Adam Nathan

Full Color

Figures and code appear as they do in Visual Studio.

Universal Windows® Apps with XAML and C#

UNLEASHED

SAMS

Adam Nathan

Universal Windows® Apps with XAML and C#

UNLEASHED

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

Universal Windows® Apps with XAML and C# Unleashed

ISBN-13: 978-0-672-33726-0
ISBN-10: 0-672-33726-6

Library of Congress Control Number: 2014919777

Printed in the United States of America

First Printing February 2015

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 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.

Special Sales

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

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

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

EDITOR-IN-CHIEF
Greg Wiegand

ACQUISITIONS EDITOR
Joan Murray

DEVELOPMENT EDITOR
Mark Renfrow

MANAGING EDITOR
Kristy Hart

SENIOR PROJECT EDITOR
Betsy Gratner

INDEXER
Lisa Stumpf

PROOFREADER
Kathy Ruiz

TECHNICAL EDITOR
Bill Wagner

EDITORIAL ASSISTANT
Cindy Teeters

COVER DESIGNER
Mark Shirar

COMPOSITOR
Nonie Ratcliff

Contents at a Glance

Introduction 1

Part I Getting Started

1 Hello, *Real* World! 7

2 Mastering XAML 43

Part II Building an App

3 Sizing, Positioning, and Transforming Elements 63

4 Layout 83

5 Handling Input: Touch, Mouse, Pen, and Keyboard 117

Part III Working with the App Model

6 App Lifecycle 161

7 Threading, Windows, and Pages 181

8 The Many Ways to Earn Money 199

Part IV Understanding Controls

9 Content Controls 227

10 Items Controls 259

11 Text 283

12 Images 315

13 Audio, Video, and Speech 355

14 Other Controls 387

Part V Leveraging the Richness of XAML

15 Vector Graphics 421

16 Animation 453

17 Styles, Templates, and Visual States 499

18 Data Binding. 529

Part VI Exploiting Windows

19 Working with Data 555

20 Supporting App Commands 583

21 Leveraging Contracts 613

22 Reading from Sensors 647

23 Controlling Devices 663

24 Thinking Outside the App: Live Tiles, Notifications, and the Lock Screen 687

Index 723

Table of Contents

Introduction 1

Who Should Read This Book? 3
Software Requirements 3
Code Examples 3
How This Book Is Organized 3
Conventions Used in This Book 5

Part I Getting Started

1 Hello, *Real* World! 7

Creating, Deploying, and Profiling an App 7
Understanding the App Packages 10
Updating XAML and C# Code 21
Making the App World-Ready 29
Making the App Accessible 35
Submitting to the Windows Store 40
Summary 42

2 Mastering XAML 43

Elements and Attributes 44
Namespaces 45
Property Elements 47
Type Converters 49
Markup Extensions 49
Children of Object Elements 52
Mixing XAML with C# 56
XAML Keywords 59
Summary 60

Part II Building an App

3 Sizing, Positioning, and Transforming Elements 63

Controlling Size 64
Controlling Position 68
Applying 2D Transforms 72
Applying 3D Transforms 79
Summary 82

4 Layout 83

Discovering Your Window Size and Location 84
`Panels` 88
Handling Content Overflow 103
Summary 115

5 Handling Input: Touch, Mouse, Pen, and Keyboard 117

Touch Input 118
Mouse Input 141
Pen Input 144
Keyboard Input 153
Summary 159

Part III Working with the App Model

6 App Lifecycle 161

Killing 163
Suspending 164
Resuming 166
Terminating 167

Launching 168
Activating 171
Managing Session State with `SuspensionManager` 173
Programmatically Launching Apps 176
Summary 179

7 Threading, Windows, and Pages 181

Understanding the Threading Model for Universal Apps 181
Displaying Multiple Windows 186
Navigating Between Pages 189
Summary 198

8 The Many Ways to Earn Money 199

Adding Advertisements to Your App 200
Supporting a Free Trial 205
Supporting In-App Purchases 210
Validating Windows Store Receipts 218
Testing Windows Store Features 220
Summary 225

Part IV Understanding Controls

9 Content Controls 227

`Button` 230
`AppBarButton` 234
`HyperlinkButton` 241
`RepeatButton` 242
`ToggleButton` 243
`AppBarToggleButton` 243
`CheckBox` 244
`RadioButton` 245
`ToolTip` 246
App Bars 249
Summary 257

10 Items Controls 259

Items in the Control 260
Items Panels 262
`ComboBox` 265
`ListBox` 267
`ListView` 269
`GridView` 273
`FlipView` 274
`SemanticZoom` 276
`MenuFlyout` 279
Summary 281

11 Text 283

`TextBlock` 283
`RichTextBlock` 296
`TextBox` 301
`RichEditBox` 309
`PasswordBox` 311
Summary 313

12 Images 315

The `Image` Element 316
Multiple Files for Multiple Environments 325
Decoding Images 330
Encoding Images 339
Rendering PDF Content as an Image 347
Summary 353

13 Audio, Video, and Speech 355
Playback 356
Capture 367
Transcoding 378
Speech Synthesis 383
Summary 386

14 Other Controls 387
Range Controls 387
`SearchBox` 390
Popup Controls 397
`Hub` 403
Date and Time Controls 407
`ProgressRing` 411
`ToggleSwitch` 412
`WebView` 413
Summary 419

Part V Leveraging the Richness of XAML

15 Vector Graphics 421
`Shapes` 421
Geometries 428
`Brushes` 436
Summary 450

16 Animation 453
Dependency Properties 454
Theme Transitions 455
Theme Animations 466
Custom Animations 472
Custom Keyframe Animations 485
Easing Functions 490
Manual Animations 495
Summary 497

17 Styles, Templates, and Visual States 499
Styles 500
Templates 509
Visual States 519
Summary 528

18 Data Binding 529
Introducing `Binding` 529
Controlling Rendering 538
Customizing the View of a Collection 546
High-Performance Rendering with `ListView` and `GridView` 550
Summary 554

Part VI Exploiting Windows

19 Working with Data 555
An Overview of Files and Folders 555
App Data 557
User Data 563
Networking 572
Summary 582

20 Supporting App Commands 583
Search 584
Share 589
Print 596
Play 604
Project 606
Settings 606
Summary 611

21 Leveraging Contracts 613
Account Picture Provider 615
AutoPlay Content and AutoPlay Device 617
File Type Associations 620
Protocol 623
File Open Picker 624
File Save Picker 627
Contact Picker 628
The Contact Contract 631
The Appointments Provider Contract 635
Background Tasks 637
Summary 646

22 Reading from Sensors 647
Accelerometer 647
Gyrometer 651
Inclinometer 651
Compass 651
Light Sensor 651
Orientation 652
Location 652
Proximity 659
Summary 662

23 Controlling Devices 663
Fingerprint Readers 664
Image Scanners 664
Barcode Scanners 668
Magnetic Stripe Readers 671
Custom Bluetooth Devices 673
Custom Bluetooth Smart Devices 676
Custom USB Devices 679
Custom HID Devices 682
Custom Wi-Fi Direct Devices 684
Summary 686

24 Thinking Outside the App: Live Tiles, Notifications, and the Lock Screen 687
Live Tiles 687
Badges 701
Secondary Tiles 703
Toast Notifications 705
Setting Up Push Notifications 711
The Lock Screen 719
Summary 721

Index 723

About the Author

Adam Nathan is a principal software architect for Microsoft, a best-selling technical author, and a prolific developer of apps for Windows. He introduced XAML to countless developers through his books on a variety of Microsoft technologies. Currently a part of Microsoft's Windows division, Adam has previously worked on Visual Studio and the Common Language Runtime. He was the founding developer and architect of Popfly, Microsoft's first Silverlight-based product, named by *PCWorld* as one of its year's most innovative products. He is also the founder of PINVOKE.NET, the online resource for .NET developers who need to access Win32. His apps have been featured on Lifehacker, Gizmodo, ZDNet, ParentMap, and other enthusiast sites.

Adam's books are considered required reading by many inside Microsoft and throughout the industry. Adam is the author of *Windows 8.1 Apps with XAML and C# Unleashed* (Sams, 2013), *101 Windows Phone 7 Apps* (Sams, 2011), *WPF 4.5 Unleashed* (Sams, 2013), *.NET and COM: The Complete Interoperability Guide* (Sams, 2002), and several other books. You can find Adam online at www.adamnathan.net, or @adamnathan on Twitter.

Dedication

To Tyler and Ryan.

Acknowledgments

I'd like to thank Ashish Shetty, Tim Heuer, Mark Rideout, Jonathan Russ, Joe Duffy, Chris Brumme, Eric Rudder, Neil Rowe, Betsy Gratner, Ginny Munroe, Bill Chiles, Valery Sarkisov, Joan Murray, Patrick Wong, Jacqueline Ting, and Michelle McCarthy. As always, I thank my parents for having the foresight to introduce me to Basic programming on our IBM PCjr when I was in elementary school.

We Want to Hear from You!

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

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

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

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

Email: feedback@samspublishing.com

Mail: Joan Murray
Acquisitions 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

If you ask me, it has never been a better time to be a software developer. Not only are programmers in high demand—due in part to an astonishingly low number of computer science graduates each year—but app stores make it easier than ever to broadly distribute your own software and even make money from it.

When I was in junior high school, I released a few shareware games and asked for $5 donations. I earned $15 total. One of the three donations was from my grandmother, who didn't even own a computer! These days, of course, adults and kids alike can make money on simple apps and games without relying on kind and generous individuals going to the trouble of mailing a check.

With universal Windows apps, it's finally possible to create an app that targets both PCs (desktops, laptops, tablets, and hybrids) and phones simultaneously. Universal apps also represent a consolidation of XAML-based technologies. First there was Windows Presentation Foundation (WPF) for traditional desktop apps, then Silverlight for the Web, then Silverlight for the phone, then the XAML UI Framework for Windows Store apps. All of these frameworks are similar but frustratingly not quite the same. The technology behind universal apps now has enough momentum that the need for these older frameworks should fade away.

In This Chapter

- Who Should Read This Book?
- Software Requirements
- Code Examples
- How This Book Is Organized
- Conventions Used in This Book

Universal apps run on the Windows Runtime, or WinRT for short. WinRT is actually based on Microsoft's Component Object Model (COM) that has been around since 1993, but most of the time you can't tell. And most of the time, it doesn't matter. This is a modern, friendlier version of COM that is more amenable to *automatic* correct usage from environments such as C#. (Contrast this to over a decade ago, when I wrote a book about mixing COM with .NET. This topic alone required over 1,600 pages!)

WinRT APIs are automatically *projected* into the programming language you use, so they look natural for that language. Projections are more than just exposing the raw APIs, however. Core WinRT data types such as `String`, collection types, and a few others are mapped to appropriate data types for the target environment. For C# or other .NET languages, this means exposing them as `System.String`, `System.Collections.Generic.IList<T>`, and so on.

Although WinRT APIs are not .NET APIs, they have metadata in the standardized format used by .NET. Therefore, you can browse them directly with familiar .NET tools, such as the IL Disassembler (ILDASM). You can find these on your computer as .winmd files. Visual Studio's "Object Browser" is also a convenient way to search and browse WinRT APIs.

In the set of APIs exposed by Windows:

- Everything under the **`Windows.UI.Xaml`** namespace is XAML-specific
- Everything under the **`Windows.UI.WebUI`** namespace is for HTML apps
- Everything under **`System`** is .NET-specific
- Everything else (which is under **`Windows`**) is general-purpose WinRT functionality

As you dig into the framework, you notice that the XAML-specific and .NET-specific APIs are indeed the most natural to use from C# and XAML. General-purpose WinRT APIs follow slightly different conventions and can sometimes look a little odd to developers familiar with .NET. For example, they tend to be exception-heavy for situations that normally don't warrant an exception (such as the user cancelling an action). Artifacts like this are caused by the projection mechanism mapping HRESULTs (COM error codes) into .NET exceptions.

I wrote this book 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 how to write universal apps and to show how commonly desired tasks are accomplished
- To be an authoritative source, thanks to input from members of the team who designed, implemented, and tested Windows and Visual Studio
- To be clear about where the technology falls short rather than blindly singing its praises

- ➔ To optimize for concise, easy-to-understand code rather than enforcing architectural patterns that can be impractical or increase the number of concepts to understand
- ➔ To be an easily navigated reference that you can constantly come back to

To elaborate on the second-to-last point: You won't find examples of patterns such as Model-View-ViewModel (MVVM) in this book. I *am* a fan of applying such patterns to code, but I don't want to distract from the core lessons in each chapter.

Whether you're new to XAML or a long-time XAML 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 apps for the Windows Store, whether they are for tablets, laptops, desktops, or phones. It does not teach you how to program, nor does it teach the basics of the C# language. However, it is designed to be understandable even for folks who are new to .NET, and does not require previous experience with XAML.

If you are already well versed in XAML, I'm confident that this book still has a lot of helpful information for you. At the very least, this book should be an invaluable reference for your bookshelf.

Software Requirements

This book targets Windows 8.1, Windows Phone 8.1, and the corresponding developer tools. The tools are a free download at the Windows Dev Center: http://dev.windows.com. The download includes the Windows SDK, a version of Visual Studio Express for Windows, and miscellaneous tools.

Although it's not required, I recommend PAINT.NET, a free download at http://getpaint.net, for creating and editing graphics, such as the set of icons needed by apps.

Code Examples

Source code for examples in this book can be downloaded from www.informit.com/title/9780672337260.

How This Book Is Organized

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

Part I: Getting Started

This part includes the following chapters:

- ➔ Chapter 1: Hello, *Real* World!
- ➔ Chapter 2: Mastering XAML

Part I provides the foundation for the rest of the book. Chapter 1 helps you understand all the tools available at your disposal, and even dives into topics such as accessibility and localization so you can be prepared to get the broadest set of customers possible for your app.

Part II: Building an App

This part includes the following chapters:

- ➔ Chapter 3: Sizing, Positioning, and Transforming Elements
- ➔ Chapter 4: Layout
- ➔ Chapter 5: Handling Input: Touch, Mouse, Pen, and Keyboard

Part II equips you with the knowledge of how to place things on the screen, how to make them adjust to the wide variety of screen types, and how to interact with the user.

Part III: Working with the App Model

This part includes the following chapters:

- ➔ Chapter 6: App Lifecycle
- ➔ Chapter 7: Threading, Windows, and Pages
- ➔ Chapter 8: The Many Ways to Earn Money

The app model for universal apps is significantly different from the app model for traditional desktop applications in a number of ways. It's important to understand how the app lifecycle works and how you need to interact with it in order to create a well-behaved app. But there are other pieces to what is sometimes called the *app model*: how one app can launch another, how to work with the Windows Store to enable free trials and in-app purchases, and how to deal with multiple windows and pages.

Part IV: Understanding Controls

This part includes the following chapters:

- ➔ Chapter 9: Content Controls
- ➔ Chapter 10: Items Controls
- ➔ Chapter 11: Text

- Chapter 12: Images
- Chapter 13: Audio, Video, and Speech
- Chapter 14: Other Controls

Part IV provides a tour of the controls built into the XAML UI Framework. There are many controls that you expect to have available, plus several that you might not expect.

Part V: Leveraging the Richness of XAML

This part includes the following chapters:

- Chapter 15: Vector Graphics
- Chapter 16: Animation
- Chapter 17: Styles, Templates, and Visual States
- Chapter 18: Data Binding

The features covered in Part V are areas in which XAML really shines. Although previous parts of the book expose some XAML richness (applying transforms to any elements, the composability of controls, and so on), these features push the richness to the next level.

Part VI: Exploiting Windows

This part includes the following chapters:

- Chapter 19: Working with Data
- Chapter 20: Supporting App Commands
- Chapter 21: Leveraging Contracts
- Chapter 22: Reading from Sensors
- Chapter 23: Controlling Devices
- Chapter 24: Thinking Outside the App: Live Tiles, Notifications, and the Lock Screen

This part of the book covers unique and powerful Windows features that are not specific to XAML or C#, but they are things that all app developers should know.

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 filenames. In code listings, *`italic monospace type`* is used for placeholder text. Code listings are colorized similarly to the way they are colorized in Visual Studio. `Blue monospace type` is used for XML elements and C# keywords, `brown monospace type` is used for XML element names and 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#.
Bold	When appropriate, bold is used for code directly related to the main lesson(s) in a chapter.

When a line of code is too long to fit on a line in the printed book, a code-continuation arrow (➥) is used.

Throughout this book, and even in this introduction, you will find a number of sidebar elements:

What is a FAQ sidebar?

A Frequently Asked Question (FAQ) sidebar presents a question you might have about the subject matter—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 something you can look into if you're curious but can ignore if you're not.

A tip offers information about design guidelines, shortcuts, or alternative approaches to produce better results, or something that makes a task easier.

This is a 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

In This Chapter

- Creating, Deploying, and Profiling an App
- Understanding the App Packages
- Updating XAML and C# Code
- Making the App World-Ready
- Making the App Accessible
- Submitting to the Windows Store

HELLO, *REAL* WORLD!

"Oh no, not another cliché 'Hello, World' example," you might be thinking as you examine this book. However, the length of this chapter alone should tell you that it is not about creating a typical "Hello, World" app.

Sure, we're going to get started with a simple, contrived app to demonstrate the anatomy of a universal app and the tooling available in Visual Studio. But we'll also see how to make it really say "hello" to the *entire* world; not just English-speaking people with no disabilities. This means understanding how to localize an app into other languages so you can exploit the vast, global scale of the Windows Store. It also means understanding how to make your app accessible to users who require assistive technologies such as screen readers or high contrast themes. No app deserves to be called "Hello, World" without considering these features.

Creating, Deploying, and Profiling an App

In Visual Studio, let's create a new Visual C# **Blank App** project under the **Universal Apps** category and call it `HelloRealWorld`. This actually gives us a solution with the following *three* projects:

- **HelloRealWorld.Windows**—A PC-specific project.
- **HelloRealWorld.WindowsPhone**—A phone-specific project.
- **HelloRealWorld.Shared**—A project referenced by both of the preceding projects.

The solution is ready to compile and run. Although pressing F5 or clicking the **Start Debugging** button in Visual Studio launches the app locally on the PC, you've got many options to choose from via the button's dropdown menu, shown in Figure 1.1.

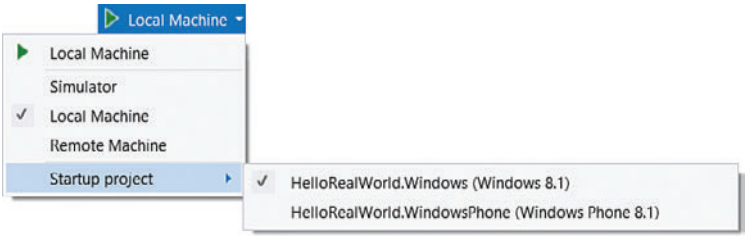


FIGURE 1.1 Options for launching your app in Visual Studio

The choices depend on whether you've got the PC project selected as the startup project (as it is by default), or the phone project selected.

For the PC project, the **Remote Machine** option enables you to deploy and debug to another PC reachable on your network (although not over the Internet). This is extremely handy for testing things on a small tablet not suitable for development work. The target device must have the Remote Tools for Visual Studio installed and running, which you can download from the Windows Dev Center.

The **Simulator** option is the next best thing to having a real tablet, as it provides mechanisms to simulate touch input, device orientations, network conditions, location services, and more. The simulator is shown in Figure 1.2. In fact, it has one huge advantage over testing on a physical device: It enables you to experience your app in a number of different resolutions and virtual screen sizes, including different aspect ratios. Given the wide variety of shapes and sizes of screens out there, not even counting phone screens, testing your app in this fashion is a must.

If you change the active project to be the phone project, you've got two launch options. The default **Device** option deploys your app to a phone that has been developer-unlocked and connected to the PC via USB. (If the phone isn't unlocked, Visual Studio gives you instructions for how to do this.) The other option is the **Emulator**, which gives you a virtual phone on the host PC, much like the simulator.

> **The simulator is your actual computer!**
>
> Although the simulator simulates several things, what you see on the virtual device is your real "host" computer running with your actual user account, apps, files, and so on. (Running the simulator is like initiating a special kind of remote desktop connection to yourself.) Changes you make inside the simulator affect your computer just as if you made them outside the simulator.

The first time you select this option, you're prompted to download the emulator, which requires you to be running on a 64-bit PC that supports Hyper-V.

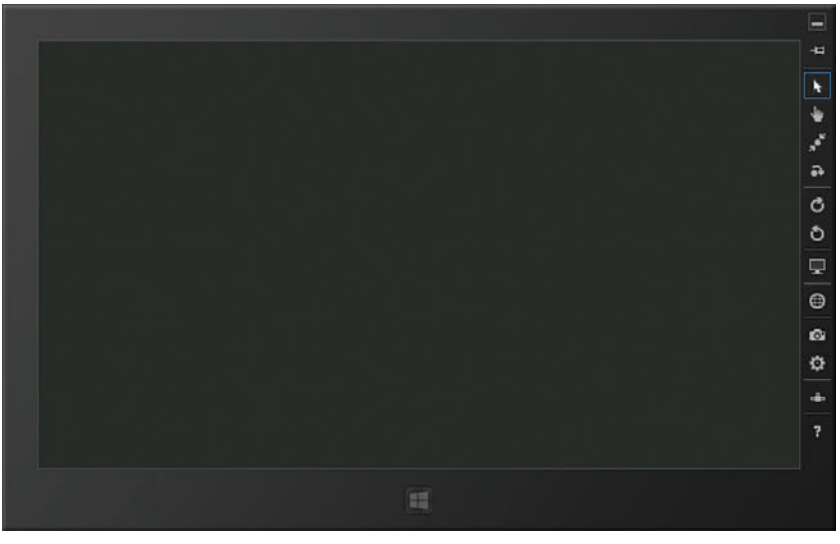


FIGURE 1.2 Testing your app on the simulator is like testing it on an army of different-sized devices.

How do I run my app outside of Visual Studio on my PC?

Although compiling your app produces an `.exe` file in the `bin` subfolder, you can't simply double-click it to run it. If you try, you get an error that explains, "This application can only run in the context of an app container." (An "app container" refers to the sandbox in which universal apps run.) Instead, you can launch it from the list of apps on the Start menu. Visual Studio automatically installs your app the first time you launch it.

When you run the `HelloRealWorld` project without any changes, you'll see why the project type was called "Blank App." The app doesn't actually do anything other than fill the screen with darkness, as seen in the simulator in Figure 1.2. (If you launch the app in debug mode, you'll also see four numbers on the top edge of the screen. These are frame rate counters described in Chapter 16, "Animation.") It does, however, set up a lot of infrastructure that would be difficult and tedious to create from scratch.

The PC project contains the following items:

- A PC-specific package manifest, a temporary certificate used to sign it, and some images
- A PC-specific main page (`MainPage.xaml` and `MainPage.xaml.cs`)
- An `AssemblyInfo.cs` file

The phone project contains the following items:

- A phone-specific package manifest, a temporary certificate used to sign it, and some images
- A phone-specific main page (`MainPage.xaml` and `MainPage.xaml.cs`)
- An `AssemblyInfo.cs` file

The shared project contains a class called `App`, implemented in `App.xaml` and `App.xaml.cs`.

The next section examines the package manifests and the images used by it. After that, we'll look at the XAML and C# files and make some code changes. The important thing to realize is that your universal app is really two separate apps—one for PC and one for phone—that can be given the same identity in the Windows Store. You have the flexibility of sharing no code, or sharing almost all of it.

Visual Studio provides some amazing tools for diagnosing performance problems in your app. You can access them by clicking **Performance and Diagnostics** on the **Debug** menu. On this page, select a tool to collect data while your app is launched. You perform the scenario you want to measure, and then stop the data collection. A rich, interactive report is then presented to you. The four tools on the Performance and Diagnostics page are:

- **XAML UI Responsiveness**—Attributes the time spent to activities such as parsing XAML and layout of your elements. Shows you the performance cost of each UI element. You can also investigate times when you're not achieving the desired 60 frames per second on the UI thread.
- **CPU Usage** and **Memory Usage**—Traditional profiling, with interactive graphs, diagrams of hot paths complete with annotated code integration, and much more.
- **Energy Consumption**—Estimates how power-hungry your app is, based on its usage of the CPU, display, and network.

In addition to the Visual Studio tools, you can download the Windows Performance Toolkit for additional analysis. This includes a Windows Performance Recorder tool for capturing a trace, and a Windows Performance Analyzer tool for analyzing the trace.

Understanding the App Packages

The *package manifest* in the PC and phone projects is a file called `Package.appxmanifest`. ("AppX" is a term sometimes used within Microsoft for app packages that stuck around in the filename.) This manifest describes your app to Windows as well as the Windows Store—its name, what it looks like, what it's allowed to do, and more. It's an XML file, although you have to tell Visual Studio to "View Source" in order to see the XML. There's usually no need to view and edit the XML directly, however. The default view is a tabbed set of forms to fill out, which is the easiest way to populate all the information. There are seven tabs:

- ➔ Application
- ➔ Visual Assets
- ➔ Requirements (phone only)
- ➔ Capabilities
- ➔ Declarations
- ➔ Content URIs
- ➔ Packaging

For our `HelloRealWorld` app, we don't need to change anything in the package manifest. But now is a good time to understand what can be done on each of these tabs. The manifests for phone and PC are almost identical. This section calls out places where they differ. When you change a setting, however, you must remember to change it in both manifest files if you want it to apply to both PCs and phones.

Application

On the Application tab, you can set the app's name and description, default language, its minimum width (PC only), whether to prevent installation to SD cards (phone only), and notification settings (if your app supports them). Notifications are covered in Chapter 24, "Thinking Outside the App: Live Tiles, Notifications, and the Lock Screen." You can even restrict the preferred orientations of your app if you'd rather not have it automatically rotate to all four of them:

- ➔ **Landscape** (horizontal)
- ➔ **Landscape-flipped** (horizontal but upside down)
- ➔ **Portrait** (vertical, with the bottom of the screen on the left)
- ➔ **Portrait-flipped** (vertical, with the bottom of the screen on the right)

Disabling the *flipped* orientations would be an odd thing to do, but disabling some orientations can make sense for certain types of games that wish to be landscape only. Note that this is just a *preference*, not a guarantee, because not all devices support rotation. For example, a portrait-only app launched on a typical desktop PC must accept the one-and-only landscape orientation. However, if a device that *does* support rotation is currently locked to a landscape orientation, a portrait-only app actually runs in the portrait orientation, ignoring the lock setting.

Visual Assets

On the Visual Assets tab, you set the characteristics of your app's tile and splash screen, as well as artwork used in a number of other contexts.

Customizing the Splash Screen

To ensure that every app's splash screen can be displayed practically instantaneously (before your app even gets loaded), you have little control over it. You specify an image (although the dimensions are different for phone versus PC) plus two optional larger sizes to support high DPI screens, and a background color for the splash screen. That's it. Visual Studio gives you an appropriately sized placeholder `SplashScreen.scale-100.png` file in

an Assets subfolder, intentionally made ugly to practically guarantee you won't forget to change it before submitting your app to the Windows Store.

When your splash screen is shown, the image is displayed centered on top of your chosen background color. Figure 1.3 shows an example `SplashScreen.scale-100.png` containing a Pixelwinks logo, and Figure 1.4 shows what this looks like on the PC simulator. The splash screen is given a yellow background for demonstration purposes. A real app should make the background color match the background of the image or simply make the image's background transparent.

FIGURE 1.3 An example `SplashScreen.scale-100.png` with a nontransparent background for demonstration purposes

FIGURE 1.4 A live splash screen shown inside the simulator with a garish yellow background to clearly show the bounds of the image

When your app is launched, the splash screen automatically animates in and automatically disappears once your app has loaded and has made a call to `Window.Current.Activate`. This gives you the flexibility to do arbitrarily complex logic before the splash screen goes away, although you should avoid doing a lot of work here. (Your app is given about fifteen seconds to remain on the splash screen before it gets terminated by Windows.)

Customizing Logo Images

The **Tile Images and Logos** section on the Visual Assets tab can be confusing and overwhelming. Besides the **Store Logo**, which supports up to 3 different sizes, it lists 5 different logo sizes, with each one actually accepting 4–8 different sizes of image files! All told, you can assign *27* different image files representing your logo! Let's start making some sense out of these images. Figure 1.5 shows what each logo *should* have been called to make things less confusing, and the following list explains each one using the terminology found in the package manifest:

- **Square 70x70 Logo** (**Square 71x71 Logo** in the phone manifest)—This is used for the **small** version of your app's tile. Although assigning an image here is optional, the small tile size is not. If you don't provide an image, the medium tile image is used (and scaled down) when a user changes your tile size to small.
- **Square 150x150 Logo**—This is used for the **medium** version of your app's tile. The medium tile size is the one required size, so at least a 100% scale image is required.
- **Wide 310x150 Logo**—This is used for the **wide** version of your app's tile, if you choose to support that tile size. If you assign at least a 100% scale image here, your app automatically supports the wide tile size. Otherwise, it doesn't.
- **Large 310x310 Logo**—This is used for the PC-only **large** version of your app's tile, if you choose to support that tile size. If you assign at least a 100% scale image here *and* for the wide logo, your app automatically supports the large tile size. (Your app can only support a large tile if it also supports a wide tile.) Otherwise, it doesn't.
- **Square 30x30 Logo** (**Square 44x44 Logo** in the phone manifest)—This is used throughout Windows. It is used by the apps list, search results, the Share pane, the file picker, an overlay on live tiles, the Alt+Tab user interface, Task Manager, file icons for associated file types, and so on. At least the 100% scale image is required. Although the image is nominally 30x30 or 44x44 pixels and supports 2–3 additional scaled sizes, this logo supports 4 additional sizes in the PC manifest to be used for file icons on the desktop (if your app has associated file types): 16x16, 32x32, 48x48, and 256x256.
- **Store Logo**—A 50x50 image (at 100% scale) used by the Windows Store. At least the 100% scale image is required.

Visual Studio provides placeholder image files for the required logo images only: the square 150x150 logo, the square 30x30 (PC) and 44x44 (phone) logo, and the store logo.

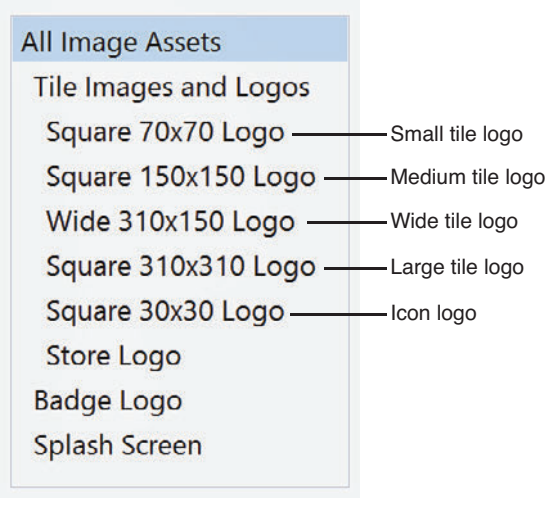


FIGURE 1.5 More understandable names for the different logo images you can provide

To make your tile look good on all devices (and to increase the chances of Microsoft promoting your app in the Windows Store or in advertisements), you should support all scale sizes for each logo you provide. It's perfectly okay to omit large tile and wide tile logos, however. Many of Microsoft's own apps omit them.

Furthermore, it's best *not* to support a large tile and/or wide tile unless you're going to make it a live tile (covered in Chapter 24). Otherwise, your pinned app occupies more space without adding any extra value.

Why does each tile logo support four different image sizes, and how are they used?

Depending on the pixel density of the screen, Windows automatically scales user interfaces to prevent items from being too small to touch or too hard to read. This applies to all universal apps as well as system UI such as the file picker. To prevent your images from looking unsightly by being scaled upward, you can provide multiple versions of any image: one at its normal size, one at 140% of its normal size, and one at 180% of its normal size. The Start screen and Start menu additionally support shrinking its content to an 80% scale.

Windows uses a file naming pattern to manage this, and the package manifest designer in Visual Studio automatically names your assigned image files accordingly. By default, the medium tile icon is assigned to `Assets\Logo.png`. However, at runtime, Windows automatically looks for a file with the following name instead, depending on the current scale being applied:

- `Assets\Logo.scale-80.png` (for 80% scale)
- `Assets\Logo.scale-100.png` (for 100% scale)

- → `Assets\Logo.scale-140.png` (for 140% scale)
- → `Assets\Logo.scale-180.png` (for 180% scale)

This is why the file in your project is actually named `Logo.scale-100.png` despite it being referenced as simply `Logo.png`. (It could drop the `.scale-100` part, however, because 100% scale is assumed for a file without that specification.) If an exact match doesn't exist for the current scale, Windows uses the next best match and scales it accordingly.

The store logo and splash screen images don't support the 80% scale size. The additional four sizes of the square 30x30 logo, assigned to `Assets\SmallLogo.png` by default, use a similar naming scheme:

- → `Assets\SmallLogo.targetsize-16.png` (for 16x16 file icons)
- → `Assets\SmallLogo.targetsize-32.png` (for 32x32 file icons)
- → `Assets\SmallLogo.targetsize-48.png` (for 48x48 file icons)
- → `Assets\SmallLogo.targetsize-256.png` (for 256x256 file icons)

You can use a similar technique for providing different files for high contrast mode, different cultures, and more. This applies not just for the images here, but for images used inside your app as well. See Chapter 12, "Images," for more details.

As with the splash screen, you can specify a background color for your tile. For the best results, this color (as well as the tile images) should match what you use in your splash screen. The desired effect of the splash screen is that your tile springs to life and fills the screen in a larger form. Even if your tile background color is completely covered by opaque tile images, there are still contexts in which the color is seen, such as the zoomed-out Start screen view or the Alt+Tab user interface. Therefore, choose your background color (and determine whether you want your images to use transparency) carefully!

You can choose a "default size," which is the initial size of your tile if the user decides to pin it to Start. This can only be set to the medium tile or the wide tile (if you support a wide tile). If unset, wide is given precedence over medium.

You can also choose a "short name," which is the text that gets overlaid on the bottom of your tile. You can even specify which tile sizes should show the text: medium, wide, and/or large. (Small tiles do not support overlaid text.) Many apps turn off the text because their images already include a logo with the name.

Finally, you can decide whether you want the overlaid text to be "light" (which means white) or "dark" (which means a dark gray). Although most apps use white text, you may need to choose the dark option if you want your tile to have a light background color.

To create a logo that fits in with the built-in apps, it should have a transparent background and the drawing inside should:

- → Be completely white
- → Be composed of simple geometric shapes
- → Use an understandable real-world metaphor

The drawing used in all logo images should look the same, just scaled to different sizes and with different margins.

For example, the drawing for the 150x150 image should generally fit in a 66x66 box centered but nudged a little higher to leave more space for any overlaid text. Typically the drawing has a 42-pixel margin on the left and right, a 37-pixel margin on top, and a 47-pixel margin on the bottom. The drawing for the 30x30 image should generally fit in a 24x24 centered box, leaving just 3 pixels of margin so it's easier to see at the small size. Similarly, the 50x50 store logo drawing should occupy a centered 40x40 square (leaving 5 pixels of margin on each side).

Creating white-on-transparent images requires some practice and patience. You'll want to use tools such as PAINT.NET, mentioned in this book's "Introduction" section. A few of the characters from fonts such as Wingdings, Webdings, and Segoe UI Symbol can even be used to help create a decent icon!

Of course, games or apps with their own strong branding usually do *not* follow these guidelines, as being consistent with their own identity outweighs being consistent with Windows.

Requirements

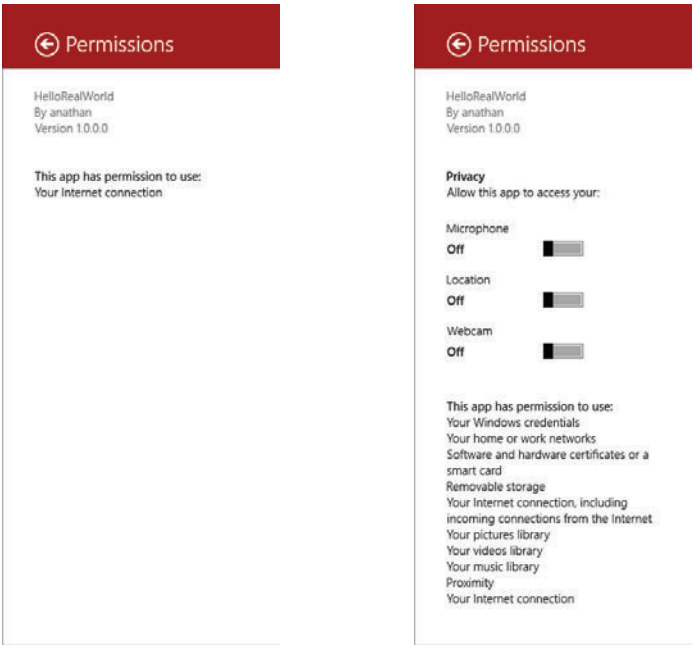
The phone-only Requirements tab enables you to prevent your app from being downloaded/installed on devices that don't meet certain hardware requirements. You can select any of the following requirements: Gyroscope, Magnetometer, NFC, Front Camera, and Rear Camera.

Capabilities

On the Capabilities tab, you select each capability required by your app. A *capability* is a special permission for actions that users might not want certain apps to perform, whether for privacy concerns or concerns about data usage charges. In the Windows Store, prospective users are told what capabilities each app requires before they decide whether to download it. To users, they are described as *permissions*, sometimes with more descriptive names.

For the most part, user approval of all requested permissions is an implicit part of downloading an app. However, the use of privacy-related capabilities, such as location services, prompts the user the first time an app invokes a relevant API when it runs on a PC. Furthermore, some capabilities can be disabled or reenabled at any time by a user. When the Settings pane is shown while a universal app is running on a PC, it contains a "Permissions" link that displays an app's capabilities and toggle switches for any that can be turned on and off. Figure 1.6 shows what this looks like while running `HelloRealWorld` on a PC, both with the default capability already chosen in our package manifest—Internet (Client)—and after selecting every listed capability in the package manifest.

You want to restrict the set of capabilities requested by your app as much as possible, because it is a competitive advantage. For example, users might decide not to buy your fun piano app if it wants permission to use the Internet!



When the app uses the Internet (Client) capability

When the app uses every listed capability

FIGURE 1.6 The "Permissions" section of the Settings pane lists the current app's capabilities, and enables turning some of them on or off at runtime.

Excluding the phone-only Appointments and Contacts capabilities, the long list of available capabilities can be grouped into four different categories:

- File capabilities
- Device capabilities
- Network capabilities
- Identity capabilities

Most of them can be used freely, although some of them are restricted. Apps that use restricted capabilities must go through extra processes when uploaded to the Windows Store and are only granted to business developer accounts with written justification. Fortunately, the restricted capabilities (called out in the upcoming lists) are for uncommon scenarios.

File Capabilities

As you'll read in Chapter 19, "Working with Data," apps can read and write their own private files in an isolated spot, and those files can even participate in automatic roaming between a user's devices. In addition, users can give apps explicit permission to read/write

other "normal" files and folders via the Windows file picker. This is all that most apps need, and does not require any capabilities.

Beyond these two features, however, programmatic reading and writing of files requires special capabilities. There is one for each of the four built-in libraries (Documents, Music, Pictures, and Videos) plus another for attached storage devices:

- **Music Library**, **Pictures Library**, and **Videos Library**—Enables enumerating and accessing all music, pictures, and videos, respectively, *without* going through the file picker.
- **Documents Library** (PC only)—Enables adding, changing, and deleting files in the Documents library on the local computer *without* going through the file picker. However, this capability is restricted to specific file type associations that must also be declared in the package manifest (on the Declarations tab). This is listed separately from the preceding three capabilities because it is a restricted capability that needs special approval from Microsoft in order to publish the app in the Windows Store. And unlike the capabilities for the Music, Pictures, and Videos libraries, this cannot be used to access Documents libraries on other computers in the same HomeGroup.
- **Removable Storage**—Enables adding, changing, and deleting files on devices such as external hard drives or thumb drives connected to the local computer, again *without* going through the file picker. As with the preceding capability, this is restricted to file type associations that must also be declared in the package manifest.

Device Capabilities

Apps can access simple sensors such as an accelerometer or devices such as a printer without any capabilities. Accessing other sensors or devices does require specific capabilities, however. The list of device types grows over time (and can be extended by third parties), but the Capabilities tab exposes four choices, listed below. For all of them except proximity, users can disable them at any time, so apps must be prepared to handle this gracefully.

- **Location**—Reveals the computer's location, either precise coordinates from a GPS sensor (if one exists) or an estimation based on network information.
- **Microphone**—Enables recording audio from a microphone.
- **Webcam**—Enables recording video—or capturing still pictures—from a camera. Note that this doesn't include sound. If you want to record audio and video, you need both Webcam and Microphone capabilities.
- **Proximity**—Enables communication with nearby devices, either via Wi-Fi Direct or near field communication (NFC).

Chapters 13, "Audio, Video and Speech," and 22, "Reading from Sensors," explain how to write apps that take advantage of these capabilities. Additional device capabilities exist

that don't appear on the Capabilities tab. These must be added manually to the package manifest XML. See Chapter 23, "Controlling Devices," for more information.

Network Capabilities

Without any network capabilities, a universal app cannot do any communication over any kind of network except for the automatic roaming of application data described in Chapter 19, the seamless opening/saving of network files enabled by the file picker, or the peer-to-peer connections enabled by the Proximity capability. Four types of network capabilities exist:

- **Internet (Client)** (PC only)—This is the only network capability that most apps need. It provides outbound access to the Internet and public networks (going through the firewall).
- **Internet (Client & Server)**—This is just like the preceding capability except it provides both inbound and outbound access, which is vital for peer-to-peer apps. It's a superset of "Internet (Client)" so if you request this capability in your manifest, then you don't need to request the other one. In the phone manifest, this is the only Internet option.
- **Private Networks (Client & Server)** (PC only)—Provides inbound and outbound access to trusted home and work networks (going through the firewall).
- **Enterprise Authentication**—Enables intranet access using the current Windows domain credentials. This is a restricted capability.

Visual Studio project templates enable an Internet capability by default!

By default, the PC manifest has "Internet (Client)" enabled, and the phone manifest has "Internet (Client & Server)" enabled. This is done because the Visual Studio team feared that it would be too confusing for developers if simple network-dependent calls failed in their brand new projects. Therefore, be sure to remove the capability if you don't need it.

Identity Capabilities

This is not really a fourth category of capabilities, but rather a single outlier that doesn't fit anywhere else. The Shared User Certificates capability enables access to digital certificates that validate a user's identity. The certificate could be installed on the computer or stored on a smart card. This is mainly for enterprise environments, and it is a restricted capability.

Declarations

The Declarations tab is the one with the most options. This is where you declare your app's support for one or more *contracts*, if applicable. Contracts enable your app to cooperate with another app, or Windows itself, to complete a well-defined task. Every contract has a *source* that initiates the task and a *target* that completes it.

Your app can be the source for a contract without doing anything in the package manifest. (It just makes various API calls.) To be the target, however, your app must

be activated in a special manner. This is what requires the declaration in the package manifest. Therefore, you can think of the list of available *declarations* as the list of available *contract targets*.

Unlike capabilities, contract target declarations are *not* listed in the Windows Store as potentially unwanted features. In fact, you should go out of your way to mention your supported contract scenarios, because they can be very useful! There's nothing about being a contract target that is inherently dangerous for the user. Supporting certain contracts does require relevant capabilities, but many don't require any. See Chapter 21, "Leveraging Contracts," for specific examples.

Content URIs

This tab only applies if you are hosting HTML content inside your app. It simply houses a list of HTTPS URLs whose JavaScript is allowed (or disallowed) to raise events that can be handled by your app. For more information, see the discussion of the `WebView` control in Chapter 14, "Other Controls."

Packaging

The Packaging tab is meant to describe information needed for the app's listing in the Windows Store. However, for apps in the store, this information is managed by the Windows Dev Center dashboard. You therefore don't normally need to change these values in your local package manifest:

- The **package name** is a unique identifier. Visual Studio automatically fills it in with a globally-unique identifier known as a GUID. That said, for easier debugging and identification of your app's local data store, it's best to replace the GUID with a human-readable name, such as *CompanyName.AppName*. This name doesn't impact real users of your app, as the Windows Store assigns this value in the package that users download.
- The **package display name** is the name of your app in the store, but this also gets replaced when you follow the procedure to upload an app, so you can leave this item alone.
- The **version**, set to 1.0.0.0 by default, is a four-part value interpreted as *Major.Minor.Build.Revision*.
- The bottom of this tab contains publisher information based on the certificate used to authenticate the package. Visual Studio configures this to work with the temporary certificate it generates, and the store upload process reconfigures it to work with your developer account.

For testing certain notification or purchase scenarios that depend on an app's identity in the Windows Store, you can automatically update your local package manifest's packaging values to match the values maintained by the Windows Store. To do this, you can select **Associate App with the Store…**, which can be found on the **Store** menu in Visual Studio Express or on the **Project, Store** menu in other editions.

Updating XAML and C# Code

With the tour of the package manifests complete, we are ready to fill our blank app with a little bit of content. Let's look at the remaining files in our project and update them where necessary.

In this example, we want to use the exact same main page for both PCs and phones. Therefore, drag `MainPage.xaml` from the PC project to the shared project, then delete the copy of the file from the PC and phone projects. (These actions automatically apply to `MainPage.xaml.cs` as well, which is a child node of `MainPage.xaml` in Solution Explorer.) Now we are ready to work on a single codebase for both types of devices.

The Main Page User Interface

Every app consists of one or more windows with one or more pages. Our `HelloRealWorld` project, created from the Blank App template, is given a single window with a single page called `MainPage`. It defines what the user sees once your app has loaded and the splash screen has gone away. `MainPage`, like any page that would be used in an app, is implemented across two files: `MainPage.xaml` contains the user interface, and `MainPage.xaml.cs` contains the logic, often called the *code-behind*. Listing 1.1 shows the initial contents of `MainPage.xaml`.

LISTING 1.1 `MainPage.xaml`—The Initial Markup for the Main Page

```
<Page
  x:Class="HelloRealWorld.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:HelloRealWorld"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">
  <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
  </Grid>
</Page>
```

At a quick glance, this file tells us:

- This is a class called `MainPage` (in the `HelloRealWorld` namespace) that derives from a class called `Page` (the root element in this file).
- It contains an empty `Grid` (an element examined in Chapter 4, "Layout") whose background is set to a theme-defined color. From running the app, we know this color is a very dark gray (`#1D1D1D`).
- It contains a bunch of XML namespaces to make adding new elements and attributes that aren't in the default namespace more convenient. These XML namespaces are discussed in the next chapter.

Listing 1.2 updates the blank-screen `MainPage.xaml` with a few elements to produce the result in Figure 1.7, shown running on a PC.

LISTING 1.2 `MainPage.xaml`—Updated Markup for the `HelloRealWorld` App

```
<Page
  x:Class="HelloRealWorld.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:HelloRealWorld"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">
  <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel Name="stackPanel" Margin="100" Background="Blue">
      <TextBlock FontSize="80" TextWrapping="WrapWholeWords" Margin="12,48">
        Hello, English-speaking world!</TextBlock>
      <TextBlock FontSize="28" Margin="12">Please enter your name:</TextBlock>
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition/>
          <ColumnDefinition Width="Auto"/>
        </Grid.ColumnDefinitions>
        <TextBox Name="nameBox" Margin="12"/>
        <Button Grid.Column="1" Click="Button_Click">Go</Button>
      </Grid>
      <TextBlock Name="result" FontSize="28" Margin="12"/>
    </StackPanel>
  </Grid>
</Page>
```

This listing adds a bunch of new content inside the topmost `Grid`. The `Grid` and `StackPanel` elements help to arrange the user-visible elements: `TextBlock`s (i.e. labels), a `TextBox`, and a `Button`. All of these elements are described in depth in upcoming chapters.

The idea for this app is to display the user's name in the `TextBlock` named `result` once he or she clicks the Go `Button`. (Granted, this is not a useful app, but it's all we need to demonstrate the concepts throughout the remainder of this chapter.) To act upon the `Button` being clicked, this XAML specifies that a method called `Button_Click` should be called when its `Click` event is raised. This method must be defined in the code-behind file, which we'll look at next.

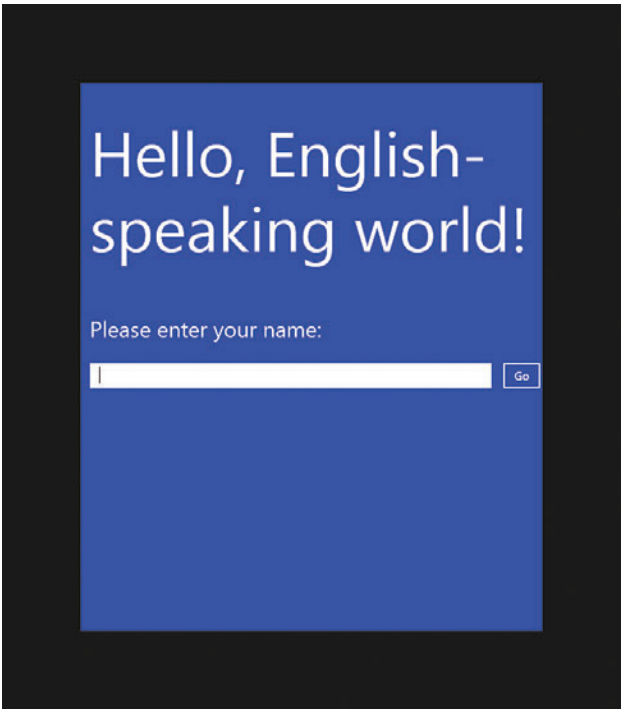


FIGURE 1.7 The `HelloRealWorld` user interface asks the user to type his or her name.

The Main Page Logic

Listing 1.3 shows the initial contents of `MainPage.xaml.cs`, the code-behind file for `MainPage.xaml`. Until we add our own logic, it contains only a required call to `InitializeComponent` that constructs the page with all the visuals defined in the XAML file. The class is marked with the `partial` keyword because its definition is shared with a hidden C# file that gets generated when the XAML file is compiled.

LISTING 1.3 `MainPage.xaml.cs`—The Initial Code-Behind for the Main Page

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
```

```
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

// The Blank Page item template is documented at
// http://go.microsoft.com/fwlink/?LinkId=234238

namespace HelloRealWorld
{
  /// <summary>
  /// An empty page that can be used on its own or navigated to within a Frame.
  /// </summary>
  public sealed partial class MainPage : Page
  {
    public MainPage()
    {
      this.InitializeComponent();
    }
  }
}
```

Never remove the call to **`InitializeComponent`** in the constructor of your code-behind class!

`InitializeComponent` is what associates your XAML-defined content with the instance of the class at run-time.

We need to add an implementation of the `Button_Click` method referenced by the XAML. It can look as follows:

```
void Button_Click(object sender, RoutedEventArgs e)
{
  this.result.Text = this.nameBox.Text;
}
```

The named elements in the XAML correspond to fields in this class, so this code updates the `result` `TextBlock` with the text from the `nameBox` `TextBox`. Figure 1.8 shows what this looks like, after the user types "Adam" then clicks the `Button`.

The Application Definition

The application definition is contained in `App.xaml` and its code-behind file, `App.xaml.cs`. `App.xaml` is a special XAML file that doesn't define any visuals, but rather defines an `App` class that can handle application-level tasks. Usually the only reason to touch this XAML file is to place new application-wide resources, such as custom styles, inside its

`Application.Resources` collection. Chapter 17, "Styles, Templates, and Visual States," contains many examples of this. Listing 1.4 shows the contents of `App.xaml` in our `HelloRealWorld` project.

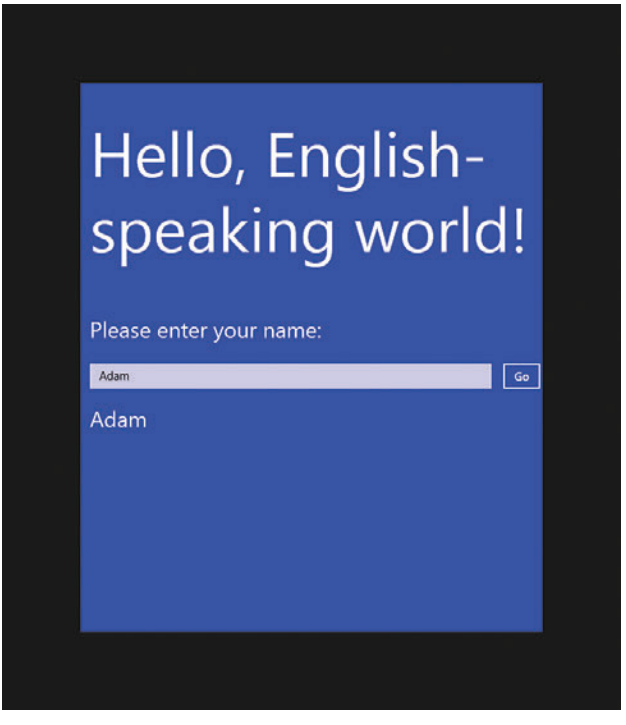


FIGURE 1.8 The `result TextBlock` contains the typed text after the user clicks the `Button`.

LISTING 1.4 `App.xaml`—The Markup for the App Class

```
<Application
  x:Class="HelloRealWorld.App"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:HelloRealWorld">
</Application>
```

Listing 1.5 contains the auto-generated contents of the code-behind file for `App.xaml`. It contains three vital pieces:

- A constructor, which is effectively the app's `main` method. The plumbing that makes it the app's entry point is enabled by an "Entry point" setting in the package manifest (on the Application tab). When you create a project, Visual Studio

automatically sets it to the namespace-qualified name of the project's `App` class (`HelloRealWorld.App` in this example).

- Logic inside an `OnLaunched` method that enables the frame rate counter overlay in debug mode, navigates to the app's first (and in this case only) page, and calls `Window.Current.Activate` to dismiss the splash screen. If you want to add a new page and make it be the starting point of the app, or if you want to customize the initialization logic, this is where you can do it. See Chapter 6, "App Lifecycle," for more information.
- An `OnSuspending` method that is attached to the base class's `Suspending` event. This gives you an opportunity to save state before your app is suspended, although the generated code does nothing here other than provide a TODO comment. Chapter 6 examines app suspension.

LISTING 1.5 `App.xaml.cs`—The Code-Behind for the App Class

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.ApplicationModel;
using Windows.ApplicationModel.Activation;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Media.Animation;
using Windows.UI.Xaml.Navigation;

namespace HelloRealWorld
{
  /// <summary>
  /// Provides application-specific behavior to supplement the base class.
  /// </summary>
  sealed partial class App : Application
  {
#if WINDOWS_PHONE_APP
    private TransitionCollection transitions;
#endif
```

```
    /// <summary>
    /// Initializes the singleton application object.  This is the first line
    /// of authored code executed; the logical equivalent of main/WinMain.
    /// </summary>
    public App()
    {
      this.InitializeComponent();
      this.Suspending += OnSuspending;
    }

    /// <summary>
    /// Invoked when the application is launched normally by the end user.
    /// Other entry points are used when the application is launched to open
    /// a specific file, to display search results, and so forth.
    /// </summary>
    /// <param name="args">Details about the launch request and process.</param>
    protected override void OnLaunched(LaunchActivatedEventArgs args)
    {
#if DEBUG
      if (System.Diagnostics.Debugger.IsAttached)
      {
        this.DebugSettings.EnableFrameRateCounter = true;
      }
#endif

      Frame rootFrame = Window.Current.Content as Frame;

      // Do not repeat app initialization when the Window already has content,
      // just ensure that the window is active
      if (rootFrame == null)
      {
        // Create a Frame and navigate to the first page
        var rootFrame = new Frame();

        // TODO: change this value to a cache size
        // that is appropriate for your application
        rootFrame.CacheSize = 1;

        if (args.PreviousExecutionState == ApplicationExecutionState.Terminated)
        {
          //TODO: Load state from previously suspended application
        }

        // Place the frame in the current Window
        Window.Current.Content = rootFrame;
      }
```

```
      if (rootFrame.Content == null)
      {
#if WINDOWS_PHONE_APP
        … Code related to animations
#endif

        // When the navigation stack isn't restored, navigate to the first page
        if (!rootFrame.Navigate(typeof(MainPage), args.Arguments))
        {
          throw new Exception("Failed to create initial page");
        }
      }

      // Ensure the current Window is active
      Window.Current.Activate();
    }

#if WINDOWS_PHONE_APP
  … More code related to animations
#endif

    /// <summary>
    /// Invoked when application execution is being suspended.  Application state
    /// is saved without knowing whether the application will be terminated or
    /// resumed with the contents of memory still intact.
    /// </summary>
    /// <param name="sender">The source of the suspend request.</param>
    /// <param name="e">Details about the suspend request.</param>
    private void OnSuspending(object sender, SuspendingEventArgs e)
    {
      var deferral = e.SuspendingOperation.GetDeferral();
      //TODO: Save application state and stop any background activity
      deferral.Complete();
    }
  }
}
```

Although the same `App.xaml.cs` file is compiled for both PC and phone projects, it demonstrates a way to write phone-specific or PC-specific code within a shared file: using conditional compilation with the `WINDOWS_PHONE_APP` symbol.

If you want to create a richer splash screen, perhaps with an animated progress graphic, the way to do this is by mimicking the splash screen with a custom page. Inside `App.OnLaunched`, you can navigate to an initial page that looks just like the real (static) splash screen but with extra UI elements and custom logic. The instance of `LaunchActivatedEventArgs` passed to `OnLaunched` even has a `SplashScreen` property that exposes an `ImageLocation` rectangle that tells you the coordinates of the real splash screen image. This makes it easy to match the splash screen's appearance no matter what the current screen's resolution is. Such a user interface is often called an "extended splash screen."

There's one more file that appears in the PC and phone projects—`AssemblyInfo.cs`—but it's not worth showing in this book. It contains a bunch of attributes where you *can* put a title, description, company name, copyright, and so on that get compiled into your assembly (the EXE or DLL). But setting these is unnecessary because all of the information used by the Windows Store is separately managed. Still, the `AssemblyVersion` and `AssemblyFileVersion` attributes, typically set to the same value, can be useful for you to keep track of distinct versions of your application:

```
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

By using *-syntax, such as `"1.0.*"`, you can even let the version number auto-increment every time you rebuild your app.

Making the App World-Ready

At this point, our `HelloRealWorld` app still only says "hello" to the English-speaking parts of the world. The Windows Store serves hundreds of markets and over a hundred different languages, so ignoring them greatly reduces the audience for your app. Making your app world-ready involves two things: *globalization* and *localization*.

Globalization refers to making your app act appropriately for different markets without any changes or customizations. An example of this is formatting the display of currency correctly for the current region without writing special-case logic. The `Windows.Globalization` namespace contains a lot of functionality for handling dates and times, geographic regions, number formatting, and more. Plus, built-in XAML controls such as `DatePicker` and `TimePicker`, discussed in Chapter 14, are globalization-ready. For many apps, these features might not apply.

Localization, which is relevant for practically every app, refers to explicit activity to adapt an app to each new market. The primary example of this is translating text in your user interface to different languages and then displaying the translations when appropriate. Performing this localization activity is the focus of this section.

To make an app ready for localization, you should remove hardcoded English strings that are user-visible, and instead mark such elements with a special identifier unique within the app. Listing 1.6 updates our XAML from Listing 1.2 to do just that.

LISTING 1.6 `MainPage.xaml`—Markup with User-Visible English Text Removed

```
<Page
  x:Class="HelloRealWorld.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:local="using:HelloRealWorld"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  mc:Ignorable="d">
  <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel x:Uid="Panel" Name="stackPanel" Margin="100">
      <TextBlock x:Uid="Greeting" FontSize="80" TextWrapping="WrapWholeWords"
                                                Margin="12,48"/>
      <TextBlock x:Uid="EnterName" FontSize="28" Margin="12"/>
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition/>
          <ColumnDefinition Width="Auto"/>
        </Grid.ColumnDefinitions>
        <TextBox Name="nameBox" Margin="12"/>
        <Button x:Uid="GoButton" Grid.Column="1" Click="Button_Click"/>
      </Grid>
      <TextBlock Name="result" FontSize="28" Margin="12"/>
    </StackPanel>
  </Grid>
</Page>
```

The `x:Uid` marking is completely independent from an element's `Name`. The former is specifically for the localization process, and the latter is for the benefit of code-behind. Note that Listing 1.6 not only removes the three hardcoded strings from the two `TextBlock`s and the `Button`, but it also removes the explicit `"Blue"` color from the `StackPanel`! This way, we can customize the color for different languages in addition to the text.

With the IDs in place and the text and color for English removed, we need to add them back in a way that identifies them as English-only. To do this, add a new folder to the shared project called **en**. This is the language code for all variations of English. If you want to target the United Kingdom separately, you could add a folder called **en-GB**. If you want to target Canada separately, you could add a folder called **en-CA**. And so forth.

Right-click on the **en** folder and select **Add**, **New Item**, then pick **Resources file** from the **General** tab. The default name of `Resources.resw` is fine. This file is a table for all your language-specific strings. Figure 1.9 shows this file populated for English.

Each value must be given a name of the form *UniqueId*.*PropertyName*.*UniqueId* must match the `x:Uid` value for the relevant element, so the `Panel.Background` entry in Figure 1.9 sets `Background` to `Blue` on the `StackPanel` marked with `x:Uid="Panel"` in Listing 1.6. From the listing, it's not obvious that `GoButton`'s relevant property is called `Content`, unlike the `TextBlock`s' property called `Text`, but as you learn about the different elements throughout this book, you'll understand which properties to set.

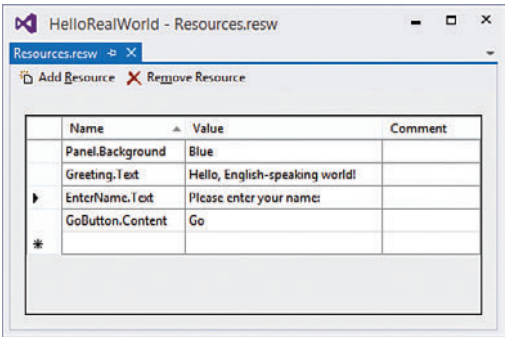


FIGURE 1.9 The `Resources.resw` file in the `en` folder is populated with English-specific values.

> **Make sure your app's default language matches the language code for your default `.resw` file!**
>
> For me, the default language in the package manifests is set to `en-US`. Because we added default resources for `en` rather than specifically for `en-US`, I must change the default language in both manifests to `en` for the rest of the features discussed in this section to work correctly. Fortunately, building your app with such a mismatch causes a warning to be reported.

After filling out the `Resources.resw` file, you can run the `HelloRealWorld` app and the result is identical to what we saw earlier in Figures 1.7 and 1.8. However, the app is now ready to be localized for other languages.

We could add additional folders named after language codes and manually populate translated resources with the help of a knowledgeable friend, a professional translator, or translation software. Depending on the current user's language settings, the appropriate resources are chosen at runtime, with a fallback to the default language if no such resources exist.

However, a better option exists. To take advantage of it, you must download and install the Multilingual App Toolkit from the Windows Dev Center. Once you do this, you can select **Enable Multilingual App Toolkit** from Visual Studio's **Tools** menu. Unfortunately, you must do this twice: once with the PC project highlighted, and once with the phone project highlighted. This automatically adds an `.xlf` file to a new subfolder added to the selected project called **MultilingualResources** for a test-only language called Pseudo Language.

We'll leverage the Pseudo Language in a moment, but first let's add support for a second *real* language: Traditional Chinese. To do this, right-click on each project in Solution Explorer and select **Add translation languages...**. This produces the dialog shown in Figure 1.10.

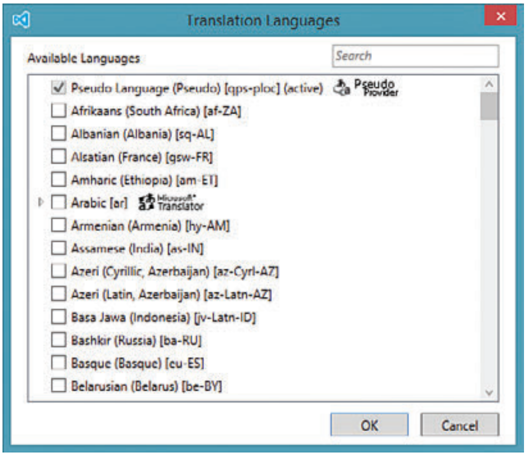


FIGURE 1.10 The Multilingual App Toolkit automates the process for supporting new languages.

In this dialog, Pseudo Language and our default English language is already selected, but we can scroll down and select **Chinese (Traditional) [zh-Hant]** from the list. After pressing OK, the **MultilingualResources** folder now has two `.xlf` files: one for Pseudo Language, and one for Traditional Chinese.

What is Pseudo Language?

Pseudo Language is designed to test how well your app handles being localized to various (real) languages. When leveraging machine translation to Pseudo Language, you get an English-looking string whose contents are still recognizable, but designed to catch problems.

Pseudo Language strings are longer than the corresponding English strings, to help you catch cases where text might get truncated or cause issues from wrapping when you translate to a *real* language whose text tends to be longer than English. Each string also begins with an ID, to help you track a problematic piece of text to its original resource. For example, a Pseudo Language translation of `Hello, English Speaking World!` can look like `[07223][!!_Ĥęłľó, Éйġļįśħ-şpêãќîиġ ώòŕłđ!_!!]`. Because of the unique appearance of Pseudo Language, it also helps you catch user-visible text in your user interface that you forgot to extract to a resource.

What are `.xlf` files?

These files, which are generated by the Multilingual App Toolkit, are XLIFF files, an industry-standard XML format for localizable data. In addition to listing source and target strings (with optional comments), these files enable a workflow in which resources can be marked as New, Needs Review, Translated, Final, or Signed Off.

The benefit of using XLIFF files to store translations is that you can send them directly to a professional translation vendor, as they should already have a workflow involving this format.

Or, if you leverage friends to do your translations, you can have them install the Multilingual App Toolkit and use its Multilingual Editor in a standalone fashion. No Visual Studio installation is necessary.

Visual Studio includes functionality for packaging and sending XLIFF files, as well as importing updated files that merge with your local content. These options can be found by right-clicking an `.xlf` file in Solution Explorer.

Now rebuild the `HelloRealWorld` app. This populates each `.xlf` file with a "translation" for each item from the default language `.resw` file. Initially, each translation is just the duplicated English text. However, for some languages, such as the two we've chosen, you can generate machine translations based on the Microsoft Translator service! To do this for the entire file, right-click on each `.xlf` file and select **Generate machine translations**. Voilà! Now we've got initial translations for all of our resources, which you can see by opening each `.xlf` file and examining the list inside the multilingual editor. This is shown in Figure 1.11.

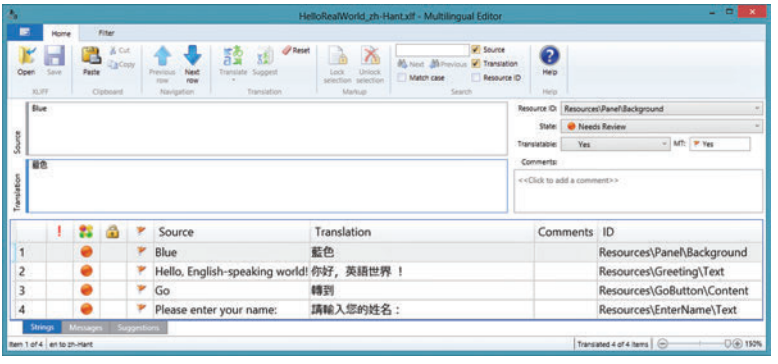
Your willingness to trust the results from machine translation is a personal decision, but at least machine translation is a good starting point. (Notice that the generated translations are automatically placed in a "Needs Review" state.) That said, we definitely don't want the `Blue` text translated to 藍色! This isn't a user-visible string, and 藍色 is not a valid value for `Background`. Instead, let's "translate" it to `Red`, which will serve as our language-specific background color. Similarly, we don't want `Blue`'s Pseudo Language translation of `[D05A0] [!!_Bļùè_!!]`, so let's change that to `Green`.

We have one more change to make. We don't want "Hello, English-speaking world!" to be translated to Chinese, but rather "Hello, Chinese-speaking world!" Both Microsoft Translator and a colleague tell me that "你好, 華語世界!" is a valid translation, so we can paste that into the appropriate spot of the Chinese `.xlf` file.

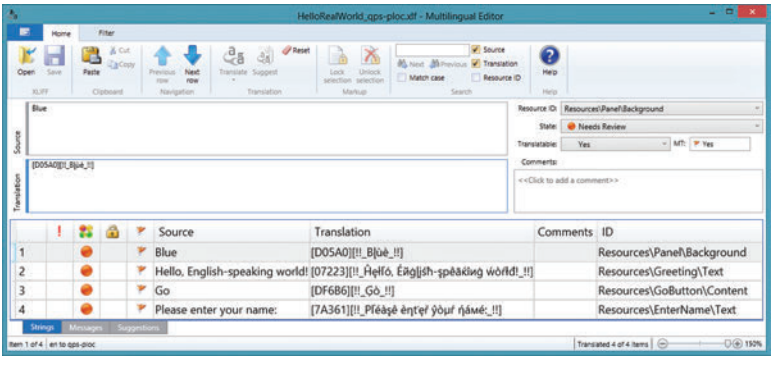
After rebuilding the solution, we are now ready to test the localized versions of `HelloRealWorld`. Just as if we had manually added separate `.resw` files in per-language folders, the translated resources are used automatically based on the current Windows language settings.

To change the default language used by Windows on a phone, you can go to the **language** section of the Settings app. To change the default language used by Windows on a PC, you can either use the PC Settings app or the Control Panel. In PC Settings, this can be found under **Time & language**; **Region & language**. In Control Panel, it's under **Clock, Language, and Region**; **Language**. Add **Chinese (Traditional)** and make it the default language to test the Traditional Chinese resources.

To add Pseudo Language (and make it the default language), you have to use a hidden trick in Control Panel. After clicking **Add languages**, type **qps-ploc** in the search box for the entry called **English (qps-ploc)** to appear. You must type *the whole thing* for this to work! This language is hidden in this way because no normal user should ever enable it.



Chinese (Traditional)



Pseudo Language

FIGURE 1.11 Each `.xlf` file contains machine-generated initial translations, courtesy of Microsoft Translator.

Figure 1.12 shows the result of running `HelloRealWorld` on a PC when Windows is set to use each of the two non-English languages. These changes are handled completely by the resource-loading mechanism. Other than the switch to marking elements with `x:Uid`, no code changes were needed. This figure also highlights Pseudo Language's knack for using really long strings that can highlight potential weaknesses in your app's layout.

You can add additional languages to your apps that have already been published in the Windows Store, thanks to *resource pack* support. As long as you don't update any code or your version number, your new resources get downloaded only to users with a matching language preference.

Chinese (Traditional) Pseudo Language

FIGURE 1.12 `HelloRealWorld` now acts appropriately for Traditional Chinese and for the test-only Pseudo Language.

The Microsoft Local Language Portal (www.microsoft.com/language) is a fantastic resource for getting translations. You can search for terms and get a translation in every language supported by Windows (over 100). These are not machine translations, but rather translations Microsoft has used in their own products. As such, they tend to be geared towards the kind of user-visible labels that are commonly found in software. The portal even shows you which products have made use of the translated terms. Just be sure you agree with the license and terms of use, which can be found on the website.

Making the App Accessible

Universal apps have a number of accessibility features built in, designed to help users with disabilities. You can test this support by enabling various features in the Ease of Access section in the PC Settings app or the Settings app on a phone. You can configure Narrator, a screen reader, and witness it convey information about your app with varying degrees of success. (You can quickly toggle Narrator on and off by pressing Windows+Enter.) You can choose a high contrast theme and watch controls used by your app automatically change to match the theme. You can turn off standard animations. And so on.

To make your app usable to the broadest set of customers, including people with disabilities, you should take steps to ensure it works even better with these assistive technologies. In this section, we look at improving the screen reading experience for our `HelloRealWorld` app, and accounting for high contrast themes.

The Windows SDK includes several tools that help you ensure that your app is accessible. The most important one is **UI Accessibility Checker**, which reports missing accessibility information in your app. Others are **Inspect**, which is a viewer for accessibility data on your elements, and **Accessible Event Watcher**, which focuses on the accessiblity events that should be raised.

Improving Screen Reading

If you turn on Narrator and launch the `HelloRealWorld` app (with English as the Windows default language), you hear the following:

> *"HelloRealWorld window"*
>
> *"Editing"*

The first utterance is triggered by the app's window getting focus, and the second utterance is triggered by the `TextBox` getting focus (which happens automatically).

This experience isn't good enough, because Narrator doesn't report the purpose of the `TextBox`. To fix this, we need to leverage the UI Automation framework, which is as simple as setting the following *automation property* on the `TextBox`:

```
<TextBox AutomationProperties.Name="Please enter your name"
         Name="nameBox" Margin="12"/>
```

If you add this property then rerun `HelloRealWorld` with Narrator on, you will hear the following:

> *"HelloRealWorld window"*
>
> ***"Please enter your name"***
>
> *"Editing"*

Note that when you give the Go `Button` focus, such as by pressing Tab, Narrator says:

> *"Go button"*

This works automatically, thanks to built-in `Button` behavior that reports its content to the UI Automation framework.

When you click the `Button`, however, Narrator gives no indication that text has been added to the screen. If a message is worth showing, then it's worth hearing as well. To fix this problem, we can add the following automation property to the `result TextBlock` that identifies it as a live region:

```
<TextBlock AutomationProperties.LiveSetting="Polite"
           Name="result" FontSize="28" Margin="12"/>
```

A live region is an area whose content changes. This `AutomationProperties.LiveSetting` property can be set to one of the following values:

- **`Off`**—This is the default value.
- **`Polite`**—Changes should be communicated, but they should not interrupt the screen reader.
- **`Assertive`**—Changes should be communicated immediately, even if the screen reader is in the midst of speaking.

Live region changes are not detected automatically, however. You must trigger them in C#. In our example, we just need to add an extra line of code to the existing `Button_Click` event handler:

```
void Button_Click(object sender, RoutedEventArgs e)
{
  this.result.Text = this.nameBox.Text;
  // Notify a screen reader to report this text
  TextBlockAutomationPeer.FromElement(this.result).RaiseAutomationEvent(
    AutomationEvents.LiveRegionChanged);
}
```

`TextBlock`, as with other controls, has a peer class in the `Windows.UI.Xaml.Automation.Peers` namespace. These classes are named with the pattern *`ElementName`*`AutomationPeer`, and have several members that are designed for accessibility as well as automated testing.

After the work we did to localize the `HelloRealWorld` app, it would be unfortunate to give screen readers a hardcoded English string, as shown earlier:

```
<TextBox AutomationProperties.Name="Please enter your name"
         Name="nameBox" Margin="12"/>
```

Fortunately, automation properties can be localized just like any other property. To do this, remove the explicit setting and give the element an `x:Uid`:

```
<TextBox x:Uid="NameBox" Name="nameBox" Margin="12"/>
```

In this example, you should then add the entry in the `Resources.resw` file named `NameBox.AutomationProperties.Name`, and its value for English should be `"Please enter your name"`.

Handling High Contrast Themes

The built-in controls automatically adjust their appearance when the user enables a high contrast theme. They adjust their colors to match the theme's eight user-customizable colors, and in some cases they change their rendering in other ways. Because of this, your app *can* automatically look correct under a high contrast theme without you doing extra work. However, when you use images or hardcoded colors, which are quite common, problems arise. Images can be a problem when they convey information but do not use enough contrast. Hardcoded colors are a problem for the same reason, but also because they can make things completely unreadable when intermixed with colors that drastically change under a high contrast theme. In general, mixing hardcoded colors with dynamic colors can be a recipe for disaster.

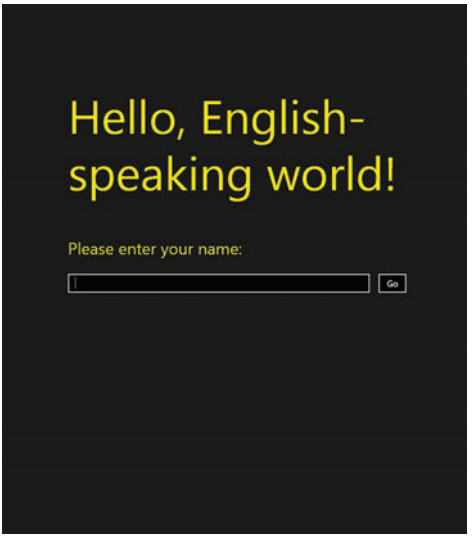
`HelloRealWorld` doesn't use any images, but Chapter 12 explains how you can provide separate versions of your images that can be used for high contrast themes only.

For `HelloRealWorld`, the hardcoded blue (or red or green) background color could be problematic as the colors of the other elements change. (Although none of the high contrast themes use blue, red, or green as a text color by default, the user could always choose it for the color of text.) We can fix this in code-behind by checking whether the app is running under high contrast and simply removing the `StackPanel`'s `Background` in that case:

```
public sealed partial class MainPage : Page
{
  Brush defaultBackground;

  public MainPage()
  {
    InitializeComponent();

    // Save the default background for later
    this.defaultBackground = this.stackPanel.Background;

    AccessibilitySettings settings = new AccessibilitySettings();

    // Update the background whenever the theme changes
    settings.HighContrastChanged += OnHighContrastChanged;

    // Set the background appropriately on initialization
    OnHighContrastChanged(settings, null);
  }

  void OnHighContrastChanged(AccessibilitySettings sender, object args)
  {
```

```
    this.stackPanel.Background =
      sender.HighContrast ? null : this.defaultBackground;
  }

  …
}
```

Because the user could change the theme while our app is running, we need to handle the `HighContrastChanged` event to adjust accordingly. The rest of the app's elements already adjust automatically. Figure 1.13 shows the result of adding this code then running the app under two different high contrast themes. Chapter 17 explains how you can define theme-specific colors without needing to write C# code such as this.



High contrast #1 theme

High contrast white theme

FIGURE 1.13 Removing the explicit `StackPanel` background makes the app look appropriate under any high contrast theme.

By defining and using the `defaultBackground` member, the code that handles the `HighContrastChanged` event preserves the language-specific background color that comes from one of the `Resources.resw` files. It does so without needing to programmatically retrieve the current resource value. However, if you need to do so, you can use code like the following for the `Panel.Background` value:

```
ResourceCandidate rc = ResourceManager.Current.MainResourceMap.GetValue(
  "Resources/Panel/Background", ResourceContext.GetForCurrentView());
string backgroundString = rc.ValueAsString;
```

If you do the following:

- ➔ check that the Windows SDK accessibility tools have no high-priority complaints about your app
- ➔ verify that your app acts appropriately when using Narrator
- ➔ verify that your app acts appropriately when running under high contrast
- ➔ verify that your app can be used when navigating using only the keyboard

then you should take credit for your work and check the "My app meets accessibility guidelines" checkbox within your app's listing in your Windows Dev Center dashboard. This fact gets advertised in the Windows Store, and it makes your app show up for users who search for accessible apps.

Submitting to the Windows Store

Once your app is finished, you can submit it to the Windows Store via items on the **Store** menu in Visual Studio Express, or via the **Project, Store** menu in other editions of Visual Studio. The Visual Studio integration works in concert with pages on the Windows Dev Center website to help you complete your submission. Before doing this, however, you have some tasks to complete:

- ➔ **Set up your developer account** at http://dev.windows.com, get it verified, and fill out your payout and tax information. This can take a couple of days for an individual account, or a couple of weeks for a business account.
- ➔ **Reserve your app name** with the Windows Store, as it requires each app's name to be unique. You can reserve names at any time, and you have up to a year to submit the app before losing each reservation. You can also reserve additional names for other languages.
- ➔ **Download, install, and run the Windows App Certification Kit** (WACK) from the Windows Dev Center. This tests your app for violations that cause it to fail the Windows Store certification process, so running it in advance can save you a lot of time.

The Windows Store certification process consists of three parts:

- ➔ **Technical checks.** This is simply running the Windows App Certification Kit on your app. If you pass its tests before submitting your app, you should have nothing to worry about here.
- ➔ **Security checks.** This ensures that your software isn't infected with a virus, which again should not be a concern for most developers.
- ➔ **Content checks.** This is the trickiest part of the process and, unlike the other two, is performed manually by human reviewers. Reviewers ensure that the app does what it claims to do and follows all the app certification requirements published in the Windows Dev Center.

The very first certification requirement is that the app "must offer customers unique, creative value or utility," so `HelloRealWorld` is bound to fail this requirement. This requirement may be obvious, but there are some requirements that often surprise people and cause many apps to fail certification:

- If your app requires a network capability, you must write a privacy statement that explains what data you collect, how you store or share it, how users can access the collected data, and so on. Requirement 4.1 in the Windows Dev Center helps you figure out how to write one. Furthermore, a link to the statement must be reachable from the Settings pane for your app, and the same link must be included in your listing in the Windows Store. See Chapter 20, "Supporting App Commands," for information about adding content to the Settings pane.
- You must select an appropriate age rating, using guidelines from the Windows Dev Center. For example, most apps that share personal information must be rated at least 12+. Regardless of your app's rating, its *listing* for the Windows Store cannot contain content that is considered too mature for a 12+ rating.
- You must provide descriptions and screenshots for every language you support. If your app is only partially localized for some languages, you must mention this in your listing.

If you fail certification, you must address the issue(s) and resubmit your app. When you do so, it goes through the entire process again, at the end of the line. Fortunately, at the time of this writing, the average length of certification is only about 2.5 days.

Don't forget to remove capabilities you don't need!

The certification process doesn't warn you about capabilities you don't actually use, so it's up to you to make sure the list is not larger than it needs to be.

Be sure to fill out the **Notes to testers** section in your Windows Dev Center dashboard to help the reviewers understand how to use any features of your app that might not be obvious. This is also the place to give them test credentials, if your app requires some sort of sign in.

To increase the chances of Microsoft promoting your app in the Windows Store, put a lot of effort into your listing. Every screenshot should be compelling, and you should feel free to enhance screenshots with explanations or other branding that increases the "wow factor" (as long as it's clear what is part of the app and what isn't). To get a feel for what makes a good description, you should look at the descriptions for apps that are already featured prominently in the Windows Store. In general, you should think of designing your listing like designing a box to sell your software in a retail store.

The optional **promotional images** are not optional at all if you want a chance for your app to be promoted. Again, they don't necessarily have to be screenshots, but they should be compelling and professional. You don't need to provide all possible sizes, but the 414x180 and 414x468 sizes are very important.

Summary

You've now seen the basic structure of a Visual Studio solution for a universal app and gotten a taste for making an app that is ready to sell across the world. Personally, I'm struck by how easy it is to localize your app and make it accessible. Software development has come a long way over the years, and you'll see evidence of this throughout the book, when it comes to handling heterogeneous screen DPI, making money through the Windows Store, communicating with slick peripherals, and much more.

Chapter 2

In This Chapter

- Elements and Attributes
- Namespaces
- Property Elements
- Type Converters
- Markup Extensions
- Children of Object Elements
- Mixing XAML with C#
- XAML Keywords

MASTERING XAML

You might be thinking, "Isn't Chapter 2 a bit early to become a *master* of XAML?" No, because this chapter focuses on the mechanics of the XAML *language*, which is a bit orthogonal to the multitude of XAML elements and APIs you'll be using when you build apps. Learning about the XAML language is kind of like learning the features of C# before delving into .NET or the Windows Runtime. Unlike the preceding chapter, this is a fairly deep dive! However, having this background knowledge before proceeding with the rest of the book will enable you to approach the examples with confidence.

XAML is a dialect of XML that Microsoft introduced in 2006 along with the first version of Windows Presentation Foundation (WPF). 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).

You can think of XAML as a clean, modern (albeit more verbose) reinvention of HTML and CSS. In universal apps, XAML serves essentially the same purpose as HTML: It provides a declarative way to represent user interfaces. That said, XAML is actually a general-purpose language that can be used in ways that have nothing to do with UI. The

preceding chapter contained a simple example of this. `App.xaml` does not define a user interface, but rather some characteristics of an app's entry point class. Note that almost everything that can be expressed in XAML can be naturally represented in a procedural language like C# as well.

The motivation for XAML is pretty much the same as any declarative markup language: Make it easy for programmers to work with others—perhaps graphic designers—and enable a powerful, robust tooling experience on top of it. XAML encourages a nice separation between visuals (and visual behavior such as animations) and the rest of the code, and enables powerful styling capabilities. XAML pages can be opened in Blend as well as Visual Studio (and Visual Studio has a convenient "Open in Blend…" item on its View menu), or entire XAML-based projects can be opened in Blend. This can be helpful for designing sophisticated artwork, animations, and other graphically rich touches. The idea is that a team's developers can work in Visual Studio while its designers work in Blend, and everyone can work on the same codebase. However, because XAML (and XML in general) is generally human readable, you can accomplish quite a bit with nothing more than a tool such as Notepad.

Elements and Attributes

The XAML specification defines rules that map object-oriented namespaces, types, properties, and events into XML namespaces, elements, and attributes. You can see this by examining the following simple XAML snippet that declares a `Button` control and comparing it to the equivalent C# code:

XAML:

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

C#:

```
Windows.UI.Xaml.Controls.Button b = new Windows.UI.Xaml.Controls.Button();
b.Content = "Stop";
```

Declaring an XML element in XAML (known as an *object element*) is equivalent to instantiating the corresponding 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` control 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="Stop" Click="Button_Click"/>
```

C#:

```
Windows.UI.Xaml.Controls.Button b = new Windows.UI.Xaml.Controls.Button();
b.Click += new Windows.UI.Xaml.RoutedEventHandler(Button_Click);
b.Content = "Stop";
```

This requires an appropriate method called `Button_Click` to be defined in a code-behind file, as seen in the preceding chapter. Note that XAML, like C#, is a case-sensitive language.

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 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 Windows Runtime namespace `Windows.UI.Xaml.Controls`. It turns out that the mapping to this and other namespaces is hard-coded. (In case you're wondering, no web page exists at the `schemas.microsoft.com` URL—it's just an arbitrary string like any namespace.) Because many Windows Runtime namespaces are mapped to the same XML namespace, the framework designers took care not to introduce two classes with the same name, despite the fact that the classes are in separate Windows Runtime namespaces.

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. `MainPage.xaml` in the preceding chapter contains the XML namespaces listed in Table 2.1.

TABLE 2.1 The XML Namespaces in Chapter 1's `MainPage.xaml`

Namespace	Typical Prefix	Description
`http://schemas.microsoft.com/winfx/2006/xaml/presentation`	(none)	The standard UI namespace. Contains elements such as `Grid`, `Button`, and `TextBlock`.
`http://schemas.microsoft.com/winfx/2006/xaml`	`x`	The XAML language namespace. Contains keywords such as `Class`, `Name`, and `Key`.
`using:HelloRealWorld`	`local`	This `using:`*`XXX`* syntax is the way to use any custom Windows Runtime or .NET namespace in a XAML file. In this case, `HelloRealWorld` is the .NET namespace generated for the project in Chapter 1 because the project itself was named "HelloRealWorld."
`http://schemas.microsoft.com/expression/blend/2008`	`d`	A namespace for design-time information that helps tools like Blend and Visual Studio show a proper preview.
`http://schemas.openxmlformats.org/markup-compatibility/2006`	`mc`	A markup compatibility namespace that can be used to mark other namespaces/elements as ignorable. Normally used with the design-time namespace, whose attributes should be ignored at runtime.

The first two namespaces are almost always used in any XAML file. The second one (with the `x` prefix) is the *XAML language namespace*, which defines some special directives for the XAML 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.

Using the UI 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 declare a `Button` in XAML as follows, and it would be equivalent to the `Button` defined previously:

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 UI 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`.

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

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.

The last two namespaces in Table 2.1, which are injected into pages generated by Visual Studio and Blend, are usually not needed.

Markup Compatibility

The markup compatibility XML namespace (`http://schemas.openxmlformats.org/markup-compatibility/2006`, typically used with an `mc` prefix) contains an `Ignorable` attribute that instructs XAML processors to ignore all elements/attributes in specified namespaces if they can't be resolved to their types/members. (The namespace also has a `ProcessContent` attribute that overrides `Ignorable` for specific types inside the ignored namespaces.)

Blend and Visual Studio take advantage of this feature to do things like add design-time properties to XAML content that can be ignored at runtime. `mc:Ignorable` can be given a space-delimited list of namespaces, and `mc:ProcessContent` can be given a space-delimited list of elements.

If you're frustrated by how long it takes to open XAML files in Visual Studio and you don't care about previewing the visuals, you might consider changing your default editor for XAML files by right-clicking on a XAML file in Solution Explorer then selecting **Open With…**, **XML (Text) Editor**, clicking **Set as Default**, then clicking **OK**. This has several major drawbacks, however, such as losing IntelliSense support and other editor shortcuts. And in Visual Studio 2013, XAML IntelliSense and editor shortcuts are better than ever!

Property Elements

Rich composition of controls is one of the highlights of XAML. This can be easily demonstrated with a `Button`, 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:

```
Windows.UI.Xaml.Controls.Button b = new Windows.UI.Xaml.Controls.Button();
b.Width = 96;
b.Height = 38;
```

```
Windows.UI.Xaml.Shapes.Rectangle r = new Windows.UI.Xaml.Shapes.Rectangle();
r.Width = 10;
r.Height = 10;
r.Fill = new Windows.UI.Xaml.Media.SolidColorBrush(Windows.UI.Colors.White);
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 10x10 `Rectangle` object. The result (when used with additional code that adds it to a page) is pictured in Figure 2.1.

FIGURE 2.1 Placing complex content inside a `Button`

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"
  Width="96" Height="38">
  <Button.Content>
    <Rectangle Width="10" Height="10" Fill="White"/>
  </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.

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="Stop" Background="Red"/>
```

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>
    Stop
  </Button.Content>
  <Button.Background>
    Red
  </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:

```
Windows.UI.Xaml.Controls.Button b = new Windows.UI.Xaml.Controls.Button();
b.Content = "Stop";
b.Background = new Windows.UI.Xaml.Media.SolidColorBrush(Windows.UI.Color.Red);
```

Wait a minute. How can `"Red"` in the previous XAML file be equivalent to the `SolidColorBrush` instance used 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 must look for a *type converter* that knows how to convert the string representation to the desired data type.

You cannot currently create your own type converters for universal apps, but type converters already exist for many common data types. Unlike the XAML language, these type converters support case-insensitive strings. Without a type converter for `Brush` (the base class of `SolidColorBrush`), 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="Stop">
  <Button.Background>
    <SolidColorBrush Color="Red"/>
  </Button.Background>
</Button>
```

And even that is only possible because of a type converter for `Color` that can make sense of the `"Red"` string. If there were no `Color` 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.

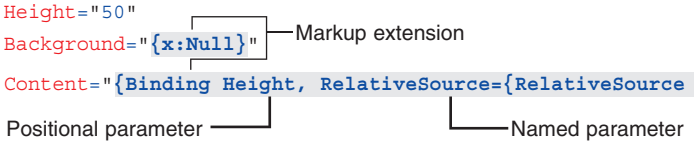
Unlike in the previous C# code, in this case, misspelling `Red` would not cause a compilation error but would cause an exception at runtime. However, Visual Studio does provide compile-time warnings for mistakes in XAML such as this.

Markup Extensions

Markup extensions, like type converters, extend the expressiveness of XAML. Both can evaluate a string attribute value at runtime and produce an appropriate object based on the string. As with type converters, you cannot currently create your own for universal apps, but several markup extensions are built in.

Unlike type converters, markup extensions are invoked from XAML with explicit and consistent syntax. Whenever an attribute value is enclosed in curly braces ({}), the XAML 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 two different markup extensions as the values for two different properties:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Height="50"
        Background="{x:Null}"
        Content="{Binding Height, RelativeSource={RelativeSource Self}}"/>
```



The first identifier in each set of curly braces is the name of the markup extension. The `Null` extension lives in the XAML language namespace, so the `x` prefix must be used. `Binding` (which happens to be a class in the `Windows.UI.Xaml.Data` namespace), can be found in the default XML namespace.

If a markup extension supports them, comma-delimited parameters can be specified. Positional parameters (such as `Height` in the example) are treated as string arguments for the extension class's appropriate constructor. Named parameters (`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, `x:Null` enables the `Background` brush to be set to `null`. This is just done for demonstration purposes, because a `null` `Background` is not very useful. `Binding`, covered in depth in Chapter 18, "Data Binding," enables `Content` to be set to the same value as the `Height` property.

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!}"/>
```

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>
```

Markup extensions can also 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.Height>
    50
  </Button.Height>
  <Button.Background>
    <x:Null/>
  </Button.Background>
  <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, `Binding` has a `Path` 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.

Markup Extensions and C#

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 `Null` and `Binding`:

```
Windows.UI.Xaml.Controls.Button b = new Windows.UI.Xaml.Controls.Button();
b.Height = 50;
// Set Background:
b.Background = null;
// Set Content:
Windows.UI.Xaml.Data.Binding binding = new Windows.UI.Xaml.Data.Binding();
binding.Path = new Windows.UI.Xaml.PropertyPath("Height");
binding.RelativeSource = Windows.UI.Xaml.Data.RelativeSource.Self;
b.SetBinding(Windows.UI.Xaml.Controls.Button.ContentProperty, binding);
```

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

Many classes designed to be used in XAML 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 just a convenient shortcut to make the XAML representation more compact.

`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="Stop"/>
```

could be rewritten as follows:

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

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

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
<Button.Content>
  <Rectangle Height="10" Width="10" Fill="White"/>
</Button.Content>
</Button>
```

could be rewritten as follows:

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <Rectangle Height="10" Width="10" Fill="White"/>
</Button>
```

There is no requirement that the content property must be called `Content`; classes such as `ComboBox` and `ListBox` (also in the `Windows.UI.Xaml.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 the `IList` interface or its generic counterpart. For example, the following XAML adds two items to a `ListBox` control whose `Items` property is an `ItemCollection` that implements `IList<object>`:

```
<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:

```
Windows.UI.Xaml.Controls.ListBox listbox =
  new Windows.UI.Xaml.Controls.ListBox();
Windows.UI.Xaml.Controls.ListBoxItem item1 =
  new Windows.UI.Xaml.Controls.ListBoxItem();
Windows.UI.Xaml.Controls.ListBoxItem item2 =
  new Windows.UI.Xaml.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 would need to wrap the items in an explicit element that instantiates the collection. The 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

A *dictionary* is any collection that implements the `IDictionary` interface or its generic counterpart. `Windows.UI.Xaml.ResourceDictionary` is a commonly used collection type that you'll see more of in later chapters. It implements `IDictionary<object, object>`, 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 dictionary. For example, the following XAML adds two `Color`s 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">White</Color>
  <Color x:Key="2">Black</Color>
</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:

```
Windows.UI.Xaml.ResourceDictionary d = new Windows.UI.Xaml.ResourceDictionary();
Windows.UI.Color color1 = Windows.UI.Colors.White;
Windows.UI.Color color2 = Windows.UI.Colors.Black;
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; 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>
```

As explained earlier, 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 objects. Using what you know so far, it would be reasonable to assume that you can't declare an instance of a class in XAML if it has no default constructor. However, even though the `Windows.UI.Xaml.Media.Brush` base class for

SolidColorBrush, LinearGradientBrush, and other brushes has no constructors at all, you can express the preceding XAML snippets as follows:

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

because the type converter for Brushes understands that this is still SolidColorBrush.

The Extensible Part of XAML

Because XAML was designed to work with the .NET type system, you can use it with just about any object, 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. A lot of care went into the design of the APIs in the Windows.UI.Xaml namespace—above and beyond the usual design guidelines—to fit XAML's declarative model.

To use an arbitrary .NET class (with a default constructor) in XAML, simply include the proper namespace with using syntax. The following XAML does this with an instance of System.Net.Http.HttpClient and System.Int64:

```
<ListBox xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <ListBox.Items>
    <sysnet:HttpClient xmlns:sysnet="using:System.Net.Http"/>
    <sys:Int64 xmlns:sys="using:System">100</sys:Int64>
  </ListBox.Items>
</ListBox>
```

The XAML language namespace defines keywords for a few common primitives so you don't need to separately include the System namespace: x:Boolean, x:Int32, x:Double, and x:String.

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 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.
3. Otherwise, if the parent supports a content property (indicated by Windows.UI.Xaml.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 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 C#

Universal apps are a mix of XAML and procedural code in a language like C#. This section covers the two ways that XAML and procedural code can be mixed together: dynamically loading and parsing XAML yourself, or leveraging the built-in support in Visual Studio projects.

Loading and Parsing XAML at Runtime

The `Windows.UI.Xaml.Markup` namespace contains a simple `XamlReader` class with a simple static `Load` method. `Load` can parse a string containing XAML, create the appropriate objects, and return an instance of the root element. So, with a string containing XAML content somewhat like `MainPage.xaml` from the preceding chapter, the following code could be used to load and retrieve the root `Page` object:

```
string xamlString = …;
// Get the root element, which we know is a Page
Page p = (Page)XamlReader.Load(xamlString);
```

After `Load` returns, the entire hierarchy of objects in the XAML file is instantiated in memory, so the XAML itself is no longer needed. 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 `Page` has a `StackPanel` object as its content, whose fifth child is a Stop button:

```
string xamlString = …;
// Get the root element, which we know is a Page
Page p = (Page)XamlReader.Load(xamlString);
// Grab the Stop button by walking the children (with hard-coded knowledge!)
StackPanel panel = (StackPanel)p.Content;
Button stopButton = (Button)panel.Children[4];
```

With a reference to the `Button` control, 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 satisfying, because 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 a `"Stop"` 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 `Button`s?

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

Naming XAML Elements

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

```
<Button x:Name="stopButton">Stop</Button>
```

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

```
string xamlString = …;
// Get the root element, which we know is a Page
Page p = (Page)XamlReader.Load(xamlString);
// Grab the Stop button, knowing only its name
Button stopButton = (Button)p.FindName("stopButton");
```

`FindName` is not unique to `Page`; it is defined on `FrameworkElement`, a base class for many important classes in the XAML UI Framework.

Naming Elements Without `x:Name`

The `x:Name` syntax can be used to name elements, but `FrameworkElement` also has a `Name` property that accomplishes the same thing. You can use either mechanism on such elements, 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. In addition, sometimes you want to name an element that doesn't derive from `FrameworkElement` (and doesn't have a `Name` property), so `x:Name` is necessary for such cases.

Visual Studio's Support for XAML and Code-Behind

Loading and parsing XAML at runtime can be interesting for some limited dynamic scenarios. Universal app projects, however, leverage work done by MSBuild and Visual Studio to make the combination of XAML and procedural code more seamless. When you compile a project with XAML files, the XAML is included as a resource in the app being built and the plumbing that connects XAML with procedural code is generated automatically.

The automatic connection between a XAML file and a code-behind file is enabled by the `Class` keyword from the XAML language namespace, as seen in the preceding chapter. For example, `MainPage.xaml` had the following:

```
<Page x:Class="HelloRealWorld.MainPage" …>
  …
</Page>
```

This causes the XAML content to be treated as a partial class definition for a class called `MainPage` (in the `HelloRealWorld` namespace) derived from `Page`. The other pieces of the partial class definition reside in auto-generated files as well as the `MainPage.xaml.cs` code-behind file. Visual Studio's Solution Explorer ties these two files together by making the code-behind file a subnode of the XAML file, but that is an optional cosmetic effect enabled by the following XML inside of the `.csproj` project file:

```
<Compile Include="MainPage.xaml.cs">
  <DependentUpon>MainPage.xaml</DependentUpon>
</Compile>
```

You can freely add members to the class in the code-behind file. And if you reference any event handlers in XAML (via event attributes such as `Click` on `Button`), this is where they should be defined.

Whenever you add a page to a Visual Studio project (via **Add New Item…**), 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.

The additional auto-generated files alluded to earlier contain some "glue code" that you normally never see and you should never directly edit. For a XAML file named `MainPage.xaml`, they are:

- `MainPage.g.cs`, which contains code that attaches event handlers to events for each event attribute assigned in the XAML file.
- `MainPage.g.i.cs`, which contains a field definition (private by default) for each named element in the XAML file, using the element name as the field name. It also contains an `InitializeComponent` method that the root class's constructor must call in the code-behind file. This file is meant to be helpful to IntelliSense, which is why it has an "i" in its name.

The "g" in both filenames stands for *generated*. Both generated source files contain a partial class definition for the same class partially defined by the XAML file and code-behind file.

If you peek at the implementation of `InitializeComponent` inside the auto-generated file, you'll see that the hookup between C# and XAML isn't so magical after all. It looks a lot like the code shown previously for manually loading XAML content and grabbing named elements from the tree of instantiated objects. Here's what the method looks like for the preceding chapter's `MainPage` if a `Button` named `stopButton` were added to it:

```
public void InitializeComponent()
{
  if (_contentLoaded)
    return;

  _contentLoaded = true;
  Application.LoadComponent(this, new System.Uri("ms-appx:///MainPage.xaml"),
    Windows.UI.Xaml.Controls.Primitives.ComponentResourceLocation.Application);

  stopButton = (Windows.UI.Xaml.Controls.Button)this.FindName("stopButton");
}
```

The `LoadComponent` method is much like `XamlReader`'s `Load` method, except it works with a reference to an app's resource file.

To reference a resource file included with your app, simply use a URI with the format `"ms-appx:///relative path to file"`. XAML files are already treated specially, but adding a new resource file to your app is as simple as adding a new file to your project with a **Build Action** of **Content**. Chapter 12, "Images," shows how to use resources such as image files with the `Image` element.

XAML Binary Format

By default, your app's package does not contain your `.xaml` source files but rather binary `.xbf` files known as XAML binary format. These files contain optimized node streams representing the original XAML content, which is great for startup performance because there is no need to load and parse XAML at runtime.

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 parser. They mostly control aspects of how elements get exposed to procedural code, but several are useful for other reasons. You've already seen some of them (such as `Key`, `Name`, and `Class`), but Table 2.2 lists all the ones relevant for universal apps. They are listed with the conventional `x` prefix because that is how they usually appear in XAML and in documentation.

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; see http://www.w3.org/XML/1998/ namespace.

TABLE 2.2 Keywords in the XAML Language Namespace, Assuming the Conventional `x` Namespace Prefix

Keyword	Valid As	Meaning
`x:Boolean`	An element	Represents a `System.Boolean`
`x:Class`	Attribute on root element	Defines a namespace-qualified class for the root element that derives from the element type
`x:Double`	An element	Represents a `System.Double`
`x:FieldModifier`	Attribute on any nonroot element but must be used with `x:Name` (or equivalent)	Defines the visibility of the field to be generated for the element (which is private by default). The value must be specified in terms of the procedural language (for example, `public`, `private`, and `internal` for C#).
`x:Int32`	An element	Represents a `System.Int32`
`x:Key`	Attribute on an element whose parent is a dictionary	Specifies the key for the item when added to the parent dictionary
`x:Name`	Attribute on any nonroot element but must be used with `x:Class` on root	Chooses a name for the field to be generated for the element, so it can be referenced from procedural code
`x:Null`	An element or an attribute value as a markup extension	Represents a `null` value
`x:StaticResource`	An element or an attribute value as a markup extension	References a XAML resource
`x:String`	An element	Represents a `System.String`
`x:Subclass`	Attribute on root element and must be used with `x:Class`	Specifies a subclass of the `x:Class` class that holds the content defined in XAML. This is only needed for languages without support for partial classes, so there's no reason to use this in a C# XAML project
`x:TemplateBinding`	An element or an attribute value as a markup extension	Binds to an element's properties from within a template, as described in Chapter 17
`x:ThemeResource`	An element or an attribute	References a theme-specific XAML resource value as a markup extension
`x:Uid`	Attribute on any element	Marks an element with an identifier used for localization

Summary

You have now seen how XAML fits in with the rest of an app's code 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.

As you proceed further, you might find that some APIs can be a little clunky to use in C# because their design is often optimized for XAML use. For example, the XAML UI Framework exposes many small building blocks to help enable rich composition, so some scenarios can involve manually creating a lot of objects. Besides the fact the XAML excels at expressing deep hierarchies of objects concisely, Microsoft 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).

In some areas, such as complicated paths and shapes, typing XAML by hand isn't practical. In fact, the trend from when XAML was first introduced in beta form has been to remove some of the handy human-typeable 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 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.

Classes in the XAML UI Framework have a 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 fundamental classes are referenced often and deserve a quick explanation before we get any further in the book. The `Page` class, for example, derives from a `UserControl` class which derives from all of the following classes, in order from most to least derived:

- **`Control`**—The base class for familiar controls such as `Button` and `ListBox`. `Control` adds many properties to its base class, such as `Foreground`, `Background`, and `FontSize`, as well as the ability to be given a completely new visual template. Part IV, "Understanding Controls," examines the built-in controls in depth.
- **`FrameworkElement`**—The base class that adds support for styles, data binding, XAML resources, and a few common mechanisms such as tooltips and context menus.
- **`UIElement`**—The base class for all visual objects with support for routed events, layout, and focus. These features are discussed in Chapter 4, "Layout," and Chapter 5, "Handling Input: Touch, Mouse, Pen, and Keyboard."
- **`DependencyObject`**—The base class for any object that can support dependency properties, discussed in Chapter 16, "Animation."
- **`Object`**—The base class for all .NET classes.

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

This page intentionally left blank

Chapter 3

In This Chapter

- Controlling Size
- Controlling Position
- Applying 2D Transforms
- Applying 3D Transforms

SIZING, POSITIONING, AND TRANSFORMING ELEMENTS

When building an app, one of the first things you must do is arrange a bunch of elements in its window. This sizing and positioning of elements is called *layout*. XAML apps are provided a feature-rich layout system that covers everything from placing elements at exact coordinates to building experiences that scale and rearrange across a wide range of screen resolutions and aspect ratios. This is essential when building a universal app. Even if you decide to build a separate user interface for phones, flexible layout is still necessary for handling the diversity of phone and PC screens, as well as intelligently handling when your app is resized on a PC.

Layout boils down to interactions between parent elements and their child elements. Parents and their children work together to determine their final sizes and positions. Although parents ultimately tell their children where to render and how much space they get, they are more like collaborators than dictators; parents also *ask* their children how much space they would like before making their final decision.

Parent elements that support the arrangement of multiple children are known as *panels*, and they derive from a class called `Panel`. All the elements involved in the layout process (both parents and children) derive from `UIElement`.

Because layout is such an important topic, this book dedicates two chapters to it. This chapter focuses on the children, examining the common ways that you can control layout on a child-by-child basis. Several properties control these aspects, most of which are summarized in Figure 3.1 for an arbitrary element inside an arbitrary panel. Size-related properties are shown in blue, and position-related properties are shown in red. In addition, elements can have transforms applied to them (shown in green) that can affect both size and position.

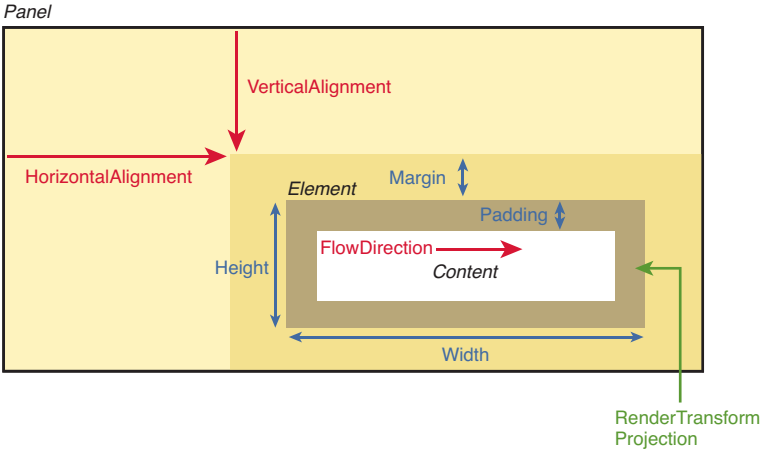


FIGURE 3.1 The main child layout properties examined in this chapter

The next chapter continues the layout story by examining the variety of built-in parent panels, each of which arranges its children in unique ways.

Controlling Size

Every time layout occurs, such as when an app's window is resized or the screen is rotated, child elements tell their parent panel their desired size. Elements tend to *size to their content*, meaning that they try to be large enough to fit their content and no larger. This size can be influenced on individual instances of children via several straightforward properties.

`Height` and `Width`

All `FrameworkElement`s have simple `Height` and `Width` properties (of type `double`), and they also have `MinHeight`, `MaxHeight`, `MinWidth`, and `MaxWidth` properties that can be used to specify a range of acceptable values. Any or all of these can be easily set on elements in C# or in XAML.

An element naturally stays as small as possible, so if you use `MinHeight` or `MinWidth`, it is rendered at that height/width unless its content forces it to grow. In addition, that growth can be limited by using `MaxHeight` and `MaxWidth`—as long as these values are larger than their `Min` counterparts. When using an explicit `Height` and `Width` at the same