Adam Nathan

WPF 4.5

UNLEASHED

SAMS

Some Praise for the First Edition of *Windows® Presentation Foundation Unleashed*

"The Nathan book is brilliant—you'll love it. Publishers, take note: I'd sure be buying a heck of a lot more technical books if more of them were like this one."
—Jeff Atwood, codinghorror.com, cofounder of Stack Overflow

"*Windows Presentation Foundation Unleashed* is a must-have book for anyone interested in learning and using WPF. Buy it, read it, and keep it close to your computer."
—Josh Smith, Microsoft MVP

"As we built the feature team that delivered the new WPF presentation layer for Visual Studio 2010, *Windows Presentation Foundation Unleashed* quickly became our must-read WPF reference book of choice, over and above other books on WPF and indeed internal documentation. Highly recommended for any developer wanting to learn how to make the most of WPF."
—James Bartlett, senior lead program manager, Microsoft Visual Studio

"I've bought nearly all available WPF books, but the only one that's still on my desk is *Windows Presentation Foundation Unleashed*. It not only covers all WPF aspects, but it does it in the right, concise way so that reading it was a real pleasure."
—Corrado Cavalli, Codeworks

"*Windows Presentation Foundation Unleashed* is the most insightful WPF book there is. Don't be misled by its size; this book has the best introduction and deepest insights. This is the must-read for anyone getting started or wanting to get the most out of WPF."
—Jaime Rodriguez, Microsoft client evangelist for Windows, WPF, Silverlight, and Windows Phone

"I found *Windows Presentation Foundation Unleashed* to be an excellent and thorough introduction and guide to programming WPF. It is clearly written, easily understood, and yet still deep enough to get a good understanding of how WPF works and how to use it. Not a simple feat to accomplish! I heartily recommend it to all the students who take DevelopMentor's WPF course! Anyone serious about doing WPF work should have a copy in their library."
—Mark Smith, DevelopMentor instructor, author of DevelopMentor's Essential WPF course

"I have read *Windows Presentation Foundation Unleashed* from cover to cover and have found it to be really the most comprehensive material on WPF. I can't think of even a single instance when I have not been able to find the solution (or a pointer to one) every time that I have picked up the book to figure out the intricacies of WPF."
—Durgesh Nayak, team leader, Axis Technical Group

"*Windows Presentation Foundation Unleashed* is the book that made WPF make so much sense for me. Without Adam's work, WPF would still be a mystery to me and my team. The enthusiasm for WPF is evident from the offset and it really rubs off on the reader."
—Peter O'Hanlon, managing director, Lifestyle Computing Ltd

"Adam Nathan's *Windows Presentation Foundation Unleashed* must surely be considered one of the seminal books on WPF. It has everything you need to help you get to grips with the learning cliff that is WPF. It certainly taught me loads, and even now, after several years of full-time WPF development, *Windows Presentation Foundation Unleashed* is never far from my hand."
—Sacha Barber, Microsoft MVP, CodeProject MVP, author of many WPF articles

"Of all the books published about WPF, there are only three that I recommend. *Windows Presentation Foundation Unleashed* is my primary recommendation to developers looking to get up to speed quickly with WPF."
—Mike Brown, Microsoft MVP, Client App Development, and president of KharaSoft, Inc.

Adam Nathan

WPF 4.5

UNLEASHED

SAMS | 800 East 96th Street, Indianapolis, Indiana 46240 USA

WPF 4.5 Unleashed

ISBN-13: 978-0-672-33697-3
ISBN-10: 0-672-33697-9

Library of Congress Control Number: 2013939232

Printed in the United States on America

First Printing July 2013

Trademarks

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

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author(s) and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the programs accompanying it.

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales

international@pearsoned.com

Editor-in-Chief
Greg Wiegand

Executive Editor
Neil Rowe

Development Editor
Mark Renfrow

Managing Editor
Kristy Hart

Senior Project Editor
Betsy Gratner

Indexer
Erika Millen

Proofreader
Kathy Ruiz

Technical Editors
Dwayne Need
Robert Hogue
Joe Castro
Jordan Parker

Publishing Coordinator
Cindy Teeters

Cover Designer
Mark Shirar

Senior Compositor
Gloria Schurick

Contents at a Glance

	Introduction	1
Part I	**Background**	
1	Why WPF?	7
2	XAML Demystified	17
3	WPF Fundamentals	55
Part II	**Building a WPF Application**	
4	Sizing, Positioning, and Transforming Elements	77
5	Layout with Panels	97
6	Input Events: Keyboard, Mouse, Stylus, and Touch	141
7	Structuring and Deploying an Application	177
8	Exploiting Windows Desktop Features	217
Part III	**Controls**	
9	Content Controls	241
10	Items Controls	255
11	Images, Text, and Other Controls	309
Part IV	**Features for Professional Developers**	
12	Resources	341
13	Data Binding	361
14	Styles, Templates, Skins, and Themes	415
Part V	**Rich Media**	
15	2D Graphics	473
16	3D Graphics	535
17	Animation	605
18	Audio, Video, and Speech	651
Part VI	**Advanced Topics**	
19	Interoperability with Non-WPF Technologies	671
20	User Controls and Custom Controls	717
21	Layout with Custom Panels	747
22	Toast Notifications	771
A	Fun with XAML Readers and Writers	783
	Index	799

Table of Contents

Introduction **1**

Who Should Read This Book? 2
Software Requirements 3
Code Examples 3
How This Book Is Organized 4
Conventions Used in This Book 6

Part I Background

1 Why WPF? **7**

A Look at the Past 8
Enter WPF 9
The Evolution of WPF 12
Summary 16

2 XAML Demystified **17**

XAML Defined 19
Elements and Attributes 20
Namespaces 22
Property Elements 25
Type Converters 26
Markup Extensions 28
Children of Object Elements 31
Mixing XAML with Procedural Code 36
XAML2009 44
XAML Keywords 49
Summary 52

3 WPF Fundamentals **55**

A Tour of the Class Hierarchy 55
Logical and Visual Trees 57
Dependency Properties 62
Summary 76

Part II Building a WPF Application

4 Sizing, Positioning, and Transforming Elements 77

Controlling Size ... 78
Controlling Position ... 83
Applying Transforms ... 86
Summary ... 95

5 Layout with Panels 97

Canvas ... 98
StackPanel ... 100
WrapPanel ... 102
DockPanel ... 105
Grid ... 108
Primitive Panels ... 120
Handling Content Overflow ... 122
Putting It All Together: Creating a Visual Studio–Like Collapsible, Dockable, Resizable Pane ... 130
Summary ... 140

6 Input Events: Keyboard, Mouse, Stylus, and Touch 141

Routed Events ... 141
Keyboard Events ... 150
Mouse Events ... 152
Stylus Events ... 156
Touch Events ... 158
Commands ... 170
Summary ... 176

7 Structuring and Deploying an Application 177

Standard Desktop Applications ... 177
Navigation-Based Desktop Applications ... 193
Gadget-Style Applications ... 205
XAML Browser Applications ... 207
Loose XAML Pages ... 213
Summary ... 215

8 Exploiting Windows Desktop Features 217

Jump Lists ... 217
Taskbar Item Customizations ... 229
Aero Glass ... 233
TaskDialog ... 236
Summary ... 239

Part III Controls

9 Content Controls 241

Buttons 243
Simple Containers 248
Containers with Headers 252
Summary 254

10 Items Controls 255

Common Functionality 256
Selectors 261
Menus 298
Other Items Controls 302
Summary 308

11 Images, Text, and Other Controls 309

The Image Control 309
Text and Ink Controls 311
Documents 318
Range Controls 334
Calendar Controls 336
Summary 340

Part IV Features for Professional Developers

12 Resources 341

Binary Resources 341
Logical Resources 349
Summary 360

13 Data Binding 361

Introducing the Binding Object 361
Controlling Rendering 373
Customizing the View of a Collection 385
Data Providers 396
Advanced Topics 403
Putting It All Together: The Pure-XAML Twitter Client 412
Summary 414

14 Styles, Templates, Skins, and Themes 415

Styles 416
Templates 430
Skins 458
Themes 465
Summary 470

Part V Rich Media

15 2D Graphics 473

Drawings 474
Visuals 491
Shapes 503
Brushes 511
Effects 527
Improving Rendering Performance 530
Summary 533

16 3D Graphics 535

Getting Started with 3D Graphics 536
Cameras and Coordinate Systems 540
Transform3D 552
Model3D 561
Visual3D 584
Viewport3D 591
2D and 3D Coordinate System Transformation 594
Summary 603

17 Animation 605

Animations in Procedural Code 606
Animations in XAML 619
Keyframe Animations 628
Easing Functions 635
Animations and the Visual State Manager 641
Summary 649

18 Audio, Video, and Speech 651

Audio 651
Video 656
Speech 662
Summary 669

Part VI Advanced Topics

19 Interoperability with Non-WPF Technologies 671

Embedding Win32 Controls in WPF Applications ... 673
Embedding WPF Controls in Win32 Applications ... 688
Embedding Windows Forms Controls in WPF Applications ... 695
Embedding WPF Controls in Windows Forms Applications ... 700
Mixing DirectX Content with WPF Content ... 704
Embedding ActiveX Controls in WPF Applications ... 710
Summary ... 714

20 User Controls and Custom Controls 717

Creating a User Control ... 719
Creating a Custom Control ... 728
Summary ... 746

21 Layout with Custom Panels 747

Communication Between Parents and Children ... 748
Creating a SimpleCanvas ... 751
Creating a SimpleStackPanel ... 756
Creating an OverlapPanel ... 759
Creating a FanCanvas ... 764
Summary ... 769

22 Toast Notifications 771

Prerequisites ... 771
Sending a Toast Notification ... 774
Toast Templates ... 775
Notification Events ... 778
Scheduled Notifications ... 779
Summary ... 780

A Fun with XAML Readers and Writers 783

Overview ... 783
The Node Loop ... 786
Reading XAML ... 787
Writing to Live Objects ... 791
Writing to XML ... 793
XamlServices ... 794

Index 799

About the Author

Adam Nathan is a principal software architect for Microsoft in the Startup Business Group. Adam was previously the founding developer and architect for Popfly, Microsoft's first product built on Silverlight, named one of the 25 most innovative products of 2007 by *PCWorld Magazine*. Having started his career on Microsoft's Common Language Runtime team, Adam has been at the core of .NET and WPF technologies since the very beginning.

Adam's books have been considered required reading by many inside Microsoft and throughout the industry. He is the author of the best-selling *WPF Unleashed* (Sams, 2006) that was nominated for a 2008 Jolt Award, *WPF 4 Unleashed* (Sams, 2010), *Windows 8 Apps with XAML and C# Unleashed* (Sams, 2012), *101 Windows Phone 7 Apps* (Sams, 2011), *Silverlight 1.0 Unleashed* (Sams, 2008), and *.NET and COM: The Complete Interoperability Guide* (Sams, 2002); a coauthor of *ASP.NET: Tips, Tutorials, and Code* (Sams, 2001); and a contributor to books including *.NET Framework Standard Library Annotated Reference, Volume 2* (Addison-Wesley, 2005) and *Windows Developer Power Tools* (O'Reilly, 2006). Adam is also the creator of PINVOKE.NET and its Visual Studio add-in. You can find him online at www.adamnathan.net or @adamnathan on Twitter.

Dedication

To Tyler and Ryan.

Acknowledgments

Although most of the process of writing a book is very solitary, this book came together because of the work of many talented and hard-working people. I'd like to take a moment to thank some of them by name.

I'd like to sincerely thank Dwayne Need, senior development manager from the WPF team. His feedback on my drafts was so thorough and insightful, the book is far better because of him. I'd like to thank Robert Hogue, Joe Castro, and Jordan Parker for their helpful reviews. David Teitlebaum, 3D expert from the WPF team, deserves many thanks for agreeing to update the great 3D chapter originally written by Daniel Lehenbauer. Having Daniel's and David's perspectives and advice captured on paper is a huge benefit for any readers thinking about dabbling in 3D.

I'd also like to thank (in alphabetical order): Chris Brumme, Eileen Chan, Brian Chapman, Beatriz de Oliveira Costa, Joe Duffy, Ifeanyi Echeruo, Dan Glick, Neil Kronlage, Rico Mariani, Mike Mueller, Oleg Ovetchkine, Lori Pearce, S. Ramini, Rob Relyea, Tim Rice, Ben Ronco, Eric Rudder, Adam Smith, Tim Sneath, David Treadwell, and Paramesh Vaidyanathan.

I'd like to thank the folks at Sams—especially Neil Rowe and Betsy Gratner, who are always a pleasure to work with. I couldn't have asked for a better publishing team. Never once was I told that my content was too long or too short or too different from a typical *Unleashed* title. They gave me the complete freedom to write the kind of book I wanted to write.

I'd like to thank my mom, dad, and brother for opening my eyes to the world of computer programming when I was in elementary school. If you have children, please expose them to the magic of writing software while they're still young enough to care about what you have to say!

Finally, I thank *you* for picking up a copy of this book and reading at least this far! I hope you continue reading and find the journey of exploring WPF 4.5 as fascinating as I have!

We Want to Hear from You!

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

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

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

E-mail: feedback@samspublishing.com

Mail: Neil Rowe
Executive Editor
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

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

This page intentionally left blank

Introduction

Thank you for picking up *WPF 4.5 Unleashed*! Windows Presentation Foundation (WPF) is Microsoft's premier technology for creating Windows desktop apps, whether they consist of plain forms, document-centric windows, animated cartoons, videos, immersive 3D environments, or all of the above. WPF is a technology that makes it easier than ever to create a broad range of applications. It's also the basis for XAML-based Windows Store apps.

Ever since WPF was publicly announced ten years ago (with the code name "Avalon"), it has gotten considerable attention for the ways in which it revolutionizes the process of creating software—especially for Windows programmers used to Windows Forms and GDI. It's relatively easy to create fun, useful, and shareable WPF samples that demonstrate all kinds of techniques that are difficult to accomplish in other technologies. WPF 4.5, released in August 2012, continues to improve on previous versions of WPF in many different dimensions.

WPF is quite a departure from previous technologies in terms of its programming model, underlying concepts, and basic terminology. Even viewing the source code for WPF (by cracking open its components with a tool such as .NET Reflector) is a confusing experience because the code you're looking for often doesn't reside where you'd expect to find it. When you combine all this with the fact that there are often several ways to accomplish any task in WPF, you arrive at a conclusion shared by many: *WPF has a very steep learning curve*.

That's where this book comes in. As WPF was developed, it was obvious that there would be no shortage of WPF books in the marketplace. But it wasn't clear to me that the books would have the right balance to guide people through the technology and its unique concepts while showing practical ways to exploit it. Therefore, I wrote the first edition of this book, *Windows Presentation Foundation Unleashed*, with the following goals in mind:

- To provide a solid grounding in the underlying concepts, in a practical and approachable fashion
- To answer the questions most people have when learning the technology and to show how commonly desired tasks are accomplished
- To be an authoritative source, thanks to input from members of the WPF team who designed, implemented, and tested the technology
- To be clear about where the technology falls short rather than selling the technology as the answer to all problems
- To be an easily navigated reference that you can constantly come back to

The first two editions of this book were far more successful than I ever imagined they would be. Now, more than six years after the first edition, I believe that this book accomplishes all the same goals but with even more depth and in the context of the modern

Windows experience that includes Windows Store apps and a different design aesthetic. Whether you're new to WPF or a long-time WPF developer, I hope you find this book to exhibit all these attributes.

Who Should Read This Book?

This book is for software developers who are interested in creating user interfaces for the Windows desktop. Regardless of whether you're creating line-of-business applications, consumer-facing applications, or reusable controls, this book contains a lot of content that helps you get the most out of the platform. It's designed to be understandable even for folks who are new to the .NET Framework. And if you are already well versed in WPF, I'm confident that this book still has information for you. At the very least, it should be an invaluable reference for your bookshelf.

Because the technology and concepts behind WPF are the same ones behind Silverlight and XAML-based Windows Store apps, reading this book can also make you a better developer for Windows Phone and the Windows Store.

Although this book's content is not optimized for graphic designers, reading this book can be a great way to understand more of the "guts" behind a product like Blend for Visual Studio.

To summarize, this book does the following:

- Covers everything you need to know about Extensible Application Markup Language (XAML), the XML-based language for creating declarative user interfaces that can be easily restyled
- Examines the WPF feature areas in incredible depth: controls, layout, resources, data binding, styling, graphics, animation, and more
- Delves into topics that aren't covered by most books: 3D, speech, audio/video, documents, effects, and more
- Shows how to create popular user interface elements and leverage built-in controls such as the new Office-style Ribbon
- Demonstrates how to create sophisticated user interface mechanisms, such as Visual Studio–like collapsible/dockable panes
- Explains how to develop and deploy all types of applications, including navigation-based applications, applications hosted in a web browser, and applications with great-looking nonrectangular windows
- Explains how to create first-class custom controls for WPF
- Demonstrates how to create hybrid WPF software that leverages Windows Forms, DirectX, ActiveX, or other non-WPF technologies
- Explains how to exploit desktop features in WPF applications, such as Jump Lists and taskbar customizations, and the same toast notifications used by Windows Store apps

This book doesn't cover every last bit of WPF. (In particular, XML Paper Specification [XPS] documents, which never really took off, are given only a small bit of attention.) WPF's surface area is so large that I don't believe any single book can. But I think you'll be pleased with the breadth and depth achieved by this book.

Examples in this book appear in XAML and C#, plus C++/CLI for interoperability discussions. XAML is used heavily for a number of reasons: It's often the most concise way to express source code, it can often be pasted into lightweight tools to see instant results without any compilation, WPF-based tools generate XAML rather than procedural code, and XAML is applicable no matter what .NET language you use, such as Visual Basic instead of C#. Whenever the mapping between XAML and a language such as C# is not obvious, examples are shown in both representations.

Software Requirements

This book targets version 4.5 of Windows Presentation Foundation, the corresponding Windows SDK, and Visual Studio 2012.

The following software is required:

- A version of Windows that supports the .NET Framework 4.5.
- The .NET Framework 4.5, which is installed by default starting with Windows 8. For earlier versions of Windows, you can download the .NET Framework 4.5 for free from http://msdn.com.

In addition, the following software is recommended:

- The Windows Software Development Kit (SDK), specifically the .NET tools it includes. This is also a free download from http://msdn.com.
- Visual Studio 2012 or later, which can be a free Express edition downloaded from http://msdn.com.

If you want additional tool support for WPF-based graphic design, Blend for Visual Studio can be extremely helpful.

A few examples are specific to features introduced in Windows Vista, Windows 7, and Windows 8. Some examples require a touchscreen (or an equivalent touch digitizer). The rest of the book applies equally to all relevant versions of Windows.

Code Examples

The source code for examples in this book can be downloaded from http://informit.com/title/9780672336973 or http://adamnathan.net/wpf.

How This Book Is Organized

This book is arranged into six main parts, representing the progression of feature areas that you typically need to understand to use WPF effectively. But if you're dying to jump ahead and learn about a topic such as 3D or animation, the book is set up to allow for nonlinear journeys as well. The following sections provide a summary of each part.

Part I: Background

This part includes the following chapters:

- Chapter 1: Why WPF?
- Chapter 2: XAML Demystified
- Chapter 3: WPF Fundamentals

Chapter 1 introduces WPF by comparing it to alternative technologies and helping you make decisions about when WPF is appropriate for your needs. Chapter 2 explores XAML in great depth, giving you the foundation to understand the XAML you'll encounter in the rest of the book and in real life. Chapter 3 highlights the most unique pieces of WPF's programming model above and beyond what .NET programmers already understand.

Part II: Building a WPF Application

This part includes the following chapters:

- Chapter 4: Sizing, Positioning, and Transforming Elements
- Chapter 5: Layout with Panels
- Chapter 6: Input Events: Keyboard, Mouse, Stylus, and Touch
- Chapter 7: Structuring and Deploying an Application
- Chapter 8: Exploiting Windows Desktop Features

Part II equips you with the knowledge to assemble and deploy a traditional-looking application (although some fancier effects, such as transforms and nonrectangular windows, are also covered). Chapters 4 and 5 discuss arranging controls (and other elements) in a user interface. Chapter 6 covers input events, including support for engaging touch user interfaces. Chapter 7 examines several different ways to package and deploy WPF-based user interfaces to make complete applications. Chapter 8 ends this part by showing slick ways to exploit features in the Windows desktop that can help make your application look modern.

Part III: Controls

This part includes the following chapters:

- Chapter 9: Content Controls
- Chapter 10: Items Controls
- Chapter 11: Images, Text, and Other Controls

Part III provides a tour of controls built into WPF. There are many that you'd expect to have available, plus several that you might not expect. Two categories of controls—content controls (Chapter 9) and items controls (Chapter 10)—are important and deep enough topics to merit their own chapters. The rest of the controls are examined in Chapter 11.

Part IV: Features for Professional Developers

This part includes the following chapters:

- Chapter 12: Resources
- Chapter 13: Data Binding
- Chapter 14: Styles, Templates, Skins, and Themes

The features covered in Part IV are not always necessary to use in WPF applications, but they can greatly enhance the development process. Therefore, they are indispensable for professional developers who are serious about creating maintainable and robust applications or components. These topics are less about the results visible to end users than they are about the best practices for accomplishing these results.

Part V: Rich Media

This part includes the following chapters:

- Chapter 15: 2D Graphics
- Chapter 16: 3D Graphics
- Chapter 17: Animation
- Chapter 18: Audio, Video, and Speech

This part of the book covers the features in WPF that typically get the most attention. The support for 2D and 3D graphics, animation, video, and more enable you to create a stunning experience. These features—and the way they are exposed—set WPF apart from previous systems. WPF lowers the barrier to incorporating such content in your software, so you might try some of these features that you never would have dared to try in the past!

Part VI: Advanced Topics

This part includes the following chapters:

- Chapter 19: Interoperability with Non-WPF Technologies
- Chapter 20: User Controls and Custom Controls
- Chapter 21: Layout with Custom Panels
- Chapter 22: Toast Notifications

The topics covered in Part VI are relevant for advanced application developers, or developers of WPF-based controls. The fact that existing WPF controls can be radically restyled greatly reduces the need for creating custom controls. The final chapter is especially interesting because it enables your WPF apps to use a feature designed for Windows Store apps.

Conventions Used in This Book

Various typefaces in this book identify new terms and other special items. These typefaces include the following:

Typeface	Meaning
Italic	Italic is used for new terms or phrases when they are initially defined and occasionally for emphasis.
`Monospace`	`Monospace` is used for screen messages, code listings, and command samples, as well as filenames. In code listings, *`italic monospace type`* is used for placeholder text. Code listings are colorized similar to the way they are colorized in Visual Studio. `Blue monospace type` is used for XML elements and C#/C++ keywords, `brown monospace type` is used for XML element names and C#/C++ strings, `green monospace type` is used for comments, `red monospace type` is used for XML attributes, and `teal monospace type` is used for type names in C# and C++.

Throughout this book, you'll find a number of sidebar elements:

FAQ

What is a FAQ sidebar?

A FAQ sidebar presents a question readers might have regarding the subject matter in a particular spot in the book—and then provides a concise answer.

DIGGING DEEPER

A Digging Deeper sidebar presents advanced or more detailed information on a subject than is provided in the surrounding text. Think of Digging Deeper material as stuff you can look into if you're curious but can ignore if you're not.

TIP

A tip is a bit of information that can help you in a real-world situation. Tips often offer shortcuts or alternative approaches to produce better results or to make a task easier or quicker.

WARNING

A warning alerts you to an action or a condition that can lead to an unexpected or unpredictable result—and then tells you how to avoid it.

CHAPTER 1

Why WPF?

IN THIS CHAPTER

- A Look at the Past
- Enter WPF
- The Evolution of WPF

In movies and on TV, the main characters are typically an exaggeration of the people you encounter in real life. They're more attractive, they react more quickly, and they somehow always know exactly what to do. The same could be said about the software they use.

This first struck me back in 1994 when watching the movie *Disclosure*, starring Michael Douglas, Demi Moore, and an email program that looks nothing like Microsoft Outlook! Throughout the movie, we're treated to various visual features of the program: a spinning three-dimensional "e," messages that unfold when you open them and crumple when you delete them, hints of inking support, and slick animations when you print messages. (The email program isn't even the most unrealistic software in the movie. I'll just say "virtual reality database" and leave it at that.)

Usability issues aside, Hollywood has been telling us for a long time that software in the real world isn't as compelling as it should be. You can probably think of several examples on your own of TV shows and movies with comically unrealistic software. In recent years, Hollywood software and real software are not so different from each other. A large part of this is due to movies and TV shows doing a better job of portraying software. But at the same time, software has gotten a lot more attractive over the past decade. Just take a moment to reflect on the experience of using a smartphone, Xbox, or Windows Store apps. These user interfaces are a lot different than what we used to see on Windows XP! Users have increasing expectations for the experience of using software, and companies are spending a great deal of time and money on user interfaces that differentiate themselves from the competition. This isn't limited to consumer-facing software; even business applications and internal tools can greatly benefit from a polished user interface.

With higher demands placed on user interfaces, traditional software development processes and technologies often fall short. Modern software usually needs to support rapid iteration and major user interface changes throughout the process—whether such changes are driven by professional graphic designers, developers with a knack for designing user interfaces, or a boss who wants the product to be more "shiny" and animated. For this to be successful, you need technology and tools that make it natural to separate the user interface from the rest of the implementation as much as possible and to decouple visual behavior from the underlying program logic. Developers should be able to create a fully functional "ugly" application that designers can directly retheme without requiring developers to translate their artwork. The Win32 style of programming, in which controls directly contain code to paint and repaint themselves, makes rapid user interface iteration far too difficult for most projects.

More importantly, the classic style of constructing user interfaces has issues on modern PCs. I recently got a new convertible laptop/tablet with a beautiful 12.5-inch 1080p screen. However, the 1920x1080 native resolution on such a small screen results in incredibly small text and other UI elements. I need to scale up the desktop (in other words, increase the dots-per-inch, or DPI) to at least 150% to make it comfortable to use. (From talking to others, I know it's not just me and my aging eyes!) And scaling the desktop can easily expose bugs in Windows desktop apps, or just make them pretty ugly. To be successful on screens such as this, desktop apps need to naturally scale to a wide variety of situations, just like Windows Store apps.

In 2006, Microsoft released a technology to help people create 21st-century software that meets these high demands: Windows Presentation Foundation (WPF). With the release of WPF 4.5 in 2012, the technology is better than ever at delivering amazing results for just about any kind of software. It is the premier technology for creating Windows desktop apps and shares a lot of the same concepts as creating XAML-based Windows Store apps.

A Look at the Past

The primary technologies behind many Windows-based user interfaces—the graphics device interface (GDI) and USER subsystems—were introduced with Windows 1.0 in 1985. That's almost prehistoric in the world of technology! In the early 1990s, OpenGL (created by Silicon Graphics) became a popular graphics library for doing advanced two-dimensional (2D) and three-dimensional (3D) graphics on both Windows and non-Windows systems. This was leveraged by people creating computer-aided design (CAD) programs, scientific visualization programs, and games. DirectX, a Microsoft technology introduced in 1995, provided a new high-performance alternative for 2D graphics, input, communication, sound, and eventually 3D (introduced with DirectX 2 in 1996).

Over the years, many enhancements have been made to both GDI and DirectX. GDI+, introduced in the Windows XP time frame, tried to improve upon GDI by adding support for features such as alpha blending and gradient brushes. It ended up being slower than GDI due to its complexity and lack of hardware acceleration. DirectX (which, by the way, is the technology behind Xbox) continually comes out with new versions that push the

limits of what can be done with computer graphics. With the introduction of .NET and managed code in 2002, developers were treated to a highly productive model for creating Windows (and web) applications. In this world, Windows Forms (built on top of GDI+) became the primary way a C#, Visual Basic, and (to a lesser degree) C++ developer started to create new user interfaces on Windows. Windows Forms has been a successful and productive technology, but it still has all the fundamental limitations of GDI+ and USER.

So although you could have developed a Windows-based email program with the 3D effects seen in *Disclosure* ever since the mid-1990s with non-GDI technologies (actually, probably mixing DirectX or OpenGL with GDI), such technologies are rarely used in mainstream Windows applications almost two decades later. There are several reasons for this: The hardware required to get a decent experience hasn't been ubiquitous until recently, it has been at least an order of magnitude harder to use alternative technologies, and GDI-based experiences have been considered "good enough."

Graphics hardware continues to get better and cheaper and consumer expectations continue to rise, but until WPF, the difficulty of creating modern user experiences had not been addressed. Some developers would take matters into their own hands to get cooler-looking applications and controls on Windows. A simple example of this is using bitmaps for buttons instead of using the standard button control. These types of customizations can not only be expensive to develop, but they also often produce a flakier experience. Such applications often aren't as accessible as they should be, don't handle high dots-per-inch (DPI) settings very well, and have other visual glitches.

Enter WPF

Microsoft recognized that something brand new was needed that escaped the limitations of GDI+ and USER yet provided the kind of productivity that people enjoy with frameworks like Windows Forms. And with the continual rise of cross-platform applications based on HTML and JavaScript, Windows desperately needed a technology that's as fun and easy to use as these, yet with the power to exploit the ever-increasing capabilities of the local computer. Windows Presentation Foundation (WPF) is the answer for software developers and graphic designers who want to create modern desktop user experiences without having to master several difficult technologies. Although "Presentation" sounds like a lofty term for what I would simply call a user interface, it's probably more appropriate for describing the higher level of visual polish that's expected of today's applications and the wide range of functionality included in WPF!

The highlights of WPF include the following:

- **Broad integration**—Prior to WPF, a Windows developer who wanted to use 3D, video, speech, and rich document viewing in addition to normal 2D graphics and controls would have to learn several independent technologies with a number of inconsistencies and attempt to blend them together without much built-in support. But WPF covers all these areas with a consistent programming model as well as tight integration when each type of media gets composited and rendered. You can apply the same kind of effects consistently across different media types, and many of the techniques you learn in one area apply to all the other areas.

- **Resolution independence**—Imagine a world in which moving to a higher resolution or DPI setting doesn't mean that everything gets smaller; instead, graphics and text simply get crisper! Envision user interfaces that look reasonable on a small tablet as well as on a 60-inch TV! WPF makes this easy and gives you the power to shrink or enlarge elements on the screen independently from the screen's resolution. A lot of this is possible because of WPF's emphasis on vector graphics.
- **Hardware acceleration**—WPF is built on Direct3D, so content in a WPF application—whether 2D or 3D, graphics, or text—is converted to 3D triangles, textures, and other Direct3D objects and then rendered by hardware. This means that WPF applications get the benefits of hardware acceleration for smoother graphics and all-around better performance (due to work being offloaded to graphics processing units [GPUs] instead of central processor units [CPUs]). It also ensures that all WPF applications (not just high-end games) receive benefits from new hardware and drivers, whose advances typically focus on 3D capabilities. But WPF doesn't *require* high-end graphics hardware; it has a software rendering pipeline as well. This enables features not yet supported by hardware, enables high-fidelity printing of any content on the screen, and is used as a fallback mechanism when encountering inadequate hardware resources (such as an outdated graphics card or even a high-end one that has simply run out of GPU resources such as video memory).
- **Declarative programming**—Declarative programming is not unique to WPF, as Win16/Win32 programs have used declarative resource scripts to define the layout of dialog boxes and menus for over 25 years. And .NET programs of all types often leverage declarative custom attributes plus configuration and resource files based on Extensible Markup Language (XML). But WPF takes declarative programming to the next level with Extensible *Application* Markup Language (XAML; pronounced "Zammel"). The combination of WPF and XAML is similar to using HTML to define a user interface—but with an incredible range of expressiveness. This expressiveness even extends beyond the bounds of user interfaces; WPF uses XAML as a document format, a representation of 3D models, and more. The result is that graphic designers are empowered to contribute directly to the look and feel of applications, as well as some behavior for which you'd typically expect to have to write code. The next chapter examines XAML in depth.
- **Rich composition and customization**—WPF controls can be composed in ways never before seen. You can create a `ComboBox` filled with animated `Buttons` or a `Menu` filled with live video clips! Although these particular customizations might sound horrible, it's important that you don't have to write a bunch of code (or any code!) to customize controls in ways that the control authors never imagined (unlike owner-draw in prior technologies). Along the same lines, WPF makes it quite easy to "skin" applications with radically different looks (covered in Chapter 14, "Styles, Templates, Skins, and Themes").

In short, WPF aims to combine the best attributes of systems such as DirectX (3D and hardware acceleration), Windows Forms (developer productivity), Adobe Flash (powerful animation support), and HTML (declarative markup). With the help of this book, I think you'll find that WPF gives you more productivity, power, and fun than any other technology you've worked with in the past!

DIGGING DEEPER

GDI and Hardware Acceleration

GDI is actually hardware accelerated on Windows XP. The video driver model explicitly supported accelerating common GDI operations. Windows Vista introduced a new video driver model that does not hardware accelerate GDI primitives. Instead, it uses a "canonical display device" software implementation of the legacy video driver for GDI. However, Windows 7 reintroduced partial hardware acceleration for GDI primitives.

FAQ

Does WPF enable me to do something that I couldn't have previously done?

Technically, the answer is "No," just like C# and the .NET Framework don't enable you to do something that you couldn't do in assembly code. It's just a question of how much work you want to do to get the desired results!

If you were to attempt to build a WPF-equivalent application from scratch without WPF, you'd not only have to worry about the drawing of pixels on the screen and interaction with input devices, you'd also need to do a ton of additional work to get the accessibility and localization support that's built in to WPF, and so on. WPF also provides the easiest way to take advantage of compelling desktop features, such as defining Jump List items with a small chunk of XAML (see Chapter 8, "Exploiting Windows Desktop Features").

So I think most people would agree that the answer is "Yes" when you factor time and money into the equation!

FAQ

When should I use DirectX instead of WPF?

DirectX is more appropriate than WPF for advanced developers writing hard-core "twitch games" or applications with complex 3D models where you need maximum performance. That said, it's easy to write a naive DirectX application that performs far worse than a similar WPF application.

DirectX is a low-level interface to the graphics hardware that exposes all the quirks of whatever GPU a particular computer has. DirectX can be thought of as assembly language in the world of graphics: You can do anything the GPU supports, but it's up to you (the application author) to support all the hardware variations. This is onerous, but such low-level hardware access enables skilled developers to make their own tradeoffs between fine-grained quality and speed. In addition, DirectX exposes cutting-edge features of GPUs as they emerge more quickly than they appear in WPF.

Continued

In contrast, WPF provides a high-level abstraction that takes a description of a scene and figures out the best way to render it, given the hardware resources available. (It's a *retained mode* system rather than an *immediate mode* system.) 2D is the primary focus of WPF; its 3D support is focused on data visualization scenarios and integration with 2D rather than supporting the full power of DirectX.

The downside of choosing DirectX over WPF is a potentially astronomical increase in development cost. A large part of this cost is the requirement to test an application on each driver/GPU combination you intend to support. One of the major benefits of building on top of the WPF is that Microsoft has already done this testing for you! You can instead focus your testing on low-end hardware for measuring performance. The fact that WPF applications can even leverage the client GPU in a partial-trust environment is also a compelling differentiator.

Note that you are able to use both DirectX and WPF in the same application. Chapter 19, "Interoperability with Non-WPF Technologies," shows how this can be done.

The Evolution of WPF

Oddly enough, WPF 4.5 is the fifth major release of WPF. It's odd because the first release had the version number 3.0! The first release in November 2006 was called WPF 3.0 because it shipped as part of the .NET Framework 3.0. The second release—WPF 3.5—came almost exactly a year later (one day shy, in fact). The third release, once again, came almost a year later (in August 2008). This release was a part of Service Pack 1 (SP1) for .NET 3.5, but this was no ordinary service pack as far as WPF was concerned—it contained many new features and improvements.

In addition to these major releases, Microsoft introduced a "WPF Toolkit" in August 2008 at http://wpf.codeplex.com that, along with miscellaneous tools and samples, gets updated periodically. The WPF Toolkit has been used as a way to ship features more quickly and in an experimental form (often with full source code). Features introduced in the WPF Toolkit often "graduate" to get included in a future release of WPF, based on customer feedback about their desirability and readiness.

When the first version of WPF was released, tool support was almost nonexistent. The following months brought primitive WPF extensions for Visual Studio 2005 and the first public preview release of Expression Blend (now called Blend for Visual Studio). Now, Visual Studio 2012 not only has first-class support for WPF development but is largely a WPF application itself! Blend, an application built 100% with WPF, has also gained a lot of functionality for designing and prototyping great user interfaces. And in the past several years, numerous WPF-based applications have been released from companies such as Autodesk, SAP, Disney, Blockbuster, Roxio, AMD, Hewlett Packard, Lenovo, and many more. Microsoft itself, of course, has a long list of software built with WPF (Visual Studio, Blend, Test and Lab Manager, Deep Zoom Composer, Songsmith, Surface, Semblio, Robotics Studio, LifeCam, Amalga, Games for Windows LIVE Marketplace, Office

TIP

To inspect the WPF elements used in any WPF-based application, you can use the Snoop tool available from http://snoopwpf.codeplex.com.

Communicator Attendant, Active Directory Administrative Center, Dynamics NAV, Pivot, PowerShell ISE, and many more).

Let's take a closer look at how WPF has changed over time.

Enhancements in WPF 3.5 and WPF 3.5 SP1

The following notable changes were made to WPF in versions 3.5 and 3.5 SP1:

- **Interactive 3D**—The worlds of 2D and 3D were woven together even more seamlessly with the `UIElement3D` base class, which gives 3D elements input, focus, and events; the odd-sounding `Viewport2DVisual3D` class, which can place any interactive 2D controls inside a 3D scene; and more. See Chapter 16, "3D Graphics."
- **First-class interoperability with DirectX**—Previously, WPF applications could only interoperate with DirectX via the lowest common denominator of Win32. Now, WPF has functionality for interacting directly with Direct3D surfaces with the `D3DImage` class rather than being forced to interact with its host HWND. One benefit from this is the ability to place WPF content on top of DirectX content and vice versa. See Chapter 19.
- **Better data binding**—WPF gained support for XLINQ binding, better validation and debugging, and output string formatting in XAML that reduces the need for custom procedural code. See Chapter 13, "Data Binding."
- **Better special effects**—The first version of WPF shipped with a handful of bitmap effects (blur, drop shadow, outer glow, emboss, and bevel) but with a warning to not use them because their performance was so poor! This has changed, with a new set of hardware-accelerated effects and a whole new architecture that allows you to plug in your own custom hardware-accelerated effects via pixel shaders. See Chapter 15, "2D Graphics."
- **High-performance custom drawing**—WPF didn't previously have a good answer for custom drawings that involve thousands of points or shapes, as even the lowest-level drawing primitives have too much overhead to make such things perform well. The `WriteableBitmap` class was enhanced so you can now specify dirty regions when drawing on it rather than getting a whole new bitmap every frame! Because `WriteableBitmap` only lets you set pixels, it is a very primitive form of "drawing," however.
- **Text improvements**—There's now better performance, better international support (improved input method editor [IME] support and improved Indic script support), and enhancements to `TextBox` and `RichTextBox`. See Chapter 11, "Images, Text, and Other Controls."
- **Enhancements to partial-trust applications**—More functionality became available in the partial-trust sandbox for .NET applications, such as the ability to use Windows Communication Foundation (WCF) for web service calls (via `basicHttpBinding`) and the ability to read and write HTTP cookies. Also, support for XAML Browser Applications (XBAPs)—the primary mechanism for running partial-trust

WPF applications—was extended to the Firefox web browser instead of just Internet Explorer (In WPF, however, the add-on that enables this is no longer installed by default.)

- **Improved deployment for applications and the .NET Framework**—This arrived in many forms: a smaller and faster .NET Framework installation process thanks to the beginnings of a .NET Framework "client profile" that excludes server-only .NET pieces such as ASP.NET; a new "bootstrapper" component that handles all .NET Framework dependencies, installations, and upgrades for you as well as enabling setups with custom branding; and a variety of new ClickOnce features.
- **Improved performance**—WPF and the underlying common language runtime implemented several changes that significantly boosted the performance of WPF applications without any code changes needed. For example, the load time (especially first-time load) has been dramatically improved, animations (especially slow ones) are much smoother, data binding is faster in a number of scenarios, and layered windows (described in Chapter 8) are now hardware accelerated. Other performance improvements were made that you must opt into due to compatibility constraints, such as improved virtualization and deferred scrolling in items controls, described in Chapter 10, "Items Controls."

Enhancements in WPF 4

WPF 4 brings the following changes, on top of the changes from previous versions:

- **Multi-touch support**—When running on computers that support multi-touch and run Windows 7 or later, WPF elements can get a variety of input events, from low-level data, to easy-to-consume manipulations (such as rotation and scaling), to high-level—including custom—gestures. The built-in WPF controls have also been updated to be multi-touch aware. See Chapter 6, "Input Events: Keyboard, Mouse, Stylus, and Touch."
- **First-class support for other Windows 7 features**—WPF provides the best way to integrate with taskbar features such as Jump Lists and icon overlays, integrate with the latest common dialogs, and more. See Chapter 8.
- **New controls**—WPF 4 includes controls such as `DataGrid`, `Calendar`, and `DatePicker`, which originally debuted in the WPF Toolkit. See Chapter 11.
- **Easing animation functions**—Eleven new animation classes such as `BounceEase`, `ElasticEase`, and `SineEase` enable sophisticated animations with custom rates of acceleration and deceleration to be performed completely declaratively. These "easing functions" and their infrastructure were first introduced in Silverlight 3 before being adopted by WPF 4.
- **Enhanced styling with Visual State Manager**—The Visual State Manager, originally introduced in Silverlight 2, provides a new way to organize visuals and their interactivity into "visual states" and "state transitions." This feature makes it easier for designers to work with controls in tools such as Blend.

- **Improved layout on pixel boundaries**—WPF straddles the line between being automatically DPI independent (which requires ignoring physical pixel boundaries) and having visual elements that look crisp (which, especially for small elements, requires being aligned on pixel boundaries). From the beginning, WPF has supported a property called `SnapsToDevicePixels` that forces "pixel snapping" on elements. But using `SnapsToDevicePixels` can be complex and doesn't help in some scenarios. Silverlight went back to the drawing board and created a property called `UseLayoutRounding` that works more naturally. WPF 4 then added this property. Just set it to `true` on a root element, and the positions of that element plus all of children will be rounded up or down to lie on pixel boundaries. The result is user interfaces that can scale *and* can easily be crisp!
- **Non-blurry text**—WPF's emphasis on DPI independence and a scalable user interface has been an issue for small text—the kind of text that occurs a lot in traditional user interfaces on 96-DPI screens. This has frustrated numerous users and developers. WPF 4 addressed this with an alternative way to render text that can make it look as crisp as GDI-based text yet with almost all the benefits that WPF brings. Visual Studio, for example, uses this rendering mode for its text documents. Because there are some limitations to the new rendering approach, you must opt into it. See Chapter 11.
- **More deployment improvements**—The .NET Framework client profile can run side-by-side with the full .NET Framework, and it can be used in just about every scenario relevant for WPF applications.
- **More performance improvements**—In order to make vector graphics perform as well as possible, WPF can cache rendered results as bitmaps and reuse them. For advanced scenarios, you can control this behavior with the `CacheMode` property. See Chapter 15. The heavy usage of WPF in Visual Studio drove a lot of miscellaneous performance improvements into WPF 4 across the board, but all WPF applications get to enjoy these improvements.

Enhancements in WPF 4.5

WPF 4.5 brings the following changes, on top of the changes from previous versions:

- **A new ribbon control**—This matches a lot of sophisticated functionality made popular by Microsoft Office, including a Quick Access Toolbar that can integrate into the window chrome.
- **Many new data-binding features**—You can now data bind to static properties and custom types that implement `ICustomTypeProvider`, automatically resort/regroup/refilter changing data, throttle data source updates, support asynchronous validation, and more. See Chapter 13.
- **Additional support for events**—You can now attach an event in XAML with a markup extension (useful for third parties only, as WPF doesn't have any relevant ones built-in), and there's now more support for the weak event pattern. See Chapter 6.

- **More performance improvements**—There are not just internal performance improvements, but also ways for you to write more efficient code. For example, data collections can now be accessed by background threads.
- **A number of miscellaneous new APIs**—These are mentioned throughout the book.

FAQ

Are there any differences with WPF, depending on the version of Windows?

WPF exposes APIs that are relevant only for Windows 7 and later, such as multi-touch functionality and various features described in Chapter 8. Besides that, WPF has a few behavioral differences when running on Windows XP (the oldest version of Windows that WPF supports). For example, 3D objects do not get antialiased.

And, of course, WPF controls have different default themes to match their host operating system (Aero2 on Windows 8 versus Luna on Windows XP).

Windows XP also has an older driver model that can negatively impact WPF applications. The driver model in later versions of Windows virtualizes and schedules GPU resources, making a system perform better when multiple GPU-intensive programs are running. Running multiple WPF or DirectX applications might bog down a Windows XP system but shouldn't cause performance issues on more recent versions of Windows.

Summary

As time passes, more software is delivering high-quality—sometimes *cinematic*—experiences, and software that doesn't risks looking old-fashioned. However, the effort involved in creating such user interfaces—especially ones that exploit Windows—has been far too difficult in the past.

WPF makes it easier than ever before to create all kinds of user interfaces, whether you want to create a traditional-looking Windows application or an immersive 3D experience worthy of a role in a summer blockbuster. Such a rich user interface can be evolved fairly independently from the rest of an application, allowing graphic designers to participate in the software development process much more effectively. But don't just take my word for it; read on to see for yourself how it's done!

CHAPTER 2

XAML Demystified

IN THIS CHAPTER

- XAML Defined
- Elements and Attributes
- Namespaces
- Property Elements
- Type Converters
- Markup Extensions
- Children of Object Elements
- Mixing XAML with Procedural Code
- XAML2009
- XAML Keywords

Throughout .NET technologies, XML is used to expose functionality in a transparent and declarative fashion. XAML, a dialect of XML, has been especially important since its introduction with the first version of WPF in 2006. It is often misunderstood to be just a way to specify user interfaces, much like HTML. Most of the time, XAML is used to describe user interfaces, but it can describe other things as well. By the end of this chapter, you will see that XAML is about much more than arranging controls on a computer screen.

The point of XAML is to make it easy for programmers to work together with experts in other fields. XAML becomes the common language spoken by all parties, most likely via development tools and field-specific design tools. But because XAML (and XML in general) is generally human readable, people can participate in this ecosystem armed with nothing more than a tool such as Notepad.

In WPF, the "field experts" are graphic designers, who can use a design tool such as Blend to create a slick user interface while developers independently write code. What enables the developer/designer cooperation is not just the common language of XAML but the fact that great care went into making functionality exposed by the relevant APIs accessible declaratively. This gives design tools a wide range of expressiveness (such as specifying complex animations or state changes) without having to worry about generating procedural code.

Even if you have no plans to work with graphic designers, you should still become familiar with XAML for the following reasons:

- XAML can be a very concise way to represent user interfaces or other hierarchies of objects.
- The use of XAML encourages a separation of front-end appearance and back-end logic, which is helpful for maintenance even if you're only a team of one.
- XAML can often be easily pasted into tools such as Visual Studio, Blend, or small standalone tools to see results without any compilation.
- XAML is the language that almost all WPF-related tools emit.

This chapter jumps right into the mechanics of XAML, examining its syntax in depth and showing how it relates to procedural code. Unlike the preceding chapter, this is a fairly deep dive! Having this background knowledge before proceeding with the rest of the book will not only help you understand the code examples but give you better insight into why the APIs in each feature area were designed the way they were. This perspective can be helpful whether you are building WPF applications or controls, designing class libraries that you want to be XAML friendly, or building tools that consume and/or produce XAML (such as validation tools, localization tools, file format converters, designers, and so on).

TIP

There are several ways to run the XAML examples in this chapter, which you can download in electronic form with the rest of this book's source code. For example, you can do the following:

- Save the content in a `.xaml` file and open it inside Internet Explorer (in Windows Vista or later, or in Windows XP with the .NET Framework 3.0 or later installed). Firefox can also work if you install an add-on. Note: By default your web browser will use the version of WPF installed with the operating system.
- Paste the content into a lightweight tool such as the XAMLPAD2009 sample included with this chapter's source code or Kaxaml (from http://kaxaml.com).
- Create a WPF Visual Studio project and replace the content of the main `Window` or `Page` element with the desired content, which might require some code changes.

Using the first two options gives you a couple great ways to get started and do some experimentation. Mixing XAML with other content in a Visual Studio project is covered at the end of this chapter.

FAQ

What happened to XamlPad?

Earlier versions of the Windows SDK shipped with a simple tool called XamlPad that allows you to type in (or paste) WPF-compatible XAML and see it rendered as a live user interface. Unfortunately, this tool is no longer being shipped due to lack of resources. (Yes, contrary to popular belief, Microsoft does not have unlimited resources!) Fortunately, there are several alternative lightweight tools for quickly experimenting with XAML, including the following:

- **XAMLPAD2009**—A sample in this book's source code. Although it lacks the bells and whistles of the other tools, it provides full source code. Plus, it's the only tool that supports XAML2009 (explained later in this chapter) at the time of writing.
- **Kaxaml**—A slick tool downloadable from http://kaxaml.com, created by Robby Ingebretsen, a former WPF team member.
- **XamlPadX**—A feature-filled tool downloadable from http://blogs.msdn.com/llobo/archive/2008/08/25/xamlpadx-4-0.aspx, created by Lester Lobo, another former WPF team member.
- **XAML Cruncher**—A ClickOnce application available at http://charlespetzold.com/wpf/XamlCruncher/XamlCruncher.application, created by Charles Petzold, prolific author and blogger.

XAML Defined

XAML is a relatively simple and general-purpose declarative programming language suitable for constructing and initializing objects. XAML is just XML, but with a set of rules about its elements and attributes and their mapping to objects, their properties, and the values of those properties (among other things).

Because XAML is just a mechanism for using .NET APIs, attempts to compare it to HTML, Scalable Vector Graphics (SVG), or other domain-specific formats/languages are misguided. XAML consists of rules for how parsers/compilers must treat XML and has some keywords, but it doesn't define any interesting elements by itself. So, talking about XAML without a framework like WPF is like talking about C# without the .NET Framework or the Windows Runtime. That said, Microsoft has formalized the notion of "XAML vocabularies" that define the set of valid elements for a given domain, such as what it means to be a WPF XAML file versus a Silverlight XAML file versus any other type of XAML file.

DIGGING DEEPER

Specifications for XAML and XAML Vocabularies

You can find detailed current and historical specifications for XAML and two XAML vocabularies (WPF and Silverlight) at http://bit.ly/Zuao1X.

The role XAML plays in relation to WPF is often confused, so it's important to reemphasize that WPF and XAML can be used independently from each other. Although XAML was originally designed for WPF, it is used by other technologies as well—even ones that have nothing to do with user interfaces, such as Windows Workflow Foundation (WF) and Windows Communication Foundation (WCF). Because of its general-purpose nature, XAML can be applied to just about any object-oriented technology if you really want it to be. Furthermore, using XAML in WPF projects is optional. Almost everything done with XAML can be done entirely in your favorite .NET procedural language instead. (But note that the reverse is not true.) However, because of the benefits listed at the beginning of the chapter, it's rare to see WPF used in the real world without XAML.

DIGGING DEEPER

XAML Functionality Unavailable in Procedural Code

There are a few things that can be done in XAML that can't be done with procedural code. These are all fairly obscure and are covered in Chapters 12 and 14:

- Creating the full range of templates. Procedural code can create templates using `FrameworkElementFactory`, but the expressiveness of this approach is limited.
- Using `x:Shared="False"` to instruct WPF to return a new instance each time an element is accessed from a resource dictionary.
- Deferred instantiation of items inside of a resource dictionary. This is an important performance optimization, and only available via compiled XAML.

Elements and Attributes

The XAML specification defines rules that map .NET namespaces, types, properties, and events into XML namespaces, elements, and attributes. You can see this by examining the following simple (but complete) XAML file that declares a WPF `Button` and comparing it to the equivalent C# code:

XAML:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Content="OK"/>
```

C#:

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
b.Content = "OK";
```

Although these two snippets are equivalent, you can instantly view the XAML in Internet Explorer and see a live button fill the browser window, as pictured in Figure 2.1, whereas the C# code must be compiled with additional code to be usable.

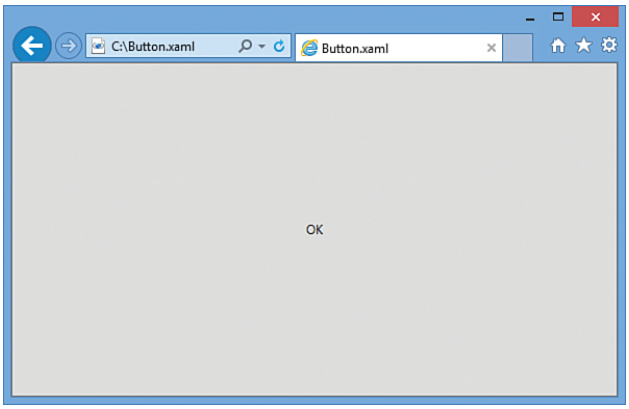

FIGURE 2.1 A simple WPF `Button` declared in a `.xaml` file.

Declaring an XML element in XAML (known as an *object element*) is equivalent to instantiating the corresponding .NET object via a default constructor. Setting an attribute on the object element is equivalent to setting a property of the same name (called a *property attribute*) or hooking up an event handler of the same name (called an *event attribute*). For example, here's an update to the `Button` that not only sets its `Content` property but also attaches an event handler to its `Click` event:

XAML:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Content="OK" Click="button_Click"/>
```

C#:

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
b.Click += new System.Windows.RoutedEventHandler(button_Click);
b.Content = "OK";
```

This requires a method called `button_Click` to be defined somewhere, with the appropriate signature, which means that the XAML file can no longer be rendered standalone, as in Figure 2.1. The "Mixing XAML with Procedural Code" section at the end of this chapter explains how to work with XAML that requires additional code. Note that XAML, like C#, is a case-sensitive language.

DIGGING DEEPER

Order of Property and Event Processing

At runtime, event handlers are always attached *before* any properties are set for any object declared in XAML (excluding the `Name` property, described later in this chapter, which is set immediately after object construction). This enables appropriate events to be raised in response to properties being set without worrying about the order of attributes used in XAML.

The ordering of multiple property sets and multiple event handler attachments is usually performed in the relative order that property attributes and event attributes are specified on the object element. Fortunately, this ordering shouldn't matter in practice because .NET design guidelines dictate that classes should allow properties to be set in any order, and the same holds true for attaching event handlers.

Namespaces

The most mysterious part about comparing the previous XAML examples with the equivalent C# examples is how the XML namespace `http://schemas.microsoft.com/winfx/2006/xaml/presentation` maps to the .NET namespace `System.Windows.Controls`. It turns out that the mapping to this and other WPF namespaces is hard-coded inside the WPF assemblies with several instances of an `XmlnsDefinitionAttribute` custom attribute. (In case you're wondering, no web page exists at the `schemas.microsoft.com` URL—it's just an arbitrary string like any namespace.)

The root object element in a XAML file must specify at least one XML namespace that is used to qualify itself and any child elements. You can declare additional XML namespaces (on the root or on children), but each one must be given a distinct prefix to be used on any identifiers from that namespace. For example, WPF XAML files typically use a second namespace with the prefix `x` (denoted by using `xmlns:x` instead of just `xmlns`):

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

This is the XAML language namespace, which maps to types in the `System.Windows.Markup` namespace but also defines some special directives for the XAML compiler or parser. These directives often appear as attributes to XML elements, so they look like properties of the host element but actually are not. For a list of XAML keywords, see the "XAML Keywords" section later in this chapter.

DIGGING DEEPER

The Implicit .NET Namespaces

WPF maps all the following .NET namespaces from a handful of WPF assemblies to the WPF XML namespace (`http://schemas.microsoft.com/winfx/2006/xaml/presentation`) used throughout this book:

- `System.Windows`
- `System.Windows.Automation`
- `System.Windows.Controls`
- `System.Windows.Controls.Primitives`
- `System.Windows.Data`
- `System.Windows.Documents`
- `System.Windows.Forms.Integration`
- `System.Windows.Ink`
- `System.Windows.Input`
- `System.Windows.Media`
- `System.Windows.Media.Animation`
- `System.Windows.Media.Effects`
- `System.Windows.Media.Imaging`
- `System.Windows.Media.Media3D`
- `System.Windows.Media.TextFormatting`
- `System.Windows.Navigation`
- `System.Windows.Shapes`
- `System.Windows.Shell`

Because this is a many-to-one mapping, the designers of WPF needed to take care not to introduce two classes with the same name, despite the fact that the classes are in separate .NET namespaces.

TIP

Most of the standalone XAML examples in this chapter explicitly specify their namespaces, but in the remainder of the book, most examples assume that the WPF XML namespace (`http://schemas.microsoft.com/winfx/2006/xaml/presentation`) is declared as the primary namespace, and the XAML language namespace (`http://schemas.microsoft.com/winfx/2006/xaml`) is declared as a secondary namespace, with the prefix `x`. If you want to view such content in your web browser or copy it into a lightweight viewer such as the XAMLPAD2009 sample, be sure to add these explicitly.

Using the WPF XML namespace (`http://schemas.microsoft.com/winfx/2006/xaml/presentation`) as a default namespace and the XAML language namespace (`http://schemas.microsoft.com/winfx/2006/xaml`) as a secondary namespace with the prefix `x` is just a convention, just like it's a convention to begin a C# file with a `using System;`

directive. You could instead write the original XAML file as follows, and it would mean the same thing:

```
<WpfNamespace:Button
  xmlns:WpfNamespace="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Content="OK"/>
```

Of course, for readability it makes sense for your most commonly used namespace (also known as the *primary* XML namespace) to be prefix free and to use short prefixes for any additional namespaces.

DIGGING DEEPER

WPF Has Accumulated Multiple WPF XML Namespaces over Time

It's practically a given that real-world WPF XAML will choose to use the WPF XML namespace as the default namespace, but it turns out that more than one XML namespace is mapped to the main WPF types in the various `System.Windows` namespaces.

WPF 3.0 shipped with support for `http://schemas.microsoft.com/winfx/2006/xaml/presentation`, but WPF 3.5 defined a new XML namespace—`http://schemas.microsoft.com/netfx/2007/xaml/presentation`—mapped to the same WPF types. (WinFX was the original name for a set of technologies introduced in the .NET Framework 3.0, including WPF, WCF, and WF. That term was abandoned, hence the change in namespace.) WPF 4.0 once again defined a new XML namespace that is mapped to the same WPF types: `http://schemas.microsoft.com/netfx/2009/xaml/presentation`. (WPF 4.5 did not add a new XML namespace.)

Despite all these options, it is best to stick with the original `http://schemas.microsoft.com/winfx/2006/xaml/presentation` namespace because it works in all versions of WPF. (Whether your *content* works with all versions of WPF is another story, as to do so it must stick to features present only in WPF 3.0.) Note that Windows Store and Silverlight apps also support the `http://schemas.microsoft.com/winfx/2006/xaml/presentation` namespace to make it easier to use XAML meant for WPF inside other project types, although Silverlight also defines its own alternative namespace, `http://schemas.microsoft.com/client/2007`, which is not supported by WPF.

The XML namespaces are confusing. They are *not* schemas. They do *not* represent a closed set of types that were available when the namespace was introduced. Instead, each version of WPF retrofits all previous namespaces with any new assembly/namespace pairs introduced in the new version. Therefore, the `winfx/2006` namespace effectively means "version 3.0 *or later*," the `netfx/2007` namespace means "version 3.5 *or later*," and so on. However, WPF 4.0 accidentally excluded some namespace/assembly pairs from the `netfx/2009` namespace, which was not corrected in WPF 4.5, making the use of omitted types (like `TextOptions`) pretty challenging!

When loose XAML is loaded into Internet Explorer, it is loaded by `PresentationHost.exe`, which decides which version of the .NET Framework to load based on the XML namespaces on the root element. If the `netfx/2009` namespace is present it will load version 4.0 or later, otherwise it will load whichever 3.x version is present. This not only impacts which features are available to your XAML, but also the appearance of controls. In Figure 2.1, for example, the `Button` looks glossy when rendered with an older version of WPF.

Property Elements

The preceding chapter mentioned that rich composition is one of the highlights of WPF. This can be demonstrated with the simple `Button` from Figure 2.1, because you can put arbitrary content inside it; you're not limited to just text! To demonstrate this, the following code embeds a simple square to make a Stop button like what might be found in a media player:

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
System.Windows.Shapes.Rectangle r = new System.Windows.Shapes.Rectangle();
r.Width = 40;
r.Height = 40;
r.Fill = System.Windows.Media.Brushes.Black;
b.Content = r; // Make the square the content of the Button
```

`Button`'s `Content` property is of type `System.Object`, so it can easily be set to the 40x40 `Rectangle` object. The result is pictured in Figure 2.2.

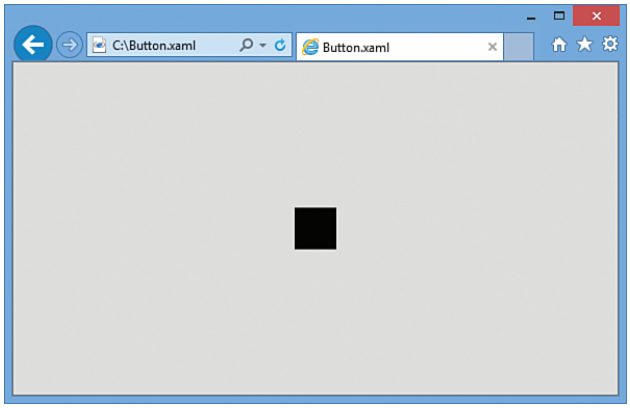

FIGURE 2.2 Updating the WPF `Button` with complex content.

That's pretty neat, but how can you do the same thing in XAML with property attribute syntax? What kind of string could you possibly set `Content` to that is equivalent to the preceding `Rectangle` declared in C#? There is no such string, but XAML fortunately provides an alternative (and more verbose) syntax for setting complex property values: *property elements*. It looks like the following:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<Button.Content>
  <Rectangle Height="40" Width="40" Fill="Black"/>
</Button.Content>
</Button>
```

The `Content` property is now set with an XML element instead of an XML attribute, making it equivalent to the previous C# code. The period in `Button.Content` is what distinguishes property elements from object elements. Property elements always take the form *`TypeName.PropertyName`*, they are always contained inside a *`TypeName`* object element, and they can never have attributes of their own (with one exception—the `x:Uid` attribute used for localization).

Property element syntax can be used for simple property values as well. The following `Button` that sets two properties with attributes (`Content` and `Background`):

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Content="OK" Background="White"/>
```

is equivalent to this `Button`, which sets the same two properties with elements:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<Button.Content>
  OK
</Button.Content>
<Button.Background>
  White
</Button.Background>
</Button>
```

Of course, using attributes when you can is a nice shortcut when hand-typing XAML.

Type Converters

Let's look at the C# code equivalent to the preceding `Button` declaration that sets both `Content` and `Background` properties:

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
b.Content = "OK";
b.Background = System.Windows.Media.Brushes.White;
```

Wait a minute. How can `"White"` in the previous XAML file be equivalent to the static `System.Windows.Media.Brushes.White` field (of type `System.Windows.Media.SolidColorBrush`) in the C# code? Indeed, this example exposes a subtlety with using strings to set properties in XAML that are a different data type than `System.String` or `System.Object`. In such cases, the XAML parser or compiler must look for a *type converter* that knows how to convert the string representation to the desired data type.

WPF provides type converters for many common data types: `Brush`, `Color`, `FontWeight`, `Point`, and so on. They are all classes deriving from `TypeConverter` (`BrushConverter`, `ColorConverter`, and so on). You can also write your own type converters for custom data types. Unlike the XAML language, type converters generally support case-insensitive strings.

Without a type converter for `Brush`, you would have to use property element syntax to set the `Background` in XAML, as follows:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Content="OK">
<Button.Background>
  <SolidColorBrush Color="White"/>
</Button.Background>
</Button>
```

And even that is only possible because of a type converter for `Color` that can make sense of the `"White"` string. If there were no `Color` type converter, you could still write the following:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Content="OK">
<Button.Background>
  <SolidColorBrush>
  <SolidColorBrush.Color>
    <Color A="255" R="255" G="255" B="255"/>
  </SolidColorBrush.Color>
  </SolidColorBrush>
</Button.Background>
</Button>
```

But *this* is only possible because of a type converter that can convert each `"255"` string into a `Byte` value expected by the `A`, `R`, `G`, and `B` properties of the `Color` type. Without this type converter, you would basically be stuck. Type converters don't just enhance the readability of XAML, they also enable values to be expressed that couldn't otherwise be expressed.

DIGGING DEEPER

Using Type Converters in Procedural Code

Although the C# code that sets `Background` to `System.Windows.Media.Brushes.White` produces the same result as the XAML declaration that assigns it to the `"White"` string, it doesn't actually use the same type conversion mechanism employed by the XAML parser or compiler. The following code more accurately represents the runtime retrieval and execution of the appropriate type converter for `Brush`:

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
b.Content = "OK";
b.Background = (Brush)System.ComponentModel.TypeDescriptor.GetConverter(
  typeof(Brush)).ConvertFromInvariantString("White");
```

Unlike in the previous C# code, in this case, misspelling `White` would not cause a compilation error but would cause an exception at runtime, as with XAML. (Although Visual Studio does provide compile-time warnings for mistakes in XAML such as this.)

DIGGING DEEPER

Finding Type Converters

So how does a XAML parser or compiler find an appropriate type converter for a property value? By looking for a `System.ComponentModel.TypeConverterAttribute` custom attribute on the property definition or on the definition of the property's data type.

For example, the `BrushConverter` type converter is used when setting `Button`'s `Background` property in XAML because `Background` is of type `System.Windows.Media.Brush`, which has the following custom attribute:

```
[TypeConverter(typeof(BrushConverter)), …]
public abstract class Brush : …
{
  …
}
```

On the other hand, the `FontSizeConverter` type converter is used when setting `Button`'s `FontSize` property because the property (defined on the base `Control` class) has the following custom attribute:

```
[TypeConverter(typeof(FontSizeConverter)), …]
public double FontSize
{
  get { … }
  set { … }
}
```

In this case, marking the type converter on the property is necessary because its data type (`double`) is too generic to always be associated with `FontSizeConverter`. In fact, in WPF, `double` is often associated with another type converter, `LengthConverter`.

Markup Extensions

Markup extensions, like type converters, enable you to extend the expressiveness of XAML. Both can evaluate a string attribute value at runtime (except for a few built-in markup extensions that are currently evaluated at compile time for performance reasons) and produce an appropriate object based on the string. As with type converters, WPF ships with several markup extensions built in.

Unlike type converters, however, markup extensions are invoked from XAML with explicit and consistent syntax. For this reason, using markup extensions is a preferred approach for extending XAML. In addition, using markup extensions enables you to overcome potential limitations in existing type converters that you don't have the power to change. For example, if you want to set a control's background to a fancy gradient brush with a simple string value, you can write a custom markup extension that supports it even though the built-in `BrushConverter` does not.

Whenever an attribute value is enclosed in curly braces ({}), the XAML compiler/parser treats it as a markup extension value rather than a literal string (or something that needs to be type-converted). The following `Button` uses three different markup extension values with three different properties:

The first identifier in each set of curly braces is the name of the markup extension class, which must derive from a class called `MarkupExtension`. By convention, such classes end with an `Extension` suffix, but you can leave it off when using it in XAML. In this example, `NullExtension` (seen as `x:Null`) and `StaticExtension` (seen as `x:Static`) are classes in the `System.Windows.Markup` namespace, so the `x` prefix must be used to locate them. `Binding` (which doesn't happen to have the `Extension` suffix) is in the `System.Windows.Data` namespace, so it can be found in the default XML namespace.

If a markup extension supports them, comma-delimited parameters can be specified. Positional parameters (such as `SystemParameters.IconHeight` in the example) are treated as string arguments for the extension class's appropriate constructor. Named parameters (`Path` and `RelativeSource` in the example) enable you to set properties with matching names on the constructed extension object. The values for these properties can be markup extension values themselves (using nested curly braces, as done with the value for `RelativeSource`) or literal values that can undergo the normal type conversion process. If you're familiar with .NET custom attributes (the .NET Framework's popular extensibility mechanism), you've probably noticed that the design and usage of markup extensions closely mirrors the design and usage of custom attributes. That is intentional.

In the preceding `Button` declaration, `NullExtension` enables the `Background` brush to be set to `null`, which isn't natively supported by `BrushConverter` (or many other type converters, for that matter). This is just done for demonstration purposes, as a `null` `Background` is not very useful. `StaticExtension` enables the use of static properties, fields, constants, and enumeration values rather than hard-coding literals in XAML. In this case, the `Button`'s `Height` is set to the operating system's current height setting for icons, exposed by the static `IconHeight` property on a `System.Windows.SystemParameters` class. `Binding`, covered in depth in Chapter 13, "Data Binding," enables `Content` to be set to the same value as the `Height` property.

DIGGING DEEPER

Escaping the Curly Braces

If you ever want a property attribute value to be set to a literal string beginning with an open curly brace ({), you must escape it so it doesn't get treated as a markup extension. This can be done by preceding it with an empty pair of curly braces, as in the following example:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        Content="{}{This is not a markup extension!}"/>
```

You can also use a backslash to escape characters such as an open curly brace, a single quote, or a double quote.

Alternatively, you could use property element syntax without any escaping because the curly braces do not have special meaning in this context. The preceding `Button` could be rewritten as follows:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<Button.Content>
  {This is not a markup extension!}
</Button.Content>
</Button>
```

Data binding (covered in Chapter 13) takes advantage of this escaping with string formatting properties that use curly braces as part of their normal string syntax.

Because markup extensions are just classes with default constructors, they can be used with property element syntax. The following `Button` is identical to the preceding one:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
<Button.Background>
  <x:Null/>
</Button.Background>
<Button.Height>
  <x:Static Member="SystemParameters.IconHeight"/>
</Button.Height>
<Button.Content>
  <Binding Path="Height">
  <Binding.RelativeSource>
    <RelativeSource Mode="Self"/>
  </Binding.RelativeSource>
  </Binding>
</Button.Content>
</Button>
```

This transformation works because these markup extensions all have properties corresponding to their parameterized constructor arguments (the positional parameters used with property attribute syntax). For example, `StaticExtension` has a `Member` property that

has the same meaning as the argument that was previously passed to its parameterized constructor, and `RelativeSource` has a `Mode` property that corresponds to its constructor argument.

DIGGING DEEPER

Markup Extensions and Procedural Code

The actual work done by a markup extension is specific to each extension. For example, the following C# code is equivalent to the XAML-based `Button` that uses `NullExtension`, `StaticExtension`, and `Binding`:

```
System.Windows.Controls.Button b = new System.Windows.Controls.Button();
// Set Background:
b.Background = null;
// Set Height:
b.Height = System.Windows.SystemParameters.IconHeight;
// Set Content:
System.Windows.Data.Binding binding = new System.Windows.Data.Binding();
binding.Path = new System.Windows.PropertyPath("Height");
binding.RelativeSource = System.Windows.Data.RelativeSource.Self;
b.SetBinding(System.Windows.Controls.Button.ContentProperty, binding);
```

However, this code doesn't use the same mechanism as the XAML parser or compiler, which rely on each markup extension to set the appropriate values at runtime (essentially by invoking each one's `ProvideValue` method). The procedural code equivalent of this mechanism is often complex, sometimes requiring context that only a parser would have (such as how to resolve an XML namespace prefix that could be used in `StaticExtension`'s `Member`). Fortunately, there is no reason to use markup extensions this way in procedural code!

Children of Object Elements

A XAML file, like all XML files, must have a single root object element. Therefore, it should come as no surprise that object elements can support child object elements (not just property elements, which aren't children, as far as XAML is concerned). An object element can have three types of children: a value for a content property, collection items, or a value that can be type-converted to the object element.

The Content Property

Most WPF classes designate a property (via a custom attribute) that should be set to whatever content is inside the XML element. This property is called the *content property*, and it is really just a convenient shortcut to make the XAML representation more compact. In some ways, these content properties are like the (often-maligned) default properties in old versions of Visual Basic.

Button's Content property is (appropriately) given this special designation, so the following Button:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Content="OK"/>
```

could be rewritten as follows:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  OK
</Button>
```

Or, more usefully, this Button with more complex content:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<Button.Content>
  <Rectangle Height="40" Width="40" Fill="Black"/>
</Button.Content>
</Button>
```

could be rewritten as follows:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <Rectangle Height="40" Width="40" Fill="Black"/>
</Button>
```

There is no requirement that the content property must actually be called Content; classes such as ComboBox, ListBox, and TabControl (also in the System.Windows.Controls namespace) use their Items property as the content property.

Collection Items

XAML enables you to add items to the two main types of collections that support indexing: lists and dictionaries.

Lists

A *list* is any collection that implements System.Collections.IList, such as System.Collections.ArrayList or numerous collection classes defined by WPF. For example, the following XAML adds two items to a ListBox control whose Items property is an ItemCollection that implements IList:

```
<ListBox xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<ListBox.Items>
  <ListBoxItem Content="Item 1"/>
  <ListBoxItem Content="Item 2"/>
</ListBox.Items>
</ListBox>
```

This is equivalent to the following C# code:

```
System.Windows.Controls.ListBox listbox = new System.Windows.Controls.ListBox();
System.Windows.Controls.ListBoxItem item1 =
  new System.Windows.Controls.ListBoxItem();
System.Windows.Controls.ListBoxItem item2 =
  new System.Windows.Controls.ListBoxItem();
item1.Content = "Item 1";
item2.Content = "Item 2";
listbox.Items.Add(item1);
listbox.Items.Add(item2);
```

Furthermore, because `Items` is the content property for `ListBox`, you can shorten the XAML even further, as follows:

```
<ListBox xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <ListBoxItem Content="Item 1"/>
  <ListBoxItem Content="Item 2"/>
</ListBox>
```

In all these cases, the code works because `ListBox`'s `Items` property is automatically initialized to any empty collection object. If a collection property is initially `null` instead (and is read/write, unlike `ListBox`'s read-only `Items` property), you need to wrap the items in an explicit element that instantiates the collection. WPF's built-in controls do not act this way, so an imaginary `OtherListBox` element demonstrates what this could look like:

```
<OtherListBox>
<OtherListBox.Items>
  <ItemCollection>
    <ListBoxItem Content="Item 1"/>
    <ListBoxItem Content="Item 2"/>
  </ItemCollection>
</OtherListBox.Items>
</OtherListBox>
```

Dictionaries

`System.Windows.ResourceDictionary` is a commonly used collection type in WPF that you'll see more of in Chapter 12, "Resources." It implements `System.Collections.IDictionary`, so it supports adding, removing, and enumerating key/value pairs in procedural code, as you would do with a typical hash table. In XAML, you can add key/value pairs to any collection that implements `IDictionary`. For example, the following XAML adds two `Colors` to a `ResourceDictionary`:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Color x:Key="1" A="255" R="255" G="255" B="255"/>
  <Color x:Key="2" A="0" R="0" G="0" B="0"/>
</ResourceDictionary>
```

This leverages the XAML `Key` keyword (defined in the secondary XML namespace), which is processed specially and enables us to attach a key to each `Color` value. (The `Color` type does not define a `Key` property.) Therefore, the XAML is equivalent to the following C# code:

```
System.Windows.ResourceDictionary d = new System.Windows.ResourceDictionary();
System.Windows.Media.Color color1 = new System.Windows.Media.Color();
System.Windows.Media.Color color2 = new System.Windows.Media.Color();
color1.A = 255; color1.R = 255; color1.G = 255; color1.B = 255;
color2.A = 0;   color2.R = 0;   color2.G = 0;   color2.B = 0;
d.Add("1", color1);
d.Add("2", color2);
```

Note that the value specified in XAML with `x:Key` is treated as a string unless a markup extension is used or the XAML2009 parser is used (see the later "XAML2009" section); no type conversion is attempted otherwise.

More Type Conversion

Plain text can often be used as the child of an object element, as in the following XAML declaration of `SolidColorBrush`:

```
<SolidColorBrush>White</SolidColorBrush>
```

This is equivalent to the following:

```
<SolidColorBrush Color="White"/>
```

even though `Color` has not been designated as a content property. In this case, the first XAML snippet works because a type converter exists that can convert strings such as `"White"` (or `"white"` or `"#FFFFFF"`) into a `SolidColorBrush` object.

Although type converters play a huge role in making XAML readable, the downside is that they can make XAML appear a bit "magical," and it can be difficult to understand how it maps to instances of .NET objects. Using what you know so far, it would be reasonable to assume that you can't declare an abstract class element in XAML because there's no way to instantiate it. However, even though `System.Windows.Media.Brush` is an abstract base class for `SolidColorBrush`, `GradientBrush`, and other concrete brushes, you can express the preceding XAML snippets as simply:

```
<Brush>White</Brush>
```

because the type converter for `Brushes` understands that this is still `SolidColorBrush`. This may seem like an unusual feature, but it's important for supporting the ability to express primitive types in XAML, as demonstrated in "The Extensible Part of XAML."

DIGGING DEEPER

Lists, Dictionaries, and the XAML2009 Parser

Although the WPF XAML parser has historically only supported `IList` and `IDictionary` collections, the XAML2009 parser (described in the later "XAML2009" section) supports more. It first looks for `IList` and `IDictionary`, then for `ICollection<T>` and `IDictionary<K,V>`, then for the presence of both `Add` and `GetEnumerator` methods.

DIGGING DEEPER

The Extensible Part of XAML

Because XAML was designed to work with the .NET type system, you can use it with just about any .NET object (or even COM objects, thanks to COM interoperability), including ones you define yourself. It doesn't matter whether these objects have anything to do with a user interface. However, the objects need to be designed in a "declarative-friendly" way. For example, if a class doesn't have a default constructor and doesn't expose useful instance properties, it's not going to be directly usable from XAML (unless you use the XAML2009 parser). A lot of care went into the design of the WPF APIs—above and beyond the usual .NET design guidelines—to fit XAML's declarative model.

The WPF assemblies are marked with `XmlnsDefinitionAttribute` to map their .NET namespaces to XML namespaces in a XAML file, but what about assemblies that weren't designed with XAML in mind and, therefore, don't use this attribute? Their types can still be used; you just need to use a special directive as the XML namespace. For example, here's some plain old C# code using .NET Framework APIs contained in `mscorlib.dll`:

```
System.Collections.Hashtable h = new System.Collections.Hashtable();
h.Add("key1", 7);
h.Add("key2", 23);
```

and here's how it can be represented in XAML:

```
<collections:Hashtable
  xmlns:collections="clr-namespace:System.Collections;assembly=mscorlib"
  xmlns:sys="clr-namespace:System;assembly=mscorlib"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <sys:Int32 x:Key="key1">7</sys:Int32>
  <sys:Int32 x:Key="key2">23</sys:Int32>
</collections:Hashtable>
```

The `clr-namespace` directive enables you to place a .NET namespace directly inside XAML. The assembly specification at the end is necessary only if the desired types don't reside in the same assembly that the XAML is compiled into. Typically the assembly's simple name is used (as with `mscorlib`), but you can use the canonical representation supported by `System.Reflection.Assembly.Load` (although with no spaces allowed), which includes additional information such as a version and/or public key token.

Two key points about this example really highlight the integration with not only the .NET type system but specific types in the .NET Framework:

- Child elements can be added to the parent `Hashtable` with the standard XAML `x:Key` syntax because `Hashtable` and other collection classes in the .NET Framework have implemented the `IDictionary` interface since version 1.0.
- `System.Int32` can be used in this simple fashion because a type converter already exists that supports converting strings to integers. This is because the type converters supported by XAML are simply classes that derive from `System.ComponentModel.TypeConverter`, a class that has also been around since version 1.0 of the .NET Framework. This is the same type conversion mechanism used by Windows Forms (enabling you to type strings into the Visual Studio property grid, for example, and have them converted to the appropriate type).

DIGGING DEEPER

XAML Processing Rules for Object Element Children

You've now seen the three types of children for object elements. To avoid ambiguity, any valid XAML parser or compiler follows these rules when encountering and interpreting child elements:

1. If the type implements `IList`, call `IList.Add` for each child.
2. Otherwise, if the type implements `IDictionary`, call `IDictionary.Add` for each child, using the `x:Key` attribute value for the key and the element for the value. (Although XAML2009 checks `IDictionary` *before* `IList` and supports other collection interfaces, as described earlier.)
3. Otherwise, if the parent supports a content property (indicated by `System.Windows.Markup.ContentPropertyAttribute`) and the type of the child is compatible with that property, treat the child as its value.
4. Otherwise, if the child is plain text and a type converter exists to transform the child into the parent type (*and* no properties are set on the parent element), treat the child as the input to the type converter and use the output as the parent object instance.
5. Otherwise, treat it as unknown content and potentially raise an error.

Rules 1 and 2 enable the behavior described in the earlier "Collection Items" section, rule 3 enables the behavior described in the section "The Content Property," and rule 4 explains the often-confusing behavior described in the "More Type Conversion" section.

Mixing XAML with Procedural Code

WPF applications can be written entirely in procedural code in any .NET language. In addition, certain types of simple applications can be written entirely in XAML, thanks to the data-binding features described in Chapter 13, the triggers introduced in the next chapter, and the fact that loose XAML pages can be rendered in a web browser. However, most WPF applications are a mix of XAML and procedural code. This section covers the two ways that XAML and code can be mixed together.

Loading and Parsing XAML at Runtime

WPF has a runtime XAML parser exposed as two classes in the `System.Windows.Markup` namespace: `XamlReader` and `XamlWriter`. And their APIs couldn't be much simpler. `XamlReader` contains a few overloads of a static `Load` method, and `XamlWriter` contains a few overloads of a static `Save` method. Therefore, programs written in any .NET language can leverage XAML at runtime without much effort. Starting with the .NET Framework 4.0, a new, separate set of XAML readers and writers exists, but with a fair number of caveats. They are not important for this discussion but are covered later in Appendix A, "Fun with XAML Readers and Writers."

`XamlReader`

The set of `XamlReader.Load` methods parse XAML, create the appropriate .NET objects, and return an instance of the root element. So, if a XAML file named `MyWindow.xaml` in the current directory contains a `Window` object (explained in depth in Chapter 7,

"Structuring and Deploying an Application") as its root node, the following code could be used to load and retrieve the `Window` object:

```
Window window = null;
using (FileStream fs =
  new FileStream("MyWindow.xaml", FileMode.Open, FileAccess.Read))
{
  // Get the root element, which we know is a Window
  window = (Window)XamlReader.Load(fs);
}
```

In this case, `Load` is called with a `FileStream` (from the `System.IO` namespace). After `Load` returns, the entire hierarchy of objects in the XAML file is instantiated in memory, so the XAML file is no longer needed. In the preceding code, the `FileStream` is instantly closed by exiting the `using` block. Because `XamlReader` can be passed an arbitrary `Stream` (or `System.Xml.XmlReader`, via a different overload), you have a lot of flexibility in retrieving XAML content.

TIP

`XamlReader` also defines `LoadAsync` instance methods that load and parse XAML content asynchronously. You'll want to use `LoadAsync` to keep a responsive user interface during the loading of large files or files over the network, for example. Accompanying these methods are a `CancelAsync` method for halting the processing and a `LoadCompleted` event for knowing when the processing is complete.

The behavior of `LoadAsync` is a bit odd, however. The work is done on the UI thread via multiple `Dispatcher.BeginInvoke` calls. (WPF tries to break the work up into 200-millisecond chunks.) Furthermore, this asynchronous processing is only used if `x:SynchronousMode="Async"` is set on the root XAML node. If this attribute is not set, `LoadAsync` will silently load the XAML synchronously.

Now that an instance of the root element exists, you can retrieve child elements by making use of the appropriate content properties or collection properties. The following code assumes that the `Window` has a `StackPanel` object as its content, whose fifth child is an OK `Button`:

```
Window window = null;
using (FileStream fs =
  new FileStream("MyWindow.xaml", FileMode.Open, FileAccess.Read))
{
  // Get the root element, which we know is a Window
  window = (Window)XamlReader.Load(fs);
}
// Grab the OK button by walking the children (with hard-coded knowledge!)
StackPanel panel = (StackPanel)window.Content;
Button okButton = (Button)panel.Children[4];
```

With a reference to the Button, you can do whatever you want: Set additional properties (perhaps using logic that is hard or impossible to express in XAML), attach event handlers, or perform additional actions that you can't do from XAML, such as calling its methods.

Of course, the code that uses a hard-coded index and other assumptions about the user interface structure isn't very satisfying, as simple changes to the XAML can break it. Instead, you could write code to process the elements more generically and look for a Button element whose content is an "OK" string, but that would be a lot of work for such a simple task. In addition, if you want the Button to contain graphical content, how can you easily identify it in the presence of multiple Buttons?

Fortunately, XAML supports naming of elements so they can be found and used reliably from procedural code.

Naming XAML Elements

The XAML language namespace has a Name keyword that enables you to give any element a name. For the simple OK button that we're imagining is embedded somewhere inside a Window, the Name keyword can be used as follows:

```
<Button x:Name="okButton">OK</Button>
```

With this in place, you can update the preceding C# code to use Window's FindName method that searches its children (recursively) and returns the desired instance:

```
Window window = null;
using (FileStream fs =
  new FileStream("MyWindow.xaml", FileMode.Open, FileAccess.Read))
{
  // Get the root element, which we know is a Window
  window = (Window)XamlReader.Load(fs);
}
// Grab the OK button, knowing only its name
Button okButton = (Button)window.FindName("okButton");
```

FindName is not unique to Window; it is defined on FrameworkElement and FrameworkContentElement, which are base classes for many important classes in WPF.

DIGGING DEEPER

Naming Elements Without x:Name

The x:Name syntax can be used to name elements, but some classes define their own property that can be treated as the element's name (by marking themselves with System.Windows.Markup.RuntimeNamePropertyAttribute). For example, FrameworkElement and FrameworkContentElement have a Name property, so they mark themselves with RuntimeNameProperty("Name"). This means that on such elements you can simply set the Name property to a string rather than use the x:Name syntax. You can use either mechanism, but you can't use both simultaneously. Having two ways to set a name is a bit confusing, but it's handy for these classes to have a Name property for use by procedural code.

TIP

In all versions of WPF, the `Binding` markup extension can be used to reference a named element as a property value:

```
<StackPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <Label Target="{Binding ElementName=box}" Content="Enter _text:"/>
  <TextBox Name="box"/>
</StackPanel>
```

In this case, assigning the `TextBox` as the `Target` of the `Label` gives it focus when the `Label`'s access key, Alt+T, is pressed. However, WPF 4.0 and later support a simpler markup extension (that finds the element at parse time rather than runtime): `System.Windows.Markup.Reference`. It can be used as follows:

```
<StackPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
            xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Label Target="{x:Reference box}" Content="Enter _text:"/>
  <TextBox Name="box"/>
</StackPanel>
```

Or, when a relevant property is marked with the `System.Windows.Markup.NameReferenceConverter` type converter (as in this case), a simple name string can be implicitly converted into the referenced instance:

```
<StackPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <Label Target="box" Content="Enter _text:"/>
  <TextBox Name="box"/>
</StackPanel>
```

Compiling XAML

Loading and parsing XAML at runtime is interesting for dynamic skinning scenarios or for .NET languages that don't have the necessary support for XAML compilation. Most WPF projects, however, leverage the XAML compilation supported by MSBuild and Visual Studio. XAML compilation involves three things: converting a XAML file into a special binary format, embedding the converted content as a binary resource in the assembly being built, and performing the plumbing that connects XAML with procedural code automatically. C# and Visual Basic are the two languages with the best support for XAML compilation.

DIGGING DEEPER

Supporting Compiled XAML with Any .NET Language

If you want to leverage XAML compilation with an arbitrary .NET language, there are two basic requirements for enabling this: having a corresponding CodeDom provider and having an MSBuild target file. In addition, language support for partial classes is helpful but not strictly required.

If you don't care about mixing procedural code with your XAML file, then all you need to do to compile it is add it to a WPF project in Visual Studio with a **Build Action** of **Page**. (Chapter 7 explains ways to make use of such content in the context of an application.) But for the typical case of compiling a XAML file *and* mixing it with procedural code, the first step is specifying a subclass for the root element in a XAML file. This can be done with the `Class` keyword defined in the XAML language namespace, for example:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="MyNamespace.MyWindow">
  …
</Window>
```

In a separate source file (but in the same project), you can define the subclass and add whatever members you want:

```
namespace MyNamespace
{
  partial class MyWindow : Window
  {
    public MyWindow()
    {
      // Necessary to call in order to load XAML-defined content!
      InitializeComponent();
      …
    }
    Any other members can go here…
  }
}
```

This is often referred to as the *code-behind file*. If you reference any event handlers in XAML (via event attributes such as `Click` on `Button`), this is where they should be defined.

The `partial` keyword in the class definition is important, as the class's implementation is spread across more than one file. If the .NET language doesn't support partial classes (for example, C++/CLI), the XAML file must also use a `Subclass` keyword in the root element, as follows:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="MyNamespace.MyWindow" x:Subclass="MyNamespace.MyWindow2">
  …
</Window>
```

With this change, the XAML file completely defines the `Subclass` (`MyWindow2` in this case) but uses the `Class` in the code-behind file (`MyWindow`) as its base class. Therefore, this

simulates the ability to split the implementation across two files by relying on inheritance.

When creating a WPF-based C# or Visual Basic project in Visual Studio, or when you use **Add New Item...** to add certain WPF items to a project, Visual Studio automatically creates a XAML file with `x:Class` on its root, creates the code-behind source file with the partial class definition, and links the two together so they are built properly.

If you're an MSBuild user and want to understand the contents of the project file that enables code-behind, you can open any of the C# project files included with this book's source code in a simple text editor such as Notepad. The relevant part of a typical project is as follows:

```
<ItemGroup>
  <Page Include="MyWindow.xaml"/>
</ItemGroup>
<ItemGroup>
  <Compile Include="MyWindow.xaml.cs">
    <DependentUpon>MyWindow.xaml</DependentUpon>
    <SubType>Code</SubType>
  </Compile>
</ItemGroup>
```

For such a project, the build system generates several items when processing `MyWindow.xaml`, including these:

- A BAML file (`MyWindow.baml`), which gets embedded in the assembly as a binary resource by default.
- A C# source file (`MyWindow.g.cs`), which gets compiled into the assembly like all other source code.

TIP

`x:Class` can only be used in a XAML file that gets compiled. But you can sometimes compile a XAML file with no `x:Class` just fine. This simply means that there is no corresponding code-behind file, so you can't use any features that rely on the presence of procedural code. Therefore, adding a XAML file to a Visual Studio project without an `x:Class` directive can be a handy way to get the deployment and performance benefits of compiled XAML without having to create an unnecessary code-behind file.

BAML

BAML, which stands for Binary Application Markup Language, is simply XAML that has been parsed, tokenized, and converted into binary form. Although almost any chunk of XAML can be represented by procedural code, the XAML-to-BAML compilation process *does not* generate procedural source code. So, BAML is not like Microsoft intermediate language (MSIL); it is a compressed declarative format that is faster to load and parse (and smaller in size) than plain XAML. BAML is basically an implementation detail of the XAML compilation process. Nevertheless, it's interesting to be aware of its existence. In fact, WPF contains a public BAML reader class (see Appendix A).

DIGGING DEEPER

There Once Was a CAML...

Prerelease versions of WPF had the ability to compile XAML into BAML *or* MSIL. This MSIL output was called CAML, which stood for *Compiled* Application Markup Language. The idea was to enable the choice of optimizing for size (BAML) or speed (CAML). But the team decided not to burden the WPF codebase with these two independent implementations that did essentially the same thing. BAML won out over CAML because it has several advantages: It's less of a security threat than MSIL, it's more compact (resulting in smaller download sizes for web scenarios), and it can be localized postcompilation. Furthermore, using CAML was not appreciably faster than using BAML, as people had theorized it would be. It generated a lot of code that would only ever run once. This is inefficient, it bloats DLLs, it doesn't take advantage of caches, and so on.

Generated Source Code

Some procedural code does get generated in the XAML compilation process (if you use `x:Class`), but it's just some "glue code" similar to what had to be written to load and parse a loose XAML file at runtime. Such files are given a suffix such as `.g.cs` (or `.g.vb`), where the `g` stands for *generated*.

Each generated source file contains a partial class definition for the class specified with `x:Class` on the root object element. This partial class contains a field (internal by default) for every named element in the XAML file, using the element name as the field name. It also contains an `InitializeComponent` method that does the grunt work of loading the embedded BAML resource, assigning the fields to the appropriate instances originally declared in XAML, and hooking up any event handlers (if any event handlers were specified in the XAML file).

Because the glue code tucked away in the generated source file is part of the same class you've defined in the code-behind file (and because BAML gets embedded as a resource), you often don't need to be aware of the existence of BAML or the process of loading and parsing it. You simply write code that references named elements just like any other class member, and you let the build system worry about hooking things together. The only thing you need to remember is to call `InitializeComponent` in your code-behind class's constructor.

WARNING

Don't forget to call `InitializeComponent` in the constructor of your code-behind class!

If you fail to do so, your root element won't contain any of the content you defined in XAML (because the corresponding BAML doesn't get loaded), and all the fields representing named object elements will be `null`.

DIGGING DEEPER

Procedural Code Inside XAML

XAML actually supports an obscure "code-inside" feature in addition to code-behind (somewhat like in ASP.NET). This can be done with the `Code` keyword in the XAML language namespace, as follows:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        x:Class="MyNamespace.MyWindow">
  <Button Click="button_Click">OK</Button>
  <x:Code><![CDATA[
    void button_Click(object sender, RoutedEventArgs e)
    {
        this.Close();
    }
  ]]></x:Code>
</Window>
```

When such a XAML file is compiled, the contents inside the `x:Code` element get plopped inside the partial class in the `.g.cs` file. Note that the procedural language is not specified in the XAML file; it is determined by the project containing this file.

Wrapping the code in `<![CDATA[…]]>` isn't required, but it avoids the need to escape less-than signs as `<` and ampersands as `&`. That's because CDATA sections are ignored by XML parsers, whereas anything else is processed as XML. (The tradeoff is that you must avoid using `]]>` anywhere in the code, because that terminates the CDATA section!)

Of course, there's no good reason to pollute your XAML files with this "code-inside" feature. Besides making the division between user interface and logic messier, loose XAML pages don't support it, and Visual Studio doesn't support any of its typical code features, such as IntelliSense and syntax coloring.

FAQ

Can BAML be decompiled back into XAML?

Sure, because BAML can be converted into a graph of live object instances, and these instances can be serialized as XAML, regardless of how they were originally declared.

The first step is to retrieve an instance that you want to be the root of the XAML. If you don't already have this object, you can call the static `System.Windows.Application.LoadComponent` method to load it from BAML, as follows:

```
System.Uri uri = new System.Uri("/WpfApplication1;component/MyWindow.xaml",
  System.UriKind.Relative);
Window window = (Window)Application.LoadComponent(uri);
```

Continued

Yes, that code is loading BAML despite the `.xaml` suffix. This differs from previous code that uses `FileStream` to load a `.xaml` file because with `LoadComponent`, the name specified as the uniform resource identifier (URI) does not have to physically exist as a standalone `.xaml` file. `LoadComponent` can automatically retrieve BAML embedded as a resource when given the appropriate URI (which, by MSBuild convention, is the name of the original XAML source file). In fact, Visual Studio's autogenerated `InitializeComponent` method calls `Application.LoadComponent` to load embedded BAML, although it uses a different overload. Chapter 12 provides more details about this mechanism of retrieving embedded resources with URIs.

After you've gotten a hold of the root element instance, you can use the `System.Windows.Markup.XamlWriter` class to get a XAML representation of the root element (and, therefore, all its children). `XamlWriter` contains five overloads of a static `Save` method, the simplest of which accepts an object instance and returns appropriate XAML as a string:

```
string xaml = XamlWriter.Save(window);
```

It might sound a little troubling that BAML can be so easily "cracked open," but it's really no different from any other software running locally or displaying a user interface locally. (For example, you can easily dig into a website's HTML, JavaScript, and Cascading Style Sheets [CSS] files.) The popular .NET Reflector tool has a BamlViewer add-in (see http://codeplex.com/reflectoraddins) that displays BAML embedded in any assembly as XAML.

XAML2009

Although XAML is a general-purpose language whose use is broader than that of WPF, WPF's XAML compiler and parsers are architecturally tied to WPF. Therefore, they are not usable by other technologies without taking a dependency on WPF. The .NET Framework 4.0 fixed this by introducing a `System.Xaml` assembly that contains a bunch of functionality for processing XAML. WPF (and WCF and WF) take a dependency on `System.Xaml`—not the other way around.

At the same time, the .NET Framework 4.0 introduced a handful of new features for the XAML language. This second generation of the XAML language is referred to as XAML2009. (To differentiate, the first generation is sometimes referred to as XAML2006.) The `System.Xaml` assembly supports XAML2009, unlike the older APIs (`System.Windows.Markup.XamlReader` and `System.Windows.Markup.XamlWriter` from the previous section), which only support XAML2006.

The XAML2009 features, outlined in this section, are nothing revolutionary but represent a nice set of incremental improvements to XAML. However, don't get too excited; for the most part, these features are not usable in WPF projects because XAML compilation still uses the XAML2006-based APIs, as do Visual Studio's WPF designer and editor.

It is unclear whether WPF will ever completely switch over to XAML2009. (Note that Silverlight and Windows Store apps don't support XAML2009 either.) In WPF 4.0 or later, however, you can take advantage of these features when using loose XAML with a host that processes the XAML with the XAML2009-based APIs, such as the XAMLPAD2009 sample from this book's source code or Internet Explorer when the `netfx/2009` XML namespace is used.

Therefore, the XAML2009 features are interesting to know about, even if they are not terribly useful. Most of them revolve around the idea of making a wider range of types directly usable from XAML. This is good news for class library authors, as XAML2009 imposes fewer restrictions for making class libraries XAML friendly. On its own, each feature provides a small improvement in expressiveness, but many of the features work together to solve real-world problems.

Full Generics Support

In XAML2006, the root element can be an instantiation of a generic class, thanks to the `x:TypeArguments` keyword. `x:TypeArguments` can be set to a type name or a comma-delimited list of type names. But because `x:TypeArguments` can only be used on the root element, generic classes generally have not been XAML friendly.

A common workaround for this limitation is to derive a non-generic class from a generic one simply so it can be referenced from XAML, as in the following example:

C#:

```
public class PhotoCollection : ObservableCollection<Photo> {}
```

XAML:

```
<custom:PhotoCollection>
  <custom:Photo …/>
  <custom:Photo …/>
</custom:PhotoCollection>
```

In XAML2009, however, `x:TypeArguments` can be used on *any* element, so a class like `ObservableCollection<Photo>` can be instantiated directly from XAML:

```
<collections:ObservableCollection TypeArguments="custom:Photo">
  <custom:Photo …/>
  <custom:Photo …/>
</collections:ObservableCollection>
```

In this case, `collections` is assumed to map to the `System.Collections.ObjectModel` namespace that contains `ObservableCollection`.

Dictionary Keys of Any Type

In XAML2009, type conversion is now attempted with `x:Key` values, so you can successfully add items to a dictionary with non-string keys without using a markup extension. Here's an example:

```
<collections:Dictionary x:TypeArguments="x:Int32, x:String">
  <x:String x:Key="1">One</x:String>
  <x:String x:Key="2">Two</x:String>
</collections:Dictionary>
```

Here, `collections` is assumed to map to the `System.Collections.Generic` namespace.

DIGGING DEEPER

Turning Off the Type Conversion of Non-String Dictionary Keys

For backwards compatibility, the XAML2009 `XamlObjectWriter` has a setting for turning off the new automatic type conversion. This is controlled by the `XamlObjectWriterSettings.PreferUnconvertedDictionaryKeys` property. When set to `true`, `System.Xaml` won't convert keys if the dictionary implements the non-generic `IDictionary` interface, unless:

- `System.Xaml` has already failed calling `IDictionary.Add` on this same instance, or
- The dictionary is a well-known type from the .NET Framework that `System.Xaml` knows requires conversion.

Built-In System Data Types

In XAML2006, using core .NET data types such as `String` or `Int32` is awkward due to the need to reference the `System` namespace from the `mscorlib` assembly, as seen previously in this chapter:

```
<sys:Int32 xmlns:sys="clr-namespace:System;assembly=mscorlib">7</sys:Int32>
```

In XAML2009, 13 .NET data types have been added to the XAML language namespace that most XAML is already referencing. With a namespace prefix of `x`, these data types are `x:Byte`, `x:Boolean`, `x:Int16`, `x:Int32`, `x:Int64`, `x:Single`, `x:Double`, `x:Decimal`, `x:Char`, `x:String`, `x:Object`, `x:Uri`, and `x:TimeSpan`. Therefore, the previous snippet can be rewritten as follows:

```
<x:Int32 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">7</x:Int32>
```

But it is typically seen as follows in a XAML file already referencing the XAML language namespace:

```
<x:Int32>7</x:Int32>
```

Instantiating Objects with Non-Default Constructors

XAML2009 introduces an `x:Arguments` keyword that enables you to specify one or more arguments to pass to a class's constructor. Consider, for example, the `System.Version` class, which has a default constructor and four parameterized constructors. You could not construct an instance of this class in XAML2006 unless someone provided an appropriate type converter (or unless you were happy with the behavior of the default constructor, which produces a version number of `0.0`).

In XAML2009, you can instantiate this class with its constructor that accepts a single string as follows:

```
<sys:Version x:Arguments="4.0.30319.1"/>
```

The constructor argument doesn't have to be a string; the attribute value undergoes type conversion as necessary.

Unlike `x:TypeArguments`, `x:Arguments` does not allow you to specify multiple arguments in the attribute value with a comma-delimited string. Instead, you can use the element form of `x:Arguments` to specify any number of arguments. For example, calling `System.Version`'s constructor that accepts four integers can be done as follows:

```
<sys:Version>
<x:Arguments>
  <x:Int32>4</x:Int32>
  <x:Int32>0</x:Int32>
  <x:Int32>30319</x:Int32>
  <x:Int32>1</x:Int32>
</x:Arguments>
</sys:Version>
```

Getting Instances via Factory Methods

With the new `x:FactoryMethod` keyword in XAML2009, you can now get an instance of a class that doesn't have *any* public constructors. `x:FactoryMethod` enables you to specify any public static method that returns an instance of the desired type. For example, the following XAML uses a `Guid` instance returned by the static `Guid.NewGuid` method:

```
<Label 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">
  <sys:Guid x:FactoryMethod="sys:Guid.NewGuid"/>
</Label>
```

When `x:FactoryMethod` is used with `x:Arguments`, the arguments are passed to the static factory method rather than to a constructor. Therefore, the following XAML calls the static `Marshal.GetExceptionForHR` method, which accepts an `HRESULT` error code as input

and returns the corresponding .NET exception that would be thrown by the common language runtime interoperability layer when encountering such an error:

```
<Label 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"
       xmlns:interop=
         "clr-namespace:System.Runtime.InteropServices;assembly=mscorlib">
  <sys:Exception x:FactoryMethod="interop:Marshal.GetExceptionForHR">
  <x:Arguments>
    <x:Int32>0x80004001</x:Int32>
  </x:Arguments>
  </sys:Exception>
</Label>
```

Figure 2.3 shows the result of the previous two `Labels` stacked in the same XAML content, as rendered by the XAMLPAD2009 sample.

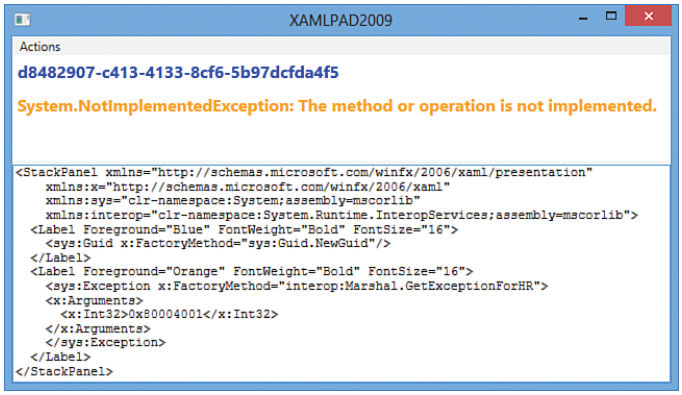

FIGURE 2.3 Displaying two instances retrieved via static factory methods.

Event Handler Flexibility

Event handlers can't be assigned in a loose XAML2006 file, but they can be assigned in a loose XAML2009 file as long as the root instance can be located and it has a method with a matching name and appropriate signature. In addition, in XAML2009, the value of an event attribute can be any markup extension that returns an appropriate delegate:

```
<Button Click="{custom:DelegateFinder Click}"/>
```

As with any markup extension, it can accept arbitrary input and perform arbitrary logic to look up the delegate.

Defining New Properties

XAML is primarily focused on instantiating existing classes and setting values of their predefined properties. Two new elements in XAML2009—`x:Members` and the corresponding `x:Property`—enable the *definition* of additional properties directly inside XAML. This functionality doesn't apply to WPF, however. You can see it used in Windows Workflow Foundation XAML, as in the following example:

```
<Activity x:Class="ActivityLibrary1.Activity1" …>
<x:Members>
  <x:Property Name="argument1" Type="InArgument(x:Int32)"/>
  <x:Property Name="argument2" Type="OutArgument(x:String)"/>
</x:Members>
…
</Activity>
```

XAML Keywords

The XAML language namespace (`http://schemas.microsoft.com/winfx/2006/xaml`) defines a handful of keywords that must be treated specially by any XAML compiler or parser. They mostly control aspects of how elements get exposed to procedural code, but several are useful even without any procedural code. You've already seen some of them (such as `Key`, `Name`, `Class`, `Subclass`, and `Code`), but Table 2.1 lists them all. They are listed with the conventional `x` prefix because that is how they usually appear in XAML and in documentation.

DIGGING DEEPER

Special Attributes Defined by the W3C

In addition to keywords in the XAML language namespace, XAML also supports two special attributes defined for XML by the World Wide Web Consortium (W3C): `xml:space` for controlling whitespace parsing and `xml:lang` for declaring the document's language and culture. The `xml` prefix is implicitly mapped to the standard XML namespace: `http://www.w3.org/XML/1998/namespace`.

TABLE 2.1 Keywords in the XAML Language Namespace, Assuming the Conventional `x` Namespace Prefix

Keyword	Valid As	Version	Meaning
`x:AsyncRecords`	Attribute on root element	2006+	Controls the size of asynchronous XAML-loading chunks.
`x:Arguments`	Attribute on or element inside any element	2009	Specifies an argument (or multiple arguments in the element syntax) to be passed to the element's constructor. When used with `x:FactoryMethod`, specifies argument(s) for the factory method.
`x:Boolean`	An element	2009	Represents a `System.Boolean`.
`x:Byte`	An element	2009	Represents a `System.Byte`.

TABLE 2.1 Continued

Keyword	Valid As	Version	Meaning
`x:Char`	An element	2009	Represents a `System.Char`.
`x:Class`	Attribute on root element	2006+	Defines a class for the root element that derives from the element type, optionally prefixed with a .NET namespace.
`x:ClassAttributes`	Attribute on root element and must be used with `x:Class`	2009	Not used by WPF; contains attributes relevant for Windows Workflow Foundation activities.
`x:ClassModifier`	Attribute on root element and must be used with `x:Class`	2006+	Defines the visibility of the class specified by `x:Class` (which is public by default). The attribute value must be specified in terms of the procedural language being used (for example, `public` or `internal` for C#).
`x:Code`	Element anywhere in XAML, but must be used with `x:Class`	2006+	Embeds procedural code to be inserted into the class specified by `x:Class`.
`x:ConnectionId`	Attribute	2006+	Not for public use.
`x:Decimal`	An element	2009	Represents a `System.Decimal`.
`x:Double`	An element	2009	Represents a `System.Double`.
`x:FactoryMethod`	Attribute on any element	2009	Specifies a static method to be called to retrieve the element instance instead of its constructor.
`x:FieldModifier`	Attribute on any nonroot element but must be used with `x:Name` (or equivalent)	2006+	Defines the visibility of the field to be generated for the element (which is internal by default). As with `x:ClassModifier`, the value must be specified in terms of the procedural language (for example, `public`, `private`, ... for C#).
`x:Int16`	An element	2009	Represents a `System.Int16`.
`x:Int32`	An element	2009	Represents a `System.Int32`.
`x:Int64`	An element	2009	Represents a `System.Int64`.
`x:Key`	Attribute on an element whose parent implements `IDictionary`	2006+	Specifies the key for the item when added to the parent dictionary.
`x:Members`	Not valid in WPF XAML	2009	Defines additional members for the root class specified by `x:Class`.
`x:Name`	Attribute on any nonroot element but must be used with `x:Class`	2006+	Chooses a name for the field to be generated for the element, so it can be referenced from procedural code.

TABLE 2.1 Continued

Keyword	Valid As	Version	Meaning
`x:Object`	An element	2009	Represents a `System.Object`.
`x:Property`	Not valid in WPF XAML	2009	Defines a property inside an `x:Members` element.
`x:Shared`	Attribute on any element in a `ResourceDictionary`, but only works if XAML is compiled	2006+	Can be set to `false` to avoid sharing the same resource instance in multiple places, as explained in Chapter 12.
`x:Single`	An element	2009	Represents a `System.Single`.
`x:String`	An element	2009	Represents a `System.String`.
`x:Subclass`	Attribute on root element and must be used with `x:Class`	2006+	Specifies a subclass of the `x:Class` class that holds the content defined in XAML, optionally prefixed with a .NET namespace (used with languages without support for partial classes).
`x:SynchronousMode`	Attribute on root element	2006+	Specifies whether the XAML content is allowed to be loaded asynchronously.
`x:TimeSpan`	An element	2009	Represents a `System.TimeSpan`.
`x:TypeArguments`	Attribute on any element in XAML2009, or attribute on root element that must be used with `x:Class` in XAML2006	2006+	Makes the class generic (for example, `List<T>`) with the specified generic argument instantiations (for example, `List<Int32>` or `List<String>`). Can be set to a comma-delimited list of generic arguments, with XML namespace prefixes for any types not in the default namespace.
`x:Uid`	Attribute on any element	2006+	Marks an element with an identifier used for localization, as described in Chapter 12.
`x:Uri`	An element	2009	Represents a `System.Uri`.
`x:XData`	Element used as the value for any property of type `IXmlSerializable`	2006+	An arbitrary XML data island that remains opaque to the XAML parser, as explained in Chapter 13.

Table 2.2 contains additional items in the XAML language namespace that can be confused as keywords but are actually just markup extensions (real .NET classes in the `System.Windows.Markup` namespace). Each class's `Extension` suffix is omitted from the table because the classes are typically used without the suffix.

TABLE 2.2 Markup Extensions in the XAML Language Namespace, Assuming the Conventional x Namespace Prefix

Extension	Meaning
`x:Array`	Represents a .NET array. An `x:Array` element's children are the elements of the array. It must be used with `x:Type` to define the type of the array.
`x:Null`	Represents a `null` reference.
`x:Reference`	A reference to a named element. It has a single positional parameter, which is the name of the referenced element.
`x:Static`	References any static property, field, constant, or enumeration value defined in procedural code. This can even be a nonpublic member in the same assembly, when XAML is compiled. Its `Member` string must be qualified with an XML namespace prefix if the type is not in the default namespace.
`x:Type`	Represents an instance of `System.Type`, just like the `typeof` operator in C#. Its `TypeName` string must be qualified with an XML namespace prefix if the type is not in the default namespace.

Summary

You have now seen how XAML fits in with WPF and, most importantly, you now have the information needed to translate most XAML examples into a language such as C# and vice versa. However, because type converters and markup extensions are "black boxes," a straightforward translation is not always going to be obvious. That said, invoking a type converter directly from procedural code is always an option if you can't figure out the conversion that the type converter is doing internally! (Many classes with corresponding type converters even expose a static `Parse` method that does the same work, for the sake of simpler procedural code.)

I love the fact that simple concepts that could have been treated specially by XAML (such as `null` or a named reference) are expressed using the same markup extension mechanism used by third parties. This keeps the XAML language as simple as possible, and it ensures that the extensibility mechanism works really well.

As you proceed further with WPF, you might find that some WPF APIs can be a little clunky from procedural code because their design is often optimized for XAML use. For example, WPF exposes many small building blocks (enabling the rich composition described in the previous chapter), so a WPF application generally must create many objects manually. Besides the fact the XAML excels at expressing deep hierarchies of objects concisely, the WPF team spent more time implementing features to effectively hide intermediate objects in XAML (such as type converters) rather than features to hide them from procedural code (such as constructors that create inner objects on your behalf).

Most people understand the benefit of WPF having the separate declarative model provided by XAML, but some lament XML as the choice of format. The following sections are two common complaints and my attempt to debunk them.

Complaint 1: XML Is Too Verbose to Type

This is true: Almost nobody enjoys typing lots of XML, but that's where tools come in. Tools such as IntelliSense and visual designers can spare you from typing a single angle bracket! The transparent and well-specified nature of XML enables you to easily integrate new tools into the development process (creating a XAML exporter for your favorite tool, for example) and also enables easy hand-tweaking or troubleshooting.

In some areas of WPF—complicated paths and shapes, 3D models, and so on—typing XAML by hand isn't even practical. In fact, the trend from when XAML was first introduced in beta form has been to remove some of the handy human-typable shortcuts in favor of a more robust and extensible format that can be supported well by tools. But I still believe that being familiar with XAML and seeing the WPF APIs through both procedural and declarative perspectives is the best way to learn the technology. It's like understanding how HTML works without relying on a visual tool.

Complaint 2: XML-Based Systems Have Poor Performance

XML is about interoperability, not about an efficient representation of data. So, why should most WPF applications be saddled with a bunch of data that is relatively large and slow to parse?

The good news is that in a normal WPF scenario, XAML is compiled into BAML, so you don't pay the full penalties of size and parsing performance at runtime. BAML is both smaller in size than the original XAML and optimized for efficient use at runtime. Performance pitfalls from XML are therefore limited to development time, which is when the benefits of XML are needed the most.

This page intentionally left blank

CHAPTER 3

WPF Fundamentals

IN THIS CHAPTER

- A Tour of the Class Hierarchy
- Logical and Visual Trees
- Dependency Properties

To finish Part I, "Background," and before moving on to the *really* fun topics, it's helpful to examine some of the main concepts that WPF introduces above and beyond what .NET programmers are already familiar with. The topics in this chapter are some of the main culprits responsible for WPF's notoriously steep learning curve. By familiarizing yourself with these concepts now, you'll be able to approach the rest of this book (or any other WPF documentation) with confidence.

Some of this chapter's concepts are unique to WPF (such as logical and visual trees), but others are just extensions of concepts that should be quite familiar (such as properties). As you learn about each one, you'll see how to apply it to a very simple piece of user interface that most programs need—an *About dialog*.

A Tour of the Class Hierarchy

WPF's classes have a very deep inheritance hierarchy, so it can be hard to get your head wrapped around the significance of various classes and their relationships. A handful of classes are fundamental to the inner workings of WPF and deserve a quick explanation before we get any further in the book. Figure 3.1 shows these important classes and their relationships.

These 12 classes have the following significance:

- **`Object`**—The base class for all .NET classes and the only class in the figure that isn't WPF specific.

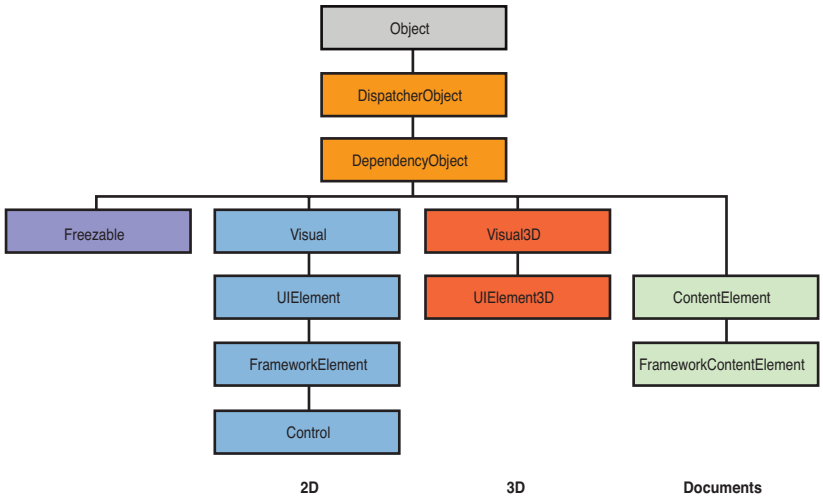

FIGURE 3.1 The core classes that form the foundation of WPF.

- **`DispatcherObject`**—The base class meant for any object that wishes to be accessed only on the thread that created it. Most WPF classes derive from `DispatcherObject` and are therefore inherently thread-unsafe. The `Dispatcher` part of the name refers to WPF's version of a Win32-like message loop, discussed further in Chapter 7, "Structuring and Deploying an Application."
- **`DependencyObject`**—The base class for any object that can support dependency properties, one of the main topics in this chapter.
- **`Freezable`**—The base class for objects that can be "frozen" into a read-only state for performance reasons. `Freezables`, once frozen, can be safely shared among multiple threads, unlike all other `DispatcherObjects`. Frozen objects can never be unfrozen, but you can clone them to create unfrozen copies. Most `Freezables` are graphics primitives such as brushes, pens, and geometries or animation classes.
- **`Visual`**—The base class for all objects that have their own 2D visual representation. `Visuals` are discussed in depth in Chapter 15, "2D Graphics."
- **`UIElement`**—The base class for all 2D visual objects with support for routed events, command binding, layout, and focus. These features are discussed in Chapter 5, "Layout with Panels," and Chapter 6, "Input Events: Keyboard, Mouse, Stylus, and Touch."
- **`Visual3D`**—The base class for all objects that have their own 3D visual representation. `Visual3Ds` are discussed in depth in Chapter 16, "3D Graphics."
- **`UIElement3D`**—The base class for all 3D visual objects with support for routed events, command binding, and focus, also discussed in Chapter 16.
- **`ContentElement`**—A base class similar to `UIElement` but for document-related pieces of content that don't have rendering behavior on their own. Instead,

ContentElements are hosted in a Visual-derived class to be rendered on the screen. Each ContentElement often requires multiple Visuals to render correctly (spanning lines, columns, and pages).

- **FrameworkElement**—The base class that adds support for styles, data binding, resources, and a few common mechanisms for controls, such as tooltips and context menus.
- **FrameworkContentElement**—The analog to FrameworkElement for content. Chapter 11, "Images, Text, and Other Controls," examines the FrameworkContentElements in WPF.
- **Control**—The base class for familiar controls such as Button, ListBox, and StatusBar. Control adds many properties to its FrameworkElement base class, such as Foreground, Background, and FontSize, as well as the ability to be completely restyled. Part III, "Controls," examines WPF's controls in depth.

Throughout the book, the simple term *element* is used to refer to an object that derives from UIElement or FrameworkElement, and sometimes ContentElement or FrameworkContentElement. The distinction between UIElement and FrameworkElement or between ContentElement and FrameworkContentElement is not important because WPF doesn't ship any other public subclasses of UIElement and ContentElement.

Logical and Visual Trees

XAML is natural for representing a user interface because of its hierarchical nature. In WPF, user interfaces are constructed from a tree of objects known as a *logical tree*.

Listing 3.1 defines the beginnings of a hypothetical About dialog, using a Window as the root of the logical tree. The Window has a StackPanel child element (described in Chapter 5) containing a few simple controls plus another StackPanel that contains Buttons.

LISTING 3.1 A Simple About Dialog in XAML

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="About WPF 4.5 Unleashed" SizeToContent="WidthAndHeight"
  Background="OrangeRed">
  <StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
      WPF 4.5 Unleashed
    </Label>
    <Label>© 2013 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
      <ListBoxItem>Chapter 1</ListBoxItem>
      <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>
```

LISTING 3.1 Continued

```
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
      <Button MinWidth="75" Margin="10">Help</Button>
      <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
  </StackPanel>
</Window>
```

Figure 3.2 shows the rendered dialog (which you can easily produce by pasting the content of Listing 3.1 into a tool such as the XAMLPAD2009 sample from the previous chapter), and Figure 3.3 illustrates the logical tree for this dialog.

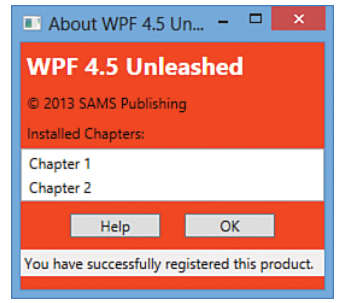

FIGURE 3.2 The rendered dialog from Listing 3.1.

Note that a logical tree exists even for WPF user interfaces that aren't created in XAML. Listing 3.1 could be implemented entirely in C#, and the logical tree would be identical.

The logical tree concept is straightforward, but why should you care about it? Because just about every aspect of WPF (properties, events, resources, and so on) has behavior tied to the logical tree. For example, property values are sometimes propagated down the tree to child elements automatically, and raised events can travel up or down the tree. This behavior of property values is discussed later in this chapter, and this behavior of events is discussed in Chapter 6.

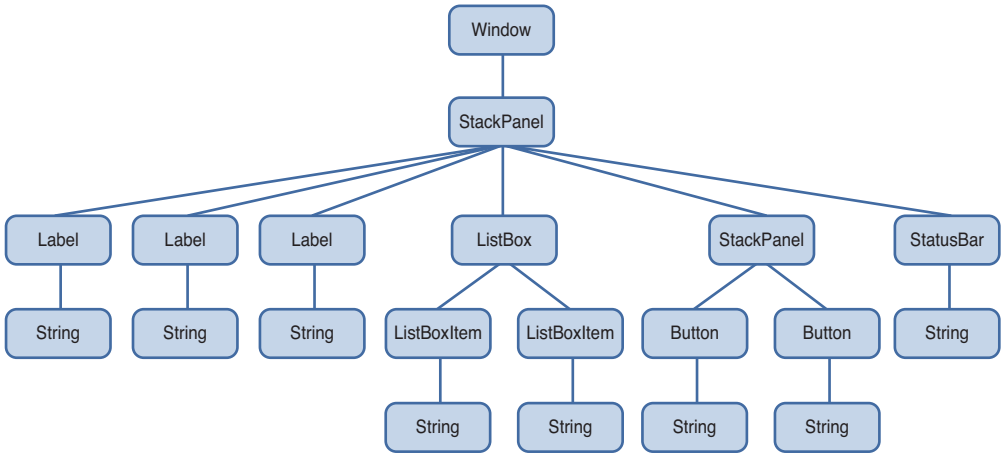

FIGURE 3.3 The logical tree for Listing 3.1.

The logical tree exposed by WPF is a simplification of what is actually going on when the elements are rendered. The entire tree of elements actually being rendered is called the

visual tree. You can think of the visual tree as an expansion of a logical tree, in which nodes are broken down into their core visual components. Rather than leaving each element as a "black box," a visual tree exposes the visual implementation details. For example, although a `ListBox` is logically a single control, its default visual representation is composed of more primitive WPF elements: a `Border`, two `ScrollBars`, and more.

Not all logical tree nodes appear in the visual tree; only the elements that derive from `System.Windows.Media.Visual` or `System.Windows.Media.Visual3D` are included. Other elements (and simple string content, as in Listing 3.1) are not included because they don't have inherent rendering behavior of their own.

TIP

Some lightweight XAML viewers, such as the XamlPadX tool mentioned in the preceding chapter, have functionality for exploring the visual tree (and property values) for the objects that it renders from XAML.

Figure 3.4 illustrates the default visual tree for Listing 3.1. This diagram exposes some inner components of the user interface that are currently invisible, such as the `ListBox`'s two `ScrollBars` and each `Label`'s `Border`. It also reveals that `Button`, `Label`, and `ListBoxItem` are all composed of the same elements. (These controls have other visual differences as the result of different default property values. For example, `Button` has a default `Margin` of `10` on all sides, whereas `Label` has a default `Margin` of `0`.)

Because they enable you to peer inside the deep composition of WPF elements, visual trees can be surprisingly complex. Fortunately, although visual trees are an essential part of the WPF infrastructure, you often don't need to worry about them unless you're radically restyling controls (covered in Chapter 14, "Styles, Templates, Skins, and Themes") or doing low-level drawing (covered in Chapter 15). Writing code that depends on a specific visual tree for a `Button`, for example, breaks one of WPF's core tenets—the separation of look and logic. When someone restyles a control such as `Button` using the techniques described in Chapter 14, its entire visual tree is replaced with something that could be completely different.

WARNING

Avoid writing code that depends on a specific visual tree!

Whereas a logical tree is static without programmer intervention (such as dynamically adding/removing elements), a visual tree can change simply because a user switches to a different Windows theme!

However, you can easily traverse both the logical and visual trees using the somewhat symmetrical `System.Windows.LogicalTreeHelper` and `System.Windows.Media.VisualTreeHelper` classes. Listing 3.2 contains a code-behind file for Listing 3.1 that, when run under a debugger, outputs a simple depth-first representation of both the logical and visual trees for the About dialog. (This requires adding `x:Class="AboutDialog"` and the corresponding `xmlns:x` directive to Listing 3.1 in order to hook it up to this procedural code.)

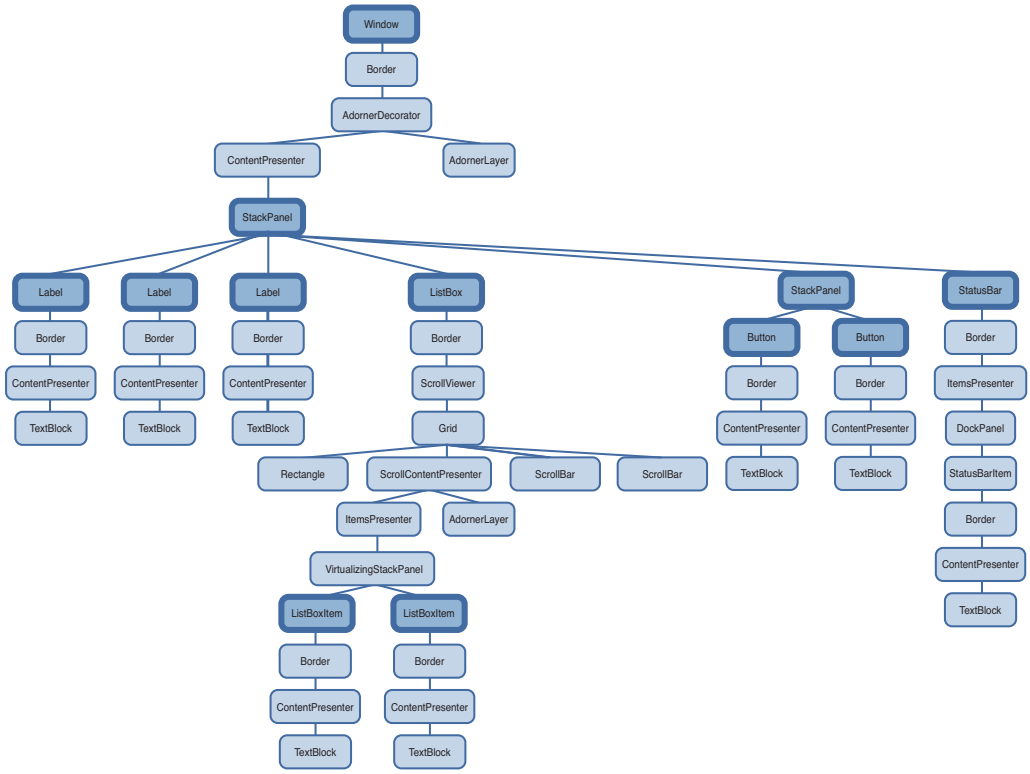

FIGURE 3.4 The visual tree for Listing 3.1, with logical tree nodes emphasized.

LISTING 3.2 Walking and Printing the Logical and Visual Trees

```
using System;
using System.Diagnostics;
using System.Windows;
using System.Windows.Media;

public partial class AboutDialog : Window
{
  public AboutDialog()
  {
    InitializeComponent();
    PrintLogicalTree(0, this);
  }

  protected override void OnContentRendered(EventArgs e)
  {
    base.OnContentRendered(e);
    PrintVisualTree(0, this);
```

LISTING 3.2 Continued

```
  }

  void PrintLogicalTree(int depth, object obj)
  {
    // Print the object with preceding spaces that represent its depth
    Debug.WriteLine(new string(' ', depth) + obj);

    // Sometimes leaf nodes aren't DependencyObjects (e.g. strings)
    if (!(obj is DependencyObject)) return;

    // Recursive call for each logical child
    foreach (object child in LogicalTreeHelper.GetChildren(
      obj as DependencyObject))
      PrintLogicalTree(depth + 1, child);
  }

  void PrintVisualTree(int depth, DependencyObject obj)
  {
    // Print the object with preceding spaces that represent its depth
    Debug.WriteLine(new string(' ', depth) + obj);

    // Recursive call for each visual child
    for (int i = 0; i < VisualTreeHelper.GetChildrenCount(obj); i++)
      PrintVisualTree(depth + 1, VisualTreeHelper.GetChild(obj, i));
  }
}
```

When calling these methods with a depth of 0 and the current `Window` instance, the result is a text-based tree with exactly the same nodes shown in Figures 3.2 and 3.3. Although the logical tree can be traversed within `Window`'s constructor, the visual tree is empty until the `Window` undergoes layout at least once. That is why `PrintVisualTree` is called within `OnContentRendered`, which doesn't get called until after layout occurs.

TIP

Visual trees like the one represented in Figure 3.4 are often referred to simply as *element trees*, because they encompass both elements in the logical tree and elements specific to the visual tree. The term *visual tree* is then used to describe any subtree that contains visual-only (illogical?) elements. For example, most people would say that `Window`'s default visual tree consists of a `Border`, an `AdornerDecorator`, an `AdornerLayer`, a `ContentPresenter`, and nothing more. In Figure 3.4, the top-most `StackPanel` is generally *not* considered to be the visual child of the `ContentPresenter`, despite the fact that `VisualTreeHelper` presents it as one.

TIP

In the Visual Studio debugger, you can click the little magnifying glass next to an instance of a `Visual`-derived variable to invoke WPF Visualizer, as shown in Figure 3.5. This enables you to navigate and visualize the visual tree.

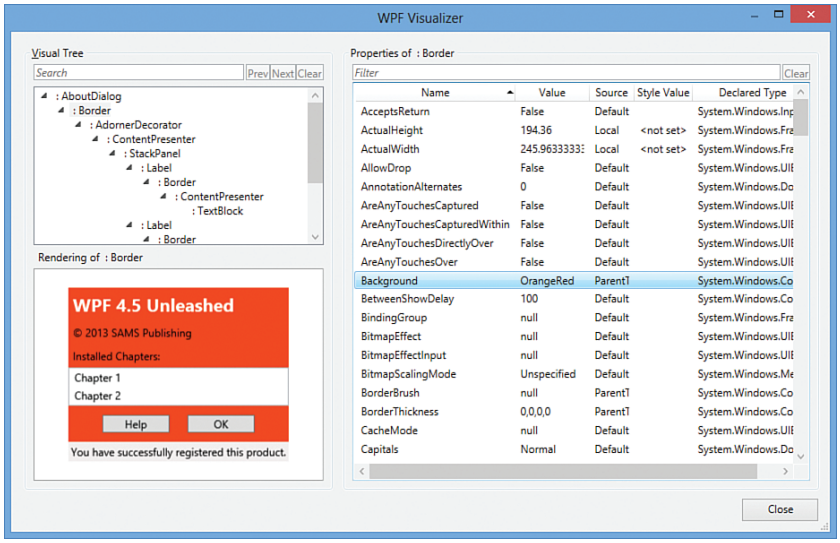

FIGURE 3.5 The WPF Visualizer in Visual Studio reveals the visual tree and details about each element.

Navigating either tree can sometimes be done with instance methods on the elements themselves. For example, the `Visual` class contains three protected members (`VisualParent`, `VisualChildrenCount`, and `GetVisualChild`) for examining its visual parent and children. `FrameworkElement`, the common base class for controls such as `Button` and `Label`, and its peer `FrameworkContentElement` both define a public `Parent` property representing the logical parent and a protected `LogicalChildren` property for the logical children. Subclasses of these two classes often publicly expose their logical children in a variety of ways, such as in a public `Children` collection. Some classes, such as `Button` and `Label`, expose a `Content` property and enforce that the element can have only one logical child.

Dependency Properties

WPF introduces a type of property called a *dependency property* that is used throughout the platform to enable styling, automatic data binding, animation, and more. You might first meet this concept with skepticism, as it complicates the picture of .NET types having simple fields, properties, methods, and events. But when you understand the problems that dependency properties solve, you will likely accept them as a welcome addition.

A dependency property *depends* on multiple providers for determining its value at any point in time. These providers could be an animation continuously changing its value, a parent element whose property value propagates down to its children, and so on. Arguably the biggest feature of a dependency property is its built-in ability to provide change notification.

The motivation for adding such intelligence to properties is to enable rich functionality directly from declarative markup. The key to WPF's declarative-friendly design is its heavy use of properties. `Button`, for example, has more than 100 public properties (most of which are inherited from `Control` and its base classes)! Properties can be easily set in XAML (directly or by using a design tool) without any procedural code. But without the extra plumbing in dependency properties, it would be hard for the simple action of setting properties to get the desired results without the need to write additional code.

In this section, we briefly look at the implementation of a dependency property to make this discussion more concrete, and then we dig deeper into some of the ways that dependency properties add value on top of plain .NET properties:

- Change notification
- Property value inheritance
- Support for multiple providers

Understanding most of the nuances of dependency properties is usually important only for custom control authors. However, even casual users of WPF need to be aware of what dependency properties are and how they work. For example, you can only style and animate dependency properties. After working with WPF for a while, you might find yourself wishing that *all* properties would be dependency properties!

A Dependency Property Implementation

In practice, dependency properties are just normal .NET properties hooked into some extra WPF infrastructure. This is all accomplished via WPF APIs; no .NET languages (other than XAML) have an intrinsic understanding of a dependency property.

Listing 3.3 demonstrates how `Button` effectively implements one of its dependency properties, called `IsDefault`.

LISTING 3.3 A Standard Dependency Property Implementation

```
public class Button : ButtonBase
{
  // The dependency property
  public static readonly DependencyProperty IsDefaultProperty;

  static Button()
  {
    // Register the property
```

```
    Button.IsDefaultProperty = DependencyProperty.Register("IsDefault",
      typeof(bool), typeof(Button),
      new FrameworkPropertyMetadata(false,
      new PropertyChangedCallback(OnIsDefaultChanged)));
    …
  }

  // A .NET property wrapper (optional)
  public bool IsDefault
  {
    get { return (bool)GetValue(Button.IsDefaultProperty); }
    set { SetValue(Button.IsDefaultProperty, value); }
  }

  // A property changed callback (optional)
  private static void OnIsDefaultChanged(
    DependencyObject o, DependencyPropertyChangedEventArgs e) { … }
  …
}
```

The static `IsDefaultProperty` field is the actual dependency property, represented by the `System.Windows.DependencyProperty` class. By convention, all `DependencyProperty` fields are public, static, and have a `Property` suffix. Several pieces of infrastructure require that you follow this convention: localization tools, XAML loading, and more.

Dependency properties are usually created by calling the static `DependencyProperty.Register` method, which requires a name (`IsDefault`), a property type (`bool`), and the type of the class claiming to own the property (`Button`). Optionally (via different overloads of `Register`), you can pass metadata that customizes how the property is treated by WPF, as well as callbacks for handling property value changes, coercing values, and validating values. `Button` calls an overload of `Register` in its static constructor to give the dependency property a default value of `false` and to attach a delegate for change notifications.

Finally, the traditional .NET property called `IsDefault` implements its accessors by calling `GetValue` and `SetValue` methods inherited from `System.Windows.DependencyObject`, the low-level base class from which all classes with dependency properties must derive. `GetValue` returns the last value passed to `SetValue` or, if `SetValue` has never been called, the default value registered with the property.

The `IsDefault` .NET property (sometimes called a *property wrapper* in this context) is not strictly necessary; consumers of `Button` could directly call the `GetValue`/`SetValue` methods because they are exposed publicly. But the .NET property makes programmatic reading

TIP

Visual Studio has a snippet called `propdp` that automatically expands into a definition of a dependency property, which makes defining one much faster than doing all the typing yourself!

and writing of the property much more natural for consumers, and it enables the property to be set via XAML. WPF should, but does not, provide generic overloads of `GetValue` and `SetValue`. This is primarily because dependency properties were invented before .NET generics were widely used.

WARNING

.NET property wrappers are bypassed at runtime when setting dependency properties in XAML!

Although the XAML compiler depends on the property wrapper at compile time, WPF calls the underlying `GetValue` and `SetValue` methods directly at runtime! Therefore, to maintain parity between setting a property in XAML and procedural code, it's crucial that property wrappers not contain any logic in addition to the `GetValue`/`SetValue` calls. If you want to add custom logic, that's what the registered callbacks are for. All of WPF's built-in property wrappers abide by this rule, so this warning is for anyone writing a custom class with its own dependency properties.

On the surface, Listing 3.3 looks like an overly verbose way of representing a simple Boolean property. However, because `GetValue` and `SetValue` internally use an efficient sparse storage system and because `IsDefaultProperty` is a static field (rather than an instance field), the dependency property implementation saves per-instance memory compared to a typical .NET property. If all the properties on WPF controls were wrappers around instance fields (as most .NET properties are), they would consume a significant amount of memory because of all the local data attached to each instance. Having more than 100 fields for each `Button`, more than 100 fields for each `Label`, and so forth would add up quickly! Instead, almost all of `Button`'s and `Label`'s properties are dependency properties.

The benefits of the dependency property implementation extend to more than just memory usage, however. The implementation centralizes and standardizes a fair amount of code that property implementers would have to write to check thread access, prompt the containing element to be re-rendered, and so on. For example, if a property requires its element to be re-rendered when its value changes (such as `Button`'s `Background` property), it can simply pass the `FrameworkPropertyMetadataOptions.AffectsRender` flag to an overload of `DependencyProperty.Register`. In addition, this implementation enables the three features listed earlier that we'll now examine one-by-one, starting with change notification.

Change Notification

Whenever the value of a dependency property changes, WPF can automatically trigger a number of actions, depending on the property's metadata. These actions can be re-rendering the appropriate elements, updating the current layout, refreshing data bindings, and much more. One of the interesting features enabled by this built-in change notification is *property triggers*, which enable you to perform your own custom actions when a property value changes, without writing any procedural code.

For example, imagine that you want the text in each `Button` from the About dialog in Listing 3.1 to turn blue when the mouse pointer hovers over it. Without property triggers, you can attach two event handlers to each `Button`, one for its `MouseEnter` event and one for its `MouseLeave` event:

```
<Button MouseEnter="Button_MouseEnter" MouseLeave="Button_MouseLeave"
        MinWidth="75" Margin="10">Help</Button>
<Button MouseEnter="Button_MouseEnter" MouseLeave="Button_MouseLeave"
        MinWidth="75" Margin="10">OK</Button>
```

These two handlers could be implemented in a C# code-behind file as follows:

```
// Change the foreground to blue when the mouse enters the button
void Button_MouseEnter(object sender, MouseEventArgs e)
{
  Button b = sender as Button;
  if (b != null) b.Foreground = Brushes.Blue;
}

// Restore the foreground to black when the mouse exits the button
void Button_MouseLeave(object sender, MouseEventArgs e)
{
  Button b = sender as Button;
  if (b != null) b.Foreground = Brushes.Black;
}
```

With a property trigger, however, you can accomplish this same behavior purely in XAML. The following concise `Trigger` object is just about all you need:

```
<Trigger Property="IsMouseOver" Value="True">
  <Setter Property="Foreground" Value="Blue"/>
</Trigger>
```

This trigger can act on `Button`'s `IsMouseOver` property, which becomes `true` at the same time the `MouseEnter` event is raised and `false` at the same time the `MouseLeave` event is raised. Note that you don't have to worry about reverting `Foreground` to black when `IsMouseOver` changes to `false`. This is automatically done by WPF!

The only trick is assigning this `Trigger` to each `Button`. Unfortunately, because of a confusing limitation, you can't apply property triggers directly to elements such as `Button`. You can apply them only inside a `Style` object, so an in-depth examination of property triggers is saved for Chapter 14. In the meantime, to experiment with property triggers, you can apply the preceding `Trigger` to a `Button` by wrapping it in a few intermediate XML elements, as follows:

```
<Button MinWidth="75" Margin="10">
<Button.Style>
  <Style TargetType="{x:Type Button}">
```

```
    <Style.Triggers>
      <Trigger Property="IsMouseOver" Value="True">
        <Setter Property="Foreground" Value="Blue"/>
      </Trigger>
    </Style.Triggers>
    </Style>
</Button.Style>
    OK
</Button>
```


Property triggers are just one of three types of triggers supported by WPF. A *data trigger* is a form of property trigger that works for all .NET properties (not just dependency properties), also covered in Chapter 14. An *event trigger* enables you to declaratively specify actions to take when a routed event (covered in Chapter 6) is raised. Event triggers always involve working with animations or sounds, so they aren't covered until Chapter 17, "Animation."

WARNING

Don't be fooled by an element's `Triggers` collection!

`FrameworkElement`'s `Triggers` property is a read/write collection of `TriggerBase` items (the common base class for all three types of triggers), so it looks like an easy way to attach property triggers to controls such as `Button`. Unfortunately, this collection can only contain event triggers, so its name and type are misleading. Attempting to add a property trigger (or data trigger) to the collection causes an exception to be thrown at runtime.

Property Value Inheritance

The term *property value inheritance* (or *property inheritance* for short) doesn't refer to traditional object-oriented class-based inheritance but rather the flowing of property values down the element tree. A simple example of this can be seen in Listing 3.4, which updates the `Window` from Listing 3.1 by explicitly setting its `FontSize` and `FontStyle` dependency properties. Figure 3.6 shows the result of this change. (Notice that the `Window` automatically resizes to fit all the content thanks to its slick `SizeToContent` setting!)

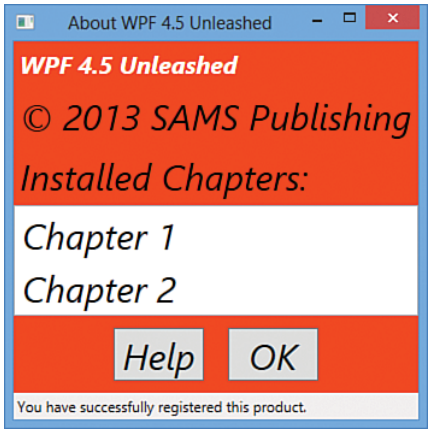

FIGURE 3.6 The About dialog with `FontSize` and `FontStyle` set on the root `Window`.

LISTING 3.4 The About Dialog with Font Properties Set on the Root Window

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="About WPF 4.5 Unleashed" SizeToContent="WidthAndHeight"
  FontSize="30" FontStyle="Italic"
  Background="OrangeRed">
  <StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
      WPF 4.5 Unleashed
    </Label>
    <Label>© 2013 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
      <ListBoxItem>Chapter 1</ListBoxItem>
      <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
      <Button MinWidth="75" Margin="10">Help</Button>
      <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
  </StackPanel>
</Window>
```

For the most part, these two settings flow all the way down the tree and are inherited by children. This affects even the `Buttons` and `ListBoxItems`, which are three levels down the logical tree. The first `Label`'s `FontSize` does not change because it is explicitly marked with a `FontSize` of `20`, overriding the inherited value of `30`. The inherited `FontStyle` setting of `Italic` affects all `Labels`, `ListBoxItems`, and `Buttons`, however, because none of them have this set explicitly.

Notice that the text in the `StatusBar` is unaffected by either of these values, despite the fact that it supports these two properties just like the other controls. The behavior of property value inheritance can be subtle in cases like this for two reasons:

- Not every dependency property participates in property value inheritance. (Internally, dependency properties can opt in to inheritance by passing `FrameworkPropertyMetadataOptions.Inherits` to `DependencyProperty.Register`.)
- There may be other higher-priority sources setting the property value, as explained in the next section.

In this case, the latter reason is to blame. A few controls, such as `StatusBar`, `Menu`, and `ToolTip`, internally set their font properties to match current system settings. This way, users get the familiar experience of controlling their font via Control Panel. The result can be confusing, however, because such controls end up "swallowing" any inheritance from

proceeding further down the element tree. For example, if you add a `Button` as a logical child of the `StatusBar` in Listing 3.4, its `FontSize` and `FontStyle` would be the default values of `12` and `Normal`, respectively, unlike the other `Buttons` outside of the `StatusBar`.

DIGGING DEEPER

Property Value Inheritance in Additional Places

Property value inheritance was originally designed to operate on the element tree, but it has been extended to work in a few other contexts as well. For example, values can be passed down to certain elements that *look like* children in the XML sense (because of XAML's property element syntax) but *are not* children in terms of the logical or visual trees. These pseudochildren can be an element's triggers or the value of *any* property (not just `Content` or `Children`), as long as it is an object deriving from `Freezable`. This may sound arbitrary and isn't well documented, but the intention is that several XAML-based scenarios "just work" as you would expect, without requiring you to think about it.

Support for Multiple Providers

WPF contains many powerful mechanisms that independently attempt to set the value of dependency properties. Without a well-defined mechanism for handling these disparate property value providers, the system would be a bit chaotic, and property values could be unstable. Of course, as their name indicates, dependency properties were designed to depend on these providers in a consistent and orderly manner.

Figure 3.7 illustrates the five-step process that WPF runs each dependency property through in order to calculate its final value. This process happens automatically, thanks to the built-in change notification in dependency properties.

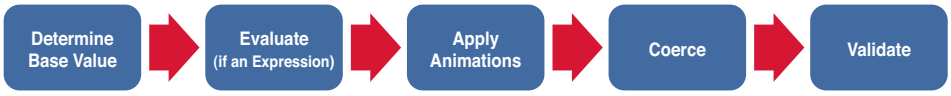

FIGURE 3.7 The pipeline for calculating the value of a dependency property.

Step 1: Determine the Base Value

Most of the property value providers factor into the base value calculation. The following list reveals the ten providers that can set the value of most dependency properties, in order from highest to lowest precedence:

1. Local value
2. Parent template trigger
3. Parent template
4. Style triggers
5. Template triggers

6. Style setters
7. Theme style triggers
8. Theme style setters
9. Property value inheritance
10. Default value

You've already seen some of the property value providers, such as property value inheritance (#9). *Local value* (#1) technically means any call to `DependencyObject.SetValue`, but this is typically seen with a simple property assignment in XAML or procedural code (because of the way dependency properties are implemented, as shown previously with `Button.IsDefault`). *Default value* (#10) refers to the initial value registered with the dependency property, which naturally has the lowest precedence. The other providers, which all involve styles and templates, are explained further in Chapter 14.

This order of precedence explains why `StatusBar`'s `FontSize` and `FontStyle` were not impacted by property value inheritance in Listing 3.4. The setting of `StatusBar`'s font properties to match system settings is done via theme style setters (#8). Although this has precedence over property value inheritance (#9), you can still override these font settings using any mechanism with a higher precedence, such as simply setting local values on `StatusBar`.

TIP

If you can't figure out where a given dependency property is getting its current value, you can use the static `DependencyPropertyHelper.GetValueSource` method as a debugging aid. This returns a `ValueSource` structure that contains a few pieces of data: a `BaseValueSource` enumeration that reveals where the base value came from (step 1 in the process) and Boolean `IsExpression`, `IsAnimated`, and `IsCoerced` properties that reveal information about steps 2 through 4.

When calling this method on the `StatusBar` instance from Listing 3.1 or 3.4 with the `FontSize` or `FontStyle` property, the returned `BaseValueSource` is `DefaultStyle`, revealing that the value comes from a theme style setter. (Theme styles are sometimes referred to as *default styles*. The enumeration value for a theme style trigger is `DefaultStyleTrigger`.)

Do *not* use this method in production code! Future versions of WPF could break assumptions you've made about the value calculation. In addition, treating a property value differently, depending on its source, goes against the way things are supposed to work in WPF applications.

DIGGING DEEPER

Clearing a Local Value

The earlier "Change Notification" section demonstrates the use of procedural code to change a `Button`'s `Foreground` to blue in response to the `MouseEnter` event and then changing it back to black in response to the `MouseLeave` event. The problem with this approach is that black is set as a local value inside `MouseLeave`, which is much different from the `Button`'s initial state, in which its black `Foreground` comes from a setter in its theme style. If the theme is changed and the new theme tries to change the default `Foreground` color (or if other providers with higher precedence try to do the same), this change is trumped by the local setting of black.

What you likely want to do instead is *clear* the local value and let WPF set the value from the relevant provider with the next-highest precedence. Fortunately, `DependencyObject` provides exactly this kind of mechanism with its `ClearValue` method. This can be called on a `Button` `b` as follows in C#:

```
b.ClearValue(Button.ForegroundProperty);
```

(`Button.ForegroundProperty` is the static `DependencyProperty` field.) After calling `ClearValue`, the local value is simply removed from the equation when WPF recalculates the base value.

Note that the trigger on the `IsMouseOver` property from the "Change Notification" section does not have the same problem as the implementation with event handlers. A trigger is either active or inactive, and when it is inactive, it is simply ignored in the property value calculation.

Step 2: Evaluate

If the value from step one is an *expression* (an object deriving from `System.Windows.Expression`), WPF performs a special evaluation step to convert the expression into a concrete result. Expressions mostly appear in data binding (the topic of Chapter 13, "Data Binding").

Step 3: Apply Animations

If one or more animations are running, they have the power to alter the current property value (using the value after step 2 as input) or completely replace it. Therefore, animations (the topic of Chapter 17) can trump all other property value providers—even local values! This is often a stumbling block for people who are new to WPF.

Step 4: Coerce

After all the property value providers have had their say, WPF passes the almost-final property value to a `CoerceValueCallback` delegate, if one was registered with the dependency property. The callback is responsible for returning a new value, based on custom logic. For example, built-in WPF controls such as `ProgressBar` use this callback to constrain its `Value` dependency property to a value between its `Minimum` and `Maximum` values, returning `Minimum` if the input value is less than `Minimum` and `Maximum` if the input value is greater than `Maximum`. If you change your coercion logic at runtime, you can call `CoerceValue` to make WPF run the new coercion and validation logic again.

Step 5: Validate

Finally, the potentially coerced value is passed to a `ValidateValueCallback` delegate, if one was registered with the dependency property. This callback must return `true` if the input value is valid and `false` otherwise. Returning `false` causes an exception to be thrown, canceling the entire process.

TIP

`DependencyObject` has a `SetCurrentValue` method that directly updates the current value of a property without changing its value source. (The value is still subject to coercion and validation.) This is meant for controls that set values in response to user interaction. For example, the `RadioButton` control modifies the value of the `IsChecked` property on other `RadioButtons` in the same group, based on user interaction.

Attached Properties

An *attached property* is a special form of dependency property that can effectively be *attached* to arbitrary objects. This may sound strange at first, but this mechanism has several applications in WPF.

For the About dialog example, imagine that rather than setting `FontSize` and `FontStyle` for the entire `Window` (as is done in Listing 3.4), you would rather set them on the inner `StackPanel` so they are inherited only by the two `Buttons`. Moving the property attributes to the inner `StackPanel` element doesn't work, however, because `StackPanel` doesn't have any font-related properties of its own! Instead, you must use the `FontSize` and `FontStyle` attached properties that happen to be defined on a class called `TextElement`. Listing 3.5 demonstrates this, introducing new XAML syntax designed especially for attached properties. This enables the desired property value inheritance, as shown in Figure 3.8.

LISTING 3.5 The About Dialog with Font Properties Moved to the Inner `StackPanel`

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="About WPF 4.5 Unleashed" SizeToContent="WidthAndHeight"
  Background="OrangeRed">
  <StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
      WPF 4.5 Unleashed
    </Label>
    <Label>© 2013 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
      <ListBoxItem>Chapter 1</ListBoxItem>
      <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>
```

```
    <StackPanel TextElement.FontSize="30" TextElement.FontStyle="Italic"
      Orientation="Horizontal" HorizontalAlignment="Center">
      <Button MinWidth="75" Margin="10">Help</Button>
      <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
  </StackPanel>
</Window>
```

`TextElement.FontSize` and `TextElement.FontStyle` (rather than simply `FontSize` and `FontStyle`) must be used in the `StackPanel` element because `StackPanel` does not have these properties. When a XAML parser or compiler encounters this syntax, it requires that `TextElement` (sometimes called the *attached property provider*) have static methods called `SetFontSize` and `SetFontStyle` that can set the value accordingly. Therefore, the `StackPanel` declaration in Listing 3.5 is equivalent to the following C# code:

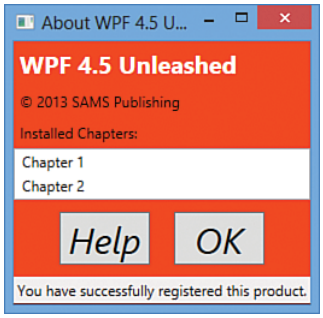

FIGURE 3.8 The About dialog with `FontSize` and `FontStyle` set on both `Buttons` via inheritance from the inner `StackPanel`.

```
StackPanel panel = new StackPanel();
TextElement.SetFontSize(panel, 30);
TextElement.SetFontStyle(panel, FontStyles.Italic);
panel.Orientation = Orientation.Horizontal;
panel.HorizontalAlignment =
HorizontalAlignment.Center;
Button helpButton = new Button();
helpButton.MinWidth = 75;
helpButton.Margin = new Thickness(10);
helpButton.Content = "Help";
Button okButton = new Button();
okButton.MinWidth = 75;
okButton.Margin = new Thickness(10);
okButton.Content = "OK";
panel.Children.Add(helpButton);
panel.Children.Add(okButton);
```

Notice that the enumeration values such as `FontStyles.Italic`, `Orientation.Horizontal`, and `HorizontalAlignment.Center` were previously specified in XAML simply as `Italic`, `Horizontal`, and `Center`, respectively. This is possible thanks to the `EnumConverter` type converter in the .NET Framework, which can convert any case-insensitive string.

Although the XAML in Listing 3.5 nicely represents the logical attachment of `FontSize` and `FontStyle` to `StackPanel`, the C# code reveals that there's no real magic here, just a method call that associates an element with an otherwise-unrelated property. One of the interesting things about the attached property abstraction is that no .NET property is a part of it!