Adam Nathan

Full Color

Learn how to build great Windows Store apps! Figures and code appear as they do in Visual Studio.

Windows® 8 Apps with XAML and C#

UNLEASHED

SAMS

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Adam Nathan

Windows® 8 Apps with XAML and C# Unleashed

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

Windows® 8 Apps with XAML and C# Unleashed

ISBN-13: 978-0-672-33601-0
ISBN-10: 0-672-33601-4

Library of Congress Cataloging-in-Publication Data is on file.

Printed in the United States on America

First Printing December 2012

Trademarks

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

Warning and Disclaimer

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

Bulk Sales

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

U.S. Corporate and Government Sales

1-800-382-3419

corpsales@pearsontechgroup.com

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

International Sales

international@pearsoned.com

EDITOR-IN-CHIEF
Greg Wiegand

EXECUTIVE EDITOR
Neil Rowe

DEVELOPMENT EDITOR
Mark Renfrow

MANAGING EDITOR
Kristy Hart

PROJECT EDITOR
Deadline Driven Publishing

COPY EDITOR
Kelly Maish

INDEXER
Angie Martin

PROOFREADER
Deadline Driven Publishing

TECHNICAL EDITOR
Ashish Shetty

PUBLISHING COORDINATOR
Cindy Teeters

COVER DESIGNER
Mark Shirar

COMPOSITOR
Bronkella Publishing

Contents at a Glance

Part V Exploiting Windows 8

Part VI Advanced Topics

Table of Contents

About the Author

Adam Nathan is a principal software architect for Microsoft, a best-selling technical author, and arguably the world's most prolific developer for Windows Phone. He introduced XAML to countless developers through his books on a variety of Microsoft technologies. Currently a part of Microsoft's Startup Business Group, 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 *101 Windows Phone 7 Apps* (Sams, 2011), *Silverlight 1.0 Unleashed* (Sams, 2008), *WPF Unleashed* (Sams, 2006), *WPF 4 Unleashed* (Sams, 2010), and *.NET and COM: The Complete Interoperability Guide* (Sams, 2002); a coauthor of *ASP.NET: Tips, Tutorials, and Code* (Sams, 2001); and a contributor to books including *.NET Framework Standard Library Annotated Reference, Volume 2* (Addison-Wesley, 2005) and *Windows Developer Power Tools* (O'Reilly, 2006). You can find Adam online at www.adamnathan.net, or @adamnathan on Twitter.

Dedication

To Tyler and Ryan.

Acknowledgments

First, I thank Lindsay Nathan for making this possible. Words fail to describe my gratitude.

I'd like to give special thanks to Ashish Shetty, Tim Heuer, Mark Rideout, Jonathan Russ, Joe Duffy, Chris Brumme, Eric Rudder, Neil Rowe, Betsy Harris, Ginny Munroe, Eileen Chan, and Valery Sarkisov. 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.

Finally, I thank *you* for picking up a copy of this book! I don't think you'll regret it!

We Want to Hear from You!

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

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

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

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

E-mail: feedback@samspublishing.com

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

Reader Services

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

This page intentionally left blank

Introduction

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!

I remember releasing a few shareware games in junior high school and asking for $5 donations. I earned $15. 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!

The Windows Store is an app store like no other. When you consider the number of people who use Windows 8 (and Windows RT) compared to the number of people who use any other operating system on the planet, you realize what a unique and enormous opportunity the Windows Store provides.

When you write a Windows Store app, you can work with whichever language and technology is most comfortable for you: JavaScript with an HTML user interface, or C#/Visual Basic/C++ with a XAML or raw DirectX user interface. (You can also componentize code to get different mixtures, such as using C# with HTML or some JavaScript in a XAML app.) Besides familiarity, your choice can have other benefits. Outside of the core Windows platform, each language and technology has different sets of reusable libraries and components. C++ has features for high-performance algorithms, for example. However, regardless of which choice you make, the Windows APIs are the same, and the graphics are hardware accelerated.

The key to the multiple-language support is the Windows Runtime, or WinRT for short. You can think of it like .NET's Common Language Runtime, except it spans both managed and unmanaged languages. To enable this, WinRT is COM-based. Most of the time, you can't tell when you interact with WinRT, however. This is a new, friendlier version of COM that is more amenable to *automatic* correct usage from environments such as .NET or JavaScript. (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. To match conventions, member names are even morphed to be Camel-cased for JavaScript and Pascal-cased for other languages, which makes the MSDN reference documentation occasionally look goofy.

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, and 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 Windows Store 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 8 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, or desktops. 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. And 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, it should be an invaluable reference for your bookshelf.

Software Requirements

This book targets Windows 8, Windows RT, and the corresponding developer tools. The tools can be downloaded for free at the Windows Dev Center: `s`. The download includes the Windows 8 SDK, a version of Visual Studio Express specifically for Windows Store apps, and Blend. It's worth noting that although this book almost exclusively refers to Windows 8, the content also applies to Windows RT.

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.samspublishing.com`.

How This Book Is Organized

This book is arranged into five 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 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, "Anatomy of a Windows Store App"
- Chapter 2, "Mastering XAML"

This part provides the foundation for the rest of the book. If you have previously created Windows Phone apps or worked with XAML in the context of other Microsoft technologies, a lot of this should be familiar to you. There are still several unique aspects for Windows 8 and the Windows Store, however.

Part II: Building an App

This part includes the following chapters:

- Chapter 3, "Sizing, Positioning, and Transforming Elements"
- Chapter 4, "Layout"
- Chapter 5, "Interactivity"
- Chapter 6, "Handling Input: Touch, Mouse, Pen, and Keyboard"
- Chapter 7, "App Model"

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. It also digs into the app model for Windows Store apps, which is significantly different from the app model for desktop applications in a number of ways.

Part III: Understanding Controls

This part includes the following chapters:

- Chapter 8, "Content Controls"
- Chapter 9, "Items Controls"
- Chapter 10, "Text"
- Chapter 11, "Images"
- Chapter 12, "Audio and Video"
- Chapter 13, "Other Controls"

Part III 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 IV: Leveraging the Richness of XAML

This part includes the following chapters:

- Chapter 14, "Vector Graphics"
- Chapter 15, "Animation"
- Chapter 16, "Styles, Templates, and Visual States"
- Chapter 17, "Data Binding"

The features covered in Part IV 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 V: Exploiting Windows 8

This part includes the following chapters:

- Chapter 18, "Data"
- Chapter 19, "Charms"
- Chapter 20, "Extensions"
- Chapter 21, "Sensors and Other Devices"
- Chapter 22, "Thinking Outside the App: Live Tiles, Toast Notifications, and the Lock Screen"

This part of the book can just as easily appear in a book about JavaScript or C++ Windows Store apps, with the exception of its code snippets. It covers unique and powerful Windows 8 features that are not specific to XAML or C#, but they are things that all Windows Store 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.

Throughout this book, and even in this introduction, you 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 in a particular spot in the book—and then provides a concise answer.

•••

Digging Deeper Sidebars

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.

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 2

In This Chapter

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 Windows Store 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 Windows Store 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. The "Mixing XAML with Procedural Code" section at the end of this chapter explains how to work with XAML that requires additional code. Note that XAML, like C#, is a case-sensitive language.

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 inside the framework. (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 `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:BlankApp`	`local`	This `using:`*XXX* syntax is the way to use any custom Windows Runtime or .NET namespace in a XAML file. In this case, `BlankApp` is the .NET namespace generated for the project in Chapter 1 because the project itself was named "BlankApp."
`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.

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`.

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:

```
<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 plopped in 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.

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 (with one exception—the x:Uid attribute used for localization).

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

```
<Button xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Content="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, 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 wasn't a `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. (Although 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, 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}}"/>
```

Markup extension

Positional parameter

Named parameter

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 also happens to be a class in the `Windows.UI.Xaml.Data` namespace), can be found in the default XML namespace. Note that the full name for `Null` is `NullExtension`, and this long form can be used as well in XAML. XAML permits dropping the `Extension` suffix on any markup extensions named with the suffix.

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 17, "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 Procedural Code

The actual work done by a markup extension is specific to each extension. For example, the following C# code is equivalent to the XAML-based `Button` that uses `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, bi nding);
```

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 `Colors` to a `ResourceDictionary`:

```
<ResourceDictionary
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Color x:Key="1">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>
```

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>
```

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

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 Procedural Code

XAML-based Windows Store apps are a mix of XAML and procedural code. This section covers the two ways that XAML and 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 Windows Runtime 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 `Buttons`?

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

Naming XAML Elements

The XAML language namespace has a `Name` keyword that enables you to give any element a name. For the simple 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. 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. Windows Store 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="BlankApp.MainPage" …>
  …
</Page>
```

This causes the XAML content to be treated as a partial class definition for a class called `MainPage` (in the `BlankApp` 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 11, "Images," shows how to use resources such as image files with the `Image` element.

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 independent of procedural code. You've already seen some of them (such as `Key`, `Name`, and `Class`), but Table 2.2 lists all the ones relevant for Windows Store 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. Can also appear as `x:NullExtension`.	Represents a `null` value.
`x:StaticResource`	An element or an attribute value as a markup extension. Can also appear as `x:StaticResourceExtension`.	References a XAML resource
`x:String`	An element.	Represents a `System.String`.

Keyword	Valid As	Meaning
`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 needed only 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 16. Can also appear as `x:TemplateBindingExtension`.
`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 from procedural code 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).

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

In some areas (such as complicated paths and shapes), typing XAML by hand isn't even practical. In fact, the trend from when XAML was first introduced in beta form has been to remove some of the handy human-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 capability to be given a completely new visual template. Part III, "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, "Interactivity."
- **`DependencyObject`**—The base class for any object that can support dependency properties, also discussed in Chapter 5.
- **`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`.

INDEX

Symbols and Numbers

A

B

C

D

E

F

G

H

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z