether the control is aligned vertically to the top, center, or bottom in its container. The special value `Stretch` makes the control try to fill its container vertically.
`VerticalContentAlignment`	Determines how content is aligned vertically within the control. For example, you can use this property to align the text within a `Label`.
`VerticalScrollBarVisibility`	Determines whether the control's vertical scrollbar is visible. This property can take the values `Auto` (displayed when needed), `Disabled` (not displayed), `Hidden` (not displayed), and `Visible` (always displayed).
`Visibility`	Determines whether the control is visible. This can take the values `Visible` (visible as usual), `Hidden` (not visible but the layout saves room for it), and `Collapsed` (not visible and the layout does not save room for it).
`Width`	Determines the control's absolute width. Often interfaces are more flexible if you let the control stretch to fill its parent rather than setting this value explicitly.

FONT PROPERTIES

Table A-2 summarizes font properties. These are useful for controls that display text as well as containers that may hold controls that display text. In particular, you can set these properties for a `Window`, and all of the controls in the `Window` will use them.

TABLE A-2: Font Properties

PROPERTY	PURPOSE
`FontFamily`	The name of the font such as Times New Roman or Arial. Microsoft's preferred font for Vista applications is Segoe (pronounced *see-go*).
`FontSize`	The font's size, by default in pixels. Append cm, in, px, or pt to specify centimeters, inches, pixels (the default), or points (1/72 inch) as in "0.25in" or "12pt."
`FontStyle`	The font's style. This can be `Normal`, `Italic`, or `Oblique` (simulates italic for fonts without an italic style).

continues

TABLE A-2 *(continued)*

PROPERTY	PURPOSE
`FontWeight`	The font's "density." This can be a number or one of the values `Thin`, `ExtraLight`, `Light`, `Normal`, `Medium`, `SemiBold`, `Bold`, `ExtraBold`, `Black`, or `ExtraBlack`, although many fonts look the same for many of these values.

DRAWING PROPERTIES

Table A-3 summarizes properties that determine the appearance of graphical objects.

TABLE A-3:

PROPERTY	PURPOSE
`Fill`	The control's background color, usually used for drawing controls such as `Ellipse` and `Rectangle`
`Stroke`	The control's foreground color, usually used for lines and curves in drawing controls such as `Ellipse` and `Rectangle`
`StrokeDashArray`	An array of values indicating the number of line widths to draw and skip while making a dashed line
`StrokeDashCap`	The style to use for the ends of dashes in a dashed line. This can be `Flat`, `Round`, `Square`, or `Triangle`.
`StrokeDashOffset`	The distance into its dash pattern where a line starts
`StrokeEndLineCap`	The style to use for the line's end. This can be `Flat`, `Round`, `Square`, or `Triangle`.
`StrokeLineJoin`	The style used to make the corners between segments in a connected series of segments, for example, in a `Polygon` or `Polyline`. This can be Bevel, Miter, or Round.
`StrokeMiterLimit`	The maximum size that will be mitered in a corner. If a corner is too sharp, the corner is beveled.
`StrokeStartLineCap`	The style to use for the line's start. This can be `Flat`, `Round`, `Square`, or `Triangle`.
`StrokeThickness`	The width of lines and curves produced by drawing controls

BITMAP EFFECT PROPERTIES

Table A-4 summarizes bitmap effect properties.

TABLE A-4:

PROPERTY	PURPOSE
`BevelBitmapEffect`	Adds beveled edges to the control.
`BlurBitmapEffect`	Makes the control blurred.
`DropShadowBitmapEffect`	Adds a shadow behind the control.
`EmbossBitmapEffect`	Gives the control an embossed appearance.
`OuterGlowBitmapEffect`	Adds a glow behind the control.

GRID ATTACHED PROPERTIES

When you place a control inside a `Grid`, you can use attached properties defined by the `Grid` to determine the control's placement. Table A-5 summarizes the `Grid` attached properties.

TABLE A-5:

PROPERTY	PURPOSE
`Grid.Column`	Determines the column that contains the control.
`Grid.ColumnSpan`	Determines the number of columns that the control spans.
`Grid.Row`	Determines the row that contains the control.
`Grid.RowSpan`	Determines the number of rows that the control spans.

Use the `Grid`'s `Grid.ColumnDefinitions` and `Grid.RowDefinitions` property elements to define the `Grid`'s rows and columns and their sizes.

DOCKPANEL ATTACHED PROPERTIES

When you place a control inside a `DockPanel`, you can use the `DockPanel.Dock` attached property to determine where the control is positioned within the `DockPanel`. This property can take the values `Left`, `Right`, `Top`, and `Bottom`.

For example, if you set a `Label`'s `DockPanel.Dock` property to `Left`, then the `Label` will fill the left edge of the remaining space in the control.

If the `DockPanel`'s `LastChildFill` property is `True` — which it is by default — then the control makes its last child fill any remaining space.

CANVAS ATTACHED PROPERTIES

When you place a control inside a `Canvas`, you can use attached properties defined by the `Canvas` to determine the control's placement. Table A-6 summarizes the `Canvas` attached properties.

TABLE A-6:

PROPERTY	PURPOSE
`Canvas.Bottom`	Determines the position of the control's bottom edge.
`Canvas.Left`	Determines the position of the control's left edge.
`Canvas.Right`	Determines the position of the control's right edge.
`Canvas.Top`	Determines the position of the control's top edge.

The `Canvas` will only set the control's position, not its size, so it will only set one of `Left/Right` and `Top/Bottom`.

B

Content Controls

This appendix summarizes controls that are designed to hold content. They display something that the user should see but should generally not modify.

The following sections very briefly summarize the WPF content controls. They list the controls' most important properties and provide simple XAML examples. For greater detail on the controls, see Chapter 5.

Many of these controls provide decoration for a single child. For example, you can place a single child inside a `Border` or `GroupBox`. Remember that this child can be a container such as a `Grid` or `StackPanel`, so being able to hold only one child isn't really much of a restriction.

BORDER

The `Border` draws a border around or background behind a single child control. The key properties are given in Table B-1.

TABLE B-1: Key Properties of Border

PROPERTY	PURPOSE
`Background`	The color used to fill the control's interior
`BorderBrush`	The color of the control's edge
`BorderThickness`	The control edge's thickness
`CornerRadius`	The radius of curvature for the control's corners. Areas outside the curve are transparent so the parent shows through.

EXAMPLE **Border and Grid**

The following XAML code makes a simple `Border` containing a `Grid` that could hold other controls:

```
<Border BorderBrush="Orange" BorderThickness="3" Background="Yellow"
 CornerRadius="20" Width="100" Height="100">
    <Grid>
        ...
    </Grid>
</Border>
```

BULLETDECORATOR

The `BulletDecorator` displays a single item and bullet in a bulleted list. The `Bullet` property defines the bullet. The control's single child defines the content. The key properties are given in Table B-2.

TABLE B-2: Key Properties of BulletDecorator

PROPERTY	PURPOSE
`Bullet`	The object displayed as the bullet
`Child`	The object displayed next to the bullet

EXAMPLE **Diamond-Shaped BulletDecorator**

The following XAML code creates a blue diamond bullet to the left of the text "Bullet Point." The code uses `Margin` properties to provide some separation between the bullet and the text.

```
<BulletDecorator>
    <BulletDecorator.Bullet>
        <Polygon Margin="2,0,0,0" Points="0,5 5,0 10,5 5,10" Fill="Blue"/>
    </BulletDecorator.Bullet>
    <TextBlock Margin="10,0,0,0" Text="Bullet Point"/>
</BulletDecorator>
```

Note that the `BulletDecorator` does not arrange controls in the same way that a `StackPanel` does. In particular, increasing the bullet's right margin does not move the `TextBlock` farther to the right. That's why this code uses the `TextBlock`'s `Margin` property.

DOCUMENTVIEWER

You can build a `FixedDocument` object inside a `DocumentViewer` in XAML code, but often a program loads the `DocumentViewer`'s content at run time.

EXAMPLE Viewing an XPS Document

The following XAML code defines a `DocumentViewer` named `docViewer`:

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="ViewXpsDocument.MainWindow"
    x:Name="Window"
    Title="ViewXpsDocument"
    Width="640" Height="480">
    <DocumentViewer Name="docViewer" />
</Window>
```

The following code-behind loads the file "Fixed Document.XPS" into the viewer when the program loads:

```
// Load the XPS document.
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Note: Mark the file as "Copy if newer"
    // to make a copy go in the output directory.
    XpsDocument xps_doc =
        new XpsDocument("Fixed Document.xps", System.IO.FileAccess.Read);
    docViewer.Document = xps_doc.GetFixedDocumentSequence();
}
```

FLOWDOCUMENT

The `FlowDocument` object represents a document with contents that can flow to use the available space.

Content Objects

Table B-3 summarizes some of the most useful objects that a `FlowDocument` can contain.

TABLE B-3: FlowDocument Objects

OBJECT	PURPOSE
`Paragraph`	Groups text. Contains other objects.
`Table`	Displays contents in rows and columns.
`List`	Displays items in a numbered or bulleted list.
`Floater`	Displays content that can float to different positions within the document to improve layout. Will appear wherever there is room.

continues

TABLE B-3 *(continued)*

OBJECT	PURPOSE
`Figure`	Displays content in a separate area with text flowing around it. You can set its alignment within the flowing content, and it can span multiple columns.
User interface elements	`TextBox`, `Button`, and other elements sitting within the `FlowDocument`.
Three-dimensional objects	Displays 3D output.

EXAMPLE FlowDocument Objects

The following XAML code demonstrates simple examples of the most common objects contained in a `FlowDocument`:

Available for download on Wrox.com

```
<FlowDocument FontFamily="Arial" FontSize="12" Background="Lime" Foreground="Black">
    <!--  A Paragraph containing a Run that uses a different font.  -->
    <Paragraph>This is a
        <Run FontStyle="Italic" Foreground="Red" FontWeight="Bold">simple</Run>
        Paragraph that contains a Run.</Paragraph>

    <!--  A Table with two rows. The first has a different style
         (perhaps to use as a header).  -->
    <Table BorderBrush="Black" BorderThickness="1" FontSize="12">
        <Table.Columns>
            <TableColumn Width="25*" />
            <TableColumn Width="75*" />
        </Table.Columns>
        <TableRowGroup>
            <TableRow Background="LightGray" FontSize="15">
                <TableCell>
                    <Paragraph>Item</Paragraph>
                </TableCell>
                <TableCell>
                    <Paragraph>Purpose</Paragraph>
                </TableCell>
            </TableRow>
            <TableRow>
                <TableCell>
                    <Paragraph>List</Paragraph>
                </TableCell>
                <TableCell>
                    <Paragraph>Display items in a list</Paragraph>
                </TableCell>
            </TableRow>
        </TableRowGroup>
    </Table>

    <!--  A block containing a user interface element.  -->
    <BlockUIContainer>
        <Button Name="btnBlockUI" Width="100" Height="50"
         Content="Block UI Button" />
```

```
        </BlockUIContainer>

        <!--  A Paragraph containing controls inline.  -->
        <Paragraph>
            Some inline stuff:
            <Button Name="btnAButton">A Button</Button>
            <TextBox>A Text Box</TextBox>
            <Ellipse Width="50" Height="30"
             Fill="Yellow" Stroke="Red" StrokeThickness="5" />
        </Paragraph>

        <!--  A Paragraph that defines a Floater.  -->
        <Paragraph>
            <Floater HorizontalAlignment="Right">
                <Paragraph BorderBrush="Black" BorderThickness="1">
                    <StackPanel Margin="2,2,2,2">
                        <Ellipse Fill="Yellow" Width="40" Height="20"/>
                        <TextBlock Margin="5" FontStyle="Italic">
                            Figure 1. A floating Ellipse.</TextBlock>
                    </StackPanel>
                </Paragraph>
            </Floater>
        </Paragraph>

        <!--  Another Paragraph that can sit beside the Floater.  -->
        <Paragraph>
            A Floater holds content that can float to other locations.
        </Paragraph>

        <!--  A List.  -->
        <List>
            <ListItem>
                <Paragraph>
                    Ellipse
                    <Ellipse Width="20" Height="20" Stroke="Black" StrokeThickness="1"/>
                </Paragraph>
            </ListItem>
            <ListItem>
                <Paragraph>
                    Triangle
                    <Polygon
                     Stroke="Black" StrokeThickness="1"
                     Points="0,0 20,0 10,20" />
                </Paragraph>
            </ListItem>
        </List>
    </FlowDocument>
```

ShowFlowDocument

Paragraph objects typically make up most of a `FlowDocument`'s content. This object has a full set of font properties in addition to `Background` and `Foreground`.

For more information on `FlowDocuments`, see Chapter 21.

FLOWDOCUMENTPAGEVIEWER

The `FlowDocumentPageViewer` displays a `FlowDocument` in page viewing mode. In this mode, the user sees a single page at a time and uses controls at the bottom of the viewer to move between pages.

EXAMPLE Using FlowDocumentPageViewer

To use a `FlowDocumentPageViewer`, simply place a `FlowDocument` inside it.

```
<FlowDocumentPageViewer>
    <!--  Insert FlowDocument here.  -->
</FlowDocumentPageViewer>
```

See the earlier entry for information about `FlowDocuments`.

FLOWDOCUMENTREADER

The `FlowDocumentReader` displays a `FlowDocument` in any of Page, Scroll, or TwoPage modes.

Viewing Mode Values

Table B-4 summarizes the `FlowDocumentReader` control's `ViewMode` values.

TABLE B-4: ViewMode Values for FlowDocumentReader

MODE	MEANING
Page	Behaves like a `FlowDocumentPageViewer`.
Scroll	Behaves like a `FlowDocumentScrollViewer`.
TwoPage	Displays the document two pages at a time side-by-side.

EXAMPLE Using FlowDocumentReader

To use a `FlowDocumentReader`, simply place a `FlowDocument` inside it. You can use the `ViewingMode` property to determine which viewing mode the control initially uses, as in the following code:

```
<FlowDocumentReader ViewingMode="TwoPage">
    <!--  Insert FlowDocument here.  -->
</FlowDocumentReader>
```

See the earlier entry for information about `FlowDocuments`.

FLOWDOCUMENTSCROLLVIEWER

The `FlowDocumentScrollViewer` displays a `FlowDocument` in Scroll mode as a long, single, scrolling page much as a web browser typically displays a long web page.

EXAMPLE Using FlowDocumentScrollViewer

To use a `FlowDocumentScrollViewer`, simply place a `FlowDocument` inside it.

```
<FlowDocumentScrollViewer>
    <!--  Insert FlowDocument here.  -->
</FlowDocumentScrollViewer>
```

See the earlier entry for information about `FlowDocuments`.

GROUPBOX

A `GroupBox` displays a header and a border around a single child. The `Header` property determines the text that the `GroupBox` displays. The `Foreground` property determines the text's color. See Table B-5 for the key properties of `GroupBox`.

TABLE B-5: Key Properties of GroupBox

PROPERTY	PURPOSE
`Background`	The color used to fill the control's interior
`BorderBrush`	The color of the control's edge
`BorderThickness`	The control edge's thickness
`Foreground`	The color used to display the `Header`
`Header`	The text displayed at the top of the `GroupBox`

EXAMPLE GroupBox and "Grid"

The following code defines a simple `GroupBox` containing a `Grid`:

```
<GroupBox Width="300" Height="200"
 Header="Customer Address" BorderBrush="Red" Foreground="Blue">
    <Grid>
        <!--  Insert controls in the Grid here.  -->

    </Grid>
</GroupBox>
```

IMAGE

An `Image` displays a picture. The key properties are listed in Table B-6.

TABLE B-6:

PROPERTY	PURPOSE
`Source`	Tells where the `Image`'s picture is stored. This can be the name of a picture that is part of the project or it can be an absolute web address.
`Stretch`	Determines how the control stretches its picture. This can be `None`, `Fill`, `Uniform`, and `UniformToFill`.

EXAMPLE Loading Images

The following code creates two `Images`. The first loads its picture from the project file MadScientist.jpg. The second `Image` gets its file from the web address `www.vb-helper.com/howto_2005_buddhabrot.jpg`.

```
<Image Width="200" Stretch="Uniform" Source=" MadScientist.jpg"/>
<Image Width="200" Stretch="Uniform"
 Source="http://www.vb-helper.com/howto_2005_buddhabrot.jpg"/>
```

In code-behind, you can set the `Source` property to a `BitmapImage` object.

The following code displays two pictures at run time. It loads the first from the file amy007.jpg stored in the project as a resource and the second from a web address.

```
img1.Source = new BitmapImage(new Uri("pack://application:,,,/amy007.jpg"));
img2.Source = new BitmapImage(new
    Uri("http://www.vb-helper.com/howto_2005_buddhabrot.jpg"));
```

LABEL

The `Label` displays text that the user can read but not modify. The key properties are given in Table B-7.

TABLE B-7: Key Properties of Label

PROPERTY	PURPOSE
`Background`	The color behind the text
`BorderBrush`	The color of the control's edge
`BorderThickness`	The control edge's thickness

PROPERTY	PURPOSE
`Content`	The control's content, usually simple text
`FontFamily`	The name of the font family as in Times New Roman, Arial, or Segoe (pronounced *see-go*)
`FontSize`	The text's font size. Append *cm*, *in*, *px*, or *pt* to specify centimeters, inches, pixels (the default), or points (1/72 inch) as in "0.25in" or "12pt."
`FontStyle`	Can be `Normal`, `Italic`, or `Oblique` (simulates italic for fonts without an italic style)
`FontWeight`	The text's *boldness*. This can be `Thin`, `ExtraLight`, `Light`, `Normal`, `Medium`, `SemiBold`, `Bold`, `ExtraBold`, `Black`, and `ExtraBlack`.
`Foreground`	The color used to draw the text

CLEVER COPYING

If you want the user to be able to copy text to the clipboard, consider using a read-only `TextBox` instead of a `Label`.

EXAMPLE Simple Label

The following code creates a simple `Label`:

```
<Label Content="First Name:" FontSize="20" FontWeight="Bold"/>
```

LISTVIEW

The `ListView` displays data items in one of several layouts. The `GridView` is probably the most common. The key properties are given in Table B-8.

TABLE B-8: Key Properties of ListView

PROPERTY	PURPOSE
`Background`	The color behind the text
`BorderBrush`	The color of the control's edge
`BorderThickness`	The control edge's thickness

continues

TABLE B-8 *(continued)*

PROPERTY	PURPOSE
`ItemsSource`	The collection used to generate the control's contents
`View`	Defines how the data is displayed. The most common view is `GridView`.

EXAMPLE ListView Binding

The following code defines a `ListView` that displays book information:

```
<ListView Name="lvwPeople"
 Background="{x:Null}"
 ItemsSource="{Binding}">
    <ListView.View>
        <GridView>
            <GridViewColumn Header="Author" Width="100"
             DisplayMemberBinding="{Binding Path=Author}"/>
            <GridViewColumn Header="Title" Width="300"
             DisplayMemberBinding="{Binding Path=Title}"/>
            <GridViewColumn Header="Year" Width="50"
             DisplayMemberBinding="{Binding Path=Year}"/>
            <GridViewColumn Header="Price" Width="50"
             DisplayMemberBinding="{Binding Path=Price}"/>
        </GridView>
    </ListView.View>
</ListView>
```

The control is bound to an `ObservableCollection` of `BookInfo` objects created in code-behind.

Each `GridViewColumn` object's `DisplayMemberBinding` property gives the name of the property in the corresponding `BookInfo` object that the column should display. For example, the first column displays the `BookInfo` object's `Author` property.

See the "ListView" section in Chapter 5 for more information about this control.

MEDIAELEMENT

The `MediaElement` displays audio or video media. See Appendix E for more information about `MediaElement`.

POPUP

The `Popup` displays a floating area over a window. `ContextMenu` and `ToolTip` controls display similar floating areas. See Table B-9 for the key properties.

> **POP-UPS THE EASY WAY**
>
> If you need a context menu or a tooltip, use a `ContextMenu` or `ToolTip` control instead of building one with a `Popup`. There's no point in making things harder than necessary.

TABLE B-9:

PROPERTY	PURPOSE
`HorizontalOffset`	Determines the horizontal offset between the target and the `Popup`'s origin.
`IsOpen`	Determines whether the `Popup` is visible. Set this to `True` to open the `Popup` in code-behind.
`Placement`	Determines how the `Popup` is positioned relative to the `PlacementTarget`. This can be `Absolute`, `Relative`, `Bottom`, `Center`, `Right`, `AbsolutePoint`, `RelativePoint`, `Mouse`, `MousePoint`, `Left`, `Top`, and `Custom`.
`PlacementRectangle`	Determines the area in which the `Popup` is positioned.
`PlacementTarget`	Determines the control relative to which the `Popup` is positioned.
`VerticalOffset`	Determines the vertical offset between the target and the `Popup`'s origin.

EXAMPLE Button and Popup

The following code creates a `Button`. It then makes a `Popup` that appears 5 pixels to the right of the `Button`. The `Popup` contains a `Label`.

```
<Button Name="btnClickMe" Content="Click Me" Width="75" Height="30"/>

<Popup Visibility="Visible" Placement="Right"
 HorizontalOffset="5" IsOpen="True"
 PlacementTarget="{Binding ElementName=btnClickMe}">
    <Label Background="Yellow" BorderBrush="Black" BorderThickness="1"
     Content="You should click this button!"/>
</Popup>
```

PROGRESSBAR

The `ProgressBar` displays progress information to the user. See Table B-10 for the key properties.

TABLE B-10:

PROPERTY	PURPOSE
`Maximum`	Determines the largest value displayed by the control.
`Minimum`	Determines the smallest value displayed by the control.
`Orientation`	Determines whether the control is oriented horizontally or vertically.
`Value`	Determines the value currently displayed by the control.

EXAMPLE Defining a ProgressBar

The following code creates a simple `ProgressBar` that can display values between 0 and 100, and that is currently displaying the value 60:

```
<ProgressBar Height="15" Width="200"
 Minimum="0" Maximum="100" Value="60"/>
```

SEPARATOR

The `Separator` displays a simple horizontal separator line in a `Menu` or a vertical separator line in a `ToolBar`.

EXAMPLE Using Separators

The following code creates a `Menu` and a `ToolBar` that include `Separators`:

```
<Menu>
    <MenuItem Header="File">
        <MenuItem Header="New"/>
        <MenuItem Header="Save"/>
        <MenuItem Header="Open"/>
        <Separator/>
        <MenuItem Header="Exit"/>
    </MenuItem>
</Menu>

<ToolBar Height="50" VerticalAlignment="Top">
    <Button><Image Source="New.jpg"/></Button>
    <Button><Image Source="Open.jpg"/></Button>
    <Button><Image Source="Save.jpg"/></Button>
    <Separator/>
    <Button><Image Source="Delete.jpg"/></Button>
</ToolBar>
```

TEXTBLOCK

The `TextBlock` displays read-only text much as a `Label` does but with additional features such as line wrapping, italics, and bold text (see Table B-11). The `TextBlock` is intended to display only a few lines of text. For more complex documents, consider using a `FlowDocument`. (See the "FlowDocument" section earlier in this appendix.)

You can use special style *inline tags* or *inlines* to give a piece of the text a special style such as bold or italic. For example, the following code displays text with the word *bold* in bold:

```
<TextBlock>A TexBlock can display <Bold>bold</Bold> text.</TextBlock>
```

TABLE B-11: Key Properties of TextBlock

PROPERTY	PURPOSE
`Background`	The color used to fill the control's interior
`FontFamily`	The name of the font family as in Times New Roman, Arial, or Segoe
`FontSize`	The text's font size. Append *cm*, *in*, *px*, or *pt* to specify centimeters, inches, pixels (the default), or points (1/72 inch) as in "0.25in" or "12pt."
`FontStyle`	Can be `Normal`, `Italic`, or `Oblique` (simulates italic for fonts without an italic style)
`FontWeight`	The text's *boldness*. This can be `Thin`, `ExtraLight`, `Light`, `Normal`, `Medium`, `SemiBold`, `Bold`, `ExtraBold`, `Black`, and `ExtraBlack`, although many fonts look the same for many of these values.
`Foreground`	The color used to draw the text
`LineHeight`	The vertical spacing between lines. Append *cm*, *in*, *px*, or *pt* to specify centimeters, inches, pixels (the default), or points (1/72 inch) as in "0.25in" or "12pt."
`LineStackingStrategy`	Determines how the control sizes its lines. This can be `MaxHeight` (a line is tall enough to hold all of its content) or `BlockLineHeight` (a line's height is determined by the control's `LineHeight` property).
`Padding`	The extra space added between the control's edges and the text it contains
`Text`	The text displayed by the control (simple text only)
`TextAlignment`	Determines how text is aligned within the `TextBlock`. This can be `Left`, `Right`, `Center`, or `Justify`.

continues

TABLE B-11 *(continued)*

PROPERTY	PURPOSE
TextTrimming	Determines how text is trimmed if it doesn't fit. This can be None (text is truncated), CharacterEllipsis (text is trimmed to the nearest character and followed by an ellipsis), or WordEllipsis (text is trimmed to the nearest word and followed by an ellipsis).
TextWrapping	Determines whether the control wraps text across multiple lines. This can be Wrap, NoWrap, and WrapWithOverflow.

Table B-12 summarizes the most useful inlines supported by the TextBlock.

TABLE B-12: Key Inlines for TextBlock

PROPERTY	PURPOSE
Bold	Makes the enclosed text bold.
Hyperlink	Creates a hyperlink. If the project is running in a web browser, Frame, or other host that provides navigation, then you can set the Hyperlink's NavigateUri property to the URL or filename that you want opened when the user clicks on the Hyperlink. If the host does not provide navigation, you can handle the Hyperlink's RequestNavigate event if the NavigateUri property is filled in. You can also catch the Hyperlink's Click event.
InlineUIContainer	Contains a user interface element such as a Button or TextBox.
Italic	Makes the enclosed text italic.
LineBreak	Inserts a line break.
Run	Contains a run of formatted text. Use the Run's Foreground, Background, and other properties to format the enclosed text.
Span	Groups other inlines. Span can contain the inlines Bold, Figure, Floater, Hyperlink, InlineUIContainer, Italic, LineBreak, Run, Span, and Underline.
Underline	Makes the enclosed text underlined.

EXAMPLE TextBlock with Inlines

The following code creates a TextBlock that demonstrates various inlines:

Available for download on Wrox.com

```
<TextBlock Grid.Row="0" Grid.Column="0" Background="White"
 HorizontalAlignment="Stretch"
 TextWrapping="NoWrap" TextAlignment="Left">
```

```
        This long line of text includes <Bold>bold</Bold>, <Italic>italic</Italic>,
        and <Underline>underlined</Underline> text.
        <LineBreak/>

        Here's a button:
        <InlineUIContainer>
            <Button Content="Click Me"/>
        </InlineUIContainer>
        You could even give it a Click event handler.
        <LineBreak/>

        Here's a really <Run FontSize="20" FontWeight="Bold">BIG</Run> word.
        <LineBreak/>

        This Hyperlink goes to the
        <Hyperlink NavigateUri="http://www.vb-helper.com">
            VB Helper Web site
        </Hyperlink>,
        although it won't work unless you run it inside a navigation host
        or add code-behind.
    </TextBlock>
```

UseTextBlock

The example program UseTextBlock uses this code to produce the `TextBlock` on the right in Figure B-1. The `TextBlock` on the left displays the same content but with the `TextWrapping` property set to `NoWrap`.

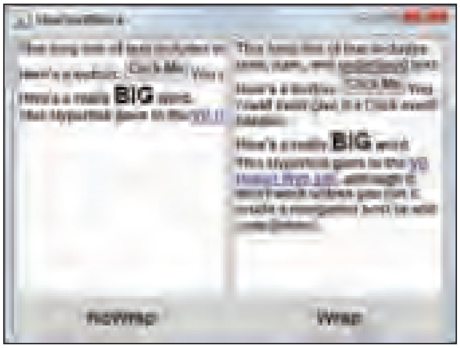


FIGURE B-1

TOOLTIP

As you can probably guess, the `ToolTip` displays a tooltip (see Table B-13). Usually it's easier to simply set the control's `ToolTip` property than it is to create a `ToolTip` object. One case in which you might want to make the `ToolTip` a separate object is if you want to change its properties.

CONSISTENT TIPS

If you do change the `ToolTip`'s properties, you will make it look different from other applications' tooltips. While that will make your program less consistent with the rest of the system, it probably won't distract the user too much as long as you don't go overboard (e.g., by displaying tooltips in a 60-point font).

TABLE B-13:

PROPERTY	PURPOSE
`Background`	The color used to fill the control’s interior
`Foreground`	The color used to draw the text
`BorderBrush`	The color of the control’s edge
`BorderThickness`	The control edge’s thickness
`FontFamily`	The name of the font family as in Times New Roman, Arial, or Segoe
`FontSize`	The text’s font size. Append *cm*, *in*, *px*, or *pt* to specify centimeters, inches, pixels (the default), or points (1/72 inch) as in “0.25in” or “12pt.”
`FontStyle`	Can be `Normal`, `Italic`, or `Oblique` (simulates italic for fonts without an italic style)
`FontWeight`	The text’s *boldness*. This can be `Thin`, `ExtraLight`, `Light`, `Normal`, `Medium`, `SemiBold`, `Bold`, `ExtraBold`, `Black`, and `ExtraBlack`, although many fonts look the same for many of these values.

EXAMPLE Using ToolTips

The following code creates two `Buttons` with `ToolTips`. The first uses property element syntax to define a `ToolTip` object and sets several of its property values. The second uses the simpler `ToolTip` property attribute syntax and uses default property values.

```
<Button Content="New">
    <Button.ToolTip>
        <ToolTip Content="Make a new file"
         Background="Yellow" Foreground="Red" BorderBrush="Blue" BorderThickness="5"
         FontWeight="Bold" FontSize="16" FontStyle="Italic"/>
    </Button.ToolTip>
</Button>

<Button Content="Open" ToolTip="Open an existing file"/>
```

TREEVIEW

The `TreeView` displays hierarchical data in a tree-like format similar to the one used by Windows Explorer to show directory structure. It should contain `TreeViewItem` objects that represent the entries.

See Chapter 18 for information about binding `TreeView` controls to data sources. Key properties of `TreeView` are given in table B-14.

TABLE B-14:

PROPERTY	PURPOSE
`Background`	The color used to fill the control's interior
`BorderBrush`	The color of the control's edge
`BorderThickness`	The control edge's thickness
`FontFamily`	The name of the font family as in Times New Roman, Arial, or Segoe
`FontSize`	The text's font size. Append *cm*, *in*, *px*, or *pt* to specify centimeters, inches, pixels (the default), or points (1/72 inch) as in "0.25in" or "12pt."
`FontStyle`	Can be `Normal`, `Italic`, or `Oblique` (simulates italic for fonts without an italic style)
`FontWeight`	The text's *boldness*. This can be `Thin`, `ExtraLight`, `Light`, `Normal`, `Medium`, `SemiBold`, `Bold`, `ExtraBold`, `Black`, and `ExtraBlack`, although many fonts look the same for many of these values.
`Padding`	The extra space added between the control's edges and its contents

The `TreeViewItem` control provides the same key properties as the `TreeView` control.

ITEM INHERITANCE

Note that a `TreeViewItem`'s subitems inherit font properties. For example, if you make an item bold, then all of its descendants in the tree are also bold.

EXAMPLE Using TreeViews

The following code builds a `TreeView` listing the Major League Baseball teams. The two topmost nodes for the American League and National League use property element syntax to display baseball pictures in addition to text. The division and team items display only text. The code shows only the first two divisions completely. The other teams are omitted to save space.

Available for download on Wrox.com

```
<TreeView>
    <TreeViewItem IsExpanded="True">
        <TreeViewItem.Header>
            <StackPanel Orientation="Horizontal">
                <Image Source="baseball.png" Height="40"/>
                <Label Content="American League" VerticalAlignment="Center"/>
            </StackPanel>
        </TreeViewItem.Header>
        <TreeViewItem Header="East" IsExpanded="True">
            <TreeViewItem Header="Baltimore Orioles"/>
            <TreeViewItem Header="Boston red Sox"/>
            <TreeViewItem Header="New York Yaknees"/>
```

```
                <TreeViewItem Header="Tampa Bay Rays"/>
                <TreeViewItem Header="Toronto Blue Jays"/>
            </TreeViewItem>
            <TreeViewItem Header="Central" IsExpanded="True">
                <TreeViewItem Header="Chicago White Sox"/>
                <TreeViewItem Header="Cleveland Indians"/>
                <TreeViewItem Header="Detroit Tigers"/>
                <TreeViewItem Header="Kansas City Royals"/>
                <TreeViewItem Header="Minnesota Twins"/>
            </TreeViewItem>
            <TreeViewItem Header="West" IsExpanded="True">
                <!--  Teams omitted.  -->
            </TreeViewItem>
        </TreeViewItem>

        <TreeViewItem IsExpanded="True">
            <TreeViewItem.Header>
                <StackPanel Orientation="Horizontal">
                    <Image Source="baseball.png" Height="40"/>
                    <Label Content="National League" VerticalAlignment "Center"/>
                </StackPanel>
            </TreeViewItem.Header>
            <TreeViewItem Header="East" IsExpanded="True">
                <!--  Teams omitted.  -->
            </TreeViewItem>
            <TreeViewItem Header="Central" IsExpanded="True">
                <!--  Teams omitted.  -->
            </TreeViewItem>
            <TreeViewItem Header="West" IsExpanded="True">
                <!--  Teams omitted.  -->
            </TreeViewItem>
        </TreeViewItem>
    </TreeView>
```

Baseball

C

Layout Controls

This appendix summarizes controls that are designed to arrange and contain other controls. They help you position the controls that make up the user interface.

The following sections very briefly summarize these controls. They list the controls' most important properties and provide simple XAML examples. For more detail, see Chapter 6.

Many of the controls define attached properties for use by child controls to tell the layout control how to position the children. Tables in this appendix summarize the key attached properties.

Because these controls are designed to hold other controls, they make logical places to define styles. For example, to make all of the `Labels` inside a `Grid` use the same font styles, you can add an unnamed `Label` style to the `Grid`'s Resources section.

CANVAS

The `Canvas` is a very simple layout control that lets you arrange its contents by explicitly setting the distances between their edges and those of the `Canvas`. See Table C-1 for the key properties.

TABLE C-1: Key Properties of Canvas

PROPERTY	PURPOSE
`Background`	The color used to fill the control's interior
`ClipToBounds`	Determines whether the control clips its children to its boundaries.

A child control can use attached properties to tell the `Canvas` how to position the child. The key attached properties for `Canvas` are given in Table C-2.

TABLE C-2: Key Attached Properties of Canvas

PROPERTY	PURPOSE
`Canvas.Bottom`	Sets the distance between the child's bottom edge and the `Canvas`'s bottom edge.
`Canvas.Left`	Sets the distance between the child's left edge and the `Canvas`'s left edge.
`Canvas.Right`	Sets the distance between the child's right edge and the `Canvas`'s right edge.
`Canvas.Top`	Sets the distance between the child's top edge and the `Canvas`'s top edge.

The control will only honor one attached property vertically or horizontally. For example, if you specify both `Canvas.Left` and `Canvas.Right`, then the `Canvas` honors the `Left` value and ignores the `Right` value.

EXAMPLE Using Canvas

The following code positions a `Label` 10 pixels from the upper-left corner of a `Canvas` and a `TextBox` 10 pixels from the lower-right corner:

```
<Canvas>
    <Label Canvas.Left="10" Canvas.Top="10" Content="First Name:"/>
    <TextBox Canvas.Right="10" Canvas.Bottom="10" Width="100"/>
</Canvas>
```

DOCKPANEL

The `DockPanel` lets you attach child controls to its top, left, right, and bottom edges. The key properties of `DockPanel` are given in Table C-3.

TABLE C-3: Key Properties of DockPanel

PROPERTY	PURPOSE
`Background`	The color used to fill the control's interior
`LastChildFill`	If `True`, then the `DockPanel` makes its last child fill all remaining available space.

A child control can use these properties to tell the `DockPanel` how to position the child. The key attached properties of `DockPanel` are given in Table C-4.

TABLE C-4: Key Attached Properties of DockPanel

PROPERTY	PURPOSE
`DockPanel.Bottom`	Makes the child fill the bottom of the `DockPanel`'s remaining area.
`DockPanel.Left`	Makes the child fill the left side of the `DockPanel`'s remaining area.
`DockPanel.Right`	Makes the child fill the right side of the `DockPanel`'s remaining area.
`DockPanel.Top`	Makes the child fill the top of the `DockPanel`'s remaining area.

EXAMPLE Using DockPanel

The following code makes a `DockPanel` that contains a `Menu` and a `Grid`. The `Menu` is attached to the `DockPanel`'s top as you would normally expect. The `Grid` occupies the rest of the `DockPanel` and would contain whatever else you want on the window.

```
<DockPanel LastChildFill="True">
    <Menu DockPanel.Dock="Top">
        <!--  MenuItems omitted.  -->
    </Menu>
    <Grid>
        <!--  Other controls omitted.  -->
    </Grid>
</DockPanel>
```

EXPANDER

The `Expander` displays a header and an icon that you can click to show or hide a single child. The key properties of `Expander` are given in Table C-5.

TABLE C-5: Key Properties of Expander

PROPERTY	PURPOSE
`Background`	The color used to fill the control's interior
`BorderBrush`	The color of the control's edge
`BorderThickness`	The control edge's thickness
`ExpandDirection`	Determines the direction in which the content is expanded. This can be `Down`, `Left`, `Right`, or `Up`.
`FontFamily`	The name of the font family as in Times New Roman, Arial, or Segoe (pronounced *see-go*)

continues

TABLE C-5 *(continued)*

PROPERTY	PURPOSE
`FontSize`	The text's font size. Append *cm*, *in*, *px*, or *pt* to specify centimeters, inches, pixels (the default), or points (1/72 inch) as in "0.25in" or "12pt."
`FontStyle`	Can be `Normal`, `Italic`, or `Oblique` (simulates italic for fonts without an italic style)
`FontWeight`	The text's *boldness*. This can be `Thin`, `ExtraLight`, `Light`, `Normal`, `Medium`, `SemiBold`, `Bold`, `ExtraBold`, `Black`, and `ExtraBlack`, although many fonts look the same for many of these values.
`Foreground`	The color used to draw the `Expander`'s `Header`
`Header`	The text displayed at the top of the `Expander` beside its Expand/Hide button
`HorizontalContentAlignment`	Determines how the content is aligned horizontally within the control. This can be `Center`, `Left`, `Right`, or `Stretch`.
`IsExpanded`	Determines whether the control is expanded to show its child.
`Padding`	The extra space added between the control's edges and its contents
`VerticalContentAlignment`	Determines how the content is aligned vertically within the control. This can be `Bottom`, `Center`, `Stretch`, or `Top`.

EXAMPLE Using Expander

The following code creates an `Expander` that contains a `StackPanel`. The `StackPanel` would contain other controls that the user could show and hide.

```
<Expander Header="Details" IsExpanded="True" BorderBrush="Blue">
    <StackPanel>
        <!--  Other controls omitted.  -->
    </StackPanel>
</Expander>
```

EXPRESSION BLEND EXAMPLE

Expression Blend's Properties tab uses `Expanders` to let you show and hide different groups of properties such as `Brushes`, `Appearance`, `Layout`, and so forth.

GRID

One of the more powerful and useful layout controls, `Grid` lets you arrange children in rows and columns. The key properties of `Grid` are given in Table C-6.

TABLE C-6: Key Properties of Grid

PROPERTY	PURPOSE
`Background`	The color used to fill the control's interior
`ColumnDefinitions`	This property element contains `ColumnDefinition` objects that define column widths
`RowDefinitions`	This property element contains `RowDefinition` objects that define row heights

A child control can use these properties to tell the `Grid` how to position the child. The key attached properties of `Grid` are given in Table C-7.

TABLE C-7: Key Attached Properties of Grid

PROPERTY	PURPOSE
`Grid.Column`	The column that contains the child
`Grid.ColumnSpan`	The number of columns that the child occupies
`Grid.Row`	The row that contains the child
`Grid.RowSpan`	The number of rows that the child occupies

EXAMPLE Making a Data Entry Grid

The following code builds a fairly standard data-entry form that lets the user enter name, address, and other information:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="40"/>
        <RowDefinition Height="30"/>
        <RowDefinition Height="30"/>
        <!--  Other row definitions omitted.  -->
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="100"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>

    <Label Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="2" Content="New Customer"
```

```
         HorizontalContentAlignment="Center" FontSize="20" FontWeight="Bold"
         Background="LightBlue"/>

        <Label Grid.Row="1" Grid.Column="0" Content="First Name:"/>
        <TextBox Grid.Row="1" Grid.Column="1"/>

        <Label Grid.Row="2" Grid.Column="0" Content="Last Name:"/>
        <TextBox Grid.Row="2" Grid.Column="1"/>

        <!--  Other rows omitted.  -->
    </Grid>
```

SCROLLVIEWER

The `ScrollViewer` displays a single child inside a scrollable region. The key properties of `ScrollViewer` are given in Table C-8.

TABLE C-8: Key Properties of ScrollViewer

PROPERTY	PURPOSE
`Background`	The color used to fill the control's interior
`HorizontalContentAlignment`	If the `ScrollViewer` is wider than its content, then this determines how the content is aligned horizontally within the control. This can be `Center`, `Left`, `Right`, or `Stretch`.
`HorizontalScrollBarVisibility`	Determines whether the horizontal scrollbar is displayed. This can be `Auto`, `Disabled`, `Hidden`, or `Visible`.
`IsDeferredScrollingEnabled`	If `True`, the control doesn't redisplay its contents until the user finishes dragging the scrollbar thumb. This is useful if the contents are very complex and take a while to display.
`Padding`	The extra space added between the control's edges and its contents
`VerticalContentAlignment`	If the `ScrollViewer` is taller than its content, then this determines how the content is aligned vertically within the control. This can be `Bottom`, `Center`, `Stretch`, or `Top`.
`VerticalScrollBarVisibility`	Determines whether the vertical scrollbar is displayed. This can be `Auto`, `Disabled`, `Hidden`, or `Visible`.

The ScrollViewer also supports font properties (FontFamily, FontSize, FontStyle, FontWeight). It doesn't draw any text itself, but contained Labels, TextBoxes, and other controls that draw text inherit these values.

SILLY SCROLLBARS

For some odd reason, the HorizontalScrollBarVisibility property defaults to Disabled and the VerticalScrollBarVisibility defaults to Visible, so the horizontal scrollbar is never visible even if it is needed, and the vertical scrollbar is always visible even if it is not needed.

You may want to set both of these properties to Auto so that they appear when needed and disappear when not needed.

EXAMPLE Using ScrollViewer

The following code creates a ScrollViewer holding an Image control that displays a large picture. The scrollbars appear as needed when the window is too small to display the entire picture.

```
<Grid>
    <ScrollViewer
     HorizontalScrollBarVisibility="Auto"
     VerticalScrollBarVisibility="Auto">
        <Image Stretch="None"
         Source="http://www.vb-helper.com/camil_moujaber_2_compressed.jpg" />
    </ScrollViewer>
</Grid>
```

STACKPANEL

The StackPanel is one of the more powerful and useful layout controls. It arranges its children either vertically or horizontally in a single column or row. If the control runs out of room before it runs out of children, it clips any remaining children. The key properties of StackPanel are given in Table C-9.

TABLE C-9: Key Properties of StackPanel

PROPERTY	PURPOSE
Background	The color used to fill the control's interior
Orientation	Determines whether the control arranges its children in a row (Orientation = Horizontal) or a column (Orientation = Vertical).

EXAMPLE Using StackPanel

The following code creates three buttons arranged in a column. The StackPanel's Resource section creates an unnamed style that gives the Buttons consistent Margin, Width, and Height properties.

```
<StackPanel>
    <StackPanel.Resources>
        <Style TargetType="Button">
            <Setter Property="Margin" Value="5"/>
            <Setter Property="Width" Value="100"/>
            <Setter Property="Height" Value="40"/>
        </Style>
    </StackPanel.Resources>

    <Button Content="Add Customer"/>
    <Button Content="Find Customer"/>
    <Button Content="Delete Customer"/>
</StackPanel>
```

STATUSBAR

The StatusBar creates an area, typically at the bottom of the window, where you can give the user status information. You can place StatusBarItems, Labels, Buttons, ComboBoxes, and other controls inside the StatusBar. The key properties of StatusBar are given in Table C-10.

TABLE C-10: Key Properties of StatusBar

PROPERTY	PURPOSE
Background	The color used to fill the control's interior
BorderBrush	The color of the control's edge
BorderThickness	The control edge's thickness

The StatusBar also supports font properties (FontFamily, FontSize, FontStyle, FontWeight). It doesn't draw any text itself, but contained Labels, TextBoxes, and other controls that draw text inherit these values.

EXAMPLE Using StatusBar

You can place controls such as Label and Button directly in the StatusBar, but you can get slightly better results if you place StatusBarItems inside the StatusBar. In that case, the StatusBar acts much like a horizontal StackPanel that stretches its last child to fill the available space. If you place controls directly inside the StatusBar, it doesn't stretch the last control.

The following code builds a `DockPanel` that displays a `Menu` on top, a `StatusBar` on the bottom, and a `Grid` in the middle:

```
<DockPanel>
    <Menu DockPanel.Dock="Top">
        <!--  Menu items omitted.  -->
    </Menu>

    <StatusBar DockPanel.Dock="Bottom">
        <StatusBarItem>
            <Label Content="Status:"/>
        </StatusBarItem>
        <StatusBarItem>
            <Label Content="Normal" Background="White" Padding="10,6,10,6">
                <Label.BitmapEffect>
                    <BevelBitmapEffect EdgeProfile="BulgedUp"/>
                </Label.BitmapEffect>
            </Label>
        </StatusBarItem>

        <StatusBarItem HorizontalAlignment="Right" >
            <Button Content="Disconnect"/>
        </StatusBarItem>
    </StatusBar>

    <Grid>
       <!--  Other controls omitted.  -->
    </Grid>
</DockPanel>
```

The `StatusBar` contains a pair of `StatusBarItems` holding `Labels`. The third `StatusBarItem` has `HorizontalAlignment = Right`, so that item is moved to the `StatusBar`'s right edge.

TABCONTROL

The `TabControl` displays a series of tabs where you can place different sets of information. The control should contain a series of `TabItems`, each of which holds a single content item (which is often a container such as a `Grid`). The key properties of `TabControl` are given in Table C-11.

TABLE C-11: Key Properties of TabControl

PROPERTY	PURPOSE
`Background`	The color used to fill the interior of the tab contents. This color does not fill the header area.
`BorderBrush`	The color of the edge of the tab contents. This does not include the header area.

continues

TABLE C-11 *(continued)*

PROPERTY	PURPOSE
`BorderThickness`	The tab contents' border thickness. This does not include the header area.
`FlowDirection`	Determines whether the tabs are arranged right-to-left or left-to-right.
`FontFamily`	The name of the font family as in Times New Roman, Arial, or Segoe
`FontSize`	The text's font size. Append *cm*, *in*, *px*, or *pt* to specify centimeters, inches, pixels (the default), or points (1/72 inch) as in "0.25in" or "12pt."
`FontStyle`	Can be `Normal`, `Italic`, or `Oblique` (simulates italic for fonts without an italic style)
`FontWeight`	The text's *boldness*. This can be `Thin`, `ExtraLight`, `Light`, `Normal`, `Medium`, `SemiBold`, `Bold`, `ExtraBold`, `Black`, and `ExtraBlack`, although many fonts look the same for many of these values.
`Padding`	The extra space added between the control's edges and the tab contents. This does not include the header area.
`SelectedIndex`	The zero-based index of the currently selected tab.
`TabStripPlacement`	Determines where the tab strip is positioned. This can be `Bottom`, `Left`, `Right`, or `Top`.

Table C-12 summarizes the most important properties of the `TabItem` objects that you place inside the `TabControl`.

TABLE C-12: Key Properties of TabItem Objects

PROPERTY	PURPOSE
`Foreground`	The color used to draw the header text
`Header`	The content displayed on the tab. Often this is plaintext.
`IsSelected`	Determines whether this tab is currently selected.
`Padding`	The extra space added between the header area's edges and the header content

EXAMPLE Using TabControl

The following code creates a `TabControl` that contains three `TabItems` with headers Customers, Products, and Employees:

```
<TabControl>
    <TabItem Header="Customers">
        <Grid>
            <!--  Controls omitted.  -->
```

```
            </Grid>
        </TabItem>
        <TabItem Header="Products">
            <Grid>
                <!--  Controls omitted.  -->
            </Grid>
        </TabItem>
        <TabItem Header="Employees">
            <Grid>
                <!--  Controls omitted.  -->
            </Grid>
        </TabItem>
    </TabControl>
```

EXAMPLE Using a Rotated TabControl

The following code is to the previous example similar except it places the tabs on the left side of the `TabControl`. The code uses the `styTabHeader` style to rotate the headers by –90 degrees so they don't take up a huge amount of space.

Available for download on Wrox.com

```
<TabControl TabStripPlacement="Left">
    <TabControl.Resources>
        <Style x:Key="styTabHeader" TargetType="TextBlock">
            <Setter Property="FontSize" Value="14"/>
            <Setter Property="FontWeight" Value="Bold"/>
            <Setter Property="Padding" Value="4,0,4,0"/>
            <Setter Property="LayoutTransform">
                <Setter.Value>
                    <RotateTransform Angle="-90"/>
                </Setter.Value>
            </Setter>
        </Style>
    </TabControl.Resources>

    <TabItem>
        <TabItem.Header>
            <TextBlock Text="Customers" Style="{StaticResource styTabHeader}"/>
        </TabItem.Header>
        <Grid>
            <!--  Controls omitted.  -->
        </Grid>
    </TabItem>
    <TabItem>
        <TabItem.Header>
            <TextBlock Text="Products" Style="{StaticResource styTabHeader}"/>
        </TabItem.Header>
        <Grid>
            <!--  Controls omitted.  -->
        </Grid>
    </TabItem>
    <TabItem>
        <TabItem.Header>
            <TextBlock Text="Employees" Style="{StaticResource styTabHeader}"/>
```

```
            </TabItem.Header>
            <Grid>
                <!--  Controls omitted.  -->
            </Grid>
        </TabItem>
    </TabControl>
```

RotatedTabs

Figure C-1 shows the result.

FIGURE C-1

TOOLBAR AND TOOLBARTRAY

The `ToolBar` displays a series of `Buttons`, `ComboBoxes`, and other tools that the user can access easily.

The `ToolBarTray` displays an area, normally at the top of the window below the menus, that contains one or more `ToolBars`. The user can drag the `ToolBars` around within the `ToolBarTray`. The key properties of `ToolBarTray` are given in Table C-13.

TABLE C-13:

PROPERTY	PURPOSE
`Background`	The color used to fill the interior of the control
`FlowDirection`	Determines whether the `ToolBars` are arranged right-to-left or left-to-right.
`IsLocked`	If `True`, then the user cannot move `ToolBars` within the `ToolBarTray`.
`Orientation`	Determines whether the `ToolBarTray` arranges the `ToolBars` horizontally or vertically.

A `ToolBarTray` can arrange its `ToolBars` in rows called *bands*. The `ToolBar`'s `Band` and `BandIndex` properties determine where a `ToolBar` initially appears within a `ToolBarTray`. These properties can be any numeric values. The `ToolBarTray` automatically creates bands as needed and arranges the `ToolBars` within them. The key properties of `ToolBar` are given in Table C-14.

TABLE C-14: Key Properties of ToolBar

PROPERTY	PURPOSE
`Background`	The color used to fill the interior of the control
`Band`	Determines which `ToolBarTray` band holds this `ToolBar`.
`BandIndex`	Determines the ordering of this `ToolBar` within its band.
`BorderBrush`	The color of the control's edge
`BorderThickness`	The control edge's thickness
`FlowDirection`	Determines whether the `ToolBar` arranges its content right-to-left or left-to-right.
`FontFamily`	The name of the font family as in Times New Roman, Arial, or Segoe
`FontSize`	The text's font size. Append *cm*, *in*, *px*, or *pt* to specify centimeters, inches, pixels (the default), or points (1/72 inch) as in "0.25in" or "12pt."
`FontStyle`	Can be `Normal`, `Italic`, or `Oblique` (simulates italic for fonts without an italic style)
`FontWeight`	The text's *boldness*. This can be `Thin`, `ExtraLight`, `Light`, `Normal`, `Medium`, `SemiBold`, `Bold`, `ExtraBold`, `Black`, and `ExtraBlack`, although many fonts look the same for many of these values.
`Header`	Defines an object to display at the beginning of the `ToolBar`. For example, you could set this to text that indicates the category of tools in this `ToolBar`. Or you could set it to a `Label` that does the same but with a smaller font and a lighter color.
`Padding`	The extra space added between the control's edges and the contents it contains

EXAMPLE Using ToolBars

The example program Toolbars shown in Figure C-2 uses the following code to create a window with a `Menu`, `ToolBarTray`, `StatusBar`, and `TabControl`. The `ToolBarTray` contains three `ToolBars` arranged in two bands. To save space, only the code that builds the `ToolBarTray` and `ToolBars` is shown here.

Available for download on Wrox.com

```
<ToolBarTray DockPanel.Dock="Top" Background="LightGray">
    <ToolBar Band="0" BandIndex="0">
        <Button>
           <Image Source="New.ico"/>
        </Button>
        <Button>
            <Image Source="Open.ico"/>
        </Button>
    </ToolBar>
    <ToolBar Band="0" BandIndex="1">
        <Button>
            <Image Source="Copy.ico"/>
        </Button>
        <Button>
            <Image Source="Cut.ico"/>
```

```
            </Button>
            <Button>
                <Image Source="Paste.ico"/>
            </Button>
        </ToolBar>
        <ToolBar Band="1" Height="30">
            <Button>
                <Image Source="Add.png"/>
            </Button>
            <Button>
                <Image Source="Cancel.png"/>
            </Button>
            <Button>
                <Image Source="Ok.png"/>
            </Button>
        </ToolBar>
    </ToolBarTray>
```

Toolbars

FIGURE C-2

UNIFORMGRID

The `UniformGrid` arranges its children in rows and columns of uniform size. The key properties of `UniformGrid` are given in Table C-15.

TABLE C-15:

PROPERTY	PURPOSE
`Background`	The color used to fill the interior of the control
`Columns`	Indicates the number of columns in the grid.

PROPERTY	PURPOSE
`FirstColumn`	Indicates the first column used in the grid's first row. Cells before that one remain empty.
`FlowDirection`	Determines whether the cells are filled in right-to-left or left-to-right order.
`Rows`	Indicates the number of rows in the grid.

EXAMPLE Using UniformGrid

The following code defines a `UniformGrid` that displays six `Buttons`:

```
<UniformGrid Rows="2" Columns="3">
    <Button Margin="5" Content="1"/>
    <Button Margin="5" Content="2"/>
    <Button Margin="5" Content="3"/>
    <Button Margin="5" Content="4"/>
    <Button Margin="5" Content="5"/>
    <Button Margin="5" Content="6"/>
</UniformGrid>
```

Also see the `WrapPanel` control described at the end of this appendix.

VIEWBOX

The `Viewbox` stretches its single child in one of several ways depending on whether its `Stretch` property is set to `Fill`, `None`, `Uniform`, or `UniformToFill`.

Normally, you don't need to use a `Viewbox` to stretch images because the `Image` control has its own `Stretch` property. The `Viewbox` is more useful for stretching other controls that don't normally stretch themselves, such as `Labels`, `Buttons`, or `Grids` containing more controls. The key properties of `Viewbox` are given in Table C-16.

TABLE C-16: Key Properties of Viewbox

PROPERTY	PURPOSE
`Stretch`	Determines how the control stretches its contents. This can be `Fill` (fill the `Viewbox` even if it distorts the contents), `None`, `Uniform` (make the contents as large as possible without distortion), or `UniformToFill` (enlarge the contents without distortion until they fill the `Viewbox` even if parts are clipped off).
`StretchDirection`	Determines whether the control can enlarge or shrink its contents, or both. This can be `Both` (enlarge or shrink), `DownOnly` (shrink only), or `UpOnly` (enlarge only).

EXAMPLE Using Viewbox

The following code creates a `Viewbox` that contains a `Grid`. The `Grid` contains several `Labels` and `TextBoxes`. The `Viewbox` stretches the `Grid` and its contents to fit, possibly squeezing or stretching the controls.

```
<Viewbox Stretch="Fill" Margin="10">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Label Grid.Row="0" Grid.Column="0" Content="First Name:"/>
        <TextBox Grid.Row="0" Grid.Column="1" Text="Rod"/>
        <Label Grid.Row="1" Grid.Column="0" Content="Last Name:"/>
        <TextBox Grid.Row="1" Grid.Column="1" Text="Stephens"/>
    </Grid>
</Viewbox>
```

WINDOWSFORMSHOST

The `WindowsFormsHost` lets you place Windows Forms controls inside a WPF application.

Before you can use a `WindowsFormsHost`, you must:

1. Add a reference to the WindowsFormsIntegration library.
2. Add a reference to the System.Windows.Forms.dll library.
3. Add a namespace declaration to the XAML file similar to this one:

```
xmlns:wf="clr-namespace:System.Windows.Forms;assembly=System.Windows.Forms"
```

The final step allows your XAML code to refer to the controls in the Windows Forms library by using the prefix *wf*.

EXAMPLE Using WindowsFormsHost

The following code creates a `WindowsFormsHost` that contains a DateTimePicker:

```
<WindowsFormsHost Margin="5" x:Name="wfhAppt">
    <WindowsFormsHost.Child>
        <wf:DateTimePicker x:Name="dtpAppt" Enabled="False" />
    </WindowsFormsHost.Child>
</WindowsFormsHost>
```

See the section “WindowsFormsHost” in Chapter 6 for more details and a more complete example.

WRAPPANEL

The `WrapPanel` displays its children in a row or column, wrapping to a new row or column when necessary. The key properties of `WrapPanel` are given in Table C-17.

TABLE C-17:

PROPERTY	PURPOSE
`Background`	The color used to fill the control's interior
`FlowDirection`	Determines whether the control arranges its contents in right-to-left or left-to-right order. Columns are always arranged in top-to-bottom order.
`ItemHeight`	Gives items a uniform height.
`ItemWidth`	Gives items a uniform width.
`Orientation`	Determines whether the control fills rows or columns. This can be `Horizontal` or `Vertical`.

EXAMPLE Using WrapPanel

The following code creates a `WrapPanel` that contains five `Buttons`, arranged in columns:

```
<WrapPanel Orientation="Vertical">
    <Button Margin="5" Width="50" Height="50" Content="1"/>
    <Button Margin="5" Width="50" Height="50" Content="2"/>
    <Button Margin="5" Width="50" Height="50" Content="3"/>
    <Button Margin="5" Width="50" Height="50" Content="4"/>
    <Button Margin="5" Width="50" Height="50" Content="5"/>
</WrapPanel>
```

D

User Interaction Controls

This appendix summarizes controls that are designed to let the user control the application. They let the user initiate and control actions.

Because these controls let the user control the program, they must have some way to interact with the program's executing code. That means they typically use event handlers written in code-behind, events in XAML code, or a mix of both.

The following sections very briefly summarize the WPF user interaction controls. They list the controls' most important properties and provide simple examples. For more detail, see Chapter 7.

Applications often use these controls to start some action and thus catch events raised by the controls. That makes these events more important than those raised by the controls described in previous appendixes.

This appendix focuses more on events than properties. It describes only properties that affect a control's behavior, not the simpler properties that determine the control's appearance.

The `IsEnabled` property is particularly important to many of the controls described here. It determines whether the control will interact with the user. When `IsEnabled` is `False`, the control may change its appearance, but it definitely won't let the user interact with it.

BUTTON

The `Button` lets the user initiate an action (see Table D-1).

TABLE D-1: Key Events for Button

EVENT	PURPOSE
`Click`	Fires when the user clicks the `Button`.

Often a `Button` displays a simple text value, but if a `Button` contains several other controls, it automatically raises its `Click` event when the user clicks on any of them.

EXAMPLE Using Button

The following code creates two `Buttons`. The first uses a simple text caption. The second contains a `StackPanel` that holds a `Label` and an `Image`.

```
<Button Content="Click Me" Click="btnClickMe_Click"/>

<Button Click="btnSmile_Click">
    <StackPanel>
        <Image Source="smiley.jpg" Height="50"/>
        <Label Content="Smile!" HorizontalAlignment="Center"/>
    </StackPanel>
</Button>
```

The previous code's `Click` attributes give the names of each `Button`'s event handler in the code-behind. The following C# code shows simple `Click` event handlers for these `Buttons`. This code just displays a message box so that you know the `Buttons` have done something.

```
private void btnClickMe_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Click me clicked");
}

private void btnSmile_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Smile clicked");
}
```

CHECKBOX

The `CheckBox` lets the user check or uncheck an option. The user can check and uncheck any number of `CheckBoxes` in any combination. (Contrast this with the `RadioButton`, where only one in a group can be selected at a time.) Table D-2 gives the key properties of `CheckBox`, and Table D-3 gives the key events.

TABLE D-2: Key Properties of CheckBox

PROPERTY	PURPOSE
`IsChecked`	Determines whether the control is currently checked.
`IsThreeState`	Determines whether the control cycles through three values — `checked` (`IsChecked = True`), `unchecked` (`IsChecked = False`), and `indeterminate` (`IsChecked = null`).

TABLE D-3: Key Events for CheckBox

EVENT	PURPOSE
`Checked`	Occurs when the control is checked.
`Unchecked`	Occurs when the control is unchecked.
`Click`	Occurs when the control is checked or unchecked.

EXAMPLE Using CheckBox

The following code creates a `CheckBox` that fires `chkLunch_Checked` and `chkLunch_Unchecked` events when it is checked or unchecked:

```
<CheckBox Name="chkLunch" Content="Lunch" IsChecked="True"
 Checked="chkLunch_Checked" Unchecked="chkLunch_Unchecked"/>
```

The following code-behind shows the control's event handlers. They hide or show a `GroupBox` named `grpLunchMenu`, which contains options that should only be available when the `CheckBox` is checked.

```
private void chkLunch_Checked(object sender, RoutedEventArgs e)
{
    grpLunchMenu.Visibility = Visibility.Visible;
}

private void chkLunch_Unchecked(object sender, RoutedEventArgs e)
{
    grpLunchMenu.Visibility = Visibility.Hidden;
}
```

COMBOBOX

The `ComboBox` lets the user pick from a list of allowed choices. The control should contain `ComboBoxItems` that hold the choices. The `SelectedIndex`, `SelectedItem`, and `SelectedValue` properties let you get and set the currently selected item (see Tables D-4 and D-5).

TABLE D-4: Key Properties of ComboBox

PROPERTY	PURPOSE
`IsEditable`	Determines whether the user can type a new value into the control.
`SelectedIndex`	Determines the index of the currently selected item.

TABLE D-5: Key Events for ComboBox

EVENT	PURPOSE
DropDownClosed	Occurs when the control closes its dropdown list.
DropDownOpened	Occurs when the control opens its dropdown list.
SelectionChanged	Occurs when the user makes a new selection.

The ComboBoxItem objects inside a ComboBox also provide some useful events (see Table D-6).

TABLE D-6: Key Events for ComboBoxItem Objects

EVENT	PURPOSE
Selected	Occurs when the item is selected.
Unselected	Occurs when the item is unselected.

EXAMPLE Using ComboBox

The following XAML code builds a ComboBox that lists the names of the solar system's planets. (I put parentheses around Pluto because it's technically no longer considered a planet.)

```
<ComboBox SelectedIndex="2">
    <ComboBoxItem Content="Mercury"/>
    <ComboBoxItem Content="Venus"/>
    <ComboBoxItem Content="Earth"/>
    <ComboBoxItem Content="Mars"/>
    <ComboBoxItem Content="Jupiter"/>
    <ComboBoxItem Content="Saturn"/>
    <ComboBoxItem Content="Uranus"/>
    <ComboBoxItem Content="Neptune"/>
    <ComboBoxItem Content="(Pluto)"/>
</ComboBox>
```

CONTEXTMENU

The ContextMenu displays a pop-up menu associated with another control (see Table D-7). The ContextMenu should contain MenuItems that act like any other MenuItems do.

TABLE D-7: Key Properties of ContextMenu

PROPERTY	PURPOSE
IsOpen	Determines whether the ContextMenu is displayed. You can use this property to display the menu programmatically.

Normally you can simply react to events raised by the MenuItems inside the ContextMenu, but you can also respond to the following events (see Table D-8).

TABLE D-8: Key Events for ContextMenu

EVENT	PURPOSE
Closed	Occurs when the ContextMenu closes.
Opened	Occurs when the ContextMenu opens.

EXAMPLE Using ContextMenu

The following code builds two Rectangles that have ContextMenus:

```
<StackPanel>
    <StackPanel.Resources>
        <ContextMenu x:Key="ctxColor">
            <MenuItem Header="Red"/>
            <MenuItem Header="Green"/>
            <MenuItem Header="Blue"/>
        </ContextMenu>
    </StackPanel.Resources>

    <Rectangle Width="20" Height="20" Fill="Red"
     ContextMenu="{StaticResource ctxColor}"/>

    <Rectangle Width="20" Height="20" Fill="Orange">
        <Rectangle.ContextMenu>
            <ContextMenu>
                <MenuItem Header="Orange"/>
                <MenuItem Header="Yellow"/>
                <MenuItem Header="Purple"/>
            </ContextMenu>
        </Rectangle.ContextMenu>
    </Rectangle>
</StackPanel>
```

The code first defines a ContextMenu resource named ctxColors that displays menu items Red, Green, and Blue. It then creates a red Rectangle, setting the Rectangle's ContextMenu property to the resource.

Next the code creates an orange Rectangle. It gives the new Rectangle a ContextMenu element attribute that has menu items Orange, Yellow, and Purple.

FRAME

The Frame displays Web or XAML content and provides navigation buttons (see Tables D-9 through D-11).

TABLE D-9:

PROPERTY	PURPOSE
`CanGoBack`	Indicates whether the control can move backward in its navigation history.
`CanGoForward`	Indicates whether the control can move forward in its navigation history.
`Source`	The location of the web page or XAML file that the `Frame` should display

TABLE D-10:

PROPERTY	PURPOSE
`GoBack`	Moves to the previous entry in the navigation history if one exists.
`GoForward`	Moves to the next entry in the navigation history if one exists.
`Navigate`	Moves to a new Source.
`Refresh`	Reloads the current Source.

TABLE D-11:

EVENT	PURPOSE
`ContentRendered`	Occurs after the `Frame` finishes displaying its content.
`LoadCompleted`	Occurs after a Source has been loaded and rendering is beginning.
`Loaded`	Occurs after a Source has been loaded and rendered, and the control is ready to interact with the user.
`Navigated`	Occurs when content has been found and is available through the `Content` property but may not yet be fully displayed.
`Navigating`	Occurs when the control is starting a new navigation.
`NavigationFailed`	Occurs when a navigation fails.
`NavigationProgress`	Occurs during navigation to give progress information.
`NavigationStopped`	Occurs when navigation stops because of a call to the `Frame`'s `StopLoading` method or because a new navigation began.

EXAMPLE Using Frame

The following simple XAML example creates a `Frame` control that loads the web page `www.vb-helper.com`:

```
<Frame Source="http://www.vb-helper.com"/>
```

EXAMPLE Using a Frame Bound to a ComboBox

The following more interesting XAML code creates a `Grid` that contains a `ComboBox` and a `Frame`. The `Frame`'s `Source` property is bound to the `ComboBox`'s `SelectedItem.Tag` value so when you pick a new entry from the `ComboBox`, the `Frame` displays the corresponding web page.

Available for download on Wrox.com

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="35"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <ComboBox Grid.Row="0" SelectedIndex="0" Name="cboSites" Margin="5">
        <ComboBoxItem Content="Beginning Database Design Solutions"
         Tag="http://www.vb-helper.com/db_design.htm"/>
        <ComboBoxItem Content="VB Helper"
         Tag="http://www.vb-helper.com"/>
        <ComboBoxItem Content="VB Helper Bookstore"
         Tag="http://astore.amazon.com/vbhelper/about"/>
        <ComboBoxItem Content="Books To Keep"
         Tag="http://www.BooksToKeep.com"/>
        <ComboBoxItem Content="Astronomy Picture of the Day"
         Tag="http://antwrp.gsfc.nasa.gov/apod"/>
    </ComboBox>
    <Frame Name="fraGo" Grid.Row="1" HorizontalAlignment="Stretch"
     VerticalAlignment="Stretch" Margin="5"
     Source="{Binding ElementName=cboSites, Path=SelectedItem.Tag}">
    </Frame>
</Grid>
```

UseFrame

The following C# code makes the `Frame` named `fraGo` navigate to the URL `http://astore.amazon.com/vbhelper/about`:

```
fraGo.Navigate(new Uri("http://astore.amazon.com/vbhelper/about"));
```

GRIDSPLITTER

The `GridSplitter` lets the user resize two adjacent rows or columns in a `Grid` control (see Table D-12).

To use a vertical `GridSplitter`, add it to a cell in the `Grid`'s first row and set its `Grid.RowSpan` property so it spans all of the `Grid`'s rows. Next, set its `HorizontalAlignment` property to `Left` or `Right`. That makes the `GridSplitter` stick to the side of its column and determines which other column it will adjust. Finally, set the control's `Width` to something big enough for the user to grab and drag, perhaps 3 or 5.

To make a horizontal `GridSplitter`, switch the roles of the `Grid`'s rows and columns, use the `GridSplitter`'s `VerticalAlignment` property, and set its `Height` to a reasonable value.

TABLE D-12:

PROPERTY	PURPOSE
`Grid.Column`	Sets the `Grid` cell where the splitter starts. For horizontal splitters, set this to 0.
`Grid.ColumnSpan`	For a horizontal splitter, set this so the control spans all columns.
`Grid.Row`	Sets the `Grid` cell where the splitter starts. For vertical splitters, set this to 0.
`Grid.RowSpan`	For a vertical splitter, set this so the control spans all rows.
`HorizontalAlignment`	For a vertical splitter, determines which side of its column the splitter sticks to and adjusts.
`ShowsPreview`	Determines whether the control actually resizes the `Grid`'s rows or columns while you drag it, or whether it only shows a preview by displaying a gray version of the `GridSplitter` moving. If `ShowsPreview` is `False`, the control resizes the rows or columns. If `ShowsPreview` is `True`, the control displays the gray splitter and only resizes the rows or columns when you release the mouse. (A preview is more efficient if the contents of the rows or columns take a long time to redraw.)
`VerticalAlignment`	For a horizontal splitter, determines which side of its column the splitter sticks to and adjusts.

Usually the `Grid` and the controls it contains can rearrange themselves when the user drags the `GridSplitter`, but the following events let you take special action during the drag if you must (see Table D-13).

TABLE D-13: Key Events for GridSplitter

EVENT	PURPOSE
`DragCompleted`	Occurs when the user has finished dragging the control.
`DragDelta`	Occurs when the user has moved the splitter during a drag.
`DragStarted`	Occurs when the user has started dragging the control.

EXAMPLE Using GridSplitter

The following code creates a `Grid` control with three rows and columns. It places two `GridSplitters` inside the middle `Grid` cell. The first splitter has `HorizontalAlignment` set to `Left` so it lets the user adjust the left and middle columns. The second splitter has `HorizontalAlignment` set to `Right` so it lets the user adjust the middle and right columns.

```
<Grid>
    <Grid.ColumnDefinitions>
```

```
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <GridSplitter Grid.Column="1" Grid.RowSpan="10"
         HorizontalAlignment="Left"
         VerticalAlignment="Stretch"
         Background="Blue"
         ShowsPreview="False"
         Width="5"/>
        <GridSplitter Grid.Column="1" Grid.RowSpan="10"
         HorizontalAlignment="Right"
         VerticalAlignment="Stretch"
         Background="Red"
         ShowsPreview="False"
         Width="5"/>
    </Grid>
```

The following code creates horizontal `GridSplitters` for the same `Grid` used in the previous example:

```
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="*"/>
       </Grid.RowDefinitions>
        <GridSplitter Grid.Row="1" Grid.ColumnSpan="20"
         HorizontalAlignment="Stretch"
         VerticalAlignment="Top"
         Background="Blue"
         ShowsPreview="False"
         Height="5"/>
        <GridSplitter Grid.Row="1" Grid.ColumnSpan="20"
         HorizontalAlignment="Stretch"
         VerticalAlignment="Bottom"
         Background="Red"
         ShowsPreview="False"
         Height="5"/>
        <Image Source="baseball.jpg" Grid.Row="1" Grid.Column="1"/>
    </Grid>
```

MAKE ROOM

Be sure to allow room for the splitters when you place other objects inside the `Grid`. For example, if you place a 5-pixel-wide `GridSplitter` on the left edge of the second column, then any other controls in that column should have a left margin of at least 5 so they don't overlap with the splitter.

LISTBOX

The `ListBox` displays a list of items and lets the user select one or more, depending on how the control is configured (see Tables D-14 and D-15).

The control should contain `ListBoxItems` that hold the choices. The `SelectedIndex`, `SelectedItem`, and `SelectedValue` properties let you get and set the currently selected item.

TABLE D-14: Key Properties of ListBox

PROPERTY	PURPOSE
`HorizontalContentAlignment`	Determines how the `ListboxItems` are aligned. This can be `Center`, `Left`, `Right`, or `Stretch`.
`SelectedIndex`	Determines the index of the currently selected item.
`SelectedItems`	Returns the currently selected items. (You cannot use `SelectedIndex`, `SelectedItem`, or other single selection properties to see which items are selected if more than one is selected.)
`SelectionMode`	Determines how many items the user can select. This can be `Single` (one item only), `Multiple` (can select multiple items by clicking on them individually), or `Extended` (can use [Shift]+click and [Ctrl]+click to select multiple items).

TABLE D-15: Key Events for ListBox

EVENT	PURPOSE
`SelectionChanged`	Occurs when the selected items change.

The `ListBoxItem` objects inside a `ListBox` also provide some useful events (see Table D-16).

TABLE D-16: Key Events for ListBoxItem Objects

EVENT	PURPOSE
`Selected`	Occurs when the item is selected.
`Unselected`	Occurs when the item is unselected.

EXAMPLE Using ListBox

The following XAML code builds a `ListBox` that lists the names of the solar system's planets. It also displays a `Label` that shows the name of the currently selected planet.

```
<StackPanel>
    <ListBox Name="lstPlanets" SelectedIndex="2">
        <ListBoxItem Content="Mercury"/>
        <ListBoxItem Content="Venus"/>
        <ListBoxItem Content="Earth"/>
        <ListBoxItem Content="Mars"/>
        <ListBoxItem Content="Jupiter"/>
        <ListBoxItem Content="Saturn"/>
        <ListBoxItem Content="Uranus"/>
        <ListBoxItem Content="Neptune"/>
        <ListBoxItem Content="(Pluto)"/>
    </ListBox>
    <Label Content="{Binding ElementName=lstPlanets, Path=SelectedItem.Content}"/>
</StackPanel>
```

MENU

The `Menu` displays a main menu for a `Window`. While you can place controls such as `Labels`, `Buttons`, and `TextBoxes` in a `Menu`, a `Menu` normally holds `MenuItem` objects that contain the `Window`'s top-level menus (see Tables D-17 through D-19).

A `MenuItem`'s `Header` property determines what it displays. A `MenuItem` can contain other `MenuItems` to build a menu hierarchy.

TABLE D-17: Key Properties of Menu

PROPERTY	PURPOSE
`IsMainMenu`	Determines whether the control is the main menu. If a `Window` contains more than one menu, all other menus should set this to `False` so they are not notified when the user presses the [Alt] key or [F10].

TABLE D-18:

PROPERTY	PURPOSE
`Command`	Determines the command associated with the menu item.
`CommandParameter`	Determines the parameter passed to a `Command`.
`CommandTarget`	Determines the object on which to raise a `Command`.
`IsCheckable`	Determines whether the menu item can be checked. If this is `True`, then the item automatically checks and unchecks itself. If this is `False`, your code-behind can still check and uncheck the item.
`IsChecked`	Determines whether the menu item is currently checked.
`IsSubmenuOpen`	Determines whether the item's submenu is open.

For more information on commands, see Chapter 19.

TABLE D-19:

EVENT	PURPOSE
`Checked`	Occurs when the menu item is checked.
`Click`	Occurs when the user clicks the menu item either with the mouse or logically by using the keyboard. Normally the item executes some action when it is clicked.
`SubmenuClosed`	Occurs when the menu's submenu is closing.
`SubmenuOpened`	Occurs when the menu's submenu is about to open. For example, you can dynamically modify the submenu in this event.
`Unchecked`	Occurs when the menu item is unchecked.

EXAMPLE Using Menu

For this example, imagine a network drawing application. The following code builds the program's `Data` menu, which provides tools for loading networks and setting options:

```
<Menu DockPanel.Dock="Top">
    <!-- Other main menus omitted. -->
    <MenuItem Name="mnuData" Header="Data"
     SubmenuOpened="mnuData_SubmenuOpened">
        <MenuItem Name="mnuDataShowCosts" Header="Show Costs"
         IsCheckable="True" IsChecked="True"
         Checked="mnuDataShowCosts_Checked" Unchecked="mnuDataShowCosts_Unchecked"/>
        <!-- Other Data menu items omitted. -->
    </MenuItem>
</Menu>
```

When the user opens the Data menu, the mnuData_SubmenuOpened event handler executes. It searches a data directory and adds menu items to this menu that list network data files that the user might want to load.

The Show Costs menu item is checkable and is initially checked. When the user checks it, the mnuDataShowCosts_Checked event handler executes and redisplays the current network with costs displayed. When the user unchecks this menu item, the mnuDataShowCosts_Unchecked event handler executes and redisplays the current network with costs hidden.

PASSWORDBOX

The PasswordBox is a textbox that hides its text on the screen, displaying a dot, asterisk, or other character for each character typed (see Tables D-20 and D-21).

TABLE D-20: Key Properties of PasswordBox

PROPERTY	PURPOSE
Password	Determines the current password in plaintext.
PasswordChar	Determines the character displayed on the screen for each character entered in the PasswordBox.
SecurePassword	Returns the current password as a SecureString.

TABLE D-21: Key Events for PasswordBox

EVENT	PURPOSE
PasswordChanged	Occurs when the password changes.

EXAMPLE Using Password

The following code creates two PasswordBoxes. The first uses the default font and PasswordChar so it displays a filled circle for each character. The second uses the Wingdings font with PasswordChar = N so it displays little skulls and crossed bones.

```
<PasswordBox Password="Secret"/>
<PasswordBox Password="Secret" FontFamily="Wingdings" PasswordChar="N"/>
```

RADIOBUTTON

The `RadioButton` lets the user select one of an exclusive set of options (see Table D-22). When the user selects one `RadioButton`, all others in the same container are deselected. (Contrast this with the `CheckBox`, where any number can be selected at a time.)

> **CLEVER COMBO**
>
> Note that the `ComboBox` also lets the user select one of a series of choices so you could consider using a `ComboBox` instead of a group of `RadioButtons`, particularly if you need to save screen space.

You can create multiple `RadioButton` groups by placing them in different containers or by using the `GroupName` property.

TABLE D-22:

PROPERTY	PURPOSE
`IsChecked`	Determines whether the control is currently checked.
`GroupName`	Defines different `RadioButton` groups. `RadioButtons` that share the same non-null `GroupName` value are part of the same group, even if they lie in different containers.
`IsThreeState`	Determines whether the control cycles through three values — `checked` (`IsChecked = True`), `unchecked` (`IsChecked = False`), and `indeterminate` (`IsChecked = null`).

Only code can set a `RadioButton` to an indeterminate state. The user can only check or uncheck a `RadioButton`.

When a `RadioButton` is in an indeterminate state, it doesn't uncheck if the user checks a different `RadioButton` in the same group (see Table D-23).

TABLE D-23:

EVENT	PURPOSE
`Checked`	Occurs when the control is checked.
`Unchecked`	Occurs when the control is unchecked.
`Click`	Occurs when the control is checked or unchecked.

EXAMPLE Using RadioButton

The following code creates three GroupBoxes that let the user select a T-shirt's color, size, and design. Each GroupBox contains a RadioButton group.

```
<GroupBox Header="Color">
    <StackPanel>
        <RadioButton Content="Bisque"/>
        <RadioButton Content="BlanchedAlmond"/>
        <RadioButton Content="Teal"/>
        <RadioButton Content="Thistle"/>
    </StackPanel>
</GroupBox>
<GroupBox Header="Size">
    <StackPanel>
        <RadioButton Content="Small"/>
        <RadioButton Content="Medium"/>
        <RadioButton Content="Large"/>
    </StackPanel>
</GroupBox>
<GroupBox Header="Design">
    <StackPanel>
        <RadioButton Content="Smiley"/>
        <RadioButton Content="Heart"/>
        <RadioButton Content="VB"/>
        <RadioButton Content="C#"/>
    </StackPanel>
</GroupBox>
```

REPEATBUTTON

The RepeatButton acts like a normal Button except it repeatedly fires its Click event as long as the user holds the button down (see Tables D-24 and D-25).

TABLE D-24: Key Properties of RepeatButton

PROPERTY	PURPOSE
Delay	Determines the time in milliseconds after the button is pressed that it starts repeating Click events.
Interval	Determines the time in milliseconds between repeated Click events.

TABLE D-25: Key Events for RepeatButton

EVENT	PURPOSE
Click	Occurs when the control is first pressed and when repeat clicks occur.

EXAMPLE Using RepeatButton

The following code creates a `Rectangle` and two `RepeatButtons`. The `btnWider_Click` and `btnThinner_Click` event handlers catch the `Buttons`' `Click` events.

```
<Rectangle Name="rectColor" Fill="Red" Width="100" Height="100"/>
<RepeatButton Content="Wider" Delay="250" Interval="50"
 Click="btnWider_Click" />
<RepeatButton Content="Thinner" Delay="250" Interval="50"
 Click="btnThinner_Click" />
```

The following C# code-behind shows the event handlers:

```
// Make the Rectangle wider.
private void btnWider_Click(object sender, RoutedEventArgs e)
{
    rectColor.Width += 10;
}

// Make the Rectangle thinner.
private void btnThinner_Click(object sender, RoutedEventArgs e)
{
    if (rectColor.Width > 10) rectColor.Width -= 10;
}
```

When you press the `Wider` button, the event handler immediately makes the rectangle 10 pixels wider. A quarter-second (250 milliseconds) later, the first repeat click occurs, and the event handler widens the rectangle again. Every 50 milliseconds after that until you release the mouse, the `Click` event fires again, and the event handler widens the control.

The Thinner button works similarly.

RICHTEXTBOX

The `RichTextBox` lets the user enter formatted text (see Table D-26). Place a `FlowDocument` with formatting inside the `RichTextBox`. For more information on `FlowDocuments`, see the section "FlowDocument" in Appendix B.

TABLE D-26: Key Properties of RichTextBox

PROPERTY	PURPOSE
`AcceptsReturn`	Determines whether the control accepts the [Return] key and creates a new line.
`AcceptsTab`	Determines whether the control accepts the [Tab] key or whether that key moves focus to another control.
`AutoWordSelection`	Determines whether the control automatically selects entire words if the user drags the mouse across them.

PROPERTY	PURPOSE
CanRedo	Returns True if the control has an undone action that it can redo.
CanUndo	Returns True if the control has an action that it can undo.
CaretPosition	Gets or sets the current insertion position.
Document	Gets or sets the FlowDocument displayed by the control.
IsReadOnly	Determines whether the control can interact with the user.
Selection	Returns a TextSelection object that represents the current selection. That object's Text property returns the selection's plaintext.
SpellCheck	Returns a SpellCheck object that manages spell-checking for the control. Its IsEnabled property determines whether the control highlights misspelled words.
SpellCheck.IsEnabled	Enables or disables spell-checking.
UndoLimit	Determines the number of actions stored in the Undo queue.

The RichTextBox has many important methods that you may need to use (see Tables D-27 and D-28).

TABLE D-27:

METHOD	PURPOSE
AppendText	Adds text to the end of the control's contents.
BeginChange	Starts adding changes into a change block created by DeclareChangeBlock.
Copy	Copies the control's current selection to the clipboard.
Cut	Cuts the control's current selection to the clipboard.
DeclareChangeBlock	Creates a change block to group subsequent changes as a single group for undo and redo.
EndChange	Stops adding changes into a change block created by DeclareChangeBlock.
GetNextSpellingErrorPosition	Returns a TextPointer that represents the position of the next spelling error.
GetSpellingError	Returns a SpellingError object for any spelling error at a given point.
GetSpellingErrorRange	Returns a TextRange object for any spelling error at a given point.

continues

TABLE D-27 *(continued)*

METHOD	PURPOSE
`LineDown`	Scrolls the control's contents down one line.
`LineLeft`	Scrolls the control's contents left one line.
`LineRight`	Scrolls the control's contents right one line.
`LineUp`	Scrolls the control's contents up one line.
`PageDown`	Scrolls the control's contents down one page.
`PageLeft`	Scrolls the control's contents left one page.
`PageRight`	Scrolls the control's contents right one page.
`PageUp`	Scrolls the control's contents up one page.
`Paste`	Pastes the clipboard's contents over the control's current selection.
`Redo`	Redoes the most recently undone action.
`ScrollToEnd`	Scrolls to the end of the control's content.
`ScrollToHome`	Scrolls to the beginning of the control's content.
`SelectAll`	Selects all of the control's contents.
`Undo`	Undoes the most recently performed action.

TABLE D-28:

EVENT	PURPOSE
`TextChanged`	Occurs when the control's contents change.

EXAMPLE Using RichTextBox

The following code creates a `RichTextBox` that contains a simple `FlowDocument`:

```
<RichTextBox Name="rchNotes" Grid.Row="1" SpellCheck.IsEnabled="True">
    <FlowDocument>
        <Paragraph FontSize="18" FontWeight="Bold">
            XAML Example
        </Paragraph>
        <Paragraph FontSize="14">
            The following code creates a RichTextBox that contains
            a simple FlowDocument.
        </Paragraph>
    </FlowDocument>
</RichTextBox>
```

The example program SimpleRichEditor, which is shown in Figure D-1 and available for download on the book's web site, adds some simple formatting commands to a `RichTextBox`.

FIGURE D-1

The following C# code shows how the SimpleRichEditor program responds when the user selects a new entry from the size `ComboBox`.

Available for download on Wrox.com

```
// Give the selection the chosen size.
private void cboSize_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (rchNotes == null) return;
    if (e.AddedItems.Count < 1) return;
    ComboBoxItem item = (ComboBoxItem)e.AddedItems[0];
    string value = item.Content.ToString();

    rchNotes.Selection.ApplyPropertyValue(TextElement.FontSizeProperty, value);
    rchNotes.Focus();
}
```

SimpleRichEditor

The code first checks that the user has made a choice. It then converts the newly selected value into a string. It uses the `RichTextBox`'s `Selection`'s `ApplyPropertyValue` method to give the current selection the new font size.

The following code shows how the program responds when the user selects a new color:

Available for download on Wrox.com

```
// Give the selection the chosen color.
private void cboColor_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (rchNotes == null) return;
    if (e.AddedItems.Count < 1) return;
    ComboBoxItem item = (ComboBoxItem)e.AddedItems[0];
    string value = item.Content.ToString();

    switch (value)
    {
        case "Black":
            rchNotes.Selection.ApplyPropertyValue(TextElement.ForegroundProperty,
                Brushes.Black);
```

```
                break;
            case "Red":
                rchNotes.Selection.ApplyPropertyValue(TextElement.ForegroundProperty,
                    Brushes.Red);
                break;
            case "Green":
                rchNotes.Selection.ApplyPropertyValue(TextElement.ForegroundProperty,
                    Brushes.Green);
                break;
        }

        rchNotes.Focus();
    }
```

SimpleRichEditor

The code again checks that the user has made a selection and converts the newly selected item into a string. It then uses a switch statement to give the text the correct color.

The code for the other `ComboBoxes` is similar to these two examples.

The following code executes when the `RichTextBox`'s selection changes. For example, if the user moves the insertion cursor or clicks-and-drags to select a new piece of text, the selection changes and this code executes. The code updates the `ComboBoxes` to display the values that actually apply to the new selection.

For example, if the selection uses the Arial font, then the code makes the font name `ComboBox` display *Arial*.

Available for download on Wrox.com

```
 <// Display the selection's properties.
private void rchNotes_SelectionChanged(object sender, RoutedEventArgs e)
{
    if (rchNotes == null) return;

    // Font family.
    FontFamily font_family = rchNotes.Selection.GetPropertyValue(
        TextElement.FontFamilyProperty) as FontFamily;
    if (font_family == null)
    {
        cboFont.Text = "";
    } else {
        cboFont.Text = font_family.Source;
    }

    // Size.
    cboSize.Text = rchNotes.Selection.GetPropertyValue(
        TextElement.FontSizeProperty).ToString();

    // Color.
    SolidColorBrush br = rchNotes.Selection.GetPropertyValue(
        TextElement.ForegroundProperty) as SolidColorBrush;
    if (br == null)
    {
        cboColor.Text = "";
    } else {
        switch (br.Color.ToString())
```

```
            {
                case "#FF000000":
                    cboColor.Text = "Black";
                    break;
                case "#FFFF0000":
                    cboColor.Text = "Red";
                    break;
                case "#FF008000":
                    cboColor.Text = "Green";
                    break;
                default:
                    cboColor.Text = "";
                    break;
            }
        }

        // Weight.
        cboWeight.Text = rchNotes.Selection.GetPropertyValue(
            TextElement.FontWeightProperty).ToString();

        // Style.
        cboStyle.Text = rchNotes.Selection.GetPropertyValue(
            TextElement.FontStyleProperty).ToString();
    }
```

SimpleRichEditor

The code first gets the `FontFamily` property value. If the value is null, the program sets the font `ComboBox`'s text to a blank string. Otherwise, it sets the `ComboBox`'s text to the font's family name.

Next, the code gets the selection's font size, converts it into a string, and displays the value in the size `ComboBox`. There's one trick here. If the current selection contains different pieces of text that have different font sizes, then the call to `GetPropertyValue` doesn't return a size. Instead it returns a `NamedObect` that has the special value `UnsetValue`. The call to `ToString` returns `"{DependencyProperty.UnsetValue}"` and the code tries to make the size `ComboBox` display that value. Because that `ComboBox` doesn't have such a value, it displays nothing.

Having displayed the font name and size, the code gets the selection's `Foreground` property. If this is not an unset value, it is a `SolidColorBrush`. The `Brush`'s `Color.ToString` property returns a hexadecimal value representing the color's red, green, and blue components. The code uses a switch statement to determine which color it is and displays the corresponding result.

Finally, the code displays the selection's weight and style property values.

The `FlowDocument` contained in a `RichTextBox` can provide far more features than those demonstrated by this simple editor. For example, it provides lists, floaters, figures, and other more advanced features that the `RichTextBox` is not particularly suited to edit. To create and edit more complex documents, you should probably use some other tool like Microsoft Word.

SCROLLBAR

The `ScrollBar` lets the user select a numeric value from a range of values (see Tables D-29 and D-30). Typically, a program uses that value to adjust some parameter or scroll some object such as an image or drawing.

> **SCANT SCROLLING**
>
> *Often you can avoid using a* `ScrollBar` *to scroll an object such as an* `Image` *by using a* `ScrollViewer`*. The* `ScrollViewer` *automatically provides scrollbars and scrolls its contents as needed.*

TABLE D-29:

PROPERTY	PURPOSE
`LargeChange`	The amount by which `Value` is changed when the user clicks between the control's thumb and an arrow
`Maximum`	The largest value the control allows
`Minimum`	The smallest value the control allows
`Orientation`	`Horizontal` or `Vertical`
`SmallChange`	The amount by which `Value` is changed when the user clicks an arrow at one of the control's ends
`Value`	The control's current value

TABLE D-30:

EVENT	PURPOSE
`Scroll`	Occurs when the user changes the `ScrollBar`'s `Value`.
`ValueChanged`	Occurs when the `ScrollBar`'s `Value` changes either programmatically or because the user changed it.

SLIDER

Like the ScrollBar, the Slider lets the user select a numeric value from a range of values (see Tables D-31 and D-32).

TABLE D-31: Key Properties of Slider

PROPERTY	PURPOSE
IsSnapToTickEnabled	If True, then the Slider's thumb snaps to the nearest tick mark.
LargeChange	The amount by which Value is changed when the user clicks between the control's thumb and an arrow
Maximum	The largest value the control allows
Minimum	The smallest value the control allows
Orientation	Horizontal or Vertical
SelectionEnd	Determines the largest value for the control's current range selection.
SelectionStart	Determines the smallest value for the control's current range selection.
SmallChange	The amount by which Value is changed when the user clicks an arrow at one of the control's ends
TickFrequency	Determines the distance between tick marks.
TickPlacement	Determines where the tick marks are positioned. This can be Both (top and bottom for horizontal Sliders, left and right for vertical Sliders), BottomRight (bottom for horizontal Sliders, right for vertical Sliders), None, or TopLeft (top for horizontal Sliders, left for vertical Sliders).
Ticks	An array that determines the positions of individual tick marks
Value	The control's current value

TABLE D-32: Key Events for Slider

EVENT	PURPOSE
ValueChanged	Occurs when the ScrollBar's Value changes either programmatically or because the user changed it.

TEXTBOX

The `TextBox` lets the user enter text (see Tables D-33 through D-35). Unlike the `RichTextBox`, the `TextBox` can display text with only a single style (font, size, color, etc.).

TABLE D-33: Key Properties of TextBox

PROPERTY	PURPOSE
`AcceptsReturn`	Determines whether the control accepts the [Return] key and creates a new line.
`AcceptsTab`	Determines whether the control accepts the [Tab] key or whether that key moves focus to another control.
`AutoWordSelection`	Determines whether the control automatically selects entire words if the user drags the mouse across them.
`CanRedo`	Returns `True` if the control has an undone action that it can redo.
`CanUndo`	Returns `True` if the control has an action that it can undo.
`CaretIndex`	Gets or sets the current insertion position.
`CharacterCasing`	Determines how the control sets the case of characters typed by the user. This can be `Lower`, `Upper`, or `Normal`.
`HorizontalContentAlignment`	Determines how the text is aligned horizontally. This can be `Center`, `Left`, `Right`, or `Stretch`.
`HorizontalScrollBarVisibility`	Determines whether the horizontal scrollbar is visible. This can be `Auto`, `Disabled`, `Hidden`, or `Visible`.
`IsReadOnly`	Determines whether the control can interact with the user.
`LineCount`	Returns the number of lines of text in the control.
`MaxLength`	Determines the maximum number of characters allowed in the control.
`MaxLines`	Determines the maximum number of visible lines.
`MinLines`	Determines the minimum number of visible lines.
`SelectedLength`	The length of the current selection.
`SelectedStart`	The index of the first character in the current selection.
`SelectedText`	Determines the currently selected text. If you set this value, the currently selected text is replaced by the new value.

PROPERTY	PURPOSE
SpellCheck	Returns a SpellCheck object that manages spell-checking for the control. Its IsEnabled property determines whether the control highlights misspelled words.
SpellCheck.IsEnabled	Enables or disables spell-checking.
Text	Determines the control's textual contents.
TextAlignment	Determines how text is aligned within the control. This can be Center, Justify, Left, or Right.
TextWrapping	Determines how text is wrapped. This can be NoWrap, Wrap, or WrapWithOverflow.
UndoLimit	Determines the number of actions stored in the Undo queue.
VerticalContentAlignment	Determines how the text is aligned vertically. This can be Bottom, Center, Stretch, or Top.
VerticalScrollBarVisibility	Determines whether the vertical scrollbar is visible. This can be Auto, Disabled, Hidden, or Visible.

TABLE D-34: Key Methods for TextBox

METHOD	PURPOSE
AppendText	Adds text to the end of the control's contents.
Clear	Clears the control's contents.
Copy	Copies the control's current selection to the clipboard.
Cut	Cuts the control's current selection to the clipboard.
DeclareChangeBlock	Creates a change block to group subsequent changes as a single group for undo and redo.
EndChange	Stops adding changes into a change block created by DeclareChangeBlock.
GetNextSpellingErrorCharacterIndex	Returns the index of the character at the start of the next spelling error.
GetSpellingError	Returns a SpellingError object for any spelling error at a given point.
GetSpellingErrorLength	Returns the length of the spelling error including the indicated character.

continues

TABLE D-34 *(continued)*

METHOD	PURPOSE
`GetSpellingErrorStart`	Returns the index of the character that begins the spelling error including the indicated character.
`LineDown`	Scrolls the control's contents down one line.
`LineLeft`	Scrolls the control's contents left one line.
`LineRight`	Scrolls the control's contents right one line.
`LineUp`	Scrolls the control's contents up one line.
`PageDown`	Scrolls the control's contents down one page.
`PageLeft`	Scrolls the control's contents left one page.
`PageRight`	Scrolls the control's contents right one page.
`PageUp`	Scrolls the control's contents up one page.
`Paste`	Pastes the clipboard's contents over the control's selection.
`Redo`	Redoes the most recently undone action.
`ScrollToEnd`	Scrolls to the end of the control's content.
`ScrollToHome`	Scrolls to the beginning of the control's content.
`ScrollToLine`	Scrolls so a particular line is visible.
`Select`	Selects a specified range of text in the control.
`SelectAll`	Selects all of the control's contents.
`Undo`	Undoes the most recently performed action.

TABLE D-35: Key Events for TextBox

EVENT	PURPOSE
`TextChanged`	Occurs when the control's contents change.

E

MediaElement Control

This appendix summarizes the `MediaElement` control's most useful properties, methods, and events.

Note that many of these properties shown in Tables E-1 through E-3 are read-only.

TABLE E-1: Key Properties of Media

PROPERTY	PURPOSE
`Balance`	Determines the speaker balance.
`BufferingProgress`	Returns the percentage of buffering complete.
`CanPause`	Returns `True` if the media can be paused.
`DownloadProgress`	Returns the percentage of download complete.
`HasAudio`	Returns `True` if the media has audio.
`HasVideo`	Returns `True` if the media has video.
`IsBuffering`	Returns `True` if the control is buffering its media.
`IsMuted`	Returns `True` if the audio is muted.
`LoadedBehavior`	Determines what the control does when it is loaded. This can be `Manual` (control "manually" in code), `Play` (play when there's valid media), `Close` (close and release all memory), `Pause` (prepare to play and then pause), or `Stop` (prepare to play and then stop).
`NaturalDuration`	Returns the media's natural duration. This can be a `TimeSpan` (use the `NaturalDuration.HasTimeSpan` to see if it has one before using `NaturalDuration.TimeSpan`), `Automatic`, or `Forever`.
`NaturalVideoHeight`	Returns the video's height.

continues

TABLE E-1 *(continued)*

PROPERTY	PURPOSE
`NaturalVideoWidth`	Returns the video's width.
`Position`	Determines the playback position.
`ScrubbingEnabled`	Determines whether the control updates itself when a seek is performed while the control is paused.
`Source`	The media's source
`SpeedRatio`	Determines the speed ratio. Use a value less than 1 for slow motion or a value greater than 1 for fast motion.
`Volume`	Determines the audio volume between 0 and 1.

TABLE E-2: Key Methods for Media

METHODS	PURPOSE
`Pause`	Pauses media playback. Use the `Play` method to restart playback.
`Play`	Starts media playback.
`Stop`	Stops media playback and resets the control to play from the beginning.

TABLE E-3: Key Events for Media

EVENT	PURPOSE
`BufferingEnded`	Occurs when the control finishes buffering.
`BufferingStarted`	Occurs when the control starts buffering.
`MediaEnded`	Occurs when the media ends.
`MediaFailed`	Occurs when the control has an error.
`MediaOpened`	Occurs when the media is loaded.

EXAMPLE Controlling Media

The ControlMedia example program shown in Figure E-1 uses a `MediaElement` to play a video. The buttons from left to right allow you to rewind 5 seconds, play, pause, stop, skip 5 seconds, and add a bookmark to the `ListBox`. If you click on a bookmark entry, the control jumps to that position in the video. The scrollbar at the bottom lets you view and change the playback position.

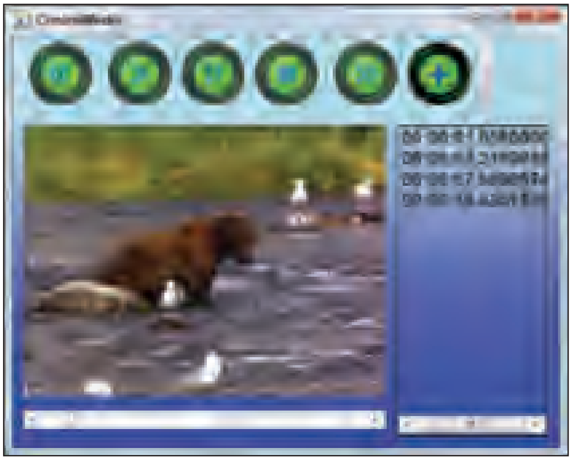


FIGURE E-1

Since this program's XAML code is reasonably straightforward, it isn't shown here. Download the example program and look it over.

The following code shows the program's C# code-behind. (As is the case for all of the examples, you can also download a Visual Basic version.)

Available for download on Wrox.com

```
private DispatcherTimer tmrProgress = new DispatcherTimer();

// Prepare the timer.
private void Window1_Loaded(object sender, RoutedEventArgs e)
{
    tmrProgress = new DispatcherTimer();
    tmrProgress.Tick += tmrProgress_Tick;
}

private void btnRewind_Click(object sender, RoutedEventArgs e)
{
    mmBear.Position = mmBear.Position.Add(new TimeSpan(0, 0, -5));
}
private void btnPlay_Click(object sender, RoutedEventArgs e)
{
    mmBear.Play();
    tmrProgress.Start();
}
private void btnPause_Click(object sender, RoutedEventArgs e)
{
    mmBear.Pause();
    tmrProgress.Stop();
}
private void btnStop_Click(object sender, RoutedEventArgs e)
{
    mmBear.Stop();
    tmrProgress.Stop();
}
```

```
private void btnForward_Click(object sender, RoutedEventArgs e)
{
    mmBear.Position = mmBear.Position.Add(new TimeSpan(0, 0, 5));
}
private void btnBookmark_Click(object sender, RoutedEventArgs e)
{
    lstBookmarks.Items.Add(mmBear.Position);
}

// Jump to this bookmark.
private void lstBookmarks_SelectionChanged(object sender,
    SelectionChangedEventArgs e)
{
    if (mmBear == null) return;
    if (e.AddedItems.Count < 1) return;
    TimeSpan bookmark = (TimeSpan)e.AddedItems[0];
    mmBear.Pause();
    mmBear.Position = bookmark;
    scrProgress.Value = mmBear.Position.TotalSeconds;
    lstBookmarks.SelectedIndex = -1;
}

// Display the media's progress.
private void tmrProgress_Tick(Object sender, System.EventArgs e)
{
    scrProgress.Value = mmBear.Position.TotalSeconds;
}

// Prepare the progress ScrollBar.
private void mmBear_MediaOpened(System.Object sender,
    System.Windows.RoutedEventArgs e)
{
    if (mmBear.NaturalDuration.HasTimeSpan)
    {
        TimeSpan ts = mmBear.NaturalDuration.TimeSpan;
        scrProgress.Maximum = ts.TotalSeconds;
        scrProgress.SmallChange = 1;
        scrProgress.LargeChange = ts.TotalSeconds / 10;
    } else {
        scrProgress.Visibility = Visibility.Hidden;
    }
}

// Go to the selected position.
private void scrPosition_Scroll(object sender,
    System.Windows.Controls.Primitives.ScrollEventArgs e)
{
    mmBear.Position = TimeSpan.FromSeconds(e.NewValue);
}
```

ControlMedia

When the window loads, the program creates a new `DispatcherTimer` and registers the `tmrProgress_Tick` event handler to receive the timer's `Tick` events.

The program's rewind, play, pause, stop, and forward event handlers are straightforward. They simply invoke the `MediaElement`'s corresponding methods.

The `bookmark Button`'s event handler adds the `MediaControl`'s current position to the `lstBookmarks ListBox`.

When you select an entry in the `ListBox`, the `lstBookmarks_SelectionChanged` event handler executes. After verifying that the `ListBox` has a selection, the code gets the selected `TimeSpan`, pauses the media playback, sets the control's position to the selected `TimeSpan`, and updates the `ScrollBar` to show the new position.

When the timer's `Tick` event occurs, the `tmrProgress_Tick` event handler sets the `ScrollBar`'s `Value` to show the control's position in the media.

When the `MediaElement` raises its `MediaOpened` event, the code determines whether the control's `NaturalDuration` property has a `TimeSpan` value and, if it does, prepares the `ScrollBar` to display the media's progress.

Finally, when you scroll the `ScrollBar`, the program adjusts the media's position accordingly.

F

Pens

This appendix summarizes Pens. A Pen defines the colors and geometry of linear features such as lines, paths, and the edges of drawn shapes like Rectangles.

Typically Pen properties are provided by some other object rather than the Pen itself and determine how the object defines its Pen.

For example, the Ellipse control's Stroke and StrokeThickness properties determine how the control draws its border. The following code draws an ellipse with a red border that's 10 pixels wide:

```
<Ellipse Width="200" Height="100" Stroke="Red" StrokeThickness="10"/>
```

Table F-1 summarizes Pen's key properties.

TABLE F-1: Key Properties of Pen

PROPERTY	PURPOSE
Stroke	Determines the Pen's Brush. This can be a solid color like red or an element attribute that specifies a more complicated brush such as a LinearGradientBrush.
StrokeDashArray	Determines the Pen's pattern of drawn and skipped segments in units of the pen's thickness. For example, the pattern 3,1 means "draw three pen widths, skip one pen width, and repeat."
StrokeDashCap	Determines the shape of the ends of each dash. This can be Flat, Round, Square, or Triangle.
StrokeDashOffset	Determines how far into the first dash the line starts.
StrokeEndLineCap	Determines the shape of the Pen's finishing end. This can be Flat, Round, Square, or Triangle.
StrokeLineJoin	Determines how the Pen draws corners. This can be Bevel, Miter, or Round.

continues

TABLE F-1 *(continued)*

PROPERTY	PURPOSE
`StrokeMiterLimit`	Determines how pointy a corner can be when two segments connect as in a `Polygon`.
`StrokeStartLineCap`	Determines the shape of the `Pen`'s starting end. This can be `Flat`, `Round`, `Square`, or `Triangle`.
`StrokeThickness`	Determines the `Pen`'s thickness.

EXAMPLE Understanding StrokeMiterLimit

The StrokeMiterLimit example program shown in Figure F-1 uses the following code to draw its shape. This example defines the `Pen`'s `Stroke` property with an element attribute containing a `RadialGradientBrush`. It also sets `StrokeMiterLimit` to 3 so the sharpest corner on the lower right is beveled.

Available for download on Wrox.com

```
<Polygon Points="100,30 250,200 30,70 200,50 75,170" StrokeThickness="20"
 StrokeLineJoin="Miter" StrokeMiterLimit="3">
    <Polygon.Stroke>
        <RadialGradientBrush>
            <GradientStop Offset="0" Color="Gray"/>
            <GradientStop Offset="0.5" Color="Gray"/>
            <GradientStop Offset="1" Color="Black"/>
        </RadialGradientBrush>
    </Polygon.Stroke>
</Polygon>
```

StrokeMiterLimit

The StrokeDashes example program shown in Figure F-2 demonstrates the `StrokeStartLineCap`, `StrokeEndLineCap`, and `StrokeDashCap` values.

FIGURE F-1

FIGURE F-2

G

Brushes

This appendix summarizes brushes. A `Brush` defines the colors used to fill an area such as a `Rectangle` or `Ellipse`. `Brushes` also define the colors used by many controls' `Foreground` and `Background` properties.

The following section briefly describes each of the types of `Brushes` available. The sections that follow describe each of the `Brush` types in greater detail.

BRUSH CLASSES

Table G-1 summarizes the `Brush` classes.

TABLE G-1:

BRUSH CLASS	PURPOSE
`DrawingBrush`	Fills areas with a drawing.
`ImageBrush`	Fills areas with one or more copies of an image.
`LinearGradientBrush`	Fills areas with a color gradient that shades between two or more colors in a linear direction.
`RadialGradientBrush`	Fills areas with a color gradient that shades between two or more colors radially.
`SolidColorBrush`	Fills areas with a solid color.
`VisualBrush`	Fills areas with a visual. (Loosely speaking, a *visual* is an object that provides rendering support for WPF. It's basically something that can draw itself such as a control.)

The Brushes example program shown in Figure G-1 demonstrates these kinds of brushes. The `SolidColorBrush` fills its `Ellipse` with dark gray, the `LinearGradientBrush` shades from white at the top to black at the bottom, and the `RadialGradientBrush` shades from white at the middle to black at the edges.

FIGURE G-1

DRAWINGBRUSH

A `DrawingBrush` fills areas with a `Drawing` object.

Table G-2 summarizes the `DrawingBrush`'s most important properties.

TABLE G-2:

PROPERTY	PURPOSE
`AlignmentX`	Determines how the `Brush` is aligned if its `Stretch` property is set to `None` so it doesn't stretch to fill the tile area.
`AlignmentY`	Determines how the `Brush` is aligned if its `Stretch` property is set to `None` so it doesn't stretch to fill the tile area.
`Drawing`	The `Brush`'s content
`Stretch`	Determines how the `Brush` is stretched to fill its tiles. This can be `None` (the drawing is used at its original size), `Uniform` (the drawing is uniformly stretched to be as large as possible within the tile), `UniformToFill` (the drawing is uniformly stretched to fill the tile), and `Fill` (the drawing is stretched to fill the tile even if it distorts the drawing).
`TileMode`	Determines how the tile is repeated if necessary to fill the area. This can be `None` (the rest of the area is unfilled), `Tile` (the tile is repeated), `FlipX` (the tile is repeated with every other tile in the X direction flipped horizontally), `FlipY` (the tile is repeated with every other tile in the Y direction flipped vertically), and `FlipXY` (both `FlipX` and `FlipY`). This only has an effect if `Viewport` is smaller than the filled area.
`Viewbox`	Defines the part of the content used for the `Brush`.

PROPERTY	PURPOSE
`ViewboxUnits`	Defines the units used by `Viewbox`. This can be `Absolute` (pixels) or `RelativeToBoundingBox` [where `(0, 0)` is the upper-left corner and `(1, 1)` is the lower right corner].
`Viewport`	Defines the part of the filled area that is covered by a single tile.
`ViewportUnits`	Defines the units used by `Viewport`. This can be `Absolute` (pixels) or `RelativeToBoundingBox` [where `(0, 0)` is the upper-left corner and `(1, 1)` is the lower right corner].

Drawing Types

Table G-3 lists the kinds of `Drawing` objects you can use to make a `DrawingBrush`.

TABLE G-3:

DRAWING TYPE	PURPOSE
`DrawingGroup`	Contains a group of other `Drawing` objects.
`GeometryDrawing`	Draws shapes.
`GlyphRunDrawing`	Draws text.
`ImageDrawing`	Draws an image.
`VideoDrawing`	Draws a video file.

EXAMPLE Using DrawingBrushes

The following code shows how the Brushes program shown in Figure G-1 creates its `DrawingBrush`:

Available for download on Wrox.com

```
<Ellipse Grid.Row="0" Grid.Column="0" Stroke="Black">
    <Ellipse.Fill>
        <DrawingBrush TileMode="Tile" Viewbox="0,0,1,1"
         Viewport="0,0,0.33,0.5" ViewportUnits="RelativeToBoundingBox">
            <DrawingBrush.Drawing>
                <GeometryDrawing Brush="Gray">
                    <GeometryDrawing.Geometry>
                        <GeometryGroup>
                            <PathGeometry>
                                <PathFigure IsClosed="True" StartPoint="10,0">
                                    <PolyLineSegment Points="20,10 10,20 0,10"/>
                                </PathFigure>
```

```
                        </GeometryDrawing.Geometry>
                        <GeometryDrawing.Pen>
                            <Pen Thickness="2">
                                <Pen.Brush>
                                    <LinearGradientBrush
                                     StartPoint="0,0" EndPoint="0,1">
                                        <GradientStop Offset="0.0" Color="Black" />
                                        <GradientStop Offset="0.5" Color="White" />
                                        <GradientStop Offset="1.0" Color="Black" />
                                    </LinearGradientBrush>
                                </Pen.Brush>
                            </Pen>
                        </GeometryDrawing.Pen>
                    </GeometryDrawing>
                </DrawingBrush.Drawing>
            </DrawingBrush>
        </Ellipse.Fill>
    </Ellipse>
```

Brushes

The code starts an `Ellipse` object and creates the `Brush` in the `Ellipse.Fill` element attribute. The `Brush` uses `TileMode = Tile` so it will repeat as needed.

The property value `Viewbox = 0,0,1,1` makes the `Brush` use the entire drawing area for its content. (By default, `ViewboxUnits` is `RelativeToBoundingBox`.)

The code sets `Viewport = 0,0,0.33,0.5` and `ViewportUnits = RelativeToBoundingBox` so the first tile fills an area that is a third of the `Ellipse`'s width and half of its height.

Next, the code defines the `Brush`'s `Drawing` property. That element contains a `GeometryDrawing`. The `GeometryDrawing`'s `Geometry` attribute contains a `GeometryGroup` that holds a `PathGeometry` and an `EllipseGeometry`.

The `PathGeometry` contains a `PathFigure` that draws a single polyline (series of connected lines). The figure is closed (the endpoint connects to the start point) and starts at the point (10, 0). The points defined by the `PolyLineSegment` draw a diamond shape.

The `EllipseGeometry` object draws an ellipse of radius 2 centered at (20, 20), which is also the center of the diamond.

After finishing the `GeometryGroup`, the code defines the `GeometryDrawing` object's `Pen`. This is a `Pen` of thickness 2 drawn with a `LinearGradientBrush` that shades from black to white and back to black.

IMAGEBRUSH

The `ImageBrush` fills an area with an image.

Table G-4 summarizes the `ImageBrush`'s most important properties.

TABLE G-4:

PROPERTY	PURPOSE
`AlignmentX`	Determines how the `Brush` is aligned if its `Stretch` property is set to `None` so it doesn't stretch to fill the tile area.
`AlignmentY`	Determines how the `Brush` is aligned if its `Stretch` property is set to `None` so it doesn't stretch to fill the tile area.
`ImageSource`	The URI of the image drawn by the `Brush`
`Stretch`	Determines how the `Brush` is stretched to fill its tiles. This can be `None` (the drawing is used at its original size), `Uniform` (the drawing is uniformly stretched to be as large as possible within the tile), `UniformToFill` (the drawing is uniformly stretched to fill the tile), and `Fill` (the drawing is stretched to fill the tile even if it distorts the drawing).
`TileMode`	Determines how the tile is repeated if necessary to fill the area. This can be `None` (the rest of the area is unfilled), `Tile` (the tile is repeated), `FlipX` (the tile is repeated with every other tile in the X direction flipped horizontally), `FlipY` (the tile is repeated with every other tile in the Y direction flipped vertically), and `FlipXY` (both `FlipX` and `FlipY`). This only has an effect if `Viewport` is smaller than the filled area.
`Viewbox`	Defines the part of the content used for the `Brush`.
`ViewboxUnits`	Defines the units used by `Viewbox`. This can be `Absolute` (pixels) or `RelativeToBoundingBox` [where `(0, 0)` is the upper-left corner and `(1, 1)` is the lower-right corner].
`Viewport`	Defines the part of the filled area that is covered by a single tile.
`ViewportUnits`	Defines the units used by `Viewport`. This can be `Absolute` (pixels) or `RelativeToBoundingBox` [where `(0, 0)` is the upper-left corner and `(1, 1)` is the lower-right corner].

EXAMPLE Using ImageBrushes

The following code shows how the Brushes program shown in Figure G-1 creates its `ImageBrush`:

Available for download on Wrox.com

```
<Ellipse Grid.Row="0" Grid.Column="1" Stroke="Black">
    <Ellipse.Fill>
        <ImageBrush ImageSource="Smiley.bmp"
         TileMode="Tile"
         Viewbox="0,0,1,1" ViewboxUnits="RelativeToBoundingBox"
         Viewport="0,0,50,50" ViewportUnits="Absolute"/>
    </Ellipse.Fill>
</Ellipse>
```

LINEARGRADIENTBRUSH

A `LinearGradientBrush` fills areas with a color gradient that shades between two or more colors in a linear direction.

Table G-5 summarizes the `LinearGradientBrush`'s most important properties.

TABLE G-5: Key Properties of LinearGradientBrush

PROPERTY	PURPOSE
`EndPoint`	The point where the gradient ends
`GradientStops`	A collection of objects that determine the `Brush`'s colors at different parts of the gradient
`MappingMode`	Determines how the `StartPoint` and `EndPoint` are mapped to the `Brush`. This can be `Absolute` (the points' coordinates are in pixels) or `RelativeToBoundingBox` [the point `(0, 0)` is the upper-left corner of the brush and `(1, 1)` is the lower-right corner].
`SpreadMethod`	Determines how an area is filled if the `Brush` doesn't completely cover it. This can be `Pad`, `Reflect`, or `Repeat`.
`StartPoint`	The point where the gradient begins

The SpreadMethods example program shown in Figure G-2 demonstrates the different `SpreadMethod` values.

The `LinearGradientBrush` should contain a series of `GradientStop` objects that define the colors along the gradient. The ordering of these objects is unimportant.

Table G-6 summarizes the `GradientStop` object's two most useful properties.

FIGURE G-2

TABLE G-6: Key Properties of GradientStop

PROPERTY	PURPOSE
`Offset`	The offset in the gradient where this color should be used. This value is usually between 0 (the start of the gradient) and 1 (the end of the gradient).
`Color`	The color that should be drawn at this location

EXAMPLE Using LinearGradientBrushes

The following code shows how the program Brushes shown in Figure G-1 creates its `LinearGradientBrush`:

Available for download on Wrox.com

```
<Ellipse Grid.Row="0" Grid.Column="2" Stroke="Black">
    <Ellipse.Fill>
        <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
            <GradientStop Offset="0" Color="White"/>
            <GradientStop Offset="1" Color="Black"/>
        </LinearGradientBrush>
    </Ellipse.Fill>
</Ellipse>
```

Brushes

RADIALGRADIENTBRUSH

A `RadialGradientBrush` fills areas with a color gradient that shades between two or more colors radially.

Table G-7 summarizes the `RadialGradientBrush`'s most important properties.

TABLE G-7: Key Properties of RadialGradientBrush

PROPERTY	PURPOSE
`Center`	The center point of the outer ellipse to which the gradient shades
`GradientOrigin`	The point where the gradient begins
`GradientStops`	A collection of objects that determine the `Brush`'s colors at different parts of the gradient
`MappingMode`	Determines how the `StartPoint` and `EndPoint` are mapped to the `Brush`. This can be `Absolute` (the points' coordinates are in pixels) or `RelativeToBoundingBox` [the point `(0, 0)` is the upper-left corner of the brush and `(1, 1)` is the lower-right corner].
`RadiusX`	The X radius (half-width) of the outer ellipse to which the gradient shades
`RadiusY`	The Y radius (half-height) of the outer ellipse to which the gradient shades
`SpreadMethod`	Determines how an area is filled if the `Brush` doesn't completely cover it. This can be `Pad`, `Reflect`, or `Repeat`.

The gradient starts at the object's `GradientOrigin` point and shades to the edges of an ellipse centered at the point Center.

The `LinearGradientBrush` should contain a series of `GradientStop` objects that define the colors along the gradient. The ordering of these objects is unimportant.

Table G-8 summarizes the `GradientStop` object's two most useful properties.

TABLE G-8: Key Properties of GradientStop

PROPERTY	PURPOSE
`Offset`	The offset in the gradient where this color should be used. This value is usually between 0 (the start of the gradient) and 1 (the end of the gradient).
`Color`	The color that should be drawn at this location

EXAMPLE Using ImageBrushes

The following code shows how the Brushes program shown in Figure G-1 creates its `RadialGradientBrush`:

Available for download on Wrox.com

```
<Ellipse Grid.Row="2" Grid.Column="0" Stroke="Black">
    <Ellipse.Fill>
        <RadialGradientBrush>
            <GradientStop Offset="0" Color="White"/>
            <GradientStop Offset="1" Color="Black"/>
        </RadialGradientBrush>
    </Ellipse.Fill>
</Ellipse>
```

Brushes

The RadialCenter example program shown in Figure G-3 demonstrates some of the `RadialGradientBrush`'s more confusing properties. The gradient shades from the point indicated by the `GradientOrigin` property (in the upper left) to the edges of the dashed circle. The `RadiusX` and `RadiusY` properties give the size of the dashed circle. The `Center` property determines where that circle is centered.

FIGURE G-3

SOLIDCOLORBRUSH

The `SolidColorBrush` has a single important property: `Color`. You can use an element attribute to make a `SolidColorBrush`, but usually it's easier to simply set the property of the control using the brush to a color.

For example, the following code creates two `Ellipses` that are both filled with green. The first uses an element attribute, while the second simply sets the `Ellipse`'s `Fill` property to Green.

```
<Ellipse Stroke="Black">
    <Ellipse.Fill>
        <SolidColorBrush Color="Green"/>
    </Ellipse.Fill>
</Ellipse>

<Ellipse Stroke="Black" Fill="Green"/>
```

VISUALBRUSH

The `VisualBrush` fills areas with a visual such as a control.

Table G-9 summarizes the `VisualBrush`'s most important properties.

TABLE G-9: Key Properties of VisualBrush

PROPERTY	PURPOSE
`AlignmentX`	Determines how the `Brush` is aligned if its `Stretch` property is set to `None` so it doesn't stretch to fill the tile area.
`AlignmentY`	Determines how the `Brush` is aligned if its `Stretch` property is set to `None` so it doesn't stretch to fill the tile area.
`Stretch`	Determines how the `Brush` is stretched to fill its tiles. This can be `None` (the drawing is used at its original size), `Uniform` (the drawing is uniformly stretched to be as large as possible within the tile), `UniformToFill` (the drawing is uniformly stretched to fill the tile), and `Fill` (the drawing is stretched to fill the tile even if it distorts the drawing).
`TileMode`	Determines how the tile is repeated if necessary to fill the area. This can be `None` (the rest of the area is unfilled), `Tile` (the tile is repeated), `FlipX` (the tile is repeated with every other tile in the X direction flipped horizontally), `FlipY` (the tile is repeated with every other tile in the Y direction flipped vertically), and `FlipXY` (both `FlipX` and `FlipY`). This only has an effect if `Viewport` is smaller than the filled area.
`Viewbox`	Defines the part of the content used for the `Brush`.
`ViewboxUnits`	Defines the units used by `Viewbox`. This can be `Absolute` (pixels) or `RelativeToBoundingBox` [where `(0, 0)` is the upper-left corner and `(1, 1)` is the lower-right corner].
`Viewport`	Defines the part of the filled area that is covered by a single tile.

continues

TABLE G-9 *(continued)*

PROPERTY	PURPOSE
`ViewportUnits`	Defines the units used by `Viewport`. This can be `Absolute` (pixels) or `RelativeToBoundingBox` [where `(0, 0)` is the upper-left corner and `(1, 1)` is the lower-right corner].
`Visual`	Determines the `Brush`'s contents. This can be a reference to an existing visual or a new one.

EXAMPLE Using ImageBrushes

The following code shows how the Brushes program shown in Figure G-1 creates its `VisualBrush` that displays the text *Visual* normally and reflected vertically:

Available for download on Wrox.com

```
<Ellipse Grid.Row="2" Grid.Column="2" Stroke="Black">
    <Ellipse.Fill>
        <VisualBrush TileMode="FlipY" AlignmentX="Left" AlignmentY="Top"
         Viewbox="0,5,50,25" ViewboxUnits="Absolute"
         Viewport="0,0,30,20" ViewportUnits="Absolute">
            <VisualBrush.Visual>
                <Label Content="Visual" FontFamily="Comic Sans MS"/>
            </VisualBrush.Visual>
        </VisualBrush>
    </Ellipse.Fill>
</Ellipse>
```

Brushes

The `Brush`'s `Visual` property is an element attribute that defines a new `Label` containing the `Brush`'s content.

The Reflections example program shown in Figure G-4 demonstrates a `VisualBrush` that uses an existing visual. On the top, it displays a horizontal `StackPanel` that holds a `TextBox`, an `Image`, and a `Button`. On the bottom, it draws a `Rectangle` filled with a reflected image of the `StackPanel`. The visual even updates at run time so if you type into the `TextBox`, the reflection immediately updates to match.

FIGURE G-4

The following code shows how the Reflections program works:

Available for download on Wrox.com

```
<StackPanel Margin="10">
    <StackPanel Name="spReflect" Height="50" Orientation="Horizontal">
        <TextBox Name="txtOriginal" Text="Reflection">
            <TextBox.Background>
                <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
                    <GradientStop Offset="0" Color="LightGray"/>
                    <GradientStop Offset="1" Color="Black"/>
                </LinearGradientBrush>
            </TextBox.Background>
```

```
            </TextBox>
            <Image Source="Smiley.bmp" Margin="5"/>
            <Button Content="Click Me!"/>
        </StackPanel>
        <Rectangle Height="50" Margin="0,3,0,0">
            <Rectangle.Fill>
                <VisualBrush Opacity="0.5" TileMode="Tile"
                 Stretch="None" AlignmentX="Left"
                 Visual="{Binding ElementName=spReflect}">
                    <VisualBrush.RelativeTransform>
                        <TransformGroup>
                            <ScaleTransform ScaleX="1" ScaleY="-1"/>
                        </TransformGroup>
                        </VisualBrush.RelativeTransform>
                </VisualBrush>
            </Rectangle.Fill>
        </Rectangle>
    </StackPanel>
```

Reflections

The first half of the program is straightforward and simply creates the first `StackPanel` and its `TextBox`, `Image`, and `Button`.

The second half of the code creates a `Rectangle` filled with a `VisualBrush`. Note how the `Brush`'s `Visual` property refers to the previously defined `StackPanel`.

The `Brush`'s `RelativeTransform` property reflects the brush vertically by multiplying the Y coordinates by a factor of –1.

REFLECTION REVIEWED

The reflection actually moves the result below the X axis so the default viewport (0, 0) to (1, 1) doesn't contain any of the image. This code sets the `Brush`'s `TileMode` property to `Tile` so the `Brush` repeats its image as needed and that fills the `Rectangle` properly.

Alternatively, you could follow the `ScaleTransform` with a `TranslateTransform` that translates by distance 1 in the Y direction to move the image back into the visible area.

VIEWPORTS AND VIEWBOXES

Two of the more confusing concepts behind tiled brushes are viewports and viewboxes. Briefly, the *viewbox* defines the part of the brush's defined content that is used to fill the area. The *viewport* defines the part of the output area that should be filled with one tile of the brush.

The ViewportsAndViewboxes example program shown in Figure G-5 demonstrates viewports and viewboxes.

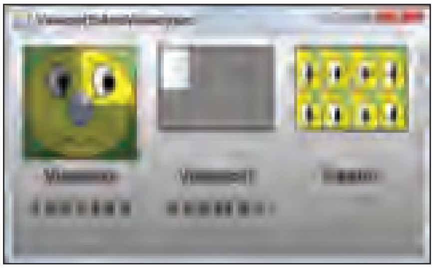


FIGURE G-5

In the image on the left, the lighter area shows the viewbox defined by the property value `Viewbox = 0.5,0,0.5,0.5`. These numbers represent the left, top, width, and height of the `Brush`'s content that is used to draw the `Brush`. In this example, the coordinates are relative so the upper-left corner is at (0, 0) and the lower-right corner is at (1, 1).

The middle of Figure G-5 shows a `Rectangle` with the viewport `0,0,0.25,0.5` filled with white. These values also give the left, top, width, and height of the area and are relative coordinates.

The right of Figure G-5 shows the result. The area defined by the viewbox on the left is stretched to fill the viewport defined in the middle. That result is then tiled to fill the `Rectangle`.

The following code shows how the program draws the final rectangle:

Available for download on Wrox.com

```
<Rectangle Width="100" Height="75" Stroke="Black">
    <Rectangle.Fill>
        <ImageBrush ImageSource="Smiley.bmp" Stretch="Fill" TileMode="Tile"
         AlignmentX="Left" AlignmentY="Top"
         Viewbox="0.5,0,0.5,0.5" ViewboxUnits="RelativeToBoundingBox"
         Viewport="0,0,0.25,0.5" ViewportUnits="RelativeToBoundingBox"/>
    </Rectangle.Fill>
</Rectangle>
```

ViewportsAndViewboxes

H

Path Mini-Language

This appendix summarizes the Path mini-language (or path markup syntax) that you can use to concisely define the lines and curves drawn by `Path` objects. Place mini-language commands in the `Path` control's `Data` property.

Table H-1 describes the Path mini-language's commands. Uppercase versions use absolute coordinates, while lowercase versions use points relative to the previous points.

TABLE H-1: Path Mini-Language Commands

COMMAND	MEANING
`F0`	Use Odd/Even fill rule. (See Figure H-1.)
`F1`	Use Non-Zero fill rule. (See Figure H-1.)
`M` or `m`	Move to the following point.
`L` or `l`	Draw lines to the following points.
`H` or `h`	Draw a horizontal line to the given X coordinate.
`V` or `v`	Draw a vertical line to the given Y coordinate.
`C` or `c`	Draw a cubic Bézier curve. This command takes three points as parameters: two control points and an endpoint. The curve starts at the current point moving toward the first control point and ends at the endpoint moving away from the second control point. (See Figure H-2.)
`S` or `s`	Draw a smooth Bézier curve. This command takes two points as parameters: a control point and an endpoint. The curve defines an initial control point by reflecting the final control point from the previous `S` command. It then draws a cubic Bézier curve using the newly defined control point and the two parameter points. This makes the second curve smoothly join with the previous one. (See Figure H-2.)

continues

TABLE H-1 *(continued)*

COMMAND	MEANING
`Q` or `q`	Draw a quadratic Bézier curve. This command takes two points as parameters: a control point and an endpoint. The curve starts at the current point moving toward the control point and ends at the endpoint moving away from the control point. (See Figure H-2.)
`T` or `t`	Draw a smooth Bézier curve defined by a single point. This command takes a single point as a parameter and draws a smooth curve to that point. It reflects the previous `T` command's control point to define a control point for the new section of curve and uses it to draw a quadratic Bézier curve. The result is a smooth curve that passes through the points sent to consecutive `T` commands. (See Figure H-2.)
`A` or `a`	Draws an elliptical arc starting at the current point and defined by five parameters: ➤ `size` — The X and Y radii of the arc ➤ `rotation_angle` — The ellipse's angle of rotation in degrees ➤ `large_angle` — 0 if the arc should span less than 180 degrees; 1 if it should span 180 or more degrees ➤ `sweep_direction` — 0 for counterclockwise; 1 for clockwise ➤ `end_point` — The point where the arc should end (see Figure H-3)
`Z` or `z`	Close the figure by drawing a line from the current point to the first point.

The FillRules example program shown in Figure H-1 demonstrates the Odd/Even and Non-Zero fill rules by drawing two shapes filled with white.

FIGURE H-1

The BezierCommands example program shown in Figure H-2 demonstrates the different Bézier drawing commands. The straight lines show where the curves' control points are.

FIGURE H-2

The Arc example program shown in Figure H-3 demonstrates the `arc` command.

FIGURE H-3

For more information on the `Path` object, see the “Path” section in Chapter 8.

I

XPath

A program that binds to XML data uses XPath to determine which data is selected and which data is displayed. This appendix summarizes XPath and provides several examples.

XML IN XAML

To include XML data inside XAML code, build an `XmlDataProvider` object containing an `x:XData` object. Inside that object, place well-formed XML data.

The `XmlXDataProvider`'s `XPath` property gives the name of the data's root. Later code can refer to this root.

The following XAML code defines XML data used by later examples in this appendix. The root data element is named `Customers`. That element contains a series of `Customer` objects, each holding `Order` objects that contain `Item` objects. An `Item` object's value gives its description. Attributes give additional information about the various kinds of objects.

Available for download on Wrox.com

```
<XmlDataProvider x:Key="customers" XPath="Customers">
    <x:XData>
        <Customers xmlns="">
            <Customer CustName="Bob Farkle">
                <Order OrderDate="4/1/2010">
                    <Item Quantity="12" UnitPrice="0.10">Pencils</Item>
                    <Item Quantity="3" UnitPrice="0.95">Notepad</Item>
                    <Item Quantity="1" UnitPrice="3.99">Cookies, Dozen</Item>
                </Order>
            </Customer>
            <Customer CustName="Edwin Cumberbund">
                <Order OrderDate="4/13/2010">
                    <Item Quantity="6" UnitPrice="0.10">Pencils</Item>
                    <Item Quantity="6" UnitPrice="1.40">Green Tea</Item>
                    <Item Quantity="1" UnitPrice="39.95">Book</Item>
                </Order>
                <Order OrderDate="5/1/2010">
                    <Item Quantity="1" UnitPrice="6.99">Napkins</Item>
```

```
                    <Item Quantity="1" UnitPrice="13.95">Rechargeable
                     Batteries</Item>
                </Order>
            </Customer>
            <Customer CustName="Emily Pickle">
                <Order OrderDate="4/1/2010">
                    <Item Quantity="1" UnitPrice="13.95">Rechargeable
                     Batteries</Item>
                    <Item Quantity="2" UnitPrice="2.99">Duct Tape</Item>
                </Order>
                <Order OrderDate="4/26/2010">
                    <Item Quantity="3" UnitPrice="1.90">Highlighter</Item>
                    <Item Quantity="1" UnitPrice="499.90">1 PB Flash Drive</Item>
                    <Item Quantity="2" UnitPrice="0.10">Pencils</Item>
                </Order>
                <Order OrderDate="4/27/2010">
                    <Item Quantity="1" UnitPrice="1.25">Soda</Item>
                </Order>
            </Customer>
        </Customers>
    </x:XData>
</XmlDataProvider>
```

XmlCustomerOrders

The XmlCustomerOrders example program shown in Figure I-1 uses bound `ListBoxes` to show different views of the data.

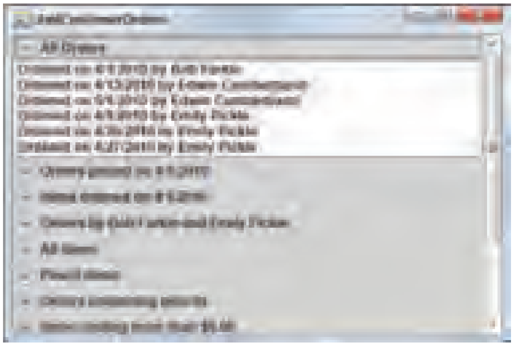


FIGURE I-1

BINDING TO XML DATA

Generally an item bound to XML data needs an item source that tells it where to get its data items and an item template that tells it how to display the items it selects. (See Chapter 18 for more detailed information on data binding.)

For example, the following code shows how the XmlCustomerOrders program displays the data shown in Figure I-1:

Available for download on Wrox.com

```
<ListBox>
    <ListBox.ItemsSource>
        <Binding Source="{StaticResource customers}" XPath="//Order"/>
    </ListBox.ItemsSource>
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="Ordered on "/>
                <TextBlock Text="{Binding XPath=@OrderDate}" FontWeight="Bold"/>
                <TextBlock Text=" by "/>
                <TextBlock Text="{Binding XPath=../@CustName}" FontWeight="Bold"/>
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

XmlCustomerOrders

The ListBox's ItemsSource property tells the control to get its data from the StaticResource named customers, which is the XmlDataProvider. The XPath value "//Order" tells the control to select all Order items from the data.

The ListBox's ItemTemplate tells the control how to display the Order objects that it selects. This template holds a StackPanel that contains several TextBlocks. The first TextBlock contains a simple string that says "Ordered on."

The second TextBlock uses a binding to extract data from the selected Order object. In this case, "@OrderDate" means to take the object's OrderDate attribute.

The third TextBlock displays the text "by." The final TextBlock uses the binding "../@CustName" to select the CustName attribute from the Order's parent node, which is a Customer.

The following sections describe the XPath used in these two ways: to select data objects and to determine what is displayed for the objects.

SELECTION

To select nodes in XML data, you build a path that is somewhat similar to a directory path leading through the data starting at the root node. For example, in the previous data, the path */Customers/Customer/Order/Item* means to follow the path from the root element Customers, enter Customer objects, enter Order objects, and finally select Item objects.

Table I-1 summarizes XPath selection symbols.

TABLE I-1: XPath Selection Symbols

SYMBOL	PURPOSE
`node_name`	Selects child nodes with the given name.
`/`	At the beginning of an XPath, selects the root node. In the middle of an XPath, separates node names.
`//`	Matches any number of nodes. For example, `//Item` matches all `Item` nodes no matter where they are in the data.
`.`	Matches the current node.
`..`	Moves to the parent node.
`@`	Selects an attribute.
`*`	Matches any single node.
`@*`	Matches any single attribute.

PREDICATES

Predicates are placed in square brackets as part of an XPath to restrict a selection. Predicates often refer to node values or attributes.

For example, the following XPath matches all `Item` nodes with a `UnitPrice` attribute greater than 5.00.

```
//Item[@UnitPrice>5.00]
```

You can combine more than one selection statement by using the | symbol. For example, the following binding selects `Order` objects where the parent node is a `Customer` with `CustName` attribute equal to either `'Bob Farkle'` or `'Emily Pickle'`. (Notice that white space is ignored, so this example uses extra spaces in the XML data to make things line up nicely.)

```
<Binding Source="{StaticResource customers}"
 XPath="//Customer[@CustName='Bob Farkle']/Order |
        //Customer[@CustName='Emily Pickle']/Order"/>
```

CONSTRAINT FUNCTIONS

XPath provides several functions that you can use in predicates to restrict the results. For example, the `contains` function selects nodes if the first parameter contains the second parameter as a substring. The following XPath statement selects `Item` nodes where the item's value (specified by the . symbol) contains the character *p*:

```
XPath="//Item[contains(.,'p')]"/>
```

Table I-2 summarizes the most useful constraint functions.

TABLE I-2:

FUNCTION	PURPOSE
`ceiling`	Returns the smallest integer greater than or equal to the input parameter.
`concat`	Concatenates two or more strings.
`contains`	Returns `True` if the first parameter contains the second as a substring. Note that the comparison is case-sensitive.
`count`	Returns a count of the parameter. For example, the XPath statement `//Order[count(Item)>2]` selects an `Order` node if it has more than two children named `Item`.
`floor`	Returns the largest integer less than or equal to the input parameter.
`name`	Returns the current node's name. For example, the XPath expression `//*[contains(name(),'Person')]` selects all nodes with node names that contain *Person*.
`normalize-space`	Removes leading and trailing white space from a string.
`position`	Returns a node's position within its parent's list of children. For example, the XPath expression `//Order[position()=1]` returns `Order` nodes that are the first `Order` inside their `Customer` node.
`round`	Rounds a number to the nearest integer.
`string-length`	Returns a string's length.
`substring`	Returns part of a string. The parameters are the original string, the 1-based start character, and the number of characters.
`substring-after`	Returns the substring that comes after some other specified substring.
`substring-before`	Returns the substring that comes before some other specified substring.
`text`	Selects a text node. For example, the values of the `Item` nodes in the XML code shown earlier in this appendix are stored in text nodes.
`translate`	Searches the first parameter for characters in the second parameter and replaces them with the corresponding characters in the third parameter. For example, the expression `translate("confusing", "aeiou", "AEIOU")` returns `'cOnfUsIng'`.

Selection paths also allow you to use arithmetic and comparison operators. For example, the following XPath expression selects `Item` nodes where the `UnitPrice` attribute is greater than 5.00.

```
//Item[@UnitPrice>5.00]
```

The following expression selects `Item` nodes where the product of the `Quantity` and `UnitPrice` attributes is greater than 5:

```
//Item[@Quantity * @UnitPrice > 5]
```

The following section describes some example XPath expressions used by the XmlCustomerOrders program to select nodes from the XAML data shown in the "XML in XAML" section earlier in this appendix. The section after that describes XAML code that the program uses to display data starting from the selected nodes.

Selection Expressions

These expressions select nodes from the XML code. The XmlCustomerOrders program uses them for `ListBox` `ItemsSource` properties as in the following example:

```
<ListBox.ItemsSource>
    <Binding Source="{StaticResource customers}"
     XPath="//Order"/>
</ListBox.ItemsSource>
```

The following list summarizes the selection XPath expressions used by the example program:

- `//Order` — All `Order` nodes
- `/Customers/Customer/Order` — `Order` nodes that are children of `Customer` nodes that are children of `Customers` nodes that are children of the data root. (In this data, the result is the same as `//Orders`.)
- `//Order[@OrderDate='4/1/2010']` — All `Order` nodes that have an `OrderDate` attribute with value `4/1/2010` — in other words, orders placed on 4/1/2010
- `//Order[@OrderDate='4/1/2010']//Item` — `Item` nodes that are children of `Order` nodes that have an `OrderDate` attribute with value `4/1/2010` — in other words, the `Items` that make up `Orders` placed on 4/1/2010
- `//Customer[@CustName='Bob Farkle']/Order |`
  `//Customer[@CustName='Emily Pickle']/Order` — `Order` nodes that are children of `Customer` nodes that have `CustName` attribute `'Bob Farkle'` plus `Order` nodes that are children of `Customer` nodes that have `CustName` attribute `'Emily Pickle'` — in other words, orders placed by Bob or Emily
- `//Item` — All `Item` nodes
- `//Item[.='Pencils']` — `Item` nodes that have the value `'Pencils'`. The value of a node is the text sitting between its opening and closing elements — in other words, all `Pencils` `Items`. Contrast this syntax with the syntax used to select attribute values, which includes an `@` symbol.
- `//Order[Item='Pencils']"` — `Order` nodes that have at least one `Item` with the value `'Pencils'` — in other words, orders that include pencils
- `//Item[@UnitPrice>5.00]` — `Item` nodes with `UnitPrice` attribute greater than 5.00

- `//Item[@Quantity * @UnitPrice > 5]` — `Item` nodes where the `Quantity` attribute times the `UnitPrice` attribute is greater than 5 — in other words, `Items` where the total price is greater than 5

> *Arithmetic expressions work for selecting nodes but don't work for displaying values. For example, you cannot bind a `TextBlock`'s `Text` property to the result of an arithmetic expression. You can work around this with type converters, but I don't know of a pure XAML solution.*

- `//Item[floor(@Quantity * @UnitPrice) > 8]` — `Item` nodes where the `floor` of the `Quantity` attribute times the `UnitPrice` attribute is greater than 8. (Recall that `floor` returns the largest integer greater than or equal to the input.)
- `//Item[contains(translate(.,'P','p'),'p')]` — `Item` nodes with values that contain `p` or `P`. The call to `translate` takes the current node's value (specified by the ".") and replaces `'P'` with `'p'`. The call to `contains` then determines whether the result contains `p`. The result includes `Items` with values such as Pencils, Napkins, and Duct Tape.
- `//Order[count(Item)>2]` — `Order` nodes that contain more than two `Item` nodes

Display Expressions

These expressions find data to display starting from a particular node. For example, the following code assumes that an `Order` node is currently selected. The first binding selects the node's `OrderDate` attribute, and the second binding selects the node's parent's `CustName` attribute. (The node's parent is a `Customer` in this example.)

```
<ListBox.ItemTemplate>
    <DataTemplate>
        <StackPanel Orientation="Horizontal">
            <TextBlock Text="Ordered on "/>
            <TextBlock Text="{Binding XPath=@OrderDate}" FontWeight="Bold"/>
            <TextBlock Text=" by "/>
            <TextBlock Text="{Binding XPath=../@CustName}" FontWeight="Bold"/>
        </StackPanel>
    </DataTemplate>
</ListBox.ItemTemplate>
```

The following sections include lists that summarize the display XPath expressions used by the XmlCustomerOrders example program while different types of nodes are currently selected. For example, the previous code assumes that an `Order` node is selected.

Order Node Selected

- `@OrderDate` — The `Order`'s `OrderDate` attribute
- `../@CustName` — The `Order`'s parent's `CustName` attribute. In this data, the `Order`'s parent is a `Customer` node.

Item Node Selected

- `@Quantity` — The `Item`'s `Quantity` attribute
- `@UnitPrice` — The `Item`'s `UnitPrice` attribute
- `.` — The `Item`'s value
- `../@OrderDate` — The `Item`'s parent's `OrderDate` attribute. In this data, the `Item`'s parent is an `Order` node.
- `../../@CustName` — The `Item`'s grandparent's `CustName` attribute. In this data, the `Item`'s grandparent is a `Customer` node.

J

Data Binding

This appendix briefly summarizes useful data-binding techniques. You can use it to refresh your memory about specific data-binding scenarios. For more information on data binding, see Chapter 18.

BINDING COMPONENTS

Data bindings have these four basic pieces:

- **Target** — The object that will use the result of the binding
- **Target Property** — The target object's property that will use the result
- **Source** — The object that provides a value for the target object to use
- **Path** — A path that locates the value within the source object

BINDING TO ELEMENTS BY NAME

The following code binds a `Label` to a `TextBox` so it displays whatever you type in the `TextBox`. This version places a separate `Binding` object inside the `TextBlock` to define its contents.

```
<TextBox Name="txtTypeHere" Margin="5" Height="30"
 VerticalAlignment="Top" Text="Type here!"/>

<Label Margin="5" BorderBrush="Yellow" BorderThickness="1">
    <Binding ElementName="txtTypeHere" Path="Text"/>
</Label>
```

The following code makes a similar binding but using an attribute:

```
<TextBox Name="txtTypeHere" Margin="5" Height="30"
 VerticalAlignment="Top" Text="Type here!"/>

<Label Margin="5" BorderBrush="Yellow" BorderThickness="1"
 Content="{Binding ElementName=txtTypeHere, Path=Text}"/>
```

BINDING TO RELATIVESOURCE

The binding's `RelativeSource` property lets you specify a source object via its relationship to the target control. Sometimes this is useful for binding two properties to each other for the same control.

For example, the following code binds a `TextBox`'s `Background` property to its `Text` property. When you type the name of a color in the `TextBox`, the control uses that color for its `Background`.

```
<TextBox Margin="10" Height="30" VerticalAlignment="Top"
 Background="{Binding RelativeSource={RelativeSource Self}, Path=Text}"/>
```

A relative source can also be `TemplatedParent` (the object to which a template is being applied), `FindAncestor` (an ancestor that contains the control), or `PreviousData` (refers to a previous item in a `ListBox` or other control that displays multiple values).

The following code makes a `Label` display the width of the third column of the grid that contains it. (The columns are numbered starting with 0, so `"[2]"` means the third column.)

```
<Label Grid.Column="2"
 Content="{Binding RelativeSource={
    RelativeSource FindAncestor, AncestorType={x:Type Grid}},
    Path=ColumnDefinitions[2].Width}"/>
```

BINDING TO CLASSES IN CODE-BEHIND

Suppose the PersonList program defines a `Person` class that has `FirstName` and `LastName` properties. (Note that the class must have an empty constructor for XAML to be able to create instances.)

The following namespace declaration inside the main `Window` element allows the XAML code to refer to the `Person` class and other classes defined in the PersonList program:

```
xmlns:local="clr-namespace:PersonList"
```

Now the XAML code can define a `Person` object like this:

```
<Window.Resources>
    <local:Person x:Key="perAuthor" FirstName="Rod" LastName="Stephens"/>
<Window.Resources>
```

The XAML code can then bind to this object's properties like this:

```
<Label Content="{Binding Source={StaticResource perAuthor}, Path=FirstName}"/>
<Label Content="{Binding Source={StaticResource perAuthor}, Path=LastName}"/>
```

You can make displaying the entire name easier by overriding the class's default `ToString` method. The following version returns the concatenated first and last names:

```
public override string ToString()
{
    return FirstName + " " + LastName;
}
```

Now if XAML code refers to the object without a `Path` parameter, the binding returns the object's default value, which is the string returned by `ToString`.

```
<Label Content="{Binding Source={StaticResource perAuthor}}">
```

BINDING TO CLASSES IN XAML CODE

The previous section described a scenario in which XAML code creates and binds to a `Person` object. This section describes the scenario where the code binds to a `Person` object that is created in code-behind.

Suppose the PersonSource program defines a `Person` class with `FirstName` and `LastName` properties, and an overridden `ToString` method as described in the previous section.

Now suppose you want the program's code-behind to create a `Person` object and you want the XAML code to bind to it.

In the code-behind, add code to create the object, and create a property that returns the object similar to the following:

```
private Person m_ThePerson = new Person {FirstName = "Rod", LastName = "Stephens"};
public Person ThePerson
{
    get { return m_ThePerson; }
    set { m_ThePerson = value; }
}
```

In the XAML code, add a namespace declaration that refers to the local program namespace:

```
xmlns:local="clr-namespace:PersonList"
```

Note that the main `Window` element also specifies the `Window`'s name by using a `Name` or `x:Name` attribute. Now the XAML code can use that element name to refer to the window that is running the code. It can then refer to that window's property that returns the `Person` object.

The following code binds a `Label` to the object returned by `TheWindow`'s `ThePerson` property:

```
<Label Content="{Binding ElementName=TheWindow, Path=ThePerson}"/>
```

The following code binds a `Label` to the same object's `FirstName` property:

```
<Label Content="{Binding ElementName=TheWindow, Path=ThePerson.FirstName}"/>
```

MAKING COLLECTIONS OF DATA

Since item controls like `ListBox`, `ComboBox`, and `TreeView` display groups of items, it makes sense that any data bound to them should contain groups of values.

The following sections explain how you can bind controls to collections defined in XAML code and in code-behind.

Collections in XAML Code

To use XAML code to make a collection that holds simple system-defined data types such as strings or integers, add the following namespace declaration to the main `Window` element:

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

Now you can use the `x:Array` element to create an array of objects that use the system data types. The following code defines an array of strings called `names`:

```
<Window.Resources>
    <x:Array x:Key="names" Type="sys:String">
        <sys:String>Brain</sys:String>
        <sys:String>Yakko</sys:String>
        <sys:String>Dot</sys:String>
        <sys:String>Wakko</sys:String>
        <sys:String>Pinky</sys:String>
    </x:Array>
</Window.Resources>
```

Now you can bind the array to an item control. The following statement makes a `ListBox` that takes its items from the `names` array:

```
<ListBox ItemsSource="{StaticResource names}"/>
```

Collections in Code-Behind

Suppose that the PersonSource program defines a `Person` class with `FirstName` and `LastName` properties and an overridden `ToString` method that returns the concatenated first and last names as described in the earlier sections.

Now suppose that you want the program's code-behind to create an array of `Person` objects and you want the XAML code to bind a `ListBox` to it.

In the code-behind, add code to create the array and create a property that returns that array. For example, the following code defines an array of `Person` objects, and the property `ThePeople` returns the array:

```
private Person[] m_ThePeople = new Person[] {
    new Person {FirstName="Ann", LastName="Archer"},
    new Person {FirstName="Bob", LastName="Baker"},
    new Person {FirstName="Cat", LastName="Carter"},
    new Person {FirstName="Dan", LastName="Duster"}
};
public Person[] ThePeople
{
    get { return m_ThePeople; }
    set { m_ThePeople = value; }
}
```

In the XAML code, add a namespace declaration that refers to the local program namespace:

```
xmlns:local="clr-namespace:PersonList"
```

Note that the main `Window` element also specifies the window's name by using a `Name` or `x:Name` attribute. The XAML code can use that element name to refer to the window that is running the code. It can then refer to that `Window`'s property that returns the array of `Person` objects.

The following code binds a `ListBox` to the array returned by the `Window`'s `ThePeople` property.

```
<ListBox ItemsSource="{Binding ElementName=TheWindow, Path=ThePeople}"/>
```

USING LISTBOX AND COMBOBOX TEMPLATES

The previous two sections explained how to create arrays of items and gave examples that bind data to `ListBoxes`. Binding `ComboBoxes` is similar. See those sections for more information on basic binding.

These controls also let you specify a `DataTemplate` that determines how the control displays each item.

Create an `ItemTemplate` element. Inside that, place a `DataTemplate` element containing the controls that you want to use to display each data item. Bind the controls to the `ListBox`'s (or `ComboBox`'s) data context.

The following code binds a `ListBox` to the array named `ThePeople` described in the previous section. The `DataTemplate` makes the control display each item in two `TextBlocks` separated by a third containing a space.

```
<ListBox ItemsSource="{Binding ElementName=TheWindow, Path=ThePeople}">
    <ListBox.ItemTemplate>
        <DataTemplate>
            <StackPanel Orientation="Horizontal">
                <TextBlock Text="{Binding FirstName}" FontWeight="Normal"/>
                <TextBlock Text=" "/>
                <TextBlock Text="{Binding LastName}" FontWeight="Bold"/>
            </StackPanel>
        </DataTemplate>
    </ListBox.ItemTemplate>
</ListBox>
```

The code for `ComboBoxes` is similar.

USING TREEVIEW TEMPLATES

To make a `TreeView` display hierarchical data, you must perform three tasks:

- Build the hierarchical classes.
- Make the hierarchical data.
- Define `HierarchicalDataTemplate` elements.
- Attach the `TreeView` to the data.

The hierarchical classes define the data. They should provide properties that return collections of objects at the next level of the hierarchy. For example, a `Department` class representing a corporate department might have `Managers` and `Projects` properties that return lists of `Manager` and `Project` objects, respectively.

To build the hierarchical data, create instances of the classes. Initialize their collection properties so they properly represent the hierarchy.

To tell the `TreeView` how to display data and how to follow the hierarchy, place one or more `HierarchicalDataTemplate` elements inside the control's Resources section. The `DataType` attribute tells what kind of object the template handles. The `ItemsSource` attribute tells which property the `TreeView` should use to descend farther into the data hierarchy after it is done with the current item. The `HierarchicalDataTemplate`'s content should define the values that the control should display for the current object. Typically, this includes `Label`, `TextBlock`, and other controls with content bound to the current hierarchical object.

The following code snippet shows a `HierarchicalDataTemplate` that handles `Department` objects. The `TextBlock` displays the current `Department` object's `Name` property. The element's `ItemsSource` attribute makes the `TreeView` visit the `Department`'s `Managers` property next.

```
<HierarchicalDataTemplate
 DataType="{x:Type local:Department}"
 ItemsSource="{Binding Path=Managers}">
    <TextBlock Text="{Binding Path=Name}" Foreground="Blue"/>
</HierarchicalDataTemplate>
```

There are a couple of ways you can attach the `TreeView` to the hierarchical data. First, you can make the control's `ItemsSource` refer to a property defined by the form. That property can return a collection holding the top-level data items.

The following XAML code attaches the `TreeView` to the `MainWindow` object's `TopLevelData` property. For more information on binding to properties defined by the `Window`, see the section, "Collections in Code-Behind," earlier in this appendix.

```
<TreeView ItemsSource="{Binding ElementName=MainWindow, Path=TopLevelData}">
```

A second approach is to make the code-behind bind the `TreeView` to the data. The following code snippet binds the `TreeView` named `trvParts` to the list named parts:

```
trvParts.ItemsSource = parts;
```

For more information on binding `TreeView` controls to data, see the section, "TreeView Templates," in Chapter 18.

BINDING TO XML DATA

See Chapter 18 and Appendix I for information about binding to XML data.

K

Commanding Classes

This appendix summarizes predefined commanding classes. WPF provides these objects as static methods in the five classes described in the following sections. For more information about commanding, see Chapter 19.

APPLICATIONCOMMANDS

The `ApplicationCommands` class provides commands that apply at the application level. Table K-1 shows the commands and their keyboard gestures (or "N/A" for commands without gestures).

TABLE K-1: ApplicationCommand Commands

COMMAND	GESTURE
`CancelPrint`	N/A
`Close`	[Ctrl]+X or [Shift]+[Delete]
`ContextMenu`	[Shift]+[F10]
`Copy`	[Ctrl]+C, [Shift]+[Delete]
`CorrectionList`	N/A
`Cut`	[Ctrl]+X
`Delete`	[Delete]
`Find`	[Ctrl]+F
`Help`	[F1]
`New`	[Ctrl]+N
`NotACommand`	N/A
`Open`	[Ctrl]+O

continues

TABLE K-1 *(continued)*

COMMAND	GESTURE
`Paste`	[Ctrl]+V
`Print`	[Ctrl]+P
`PrintPreview`	[Ctrl]+[F2]
`Properties`	[F4]
`Redo`	[Ctrl]+Y
`Replace`	[Ctrl]+H
`Save`	[Ctrl]+S
`SaveAs`	[Ctrl]+H
`SelectAll`	[Ctrl]+A
`Stop`	[Escape]
`Undo`	[Ctrl]+Z

COMPONENTCOMMANDS

The `ComponentCommands` class defines commands that make sense for many kind of components. Many of these deal with changing focus or selection. Table K-2 shows the commands and their keyboard gestures.

TABLE K-2: ComponentCommands Commands

COMMAND	GESTURE
`ExtendSelectionDown`	[Shift]+Down
`ExtendSelectionLeft`	[Shift]+Left
`ExtendSelectionRight`	[Shift]+Right
`ExtendSelectionUp`	[Shift]+Up
`MoveDown`	Down
`MoveFocusBack`	[Ctrl]+Left
`MoveFocusDown`	[Ctrl]+Down
`MoveFocusForward`	[Ctrl]+Right
`MoveFocusPageDown`	[Ctrl]+[PageDown]

COMMAND	GESTURE
`MoveFocusPageUp`	[Ctrl]+[PageUp]
`MoveFocusUp`	[Ctrl]+Up
`MoveLeft`	Left
`MoveRight`	Right
`MoveToEnd`	[End]
`MoveToHome`	[Home]
`MoveToPageDown`	[PageDown]
`MoveToPageUp`	[PageUp]
`MoveUp`	Up
`ScrollByLine`	N/A
`ScrollPageDown`	[PageDown]
`ScrollPageLeft`	N/A
`ScrollPageRight`	N/A
`ScrollPageUp`	[PageUp]
`SelectToEnd`	[Shift]+[End]
`SelectToHome`	[Shift]+[Home]
`SelectToPageDown`	[Shift]+[PageDown]
`SelectToPageUp`	[Shift]+[PageUp]

EDITING COMMANDS

The `EditingCommands` class defines commands for use by editing applications. These make the most sense for the text editing applications that inspired them, but you can apply some of them to other types of editing applications such as drawing programs. Table K-3 shows the commands and their keyboard gestures.

TABLE K-3: EditingCommands Commands

COMMAND	GESTURE
`AlignCenter`	[Ctrl]+E
`AlignJustify`	[Ctrl]+J

continues

TABLE K-3 *(continued)*

COMMAND	GESTURE
AlignLeft	[Ctrl]+L
AlignRight	[Ctrl]+R
Backspace	[Backspace]
CorrectSpellingError	N/A
DecreaseFontSize	[Ctrl]+OemOpenBracket
DecreaseIndentation	[Ctrl]+[Shift]+T
Delete	[Delete]
DeleteNextWord	[Ctrl]+[Delete]
DeletePreviousWord	[Ctrl]+[Backspace]
EnterLineBreak	[Shift]+[Enter]
EnterParagraphBreak	[Enter]
IgnoreSpellingError	N/A
IncreaseFontSize	[Ctrl]+OemCloseBracket
IncreaseIndentation	[Ctrl]+T
MoveDownByLine	Down
MoveDownByPage	[PageDown]
MoveDownByParagraph	[Ctrl]+Down
MoveLeftByCharacter	Left
MoveLeftByWord	[Ctrl]+Left
MoveRightByCharacter	Right
MoveRightByWord	[Ctrl]+Right
MoveToDocumentEnd	[Ctrl]+[End]
MoveToDocumentStart	[Ctrl]+[Home]
MoveToLineEnd	[End]
MoveToLineStart	[Home]
MoveUpByLine	Up

COMMAND	GESTURE
MoveUpByPage	[PageUp]
MoveUpByParagraph	[Ctrl]+Up
SelectDownByLine	[Shift]+Down
SelectDownByPage	[Shift]+[PageDown]
SelectDownByParagraph	[Ctrl]+[Shift]+Down
SelectLeftByParagraph	[Shift]+Left
SelectLeftByWord	[Ctrl]+[Shift]+Left
SelectRightByParagraph	[Shift]+Right
SelectRightByWord	[Ctrl]+[Shift]+Right
SelectToDocumentEnd	[Ctrl]+[Shift]+[End]
SelectToDocumentStart	[Ctrl]+[Shift]+[Home]
SelectToLineEnd	[Shift]+[End]
SelectToLineStart	[Shift]+[Home]
SelectUpByLine	[Shift]+Up
SelectUpByPage	[Shift]+[PageUp]
SelectUpByParagraph	[Ctrl]+[Shift]+Up
TabBackward	[Shift]+[Tab]
TabForward	[Tab]
ToggleBold	[Ctrl]+B
ToggleBullets	[Ctrl]+[Shift]+L
ToggleInsert	[Insert]
ToggleItalic	[Ctrl]+I
ToggleNumbering	[Ctrl]+[Shift]+N
ToggleSubscript	[Ctrl]+OemPlus
ToggleSuperscript	[Ctrl]+[Shift]+OemPlus
ToggleUnderline	[Ctrl]+U

MEDIACOMMANDS

The `MediaCommands` class defines commands that are appropriate for different kinds of media such as audio and video. None of these commands have predefined gestures. Table K-4 shows the commands.

TABLE K-4: MediaCommands Commands

COMMAND
`BoostBass`
`ChannelDown`
`ChannelUp`
`DecreaseBass`
`DecreaseMicrophoneVolume`
`DecreaseTreble`
`DecreaseVolume`
`FastForward`
`IncreaseBass`
`IncreaseMicrophoneVolume`
`IncreaseTreble`
`IncreaseVolume`
`MuteMicrophoneVolume`
`MuteVolume`
`NextTrack`
`Pause`
`Play`
`PreviousTrack`
`Record`
`Rewind`
`Select`
`Stop`
`ToggleMicrophoneOnOff`
`TogglePlayPause`

NAVIGATIONCOMMANDS

The `NavigationCommands` class defines commands that make sense to applications that follow a Web-like navigation model. Table K-5 shows the commands and their keyboard gestures.

TABLE K-5:

COMMAND	GESTURE
`BrowseBack`	[Alt]+Left
`BrowseForward`	[Alt]+Right
`BrowseHome`	[Alt]+[Home]
`BrowseStop`	[Alt]+[Escape]
`DecreaseZoom`	N/A
`Favorites`	[Ctrl]+I
`FirstPage`	N/A
`GoToPage`	N/A
`IncreaseZoom`	N/A
`LastPage`	N/A
`NavigateJournal`	N/A
`NextPage`	N/A
`PreviousPage`	N/A
`Refresh`	[F5]
`Search`	[F3]
`Zoom`	N/A

COMMANDS IN XAML

To use a command in XAML code, simply set a control's `Command` property to the appropriate object. For example, the following code defines several items in an Edit menu and sets their `Command` properties:

```
<MenuItem Header="Edit">
    <!--  Menu items that invoke default commands.  -->
    <MenuItem Command="ApplicationCommands.Copy"/>
    <MenuItem Command="ApplicationCommands.Cut"/>
    <MenuItem Command="ApplicationCommands.Paste"/>
    <Separator/>
    <MenuItem Command="ApplicationCommands.Undo"/>
```

```
        <MenuItem Command="ApplicationCommands.Redo"/>
    </MenuItem>
```

Use the `CommandTarget` property to make a command apply to a specific control. The following code makes a `Button` that applies the `ApplicationCommands.Cut` command to the `TextBox` named `txtTop`:

```
<Button Grid.Row="0" Grid.Column="1" Margin="2"
 Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"
 Command="ApplicationCommands.Cut"
 CommandTarget="{Binding ElementName=txtTop}"/>
```

COMMANDS IN CODE-BEHIND

Some commands, such as the editing commands, have predefined behaviors. For other commands, you need to define behaviors.

The following code shows how a program can define command bindings. These tie commands like `New` to the code-behind event handlers that the command needs to determine when it is available and what actions it should perform.

```
<Window.CommandBindings>
    <CommandBinding Command="New"
     CanExecute="DocumentNewAllowed" Executed="DocumentNew"/>
    <CommandBinding Command="Open"
     CanExecute="DocumentOpenAllowed" Executed="DocumentOpen"/>
    <CommandBinding Command="Save"
     CanExecute="DocumentSaveAllowed" Executed="DocumentSave"/>
    <CommandBinding Command="SaveAs"
     CanExecute="DocumentSaveAsAllowed" Executed="DocumentSaveAs"/>
    <!--  The Close binding is defined in code
          just to show how it's done.
    <CommandBinding Command="Close"
     CanExecute="DocumentCloseAllowed" Executed="DocumentClose"/>
     -->
</Window.CommandBindings>
```

The following C# code shows the code-behind for the `New` command:

```
// Return True if we can perform the New action.
private void DocumentNewAllowed(Object sender, CanExecuteRoutedEventArgs e)
{
    e.CanExecute = (!m_IsDirty);
}

// Perform the New action.
private void DocumentNew(Object sender, ExecutedRoutedEventArgs e)
{
    // Remove any existing controls on the Canvas.
    cvsDrawing.Children.Clear();

    // Display the canvas.
    borDrawing.Visibility = Visibility.Visible;
    m_FileName = null;
}
```

L

Bitmap Effects

Table L-1 summarizes WPF's bitmap effects.

TABLE L-1:

EFFECT CLASS	RESULT
`BevelBitmapEffect`	Beveled appearance
`BlurBitmapEffect`	Blurred appearance
`DropShadowBitmapEffect`	Drop shadow behind the object
`EmbossBitmapEffect`	Embossed appearance
`OuterGlowBitmapEffect`	Glow behind the object

The BitmapEffects example program shown in Figure L-1 demonstrates the bitmap effect classes.

FIGURE L-1

The following XAML code shows how the program BitmapEffects displays drop shadows. The other bitmap effect classes work similarly.

```
<StackPanel>
    <Image Source="Volleyball.png" Stretch="Uniform" Height="90">
        <Image.BitmapEffect>
            <DropShadowBitmapEffect/>
        </Image.BitmapEffect>
    </Image>
    <Label Content="Drop Shadow">
        <Label.BitmapEffect>
            <DropShadowBitmapEffect/>
        </Label.BitmapEffect>
    </Label>
</StackPanel>
```

M

Styles

A Style lets you define a package of properties that you can later assign as a group to make controls more consistent and to make the code easier to read.

You can define Styles in a control's Resources section. If you are going to share a Style, it is most convenient to place it in a container's Resources section.

Typically a Style contains Setter and EventSetter objects that define specific property values and event handlers for the controls that use the Style.

You can make two kinds of Styles: named and unnamed.

NAMED STYLES

To make a named Style, give it an x:Name attribute. Later you can refer to the Style as a StaticResource.

The following XAML code defines and uses a Style named styButton:

```
<StackPanel Orientation="Horizontal">
    <StackPanel.Resources>
        <Style x:Key="styButton" TargetType="Button">
            <Setter Property="Width" Value="100"/>
            <Setter Property="Height" Value="30"/>
            <Setter Property="Margin" Value="5"/>
            <Setter Property="FontSize" Value="14"/>
            <Setter Property="FontWeight" Value="Bold"/>
            <EventSetter Event="Click" Handler="MenuButton_Clicked"/>
        </Style>
    </StackPanel.Resources>

    <Button Style="{StaticResource styButton}" Content="Customers"
     Width="150"/>
    <Button Style="{StaticResource styButton}" Content="Orders"/>
    <Button Style="{StaticResource styButton}" Content="Inventory"/>
    <Button Content="Maintenance" Width="75" Height="25"
     Click="Maintenance_Clicked"/>
</StackPanel>
```

The Style defines the Button properties Width, Height, Margin, FontSize, and FontWeight. It also makes the Click event of any Button that uses the Style invoke the MenuButton_Clicked code-behind routine.

The code following the Resources section defines four Buttons. The first three use the Style so they have a consistent appearance and behavior. The first Button overrides the Style's Width property by assigning its own value.

The fourth Button does not use the Style so it gets the default Button appearance. It sets its own Width and Height properties, and Click event handler.

UNNAMED STYLES

To make an unnamed Style, simply omit the x:*Name* attribute. Unnamed Styles apply to all objects of the appropriate target type within their scope. For example, if you define a Button Style in a StackPanel's Resources section, then any Button within the StackPanel uses the Style.

The following code defines an unnamed Style similar to the one described in the previous section inside a StackPanel's Resources section:

```
<StackPanel Orientation="Horizontal">
    <StackPanel.Resources>
        <Style TargetType="Button">
            <Setter Property="Width" Value="100"/>
            <Setter Property="Height" Value="30"/>
            <Setter Property="Margin" Value="5"/>
            <Setter Property="FontSize" Value="14"/>
            <Setter Property="FontWeight" Value="Bold"/>
            <EventSetter Event="Click" Handler="MenuButton_Clicked"/>
        </Style>
    </StackPanel.Resources>

    <Button Content="Customers" Width="150"/>
    <Button Content="Orders"/>
    <Button Content="Inventory"/>
    <Button Content="Maintenance" Style="{x:Null}"
     Width="75" Height="25" Click="Maintenance_Clicked"/>
</StackPanel>
```

The code following the Resources section defines four Buttons. The first three automatically use the unnamed Style so they have a consistent appearance and behavior. The first Button overrides the Style's Width property by assigning its own value.

The fourth Button explicitly sets its Style attribute to {x:Null} so it doesn't use a Style.

TEMPTING TARGETS

In an unnamed `Style`, the `TargetType` attribute is required, but it need not be a specific type of control. For example, you could set `TargetType = Control` to make the `Style` available for any control.

In that case, the `Style` can only specify properties and events that are available to the `Control` class. For example, it could not specify a `Click` event handler because the `Control` class doesn't define a `Click` event.

INHERITED STYLES

You can make one `Style` inherit from another by setting its `BasedOn` attribute. The second `Style` can add new properties or override values defined by the first `Style`.

While a *parent* `Style` must have a name, you can define an unnamed `Style` based on a named `Style`.

The following code creates a `Style` named `styAllButtons`. It then creates another `Style` based on it named `styMainButtons`. Next the code defines an unnamed `Style` based on `styAllButtons` so any `Button` that does not have an explicit `Style` uses `styAllButtons`.

```
<StackPanel Orientation="Horizontal">
    <StackPanel.Resources>
        <Style x:Key="styAllButtons" TargetType="Button">
            <Setter Property="Width" Value="100"/>
            <Setter Property="Height" Value="30"/>
            <Setter Property="Margin" Value="5"/>
        </Style>
        <Style x:Key="styMainButtons" TargetType="Button"
         BasedOn="{StaticResource styAllButtons}">
            <Setter Property="FontSize" Value="14"/>
            <Setter Property="FontWeight" Value="Bold"/>
        </Style>
        <Style TargetType="Button" BasedOn="{StaticResource styAllButtons}"/>
    </StackPanel.Resources>

    <Button Content="Customers" Style="{StaticResource styAllButtons}"/>
    <Button Content="Orders" Style="{StaticResource styMainButtons}"/>
    <Button Content="Inventory"/>
    <Button Content="Maintenance" Style="{x:Null}"
     Width="75" Height="25" Click="Maintenance_Clicked"/>
</StackPanel>
```

N

Templates

This appendix summarizes control templates. For more detailed information, see Chapter 15.

A *control template* defines a control's constituent controls and how they behave. Typically a `ControlTemplate` object contains the controls that make up the template. The `ControlTemplate` can have a Triggers section that determines the control's behavior, and it can have a Resources section to define necessary resources.

The following sections show examples of simple templates for different kinds of controls. You can use these as starting points for your own templates.

LABEL

The following template makes a `Label` control display its contents in a wrapped `TextBlock`. It provides one `Trigger` that grays out the contents and covers the control with translucent gray stripes when `IsEnabled` is `False`.

Available for download on Wrox.com

```
<ControlTemplate x:Key="temWrappedLabel" TargetType="Label">
    <Grid>
        <Border Name="brdMain"
         Background="{TemplateBinding Background}"
         BorderBrush="{TemplateBinding BorderBrush}"
         BorderThickness="{TemplateBinding BorderThickness}">
            <TextBlock Name="txtbContent" Margin="4" TextWrapping="Wrap"
             Text="{TemplateBinding ContentPresenter.Content}"/>
        </Border>
        <Canvas Name="canDisabled" Opacity="0">
            <Canvas.Background>
                <LinearGradientBrush StartPoint="0,0" EndPoint="3,3"
                 MappingMode="Absolute"
                 SpreadMethod="Repeat">
                    <GradientStop Color="LightGray" Offset="0"/>
                    <GradientStop Color="Black" Offset="1"/>
                </LinearGradientBrush>
            </Canvas.Background>
```

```
            </Canvas>
        </Grid>
        <ControlTemplate.Triggers>
            <Trigger Property="IsEnabled" Value="False">
                <Setter TargetName="canDisabled"
                 Property="Opacity" Value="0.5"/>
                <Setter TargetName="txtbContent"
                 Property="Foreground" Value="Gray"/>
            </Trigger>
        </ControlTemplate.Triggers>
    </ControlTemplate>
```

BareBonesLabel

The example BareBonesLabel program demonstrates this template.

CHECKBOX

The following template makes a `CheckBox` control display a different appearance. The `CheckBox` displays a gray `Border` surrounding a second `Border`. The inner `Border` changes background color when the `CheckBox` is checked, unchecked, or in an indeterminate state.

Available for download on Wrox.com

```
    <ControlTemplate x:Key="temCheckBox" TargetType="CheckBox">
        <BulletDecorator>
            <BulletDecorator.Bullet>
                <Grid Height="15" Width="15">
                    <Border CornerRadius="2" BorderBrush="Black" BorderThickness="1"/>
                    <Border Name="brdChecked" Margin="2" CornerRadius="6"/>
                </Grid>
            </BulletDecorator.Bullet>
            <ContentPresenter Margin="5,0,0,0"/>
        </BulletDecorator>
        <ControlTemplate.Triggers>
            <Trigger Property="IsChecked" Value="True">
                <Setter TargetName="brdChecked" Property="Background" Value="Green"/>
            </Trigger>
            <Trigger Property="IsChecked" Value="False">
                <Setter TargetName="brdChecked" Property="Background" Value="Red"/>
            </Trigger>
            <Trigger Property="IsChecked" Value="{x:Null}">
                <Setter TargetName="brdChecked" Property="Background" Value="Gray"/>
            </Trigger>
        </ControlTemplate.Triggers>
    </ControlTemplate>
```

BareBonesCheckBox

You could add other `Triggers` such as one to change the control's appearance when its `IsEnabled` property is `False`.

The BareBonesCheckBox example program demonstrates this template.

RADIOBUTTON

The following template makes a `RadioButton` control display a different appearance. Instead of the usual filled circle, the template displays an X if the control is checked.

Available for download on Wrox.com

```
<ControlTemplate x:Key="temRadioButton" TargetType="RadioButton">
    <BulletDecorator>
        <BulletDecorator.Bullet>
            <Grid Height="15" Width="15">
                <Path Name="pathChecked" Data="M4,11 L11,4 M4,4 L11,11"
                 Stroke="Blue" StrokeThickness="2" Opacity="1"/>
                <Ellipse Stroke="Black" StrokeThickness="1"/>
            </Grid>
        </BulletDecorator.Bullet>
        <ContentPresenter Margin="5,0,0,0"/>
    </BulletDecorator>
    <ControlTemplate.Triggers>
        <Trigger Property="IsChecked" Value="True">
            <Setter TargetName="pathChecked" Property="Stroke" Value="Green"/>
            <Setter TargetName="pathChecked" Property="Opacity" Value="1"/>
        </Trigger>
        <Trigger Property="IsChecked" Value="False">
            <Setter TargetName="pathChecked" Property="Opacity" Value="0"/>
        </Trigger>
        <Trigger Property="IsChecked" Value="{x:Null}">
            <Setter TargetName="pathChecked" Property="Stroke" Value="Gray"/>
            <Setter TargetName="pathChecked" Property="Opacity" Value="1"/>
        </Trigger>
    </ControlTemplate.Triggers>
</ControlTemplate>
```

BareBonesRadioButton

You could add other `Triggers` such as one to change the control's appearance when its `IsEnabled` property is `False`.

The BareBonesRadioButton example program demonstrates this template.

PROGRESSBAR

The following template redefines how a `ProgressBar` works:

Available for download on Wrox.com

```
<ControlTemplate x:Key="temProgressBar" TargetType="ProgressBar">
    <Border BorderBrush="Green" BorderThickness="5">
        <Border Name="PART_Track" Background="Red">
            <Rectangle Name="PART_Indicator" HorizontalAlignment="Left"
             Fill="Blue"/>
        </Border>
    </Border>
</ControlTemplate>
```

BareBonesProgressBar

The template uses a `Border` to draw the control's background track area. The special name `PART_Track` tells the underlying `ProgressBar` what control to use as its background.

The template uses a `Rectangle` control to display the `ProgressBar`'s current value. The special name `PART_indicator` tells the underlying `ProgressBar` what control to use for this.

The `ProgressBar` uses the `PART_Track` and `PART_Indicator` controls to determine how to display progress. It assumes that the indicator completely fills the track when the `Value` property has its maximum value. If that isn't true, then the control may behave oddly.

For example, suppose the track has a 5-pixel border. Then the indicator sits 5 pixels inside the track so it grows to completely fill the track 10 pixels early (5 pixels for each side), before the `Value` property reaches its maximum. It continues to grow beyond that point, but the rest of the indicator isn't visible because it is hidden by the track's border.

The BareBonesProgressBar example program demonstrates this template.

ORIENTED PROGRESSBAR

The template shown in the previous section only works for a `ProgressBar` that is oriented horizontally. The indicator's `HorizontalAlignment` property is set to `Left`, and its `VerticalAlignment` property defaults to `Stretch`, so the indicator fits properly within the track. For a vertical alignment, you would need to change the indicator's `HorizontalAlignment` to `Stretch` and its `VerticalAlignment` to `Bottom`.

To allow the control to handle either orientation, you can use the `ControlTemplate`'s Triggers section to either modify the properties or select different styles.

The following code shows such a Triggers section:

Available for download on Wrox.com

```
<ControlTemplate.Triggers>
    <Trigger Property="Orientation" Value="Horizontal">
        <Setter TargetName="PART_Indicator"
         Property="VerticalAlignment" Value="Stretch"/>
        <Setter TargetName="PART_Indicator"
         Property="HorizontalAlignment" Value="Left"/>
    </Trigger>
    <Trigger Property="Orientation" Value="Vertical">
        <Setter TargetName="PART_Indicator"
         Property="VerticalAlignment" Value="Bottom"/>
        <Setter TargetName="PART_Indicator"
         Property="HorizontalAlignment" Value="Stretch"/>
    </Trigger>
</ControlTemplate.Triggers>
```

OrientedProgressBar

You could add other `Triggers` such as one to change the control's appearance when its `IsEnabled` property is `False`.

The OrientedProgressBar example program demonstrates the template with the Triggers section.

LABELED PROGRESSBAR

The following template changes the `ProgressBar`'s appearance even more than the previous versions. In addition to the track and indicator controls, this version adds a `Label` centered in the track. It binds the `Label`'s `Content` property to the `ProgressBar`'s `Value` property so the `Label` displays the control's `Value`.

Available for download on Wrox.com

```
<ControlTemplate x:Key="temProgressBar" TargetType="ProgressBar">
    <Border BorderBrush="Green" BorderThickness="1">
        <Grid Name="PART_Track" Background="Red">
            <Rectangle Name="PART_Indicator" Fill="Blue"/>
            <Label Foreground="Yellow" FontWeight="Bold"
             HorizontalAlignment="Center" VerticalAlignment="Center"
             ContentStringFormat="0" Content="{TemplateBinding Value}"/>
            <!--  Alternate using a TextBox:
            <TextBox HorizontalAlignment="Left"
             Text="{Binding Path=Value,
                 RelativeSource={RelativeSource TemplatedParent},
             StringFormat=0}"/>
             -->
        </Grid>
    </Border>
    <ControlTemplate.Triggers>
        <Trigger Property="Orientation" Value="Horizontal">
            <Setter TargetName="PART_Indicator"
             Property="VerticalAlignment" Value="Stretch"/>
            <Setter TargetName="PART_Indicator"
             Property="HorizontalAlignment" Value="Left"/>
        </Trigger>
        <Trigger Property="Orientation" Value="Vertical">
            <Setter TargetName="PART_Indicator"
             Property="VerticalAlignment" Value="Bottom"/>
            <Setter TargetName="PART_Indicator"
             Property="HorizontalAlignment" Value="Stretch"/>
        </Trigger>
    </ControlTemplate.Triggers>
</ControlTemplate>
```

LabeledProgressBar

The code also includes a commented `TextBox` that can replace the `Label`. The `Label` is probably a better choice for this template. The `TextBox` code is just included to show how to bind a `TextBox` to the template control's `Value` property.

You could add other `Triggers` such as one to change the control's appearance when its `IsEnabled` property is `False`.

The LabeledProgressBar example program demonstrates this template.

SCROLLBAR

A normal `ScrollBar` displays five pieces:

- A thumb that the user can drag back and forth

- Two long repeat buttons on either side of the thumb. When the user clicks on one of these, the `ScrollBar` adds or subtracts its `LargeChange` value.
- Two smaller repeat buttons on the ends of the control. When the user clicks on one of these, the `ScrollBar` adds or subtracts its `SmallChange` value.

Like the `ProgressBar`, this control has a component with a special name — `PART_Track`. This represents the area containing the `LargeChange` `RepeatButtons` and the thumb.

The `RepeatButtons` can make the control perform the normal scrolling actions by executing these commands:

- `ScrollBar.LineLeftCommand`
- `ScrollBar.PageLeftCommand`
- `ScrollBar.PageRightCommand`
- `ScrollBar.LineRightCommand`
- `ScrollBar.LineUpCommand`
- `ScrollBar.PageUpCommand`
- `ScrollBar.PageDownCommand`
- `ScrollBar.LineDownCommand`

The following code shows the layout portion of a `ScrollBar` template:

Available for download on Wrox.com

```
<ControlTemplate x:Key="temScrollBar" TargetType="ScrollBar">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>

        <RepeatButton Name="btnSmallDown" Grid.Column="0"
         Command="ScrollBar.LineLeftCommand">
            <Path HorizontalAlignment="Center" VerticalAlignment="Center"
             Data="M2,3 L10,3" Stroke="Blue" StrokeThickness="3"
             StrokeStartLineCap="Round" StrokeEndLineCap="Round"/>
        </RepeatButton>
        <Track Grid.Column="1" Name="PART_Track">
            <Track.DecreaseRepeatButton>
                <RepeatButton Name="btnDecrease"
                 Command="ScrollBar.PageLeftCommand" />
            </Track.DecreaseRepeatButton>
            <Track.IncreaseRepeatButton>
                <RepeatButton Name="btnIncrease"
                 Command="ScrollBar.PageRightCommand" />
            </Track.IncreaseRepeatButton>
```

```
            <Track.Thumb>
                <Thumb Background="Blue" />
            </Track.Thumb>
        </Track>
        <RepeatButton Name="btnSmallUp" Grid.Column="2"
         Command="ScrollBar.LineRightCommand">
            <Path HorizontalAlignment="Center" VerticalAlignment="Center"
             Data="M2,3 L10,3 M6,0 L6,7" Stroke="Blue" StrokeThickness="3"
             StrokeStartLineCap="Round" StrokeEndLineCap="Round"/>
        </RepeatButton>
    </Grid>
    <ControlTemplate.Triggers>
        ... Triggers omitted ...
    </ControlTemplate.Triggers>
</ControlTemplate>
```

BareBonesScrollBar

The template's `Triggers` change the locations of the components in the `Grid` depending on the control's `Orientation` property. They also set the `RepeatButtons`' `Command` properties to the `Left/Right` commands or the `Up/Down` commands. To save space, the Triggers section isn't shown here. Download the example program to see the full code.

You could add other `Triggers` such as one to change the control's appearance when its `IsEnabled` property is `False`.

The BareBonesScrollBar example program demonstrates this template.

MODIFIED SCROLLBAR

The `ScrollBar` template described in the previous section makes a `ScrollBar` that looks like it's made up of a bunch of buttons (which it is). The following template gives a scrollbar a more distinctive appearance:

Available for download on Wrox.com

```
<ControlTemplate x:Key="temScrollBar" TargetType="ScrollBar">
    <ControlTemplate.Resources>
        <ControlTemplate x:Key="temSmallChangeButtons" TargetType="RepeatButton">
            <Grid>
                <Ellipse Fill="Blue" MinHeight="12" MinWidth="10"/>
                <ContentPresenter/>
            </Grid>
        </ControlTemplate>
        <ControlTemplate x:Key="temLargeChangeButtonsH" TargetType="RepeatButton">
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="*"/>
                    <RowDefinition Height="2*"/>
                    <RowDefinition Height="*"/>
                </Grid.RowDefinitions>
                <Rectangle Grid.Row="1" RadiusX="10" RadiusY="10"
                 Fill="SkyBlue" Stroke="Blue"/>
                <ContentPresenter/>
            </Grid>
        </ControlTemplate>
```

```
        <ControlTemplate x:Key="temLargeChangeButtonsV" TargetType="RepeatButton">
            <Grid>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="*"/>
                    <ColumnDefinition Width="2*"/>
                    <ColumnDefinition Width="*"/>
                </Grid.ColumnDefinitions>
                <Rectangle Grid.Column="1" RadiusX="10" RadiusY="10"
                 Fill="SkyBlue" Stroke="Blue"/>
                <ContentPresenter/>
            </Grid>
        </ControlTemplate>
        <ControlTemplate x:Key="temThumb" TargetType="Thumb">
            <Ellipse Fill="Blue"/>
        </ControlTemplate>
    </ControlTemplate.Resources>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>

        <RepeatButton Name="btnSmallDown" Grid.Column="0"
         Command="ScrollBar.LineLeftCommand"
         Template="{StaticResource temSmallChangeButtons}">
            <Path Margin="1" HorizontalAlignment="Center" VerticalAlignment="Center"
             Data="M2,3 L10,3" Stroke="Yellow" StrokeThickness="3"
             StrokeStartLineCap="Round" StrokeEndLineCap="Round"/>
        </RepeatButton>
        <Track Grid.Column="1" Name="PART_Track">
            <Track.DecreaseRepeatButton>
                <RepeatButton Name="btnDecrease" Command="ScrollBar.PageLeftCommand"
                 Template="{StaticResource temLargeChangeButtonsH}" />
            </Track.DecreaseRepeatButton>
            <Track.IncreaseRepeatButton>
                <RepeatButton Name="btnIncrease"
                 Command="ScrollBar.PageRightCommand"
                 Template="{StaticResource temLargeChangeButtonsH}" />
            </Track.IncreaseRepeatButton>
            <Track.Thumb>
                <Thumb Background="Blue" Template="{StaticResource temThumb}" />
            </Track.Thumb>
        </Track>
        <RepeatButton Name="btnSmallUp" Grid.Column="2"
         Command="ScrollBar.LineRightCommand"
         Template="{StaticResource temSmallChangeButtons}">
            <Path Margin="1" HorizontalAlignment="Center" VerticalAlignment="Center"
             Data="M2,4 L10,4 M6,0 L6,8" Stroke="Yellow" StrokeThickness="3"
             StrokeStartLineCap="Round" StrokeEndLineCap="Round"/>
        </RepeatButton>
```

```
        </Grid>
        <ControlTemplate.Triggers>
            <Trigger Property="Orientation" Value="Horizontal">
                ... Some Setters omitted ...
                <Setter TargetName="btnDecrease" Property="Template"
                 Value="{StaticResource temLargeChangeButtonsH}"/>
                <Setter TargetName="btnIncrease" Property="Template"
                 Value="{StaticResource temLargeChangeButtonsH}"/>
            </Trigger>
            <Trigger Property="Orientation" Value="Vertical">
                ... Some Setters omitted ...
                <Setter TargetName="btnDecrease" Property="Template"
                 Value="{StaticResource temLargeChangeButtonsV}"/>
                <Setter TargetName="btnIncrease" Property="Template"
                 Value="{StaticResource temLargeChangeButtonsV}"/>
            </Trigger>
        </ControlTemplate.Triggers>
    </ControlTemplate>
```

ModifiedScrollBar

The code starts by defining templates for the `ScrollBar`'s pieces. It then creates the components that make up the `ScrollBar`.

The Triggers section changes properties appropriately for vertical and horizontal orientations. A new technique shown in this example is the way the `Triggers` change the template used by the `LargeChange Buttons`. This lets those `Buttons` use very different layouts depending on the control's orientation.

You could add other `Triggers` such as one to change the control's appearance when its `IsEnabled` property is `False`.

The ModifiedScrollBar example program demonstrates this template.

BUTTON

Many applications customize `Buttons` extensively to give them new shapes, highlights, translucency, and other graphical effects. To keep this code simple, the following template makes its `Button` from a simple `Rectangle` and changes the `Rectangle`'s color to show different `Button` states:

Available for download on Wrox.com

```
<ControlTemplate x:Key="temButton" TargetType="Button">
    <!--  Styles for different states.  -->
    <ControlTemplate.Resources>
        <!--  Style for "normal" status.  -->
        <Style TargetType="Rectangle">
            <Setter Property="Fill" Value="Green"/>
        </Style>
        <Style x:Key="styIsDefaulted" TargetType="Rectangle">
            <Setter Property="Fill" Value="DarkGreen"/>
        </Style>
        <Style x:Key="styDisabled" TargetType="Rectangle">
            <Setter Property="Fill" Value="Gray"/>
        </Style>
        <Style x:Key="styIsMouseOver" TargetType="Rectangle">
```

```
                <Setter Property="Fill" Value="Lime"/>
            </Style>
            <Style x:Key="styIsFocused" TargetType="Rectangle">
                <Setter Property="Fill" Value="White"/>
            </Style>
            <Style x:Key="styIsPressed" TargetType="Rectangle">
                <Setter Property="Fill" Value="Yellow"/>
            </Style>
        </ControlTemplate.Resources>

        <!--  The controls that make up the Button.  -->
        <Grid Name="grdMain" ClipToBounds="True"
         Width="{TemplateBinding Width}"
         Height="{TemplateBinding Height}">
            <Rectangle Name="rectMain"/>
            <ContentPresenter VerticalAlignment="Center" HorizontalAlignment="Center"/>
        </Grid>

        <!--  Behaviors.  -->
        <ControlTemplate.Triggers>
            <Trigger Property="IsMouseOver" Value="True">
                <Setter TargetName="rectMain" Property="Style"
                 Value="{StaticResource styIsMouseOver}"/>
            </Trigger>
            <Trigger Property="IsFocused" Value="True">
                <Setter TargetName="rectMain" Property="Style"
                 Value="{StaticResource styIsFocused}"/>
            </Trigger>
            <Trigger Property="IsDefaulted" Value="True">
                <Setter TargetName="rectMain" Property="Style"
                 Value="{StaticResource styIsDefaulted}"/>
            </Trigger>
            <Trigger Property="IsPressed" Value="True">
                <Setter TargetName="rectMain" Property="Style"
                 Value="{StaticResource styIsPressed}"/>
            </Trigger>
            <Trigger Property="IsEnabled" Value="False">
                <Setter TargetName="rectMain" Property="Style"
                 Value="{StaticResource styDisabled}"/>
            </Trigger>
        </ControlTemplate.Triggers>
    </ControlTemplate>
```

BareBonesButton

The code begins by defining different `Styles` for the `Button`'s different states: normal, defaulted, disabled, mouse over, focused, and pressed.

The template then creates the `Button` from a simple `Rectangle`.

Next, the template includes a series of `Triggers` that handle the `Button`'s different states. Each `Trigger` simply selects the appropriate `Style`.

Keeping all of the details in `Styles` makes the template easier to read than it would be if all of the property settings were contained directly in the template's `Triggers`. In this example, the property-setting code is quite simple so the difference is small. In a more complex example that uses lots of decorative controls and animations, the difference can be significant.

O

Triggers and Animation

This appendix summarizes triggers and animations. For more detailed information, see Chapter 14.

EVENTTRIGGERS

You can place `EventTriggers` inside a control's Triggers section to take action when an event occurs. Inside the `EventTrigger`, you can place actions that should occur when the event takes place.

> **TEMPTING TRIGGERS**
>
> Despite its general-sounding name and the fact that IntelliSense lists other options, the Triggers section can only hold `EventTriggers`, not property `Triggers`.

The most useful of these actions include:

- `PauseStoryboard` — Pauses a Storyboard.
- `ResumeStoryboard` — Resumes a paused Storyboard.
- `SeekStoryboard` — Moves the Storyboard to a specific position in its timeline.
- `StopStoryboard` — Stops a Storyboard. This resets properties to their original values.
- `RemoveStoryboard` — Stops a Storyboard and frees its resources.

The following code gives a `Button` an `EventTrigger`. When the `Button` raises its `Click` event, the `EventTrigger` starts a Storyboard that changes the `Button`'s `Canvas.Left` property to 300 over a period of 0.5 seconds. The animation's `AutoReverse` property is `True`, so after the animation completes, it changes the `Canvas.Left` property back to its original value.

```
<Button Name="btnClickMe" Canvas.Left="10" Canvas.Top="10" Content="Click Me">
    <Button.Triggers>
```

```
        <EventTrigger RoutedEvent="Button.Click">
            <EventTrigger.Actions>
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation Duration="0:0:0.5" To="300"
                         Storyboard.TargetProperty="(Canvas.Left)"
                         AutoReverse="True"/>
                    </Storyboard>
                </BeginStoryboard>
            </EventTrigger.Actions>
        </EventTrigger>
    </Button.Triggers>
</Button>
```

PROPERTY TRIGGERS

While you cannot place event triggers inside a control's Triggers section, you can add them to a `Style` and then apply the `Style` to a control in its Triggers section. The following code defines a `Style` named `styGrowButton`:

```
<Style x:Key="styGrowButton" TargetType="Button">
    <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
            <Setter Property="Width" Value="150"/>
            <Setter Property="Height" Value="75"/>
        </Trigger>
    </Style.Triggers>
</Style>
```

This code's property trigger occurs when the `IsMouseOver` property becomes `True`. As long as that property's value is `True`, the trigger changes the control's `Width` and `Height` values. When the property is no longer `True`, the control's `Width` and `Height` return to their normal values.

The following code creates a `Button` that uses the `styGrowButton` `Style`:

```
<Button Style="{StaticResource styGrowButton}"
 Canvas.Left="10" Canvas.Top="10" Content="Grow Me"/>
```

In addition to `Setters`, a `Trigger` element can contain `Trigger.EnterActions` and `Trigger.ExitActions` sections that can contain `Storyboard` control commands (such as `BeginStoryboard`) to take action when the property gets and loses its target value.

The following code defines a `Style` that makes a `Button` larger when the `IsMouseOver` property becomes `True` and makes it smaller when `IsMouseOver` becomes `False`:

```
<Style x:Key="styGrowActions" TargetType="Button">
    <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
            <Trigger.EnterActions>
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation Duration="0:0:0.25" To="300"
                         Storyboard.TargetProperty="Width"/>
```

```
                    </Storyboard>
                </BeginStoryboard>
            </Trigger.EnterActions>
            <Trigger.ExitActions>
                <BeginStoryboard>
                    <Storyboard>
                        <DoubleAnimation Duration="0:0:0.25" To="100"
                         Storyboard.TargetProperty="Width"/>
                    </Storyboard>
                </BeginStoryboard>
            </Trigger.ExitActions>
        </Trigger>
    </Style.Triggers>
</Style>
```

Unlike event triggers, property triggers do not undo their changes when the property no longer has its target value. In this example, that means the `Button` is left with a `Width` of 100 no matter what its initial `Width` was.

STORYBOARD PROPERTIES

Table O-1 summarizes the most useful `Storyboard` properties.

TABLE O-1: Storyboard Properties

PROPERTY	PURPOSE
`AccelerationRatio`	Determines the percentage of the `Storyboard`'s `Duration` that it spends accelerating to full speed.
`AutoReverse`	Determines whether the `Storyboard` should replay itself backward after it finishes running.
`BeginTime`	Determines when the `Storyboard` starts running animations after it begins. This can be useful to coordinate multiple `Storyboards` running at the same time.
`DecelerationRatio`	Determines the percentage of the `Storyboard`'s `Duration` that it spends decelerating from full speed to a stop.
`Duration`	Determines the `Storyboard`'s duration in days, hours, minutes, and seconds. For example, the value `0.1:23:45.67` means 0 days, 1 hour, 23 minutes, 45.67 seconds.
`RepeatBehavior`	Determines whether the `Storyboard` repeats and how many times. This can be an iteration count (`2x` or `10x`), a duration during which the `Storyboard` should repeat (`0.0:0:2.5`), or the word `Forever`.
`SpeedRatio`	Determines the `Storyboard`'s speed relative to its parent. Values between 0 and 1 make the `Storyboard` slower. Values greater than 1 make the `Storyboard` faster.

ANIMATION CLASSES

WPF provides many animation classes that you can use in Storyboards. For each data type that these classes animate, there are three possible kinds of animation.

A basic animation moves a property from one value to another in a straightforward way. These animation classes have names ending with *Animation*. For example, the basic class that animates the `Double` data type is `DoubleAnimation`.

A key frame animation uses key frames to define points along the animation. These classes have names ending with *AnimationUsingKeyFrames*, as in `DoubleAnimationUsingKeyFrames`.

A path animation uses a `PathGeometry` to specify values during the animation. These classes have names ending with *AnimationUsingPath*, as in `DoubleAnimationUsingPath`.

Table O-2 summarizes WPF's most useful animation classes. The Basic, Key Frame, and Path columns indicated the types of animations that WPF provides for each data type.

TABLE O-2: Animation Classes

DATA TYPE	BASIC	KEY FRAME	PATH
`Boolean`		X	
`Byte`	X		
`Char`		X	
`Color`	X	X	
`Decimal`	X	X	
`Double`	X	X	X
`Int16`	X	X	
`Int32`	X	X	
`Int64`	X	X	
`Matrix`		X	
`Point3D`	X	X	X
`Rect`	X	X	
`Rotation3D`	X	X	
`Single`	X	X	
`Size`	X	X	
`String`		X	

DATA TYPE	BASIC	KEY FRAME	PATH
`Thickness`	X	X	
`Vector`	X	X	
`Vector3D`	X	X	

Key frame animations use a collection of key frames to define the values that the animation should visit and the way in which the values change from one to another. For example, a `LinearDoubleKeyFrame` object makes a `Double` property value vary linearly from its previous value to the one defined by the key frame object.

Table O-3 lists the key frame classes. The name of a particular key frame class starts with the value shown in this table, followed by the data type, followed by *KeyFrame*, as in `LinearDoubleKeyFrame`.

TABLE O-3: Key Frame Classes

KEY FRAME TYPE	PURPOSE
`Discrete`	Makes the property jump to its new value discretely.
`Linear`	Makes the property move linearly to its new value.
`Spline`	Makes the property move smoothly between values using a spline to control the movement's speed. For example, you can use a spline to make the property's change start slowly, speed up in the middle, and then slow to a stop.

P

Index of Example Programs

CHAPTER 1

Chapter 1 contains the following example programs:

- **Clutter** — A cluttered interface with garish colors, too much animation, and annoying sound effects. [Page 2, 3].
- **Clutter_FakeRotation** — A static version of Clutter to simulate rotation. Figure 1-1. [Page 2].
- **Critters** — A simple demonstration of properly nested XAML elements, in this case, a horizontal `StackPanel` holding a vertical `StackPanel`. Figure 1-4. [Page 6].
- **FlowDocument** — Demonstrates many `FlowDocument` elements including tables, controls, shapes, lists, figures, and even a rotating three-dimensional shape. Figure 1-14. [Page 19].
- **Gasket3D** — Uses code-behind to build a complex rotating three-dimensional object. Figure 1-9. [Page 12].
- **GrowingButtons** — Makes buttons that grow and shrink when the mouse moves on and off them. Figure 1-12. [Page 14].
- **Multimedia** — Plays video and audio. Figure 1-6. [Page 11].
- **SetBackgrounds** — Sets background linear gradient brushes in code-behind.
- **SimpleCritters** — A simple demonstration of properly nested XAML elements, in this case, a horizontal `StackPanel` holding a vertical `StackPanel`. Figures 1-2 and 1-3. [Page 6].
- **StackPanelButton** — Displays buttons that contain `StackPanels`. Figure 1-13. [Page 16].

- **Star** — Demonstrates the `Polygon` object. Figure 1-10. [Page 13].
- **TransformedControls** — Displays labels that are rotated 90 degrees. Figure 1-7. [Page 11].
- **TransformedVideo** — Displays video that is rotated, skewed, and stretched. Figure 1-8. [Page 12].
- **Zoom** — Shows the difference between enlarged bitmap images and enlarged vector-based images. Figure 1-11. [Page 13].

CHAPTER 2

Chapter 2 contains the following example program:

- **ImageColors** — Demonstrates different ways to attach buttons to `Click` event handlers in the code-behind. Figure 2-12. [Page 32].

CHAPTER 3

Chapter 3 contains the following example programs:

- **GradientBrushes** — Demonstrates linear gradient and radial gradient brushes. Figure 3-8. [Page 45].
- **MakeDrawingBrush** — Used in a walk-through that creates a drawing brush. Figure 3-17. [Page 50].
- **MakeImageBrush** — Used in a walk-through that creates an image brush. Figures 3-12 through 3-15. [Page 46-47].
- **MissionBriefing** — Used to demonstrate various Blend features. Figures 3-1 through 3-4. [Page 38, 39, 41, 42].
- **ModifiedGradientBrushes** — Used to demonstrate how to modify gradient brushes. Figures 3-9 through 3-11. [Page 45].
- **ReflectingBrush** — Uses visual brushes to simulate reflections of textboxes. Figure 3-19. [Page 52].
- **ResourcesAndStyles** — Demonstrates how to use resources and styles. Figures 3-22 through 3-25. [Page 53-54].
- **SpinningButtons** — Demonstrates a storyboard that make a button spin 360 degrees. Figure 3-28. [Page 56].
- **UseBrushes** — Displays a rectangle and a polygon that use gradient brushes for interiors and lines. Figure 3-5. [Page 43].
- **UseImageBrushes** — Displays image brushes with different viewbox and viewport values. Figure 3-16. [Page 48].
- **UsePens** — Demonstrates a `Polygon`’s `StrokeThickness`, `StrokeMiterLimit`, and `Stroke` properties. Figure 3-21. [Page 52].

- **UseTileBrushes** — Uses image, drawing, and visual brushes. Figure 3-20. [Page 52].
- **UseVisualBrush** — Uses a visual brush. Figure 3-18. [Page 51].

CHAPTER 4

Chapter 4 contains the following example programs:

- **FontProperties** — Demonstrates an assortment of font property values. Figure 4-5. [Page 66].
- **FontWeights** — Demonstrates `FontWeight` values for Segoe and Arial fonts. Figure 4-6. [Page 66].
- **GradientOpacityMask** — Uses a `RadialGradientBrush` as an `OpacityMask` for an `Image`. Figure 4-8. [Page 67].
- **GroupBoxColors** — Demonstrates `Foreground` and `Background` properties in a `GroupBox`. Figure 4-7. [Page 67].
- **ImageOpacityMask** — Uses an `Image` to make an `ImageBrush` and uses it as an `OpacityMask` for an `Image` and a `Button`. Figure 4-9. [Page 69].
- **Margins** — Displays `Buttons` with different `Margin` values. Figure 4-3. [Page 64].
- **MaxMinSizes** — Demonstrates `MinHeight` and `MaxHeight` properties as a window is resized. Figure 4-4. [Page 65].
- **SizeInContainers** — Demonstrates how controls are sized inside different containers. Figure 4-1. [Page 62].
- **SizeInSingleChildContainers** — Demonstrates how controls are sized inside different containers that can hold only a single child. Figure 4-2. [Page 63].

CHAPTER 5

Chapter 5 contains the following example programs:

- **PopupPlacement** — Demonstrates the `Popup` control's `Placement` values `Top`, `Left`, `Right`, and `Bottom`. Figure 5-10. [Page 86].
- **UseBorder** — Displays a `Border` control with curved corners. Figure 5-14. [Page 90].
- **UseBulletDecorator** — Uses the `BulletDecorator` to make a bulleted list. Figure 5-15. [Page 91].
- **UseDocumentViewer** — Displays a fixed format document in a `DocumentViewer`. Figure 5-4. [Page 80].
- **UseFlowDocument** — Displays a `FlowDocument` that is not inside a viewer (so it behaves like it's in a `FlowDocumentReader`).
- **UseFlowDocumentPageViewer** — Displays a `FlowDocument` in a `FlowDocumentPageViewer` (in Page mode). Figure 5-5. [Page 81].

- **UseFlowDocumentReader** — Displays a `FlowDocument` in a `FlowDocumentReader` (in Page, Scroll, or TwoPage mode). Figure 5-6. [Page 82].
- **UseFlowDocumenScrollViewer** — Displays a `FlowDocument` in a `FlowDocumentScrollViewer` (in Scroll mode). Figure 5-7. [Page 82].
- **UseGroupBox** — Demonstrates the `GroupBox` control. Figure 5-16. [Page 84].
- **UseImage** — Demonstrates the `Image` control. Figure 5-2. [Page 86].
- **UseLabel** — Demonstrates the `Label` control. Figure 5-8. [Page 87].
- **UseListView** — Demonstrates the `ListView` control. Figure 5-17. [Page 92].
- **UseMediaElement** — Demonstrates the `MediaElement` control to play audio and video. Also demonstrates the XAML `SoundPlayerAction` command to play sounds. Figure 5-3. [Page 78].
- **UsePopup** — Demonstrates the `Popup` control to display information above the window. Figure 5-9. [Page 84].
- **UseProgressBar** — Demonstrates the `ProgressBar` control. Figure 5-18. [Page 97].
- **UseSeparator** — Demonstrates the `Separator` control in menus and toolbars. Figure 5-19. [Page 97].
- **UseTextBlock** — Demonstrates the `TextBlock` control and many of the `inlines` and other objects that it can contain. Figure 5-12. [Page 89].
- **UseTextTrimming** — Demonstrates the `TextBlock`'s `TextTrimming` property. Figure 5-11. [Page 87].
- **UseToolTip** — Shows how to make tooltips with the `ToolTip` attribute and with a separate `ToolTip` object. Figure 5-13. [Page 89].
- **UseTreeView** — Demonstrates the `TreeView` control. Figure 5-20. [Page 98].

CHAPTER 6

Chapter 6 contains the following example programs:

- **ScrollBarVisibility** — Shows the difference between the `ScrollBar` control's `Visibility` property values `Disabled` and `Hidden`. Figure 6-5. [Page 107].
- **UseCanvas** — Demonstrates the `Canvas` control. Figure 6-1. [Page 102].
- **UseDockPanel** — Demonstrates the `DockPanel` control. Figure 6-2. [Page 104].
- **UseExpander** — Demonstrates the `Expander` control. Figure 6-3. [Page 105].
- **UseGrid** — Demonstrates the `Grid` control. Figure 6-4. [Page 107].
- **UseScrollViewer** — Demonstrates the `ScrollViewer` control. Figure 6-6. [Page 108].
- **UseStackPanel** — Demonstrates the `StackPanel` control. Figure 6-7. [Page 108].
- **UseStatusBar** — Demonstrates the `StatusBar` control. Figure 6-8. [Page 109].

- **UseTabControl** — Demonstrates the `TabControl` control. Figure 6-9. [Page 110].
- **UseToolBar** — Demonstrates the `ToolBarTray` and `ToolBar` controls. Figures 6-10 and 6-11. [Page 112].
- **UseUniformGrid** — Demonstrates the `UniformGrid` control. Figure 6-13. [Page 114].
- **UseVerticalToolBar** — Demonstrates the `ToolBar` control with vertical orientation. Figure 6-12. [Page 113].
- **UseViewbox** — Demonstrates the `Viewbox` control and its different `Stretch` property values. Figure 6-14. [Page 115].
- **UseWindowsFormsHost** — Demonstrates the `WindowsFormsHost` control. Figure 6-15. [Page 116].
- **UseWrapPanel** — Demonstrates the `WrapPanel` control. Figure 6-16. [Page 117].

CHAPTER 7

Chapter 7 contains the following example programs:

- **UseButtons** — Demonstrates `Button` controls. Figure 7-1. [Page 121].
- **UseCheckBoxes** — Demonstrates `CheckBox` controls. Figure 7-2. [Page 121].
- **UseComboBox** — Demonstrates `ComboBox` controls including a `ComboBox` with complex items. Figures 7-3 and 7-4. [Page 122-123].
- **UseContextMenu** — Demonstrates `ComboBox` controls including a `ComboBox` with complex items. Figure 7-5. [Page 124].
- **UseFrame** — Demonstrates a `Frame` used to display web pages. Figure 7-6. [Page 126].
- **UseGridSplitter** — Demonstrates `GridSplitter` controls that let the user resize a `Grid` control's rows and columns. Figure 7-7. [Page 127].
- **UseListBox** — Demonstrates the `ListBox` control making an extended selection. Figure 7-8. [Page 129].
- **UseMenu** — Demonstrates the `Menu` control with `MenuItems` that display accelerators, shortcuts, and complex content such as images. Figures 7-9 and 7-10. [Page 131, 131].
- **UsePasswordBox** — Demonstrates the `PasswordBox` control to let the user enter a password that is hidden on the screen. Figure 7-11. [Page 132].
- **UseRadioButton** — Demonstrates three groups of `RadioButtons`. Figure 7-12. [Page 133].
- **UseRepeatButton** — Demonstrates the `RepeatButton` control. Figure 7-13. [Page 134].
- **UseRichTextBox** — Demonstrates the `RichTextBox` control. Menus let you change the selected text's style (**bold**, *italic*, underline), size (small, medium, large), color, background color, and font family (Times New Roman, Arial, `Courier New`). Menus also let you control paragraph alignment, bulleted and numbered lists, undo, and redo. Figure 7-14. [Page 137].

- **UseScrollBar** — Demonstrates three `ScrollBars` to let the user define a custom color. Figure 7-15. [Page 140].
- **UseSlider** — Demonstrates three `Sliders` to let the user define a custom color. Figure 7-16. [Page 141].
- **UseTextBox** — Demonstrates the `TextBox` control. Menus let you change the selected text's style (**bold**, *italic*, underline), size (small, medium, large), color, background color, and font family (Times New Roman, Arial, `Courier New`). Menus also let you control paragraph alignment, undo, and redo. Figure 7-17. [Page 143].

CHAPTER 8

Chapter 8 contains the following example programs:

- **PathBezier** — Uses the `Path` mini-language's Bézier curve commands to draw cubic, smooth, quadratic, and smooth "T" Bézier curves. Figure 8-4. [Page 149].
- **PathFillRules** — Shows the difference between the Odd/Even and Non-Zero fill rules. Figure 8-3. [Page 149].
- **PathObjects** — Draws shapes by using the `Path` mini-language and using objects (which is much more verbose). Figure 8-5. [Page 150].
- **PolygonPolylineDifference** — Demonstrates the difference between a `Polygon` and a `Polyline` that has first and last points at the same place. Figure 8-6. [Page 151].
- **UseEllipseLine** — Demonstrates the `Ellipse` and `Line` controls. Figure 8-1. [Page 147].
- **UsePathPolygonPolyline** — Demonstrates the `Path`, `Polygon`, and `Polyline` controls. Figure 8-2. [Page 148].
- **UseRectangle** — Demonstrates the `Rectangle` control including one with a thick dashed border. Figure 8-7. [Page 152].

CHAPTER 9

Chapter 9 contains the following example programs:

- **AttachedProperties** — Demonstrates several attached properties and property elements. Figure 9-5. [Page 162].
- **ComplexBrush** — Fills a form with a many-colored radial gradient brush. Figure 9-2. [Page 155].
- **Heart** — Uses a `Path` to draw a filled heart shape. Figure 9-1. [Page 154].
- **InheritedProperties** — Demonstrates that some properties are inherited from a control's container and some are not. Figure 9-4. [Page 160].
- **MakeComplexBrush** — Builds a complex radial gradient brush in code-behind similar to the one shown in Figure 9-2. [Page 155].
- **PropertyElements** — Demonstrates several property elements. Figure 9-3. [Page 158].

CHAPTER 10

Chapter 10 contains the following example programs:

- **FillRules** — Demonstrates the `FillRule` property values `Nonzero` and `EvenOdd`. Figure 10-12. [Page 170].
- **GradientPens** — Draws several shapes with pens that use linear color gradients. Figure 10-4. [Page 167].
- **ImageBrushTileModes** — Uses `ImageBrushes` to demonstrate the `TileMode` property values `FlipX`, `FlipY`, `FlipXY`, `Tile`, and `None`. Figure 10-16. [Page 174].
- **MagnifiedDrawingBrush** — Draws `Rectangles` filled with a `DrawingBrush` at various scales. Figure 10-17. [Page 175].
- **MagnifiedLines** — Draws the same line magnified to show that you can tell the difference between dash caps only when a line is fairly thick. Figure 10-2. [Page 166].
- **MagnifiedVisualBrush** — Draws `Rectangles` filled with a `VisualBrush` at various scales. The result looks similar to Figure 10-17. [Page 177].
- **Opacity** — Draws several overlapping translucent shapes on top of some text. Figure 10-3. [Page 167].
- **PensAndBrushes** — Draws an ellipse to demonstrate a simple pen and brush. Figure 10-1. [Page 165].
- **PictureFilledText** — Uses an `ImageBrush` to draw text that is filled with a picture. Figure 10-11. [Page 170].
- **SpreadMethods** — Demonstrates the `SpreadMethod` property's `Pad`, `Reflect`, and `Repeat` values for gradient brushes. Figure 10-13. [Page 171].
- **StrokeDashArrays** — Draws lines with different `StrokeDashArray` values. Figure 10-5. [Page 168].
- **StrokeDashCaps** — Draws lines with different `StrokeDashCap` values. Figure 10-6. [Page 168].
- **StrokeDashOffsets** — Draws lines with different `StrokeDashOffset` values. Figure 10-7. [Page 168].
- **StrokeLineCaps** — Draws lines with different `StrokeEndLineCap` and `StrokeStartLineCap` values. Figure 10-8. [Page 168].
- **StrokeLineJoins** — Draws lines with different `StrokeLineJoin` values. Figure 10-9. [Page 169].
- **StrokeMiterLimits** — Draws lines with different `StrokeMiterLimit` values. Figure 10-10. [Page 169].
- **UseLinearGradientBrush** — Draws several different kinds of `LinearGradientBrush`. Figure 10-14. [Page 172].
- **UseRadialGradientBrush** — Draws several different kinds of `RadialGradientBrush`. Figure 10-15. [Page 174].
- **UseSolidBrush** — Draws two overlapping ellipses that are filled with solid colors. [Page 171].

CHAPTER 11

Chapter 11 contains the following example programs:

- **EventNameAttributes** — Attaches event handlers to controls by placing event name attributes in the XAML code. Figure 11-1. [Page 181].
- **EventNameAttributesRelaxed** — This Visual Basic program is similar to the program EventNameAttributes except it uses relaxed delegates to simplify the code-behind. The result looks similar to Figure 11-1. [Page 188].
- **HandlesClause** — This Visual Basic program uses `Handles` clauses to attach event handlers to events. The result looks similar to Figure 11-1. [Page 190].
- **ImageColors** — Demonstrates various techniques for attaching buttons to code-behind. Figure 11-2. [Page 181].
- **RuntimeEventHandlers** — Attaches event handlers to controls at run time. The result looks similar to Figure 11-1. [Page 189].

CHAPTER 12

Chapter 12 contains the following example programs:

- **ButtonResources** — Displays several buttons that use resources to achieve a common appearance. The XAML file contains a second set of resources commented out so you can easily switch to another appearance. Figures 12-1 and 12-2. [Page 195, 197].
- **ContentResource** — Demonstrates that you can use a resource that is a control by setting a `Button`'s `Content` property to a `TextBlock`. Figure 12-3. [Page 197].
- **ContextMenuResource** — Uses resources to make several controls use the same `ContextMenu`. Shows how to tell which control displayed the `ContextMenu`. Figure 12-4. [Page 199].
- **DynamicLabelResource** — Uses a `DynamicResource` to allow a `Label` to use a value in its own resources. [Page 208].
- **MultiWindowResource** — Uses merged resource dictionaries to allow several windows to share resources, giving a common appearance. Figure 12-8. [Page 206].
- **ResourceDictionaries** — Uses two merged resource dictionaries in external files to display one of two appearances. Figures 12-9 and 12-10. [Page 207].
- **ResourceHierarchy** — Demonstrates a resource hierarchy in which `Labels` find resources of their own, in their container, in the window, and in the application. Figures 12-6 and 12-7. [Page 204].
- **SimpleClock** — Demonstrates static and dynamic resources by displaying a resource changed by code-behind and by using system colors as they change. Figure 12-11. [Page 211].
- **SysDataTypes** — Displays examples of the simple data types defined in mscorlib. Figure 12-5. [Page 200].

CHAPTER 13

Chapter 13 contains the following example programs:

- **ButtonResources** — Displays button-like rectangles similar to those displayed by the program ButtonValues except it uses Resources to make the code a bit more consistent. Similar to Figure 13-1. [Page 213-215].
- **ButtonStyles** — Displays button-like rectangles similar to those displayed by the program ButtonValues except it uses `Styles` to make the code much simpler and more maintainable. Similar to Figure 13-1. [Page 220, 221, 226, 227, 229].
- **ButtonValues** — Displays button-like rectangles by setting individual property values for each button. Figure 13-1. [Page 213].
- **ControlStyle** — Displays two `Buttons` and a `Label` that use various `Styles`. One `Button` and the `Label` share the same `Style`. Figure 13-3. [Page 220].
- **ImageTriggers** — Displays a series of `Images` and `Labels` with `Opacity = 0.5`. When `IsMouseOver` is `True`, a trigger changes the `Opacity` to 1 and sets the controls' `LayoutTransform` property to a `ScaleTransform` object that enlarges the controls. Figure 13-10. [Page 231].
- **InheritedStyles** — Demonstrates Styles that inherit from each other by using the `BasedOn` keyword. Figure 13-6. [Page 226].
- **IsActiveTrigger** — Uses `Triggers` on the `Window`'s `IsActive` property and on the `TextBox` `IsFocused` property. Figure 13-11. [Page 233].
- **IsMouseOverTriggers** — Demonstrates `Triggers` on `Button` `IsMouseOver` properties to enlarge a `Button` when the mouse is over it. Figure 13-8. [Page 230].
- **LabelAndRectStyle** — Displays a `Button`, `Label`, and `Rectangle` that all share a common `Style`. Figure 13-4. [Page 221].
- **MenuMouseOverTriggers** — Uses a `MenuItem` `IsMouseOver` trigger to change the `MenuItem`'s `LayoutTransform` and `BitmapEffect` properties. Figure 13-9. [Page 231].
- **RedRectangles** — Uses a named style to set properties on rectangles. Figure 13-2. [Page 217].
- **TextTriggers** — Demonstrates `Triggers` on `TextBox` and `ComboBox` `Text` properties to high-

- **AnimatedText** — Demonstrates a `StringAnimationUsingKeyFrames` with `DiscreteStringKeyFrame` objects to make text appear a few characters at a time. Figure 14-10. [Page 254].
- **AnimationWithoutStoryboards** — Uses code-behind to animate an `Ellipse`'s `Canvas.Left` and `Canvas.Top` properties. Figure 14-12. [Page 259].
- **BouncingBall** — Displays a bouncing ball that plays a sound whenever it touches the "ground." Demonstrates spline key frames and `ParallelTimeline` objects to play a reversing animation with sounds. Figure 14-11. [Page 258].
- **CentralizedTriggers** — Uses `EvenTriggers` centralized in a `Canvas` control to handle `MouseLeftButtonDown` events raised by the controls contained in the `Canvas`. Figure 14-2. [Page 238].
- **GrowingButtons** — Demonstrates the three combinations of property triggers with `Setters`, property triggers with `Storyboards`, and event triggers with `Storyboards`. Figure 14-4. [Page 243].
- **JumpingButton** — Shows how complex the code is for even a very simple `Storyboard` that animates a single control property. [Page 261].
- **PropertyTriggerButton** — Demonstrates a property trigger that executes `Storyboards` in its `EnterActions` and `ExitActions` sections. Figure 14-3. [Page 242].
- **RepeatingSound** — Runs a repeating animation that plays a sound. [Page 257].
- **RovingButton** — Displays a `Button` that follows a rectangular path. Demonstrates methods to start, pause, resume, stop, and remove a `Storyboard`. Figure 14-6. [Page 248].
- **RovingButtonDiscrete** — Displays a `Button` that follows a rectangular path in discrete jumps. Demonstrates methods to start, pause, resume, stop, and remove a `Storyboard`. Similar to Figure 14-6. [Page 251].
- **RovingButtonMixedKeyFrames** — Makes a `Button` follow a path by using linear key frames, discrete key frames, path animation, and simple linear animation. Demonstrates methods to start, pause, resume, stop, and remove a `Storyboard`. Figure 14-9. [Page 252].
- **RovingButtonWithKeyFrames** — Uses a `DoubleAnimationUsingKeyFrames` object with linear key frames to make a button follow a rectangular path. Demonstrates methods to start, pause, resume, stop, and remove a `Storyboard`. Similar to Figure 14-6. [Page 249].
- **RovingButtonWithPath** — Displays a `Button` that follows a path. Demonstrates methods to start, pause, resume, stop, and remove a `Storyboard`. Figure 14-8. [Page 251].
- **RovingButtonWithSplines** — Uses a `DoubleAnimationUsingKeyFrames` object with spline key frames to make a button follow a rectangular path. The button starts crossing each side of the path slowly, accelerates, and then slows into the corner. Similar to Figure 14-6. [Page 250].
- **SoundEvents** — Plays sound files when the user clicks on a button or moves the mouse over a second button. [Page 256].
- **SpinButton** — Makes a button spin 360 degrees when clicked. Demonstrates a `Storyboard` defined as a resource. Figure 14-1. [Page 237].
- **SplineGraph** — Displays a drawing of a spline curve with its control points. Figure 14-7. [Page 249].

CHAPTER 15

Chapter 15 contains the following example programs:

- **BetterLabelTemplate** — Demonstrates a `Label` template that honors the client control's `Background`, `BorderBrush`, `BorderThickness`, `HorizontalContentAlignment`, and `VerticalContentAlignment` properties. Figure 15-3. [Page 266].
- **DisabledLabelTemplate** — Demonstrates a `Label` template that displays multi-line text and a distinctive appearance when the `Label` is disabled. Figure 15-5. [Page 268].
- **EllipseButton** — Demonstrates an elliptical `Button` template that uses different appearances for normal, disabled, and defaulted buttons. It uses animation when the mouse is over the button to display a moving "sparkle" highlight. Figures 15-9 through 15-12. [Page 275–277].
- **GlassButton** — Demonstrates a semitransparent `Button` template that uses different appearances for normal, disabled, and defaulted buttons. Figures 15-6 through 15-8. [Page 270–271].
- **InterestingLabelTemplates** — Demonstrates two `Label` templates — one that displays a double border and one that displays multi-line text. Figure 15-4. [Page 266].
- **ShowTemplate** — Displays a control's default template's XAML code. Figure 15-13. [Page 281].
- **SimpleLabelTemplate** — Demonstrates a simple `Label` template with preset brushes. Figure 15-2. [Page 265].
- **SliderParts** — This program just displays a `Slider` so it's easy to see its parts. Figure 15-1. [Page 263].

CHAPTER 16

Chapter 16 contains the following example programs:

- **AnimatedSkins** — Lets the user switch skins by animating property values so the user sees one skin morph into another. Figures 16-9 through 16-11. [Page 295-296].
- **OrderTracking** — Uses resource dictionaries to display one of three skins for use by different kinds of users: manager, billing clerk, or order entry clerk. Figure 16-7. [Page 289].
- **ShowThemes** — Displays controls that use the default, Aero, Classic, Luna, and Royale themes. Figures 16-1 and 16-3. [Page 284, 285].
- **SkinInterfaces** — Lets the user switch skins by loading XAML code at run time. Figures 16-12 and 16-13. [Page 298].
- **Skins** — Lets the user switch skins by loading resource files at run time. Figure 16-8. [Page 293].

CHAPTER 17

Chapter 17 contains the following example programs:

- **PrintFixedDocument** — Uses the `PrintDialog`'s `PrintDocument` method to print a `FixedDocument`. Figure 17-10. [Page 315].

- **PrintFlowDocument** — Uses the `PrintDialog`'s `PrintDocument` method to print a `FlowDocument`. Figures 17-6 through 17-9. [Page 314-315].
- **PrintShapes** — Uses a `DocumentPaginator` class to print four pages of shapes generated by code at print time. Figure 17-5. [Page 310].
- **PrintWindow** — Prints a window centered and optionally enlarged. Figures 17-1, 17-3, and 17-4. [Page 304, 306, 309].
- **SimplePrintWindow** — Uses the `PrintDialog`'s `PrintVisual` method to print an image of a window. The result is clipped on the upper and left sides. [Page 305].

CHAPTER 18

Chapter 18 contains the following example programs:

- **ColorList** — Uses bindings to make a `Label` display the color selected in a `ListBox`. The data source is in the parent `Grid`'s `DataContext` so the `ListBox` and `Label` share the same binding source. The `Label`'s bindings demonstrate setting `Source = /`. Figure 18-7. [Page 324].
- **ColumnWidths** — Uses a binding with `RelativeSource = FindAncestor` to display `Grid` column widths in `Labels`. Figure 18-5. [Page 321].
- **DockPanelValues** — Uses a binding with `RelativeSource = Self and Path = (DockPanel.Dock)` to make `Labels` display their `DockPanel.Dock` values. [Page 324].
- **GridColumns** — Uses a binding with `RelativeSource = Self and Path = (Grid.Column)` to make a `Label` display its own `Grid.Column` value. [Page 323].
- **NumberList** — Makes a `ListBox` bound to an array of `Int32` defined in XAML code. Figure 18-8. [Page 325].
- **OrgChartMasterDetail** — Displays an organizational chart by using `ListBoxes` to show the master–detail relationships among objects. Figure 18-13. [Page 332].
- **OrgChartTreeView** — Uses `TreeView` controls to display two different views of a hierarchical organizational chart built with objects in code-behind. Figure 18-12. [Page 330].
- **OrgChartXaml** — Uses `TreeView` controls to display two different views of a hierarchical organizational chart built with objects defined in the XAML code. Similar to Figure 18-12. [Page 334–335, 338].
- **OrgChartXml** — Uses `TreeView` controls to display two different views of a hierarchical organizational chart built with objects defined by XML code. Similar to Figure 18-12. [Page 336–337].
- **PersonList** — Displays a simple `Person` class in a bound `ListBox` with and without an overridden `ToString` method. Figure 18-9. [Page 326].
- **PersonSource** — Uses a binding with `Source` set to a `Person` object. The `Person` class is defined in code-behind, and it is created in XAML code. Figure 18-2. [Page 320].

- **PersonSource2** — Uses a binding with `DataContext` set to a `Person` object so controls can share the context. The `Person` class is defined in code-behind, and it is created in XAML code. Similar to Figure 18-2. [Page 323].
- **Planets** — Demonstrates the `ItemTemplates` property for `ListBox` and `ComboBox` controls. Figure 18-10. [Page 327].
- **PlanetsPanel** — Demonstrates the `ItemsPanel` property for a `ListBox` control. Figure 18-11. [Page 329].
- **PreviousData** — Uses a binding with `RelativeSource = PreviousData` to display a list of values and their previous values. Figure 18-6. [Page 322].
- **SliderToHeight** — When you drag a `Slider`, uses simple binding to resize the window and display the `Slider`'s value in a `Label`.
- **StudentData** — Uses ADO.NET to display data from a database. Demonstrates how to link a `ComboBox` to a lookup table and how to display master-detail data. Figure 18-14. [Page 338].
- **TemplatedParent** — Uses a binding with `RelativeSource = TemplatedParent` to display the margins in a `Label` that uses a `Template`. Figure 18-4. [Page 321].
- **TextBoxToLabel** — Uses simple bindings to make two `Label` controls display whatever is typed in a textbox. Figure 18-1. [Page 318].
- **TypeAColor** — Uses a binding with `RelativeSource = Self` to make a `TextBox` use its text as its background color name. Figure 18-3. [Page 320].

CHAPTER 19

Chapter 19 contains the following example programs:

- **CommandTarget** — Demonstrates command objects that have specific command targets. Figure 19-3. [Page 351].
- **DocumentCommands** — Demonstrates the predefined document-oriented command objects `New`, `Open`, `Save`, `SaveAs`, and `Close`. Figure 19-4. [Page 353].
- **ImageColors** — Demonstrates custom `RoutedUICommand` objects. Figure 19-5. [Page 356].
- **TextBoxCommands** — Demonstrates predefined commands that are supported by the `TextBox` control: `Copy`, `Cut`, `Paste`, `Undo`, and `Redo`. Figure 19-2. [Page 349].

CHAPTER 20

Chapter 20 contains the following example programs:

- **BevelEffects** — Demonstrates `BevelBitmapEffect` styles. Figure 20-5. [Page 364].
- **CombinedTransformations** — Demonstrates combined rotation and translation transformation and shows that the transformation order is important. Figure 20-2. [Page 361].
- **Effects** — Demonstrates bitmap effects. Figure 20-4. [Page 364].

- **Games** — Demonstrates transformations and subtler bitmap effects that are more useful than the other examples in this chapter. Figure 20-7. [Page 365].
- **LayoutVsRender** — Demonstrates the difference between `LayoutTransform` and `RenderTransform` properties. Figure 20-3. [Page 362].
- **Transformations** — Demonstrates rotation, scaling, skew, translation, and matrix render transformations. Figure 20-1. [Page 360].
- **TransparentEffects** — Demonstrates bitmap effects on unfilled objects. Figure 20-6. [Page 365].

CHAPTER 21

Chapter 21 contains the following example programs:

- **SimpleFixedDocument** — Draws a simple `FixedDocument` containing four pages of shapes. Figure 21-2. [Page 371].
- **SaveFixedDocument** — Draws a simple `FixedDocument` much as the program SimpleFixedDocument does and saves it into an XPS file. [Page 372].
- **ShowFlowDocument** — Draws a `FlowDocument` that demonstrates many flow document objects including `BlockUIContainer`, `List`, `Paragraph`, `Figure`, `Floater`, `Run`, `Span`, and `Table`. Figures 21-3 through 21-6. [Page 373-375, 378].
- **ViewFixedDocument** — Loads an XPS file created externally in Microsoft Word, WordPad, or some other application. Figure 21-1. [Page 369].

CHAPTER 22

Chapter 22 contains the following example programs:

- **FrameApp** — Displays `Page` objects inside a `Frame`. Figure 22-5. [Page 386].
- **PageApp** — Displays several `Pages`. Code-behind uses a `NavigationService`'s `Navigate` method to move between pages and to a Web URL. Figures 22-3 and 22-4. [Page 383, 384].
- **PageBorder** — Displays a `Page` with a `Style` targeted at `Border` controls to show that a `Page` displays a `Border`. Figure 22-1. [Page 380].
- **PageDocument** — Displays `Pages` containing `FlowDocuments` that navigate using hyperlinks. Figure 22-2. [Page 381].

CHAPTER 23

Chapter 23 contains the following example programs:

- **BarChart** — Draws a three-dimensional bar chart. Figure 23-11. [Page 403].
- **CameraTypes** — Displays the same scene with perspective and orthographic cameras. Figure 23-6. [Page 395].

- **Graph** — Draws a three-dimensional ribbon graph. Figure 23-12. [Page 403].
- **LabeledBarChart** — Draws a three-dimensional bar chart with labels on the end of each bar. Figure 23-13. [Page 404].
- **Lights** — Demonstrates the differences between `AmbientLight`, `DirectionalLight`, `PointLight`, and `SpotLight`. Figure 23-7. [Page 397].
- **MakeSurface** — Draws a three-dimensional surface. Figure 23-14. [Page 405].
- **Materials** — Demonstrates diffuse, specular, and emissive material types. Figure 23-9. [Page 399].
- **RectanglesAndBoxes** — Demonstrates code-behind routines that build textured rectangles, boxes, cylinders, and spheres. Figure 23-10. [Page 401].
- **SingleMeshSpheres** — Draws spheres such that each sphere is defined by a single `MeshGeometry3D` object. Figure 23-4. [Page 393].
- **SpheresWithNormals** — Draws spheres wherein each triangle is a separate `MeshGeometry3D` object with and without explicit normals. Figure 23-3. [Page 392].
- **Tetrahedrons** — Shows how you generally need many light sources to produce a realistic result. Figure 23-8. [Page 398].
- **TexturedBlock** — Draws a three-dimensional block with sides textures with various materials that you can rotate. Figure 23-1. [Page 387].
- **TextureCoordinates** — Shows how to map texture coordinates to points in a `MeshGeometry3D`. Figure 23-5. [Page 393].

CHAPTER 24

Chapter 24 contains the following example programs:

- **Bouncing Balls** — Silverlight application that displays balls bouncing around on the browser. Figure 24-5. [Page 412].
- **SelectColor** — Silverlight application that lets the user pick a color by adjusting three scrollbars. Figure 24-4. [Page 410].

APPENDIX A

There are no example programs for Appendix A.

APPENDIX B

Appendix B contains the following example program:

- **UseTextBlock** — Demonstrates `TextBlock` inlines. Figure B-1. [Page 438].

APPENDIX C

Appendix C contains the following example programs:

- **RotatedTabs** — Displays a `TabControl` with the tabs on the left and rotated sideways. Figure C-1. [Page 454].
- **Toolbars** — Displays a `ToolBarTray` containing three `ToolBars` in two bands. Figure C-2. [Page 456].

APPENDIX D

Appendix D contains the following example program:

- **SimpleRichEditor** — Implements a simple RichtextBox Editor that lets you change the selected text's font, size, color, weight, and style. Figure D-1. [Page 479].
- **UseFrame** — Uses a Frame control to provide navigation between several web sites. [Page 467].

APPENDIX E

There are no example programs for Appendix E.

APPENDIX F

Appendix F contains the following example programs:

- **StrokeDashes** — Demonstrates the `StrokeStartLineCap`, `StrokeEndLineCap`, and `StrokeDashCap` values. Figure F-2. [Page 494].
- **StrokeMiterLimit** — Draws a `Polygon` that defines its `Stroke` property with an element attribute and that demonstrates the `StrokeMiterLimit` property. Figure F-1. [Page 494].

APPENDIX G

Appendix G contains the following example programs:

- **Brushes** — Demonstrates solid, linear gradient, and radial gradient brushes. Figure G-1. [Page 496].
- **RadialCenter** — Demonstrates the `RadialGradientBrush Center`, and `GradientOrigin` properties. Figure G-3. [Page 502].
- **Reflections** — Makes a `VisualBrush` that displays a reflection of a `StackPanel` containing several other controls. Figure G-4. [Page 504].
- **SpreadMethods** — Demonstrates the different `SpreadMethod` values. Figure G-2. [Page 500].
- **ViewportsAndViewboxes** — Demonstrates viewports and viewboxes used by tiled brushes. Figure G-5. [Page 506].

APPENDIX H

Appendix H contains the following example programs:

- **Arc** — Illustrates the parameters for the Path mini-language `arc` command. Figure H-3. [Page 509].
- **BezierCommands** — Uses the Path mini-language's Bézier curve commands to draw cubic, smooth, quadratic, and smooth "T" Bézier curves. Figure H-2. [Page 509].
- **FillRules** — Shows the difference between the Odd/Even and Non-Zero fill rules. Figure H-1. [Page 508].

APPENDIX I

Appendix I contains the following example program:

- **XmlCustomerOrders** — Demonstrates various XPath expressions. Figure I-1. [Page 512].

APPENDIX J

There are no example programs for Appendix J.

APPENDIX K

There are no example programs for Appendix K.

APPENDIX L

Appendix L contains the following example program:

- **BitmapEffects** — Demonstrates the `BitmapEffect` classes. Figure L-1. [Page 533].

APPENDIX M

There are no example programs for Appendix M.

APPENDIX N

Appendix N contains the following example programs:

- **BareBonesButton** — Uses a template that changes a button's appearance. [Page 547–548].
- **BareBonesCheckBox** — Uses a template that changes a checkbox's appearance and changes its dot color depending on whether it is checked, unchecked, or in an indeterminate state. [Page 540].

- **BareBonesLabel** — Uses a template that makes a `Label` wrap its text and display a different appearance when `IsEnabled` is `False`. [Page 539–540].
- **BareBonesProgressBar** — Uses a template that changes a `ProgressBar`'s appearance. [Page 541].
- **BareBonesRadioButton** — Uses a template that changes a `RadioButton`'s appearance to display an **X** when it is checked. [Page 541].
- **BareBonesScrollBar** — Uses a template that changes a `ScrollBar`'s appearance. It contains a Triggers section to handle horizontal and vertical orientations. [Page 544–545].
- **LabeledProgressBar** — Uses a template to make a `ProgressBar` containing a `Label` that displays the control's current value. [Page 543].
- **ModifiedScrollBar** — Uses a template that changes a `ScrollBar`'s appearance drastically. It contains a Triggers section to handle horizontal and vertical orientations. [Page 545–547].
- **OrientedProgressBar** — Uses a template that changes a `ProgressBar`'s appearance. It contains a Triggers section to handle horizontal and vertical orientations. [Page 542].

APPENDIX O

There are no example programs for Appendix O.

APPENDIX P

There are no example programs for Appendix P.

INDEX

Numbers

A

B

C

D

H

I

K

L

M

N

P

Q

R

T

U

X

Y

Z

Related Wrox Books

Beginning ASP.NET 4: in C# and VB

ISBN: 9780470502211

This introductory book offers helpful examples and step-by-step format and has code examples written in both C# and Visual Basic. With this book you will gradually build a Web site example that takes you through the processes of building basic ASP.NET Web pages, adding features with pre-built server controls, designing consistent pages, displaying data, and more.

Beginning Microsoft Visual Basic 2010

ISBN: 9780470502228

This book not only shows you how to write Windows applications, Web applications with ASP.NET, and Windows mobile and embedded CE apps with Visual Basic 2010, but you'll also get a thorough grounding in the basic nuts and bolts of writing good code. You'll be exposed to the very latest VB tools and techniques with coverage of both the Visual Studio 2010 and .NET 4 releases.

Beginning Microsoft Visual C# 2010

ISBN: 9780470502266

Using this book, you will first cover the fundamentals such as variables, flow control, and object-oriented programming and gradually build your skills for Web and Windows programming, Windows forms, and data access. Step-by-step directions walk you through processes and invite you to "Try it Out," at every stage. By the end, you'll be able to write useful programming code following the steps you've learned in this thorough, practical book. If you've always wanted to master Visual C# programming, this book is the perfect one-stop resource.

Professional ASP.NET 4: in C# and VB

ISBN: 9780470502204

Written by three highly recognized and regarded ASP.NET experts, this book provides all-encompassing coverage on ASP.NET 4 and offers a unique approach of featuring examples in both C# and VB, as is the incomparable coverage of core ASP.NET. After a fast-paced refresher on essentials such as server controls, the book delves into expert coverage of all the latest capabilities of ASP.NET 4. You'll learn site navigation, personalization, membership, role management, security, and more.

Professional C# 4 and .NET 4

ISBN: 9780470502259

After a quick refresher on C# basics, the author dream team moves on to provide you with details of language and framework features including LINQ, LINQ to SQL, LINQ to XML, WCF, WPF, Workflow, and Generics. Coverage also spans ASP.NET programming with C#, working in Visual Studio 2010 with C#, and more. With this book, you'll quickly get up to date on all the newest capabilities of C# 4.

Professional Visual Basic 2010 and .NET 4

ISBN: 9780470502242

If you've already covered the basics and want to dive deep into VB and .NET topics that professional programmers use most, this is your guide. You'll explore all the new features of Visual Basic 2010 as well as all the essential functions that you need, including .NET features such as LINQ to SQL, LINQ to XML, WCF, and more. Plus, you'll examine exception handling and debugging, Visual Studio features, and ASP.NET web programming.

Professional Visual Studio 2010

ISBN: 9780470548653

Written by an author team of veteran programmers and developers, this book gets you quickly up to speed on what you can expect from Visual Studio 2010. Packed with helpful examples, this comprehensive guide explains and examines the features of Visual Studio 2010, which allows you to create and manage programming projects for the Windows platform. It walks you through every facet of the Integrated Development Environment (IDE), from common tasks and functions to its powerful tools.

Visual Basic 2010 Programmer's Reference

ISBN: 9780470499832

This reference guide provides you with a broad, solid understanding of essential Visual Basic 2010 topics and clearly explains how to use this powerful programming language to perform a variety of tasks. As a tutorial, the book describes the Visual Basic language and covers essential Visual Basic topics. The material presents categorized information regarding specific operations and reveals useful tips, tricks, and tidbits to help you make the most of the new Visual Basic 2010.

Go beyond what you thought possible in user interface development

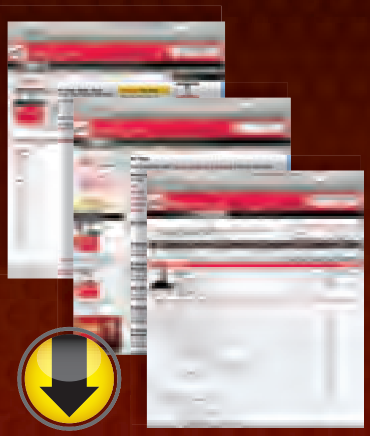
Windows Presentation Foundation (WPF) enables you to build effective and unique graphical user interfaces. However, it takes a steep learning curve to master the exceptions and shortcuts that are built into WPF. This reference provides you with a solid foundation of fundamental WPF concepts so you can start building attractive, dynamic, and interactive applications quickly and easily. As the book progresses, topics gradually become more advanced, and you'll discover how to use WPF to build applications that run in more environments, on more hardware, using more graphical tools, and providing a more engaging visual experience than is normally possible with Windows Forms.

WPF Programmer's Reference:

- Explains with full color code examples how code is connected to the user interface and shows how operations can be performed using both XAML and C#
- Features a series of essential appendices that summarize WPF syntax and concepts for easy reference
- Covers the latest release of WPF, along with Visual Studio® 2010, Expression Blend™ 3, and .NET 4
- Shows how to position and arrange content, layout, interaction, and drawing controls; define their properties; and manipulate those properties to produce stunning visual effects
- Addresses event triggers and animation, templates, themes and skins, data binding, and transformations and effects
- Provides comparable Visual Basic versions of all code examples on the companion web site

Rod Stephens is a professional software developer who has built a wide variety of software and database applications in his career that spans two decades. He is the author of more than twenty books and 250 articles, and is a regular contributor to DevX.com (www.devx.com).

Wrox guides are crafted to make learning programming languages and technologies easier than you think. Written by programmers for programmers, they provide a structured, tutorial format that will guide you through all the techniques involved.



wrox.com

Programmer Forums

Join our Programmer to Programmer forums to ask and answer programming questions about this book, join discussions on the hottest topics in the industry, and connect with fellow programmers from around the world.

Code Downloads

Take advantage of free code samples from this book, as well as code samples from hundreds of other books, all ready to use.

Read More

Find articles, ebooks, sample chapters and tables of contents for hundreds of books, and more reference resources on programming topics that matter to you.

Wrox™
An Imprint of
WILEY

Programming/C# (.NET)

$54.99 USA
$65.99 CAN

ISBN 978-0-470-47722-9
55499
9780470477229