Microsoft

Programming Windows®

SIXTH EDITION

Consumer Preview eBook

Writing Windows 8 Apps
With C# and XAML

Charles Petzold

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

ISBN: 978-0-7356-7176-8

Acquisitions, Developmental, and Project Editor: Devon Musgrave
Technical Reviewer: Marc Young
Cover: Twist Creative • Seattle

Introduction

This book—the 6th edition of *Programming Windows*—is a guide to programming applications that run under Microsoft Windows 8. At the time of this writing (May 1, 2012), Windows 8 is not yet complete and neither is this book. What you are reading right now is a preview ebook version of the book. This preview version is based on the Consumer Preview of Windows 8, which was released on February 29, 2012. Microsoft has announced that the next preview of Windows 8—called the Release Preview—will be available in June. The second preview ebook version of this book, which will update the seven chapters included here and add more chapters, will probably be available in July. If you are reading this in August 2012 or later, you are very likely not reading the most recent version.

To use this book, you'll need to download and install the Windows 8 Consumer Preview, as well as Microsoft Visual Studio 11 Express Beta for Windows 8. Both downloads are accessible from the Windows 8 developer portal:

http://msdn.microsoft.com/windows/apps

To install Visual Studio, follow the "Download the tools and SDK" link on that page.

The Versions of Windows 8

For the most part, Windows 8 is intended to run on the same class of personal computers as Windows 7, which are machines built around the 32-bit or 64-bit Intel x86 microprocessor family. When Windows 8 is released later this year, it will be available in a regular edition called simply Windows 8 and also a Windows 8 Pro edition with additional features that appeal to tech enthusiasts and professionals.

Both Windows 8 and Windows 8 Pro will run two types of programs:

- Desktop applications
- What are currently referred to as "Metro style" applications

Desktop applications are traditional Windows programs that currently run under Windows 7 and that interact with the operating system through the Windows application programming interface, known familiarly as the Win32 API. Windows 8 includes a familiar Windows desktop screen for running these applications.

The applications known as "Metro style" are new with Windows 8. These applications incorporate the "Metro" design paradigm developed at Microsoft, so named because it's been inspired by public signage common in metropolitan areas. Metro design is characterized by the use of unadorned fonts, clean open styling, and a tile-based interface.

Internally and externally, Metro style applications represent a radical break with traditional Windows. The programs generally run in a full-screen mode—although two programs can share the screen in a "snap" mode—and many of these programs will probably be optimized for touch and tablet use. Metro style applications will be purchasable and installable only from an application store run by Microsoft.

In addition to the versions of Windows 8 that run on x86 processors, there will also be a version of Windows 8 that runs on ARM processors, most likely in low-cost smartphones and tablets. This version of Windows 8 will be called Windows RT, and it will come preinstalled on these machines. Aside from some preinstalled desktop applications, Windows RT will run Metro style applications only.

Many developers were first introduced to Metro design principles with Windows Phone 7, so it's interesting to see how Microsoft's thinking concerning large and small computers has evolved. In years gone by, Microsoft attempted to adapt the design of the traditional Windows desktop to smaller devices such as hand-held computers and phones. Now a user-interface design for the phone is being moved up to tablets and the desktop.

One important characteristic of this new environment is an emphasis on multitouch, which has dramatically changed the relationship between human and computer. In fact, the term "multitouch" is now outmoded because virtually all new touch devices respond to multiple fingers. The simple word "touch" is now sufficient. Part of the new programming interface for Metro style applications treats touch, the mouse, and a stylus in a unified manner so that applications are automatically usable with all three input devices.

The Focus of This Book

This book focuses exclusively on writing Metro style applications. Plenty of other books already exist for writing desktop applications, including the 5th edition of *Programming Windows*.

For writing Metro style applications, a new object-oriented API has been introduced called the Windows Runtime or WinRT (not to be confused with the version of Windows 8 that runs on ARM processors, called Windows RT). Internally, the Windows Runtime is based on COM (Component Object Model) with interfaces exposed through metadata files with the extension .winmd located in the */Windows/System32/WinMetadata* directory.

From the application programmer's perspective, the Windows Runtime resembles Silverlight, although internally it is not a managed API. For Silverlight programmers, perhaps the most immediate difference involves namespace names: the Silverlight namespaces beginning with *System.Windows* have been replaced with namespaces beginning with *Windows.UI.Xaml*.

Most Metro style applications will be built from both code and markup, either the industry-standard HyperText Markup Language (HTML) or Microsoft's eXtensible Application Markup Language (XAML). One advantage of splitting an application between code and markup is potentially splitting the

development of the application between programmers and designers.

Currently there are three main options for writing Metro style applications, each of which involves a programming language and a markup language:

- C++ with XAML
- C# or Visual Basic with XAML
- JavaScript with HTML5

In each case, the Windows Runtime is supplemented by another programming interface appropriate for that language. Although you can't mix languages within a single application, you can create language-independent libraries with their own .winmd files.

The C++ programmer uses a dialect of C++ called C++ with Component Extensions, or C++/CX, that allows the language to make better use of WinRT. The C++ programmer also has access to a subset of the Win32 and COM APIs, as well as DirectX.

Programmers who use the managed languages C# or Visual Basic .NET will find WinRT to be very familiar territory. Metro style applications written in these languages can't access Win32, COM, or DirectX APIs, but a stripped-down version of .NET is available for performing low-level tasks.

For JavaScript, the Windows Runtime is supplemented by a Windows Library for JavaScript, or WinJS, which provides a number of system-level features for Metro style apps written in JavaScript.

After much consideration (and some anguish), I decided that this book would use the C# and XAML option exclusively. For at least a decade I have been convinced of the advantages of managed languages for development and debugging, and for me C# is the language that has the closest fit to the Windows Runtime. I hope C++ programmers find C# code easy enough to read to derive some benefit from this book.

I also believe that a book focusing on one language option is more valuable than one that tries for equal coverage among several. There will undoubtedly be plenty of other Windows 8 books that show how to write Metro style applications using the other options.

The Approach

In writing this book, I've made a couple assumptions about *you*, the reader. I assume that you are comfortable with C#. If not, you might want to supplement this book with a C# tutorial. If you are coming to C# from a C or C++ background, my free online book *.NET Book Zero: What the C or C++ Programmer Needs to Know About C# and the .NET Framework* might be adequate. This book is available in PDF or XPS format at www.charlespetzold.com/dotnet. (I hope to update this book later this year to make it more specific to Windows 8.) I also assume that you know the rudimentary syntax of XML (eXtensible Markup Language) because XAML is based on XML.

This is an API book rather than a tools book. The only programming tools I use in this book are Microsoft Visual Studio 11 Express Beta for Windows 8 (which I'll generally simply refer to as Visual Studio), and XAML Cruncher, which is a program that I've written and which is featured in Chapter 7.

Markup languages are generally much more toolable than programming code. Indeed, some programmers even believe that markup such as XAML should be entirely machine-generated. Visual Studio has a built-in interactive XAML designer that involves dragging controls to a page, and many programmers have come to know and love Microsoft Expression Blend for generating complex XAML for their applications.

While such tools are great for experienced programmers, I think that the programmer new to the environment is better served by learning how to write XAML by hand. That's how I'll approach XAML in this book. The XAML Cruncher tool featured in Chapter 7 is very much in keeping with this philosophy: it lets you type in XAML and interactively see the objects that are generated, but it does not try to write XAML for you.

On the other hand, some programmers become so skilled at working with XAML that they forget how to create and initialize certain objects in code! I think both skills are important, and consequently I often show how to do similar tasks in both code and markup.

Source Code Learning a new API is similar to learning how to play basketball or the oboe: You don't get the full benefit by watching someone else do it. Your own fingers must get involved. The source code in these pages is downloadable from the same web page where you purchased the book via the "Companion Content" link on that page, but you'll learn better by actually typing in the code yourself.

As I began working on this book, I contemplated different approaches to how a tutorial about the Windows Runtime can be structured. One approach is to start with rather low-level graphics and user input, demonstrate how controls can be built, and then describe the controls that have already been built for you.

I have instead chosen to focus initially on those skills I think are most important for most mainstream programmers: assembling the predefined controls in an application and linking them with code and data. This is what I intend to be the focus of the book's Part I, "Fundamentals." The first 7 chapters out of the 10 (or so) that will eventually make up Part I are included in this first preview version. One of my goals in Part I is to make comprehensible all the code and markup that Visual Studio generates in the various project templates it supports, so the remaining chapters in Part I obviously need to cover templates, collection controls (and data), and navigation.

In the current plan for the book, the book will get more interesting as it gets longer: Part II, "Infrastructure," will cover more low-level tasks, such as touch, files, networking, security, globalization, and integrating with the Windows 8 charms. Part III, "Specialities," will tackle more esoteric topics, such as working with the sensors (GPS and orientation), vector graphics, bitmap graphics, media, text, printing, and obtaining input from the stylus and handwriting recognizer.

My Setup

For writing this book, I used the special version of the Samsung 700T tablet that was distributed to attendees of the Microsoft Build Conference in September 2011. This machine has an Intel Core i5 processor running at 1.6 GHz with 4 GB of RAM and a 64-GB hard drive. The screen (from which all the screenshots in the book were taken) has 8 touch points and a resolution of 1366 × 768 pixels, which is the lowest resolution for which snap views are supported.

Although the machines were distributed at Build with the Windows 8 Developer Preview installed, I replaced that with a complete install of the Consumer Preview (build 8250) in March 2012.

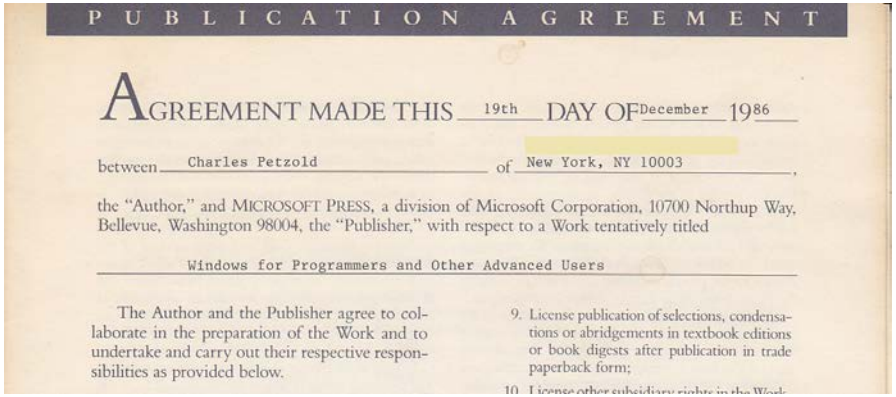
Except when testing orientation or sensors, I generally used the tablet in the docking port with an external 1920×1080 HDMI monitor, an external Microsoft Natural Ergonomic Keyboard 4000, and a Microsoft Comfort Mouse 300.

Running Visual Studio on the large screen and the resultant applications on the tablet turned out to be a fine development environment, particularly compared with the setup I used to write the first edition of *Programming Windows*.

But that was 25 years ago.

The *Programming Windows* Heritage

I still get a thrill when I look at my very first book contract:



PUBLICATION AGREEMENT

AGREEMENT MADE THIS 19th DAY OF December 1986

between Charles Petzold of New York, NY 10003,

the "Author," and MICROSOFT PRESS, a division of Microsoft Corporation, 10700 Northup Way, Bellevue, Washington 98004, the "Publisher," with respect to a Work tentatively titled

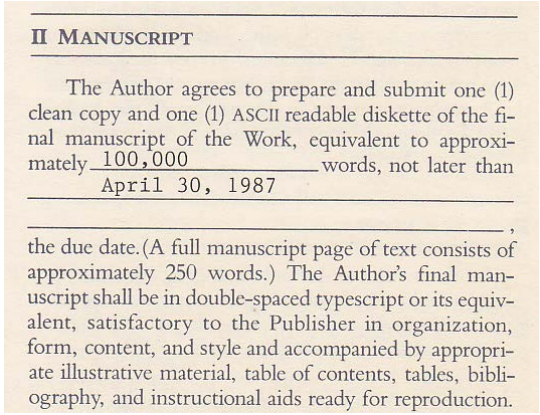
Windows for Programmers and Other Advanced Users

The Author and the Publisher agree to collaborate in the preparation of the Work and to undertake and carry out their respective responsibilities as provided below.

9. License publication of selections, condensations or abridgements in textbook editions or book digests after publication in trade paperback form;

10. License other subsidiary rights in the Work

Perhaps the most amusing part of this contract occurs further down the first page:



II MANUSCRIPT

The Author agrees to prepare and submit one (1) clean copy and one (1) ASCII readable diskette of the final manuscript of the Work, equivalent to approximately 100,000 words, not later than April 30, 1987, the due date. (A full manuscript page of text consists of approximately 250 words.) The Author's final manuscript shall be in double-spaced typescript or its equivalent, satisfactory to the Publisher in organization, form, content, and style and accompanied by appropriate illustrative material, table of contents, tables, bibliography, and instructional aids ready for reproduction.

The reference to "typescript" means that the pages must as least resemble something that came out of a typewriter. A double-spaced manuscript page with a fixed-pitch font has about 250 words, as the description indicates. A book page is more in the region of 400 words, so Microsoft Press obviously wasn't expecting a very long book.

For writing the book I used an IBM PC/AT with an 80286 microprocessor running at 8 MHz with 512 KB of memory and two 30 MB hard drives. The display was an IBM Enhanced Graphics Adapter, with a maximum resolution of 640 × 350 with 16 simultaneous colors. I wrote some of the early chapters using Windows 1 (introduced over a year earlier in November 1985), but beta versions of Windows 2 soon became available.

In those years, editing and compiling a Windows program occurred outside of Windows in MS-DOS. For editing source code, I used WordStar 3.3, the same word processor I used for writing the chapters. From the MS-DOS command line, you would run the Microsoft C compiler and then launch Windows with your program to test it out. It was necessary to exit Windows and return to MS-DOS for the next edit-compile-run cycle.

As I got deeper into writing the book, much of the rest of my life faded away. I stayed up later and later into the night. I didn't have a television at the time, but the local public radio station, WNYC-FM, was on almost constantly with classical music and other programming. For a while, I managed to shift my day to such a degree that I went to bed after *Morning Edition* but awoke in time for *All Things Considered*.

As the contract stipulated, I sent chapters to Microsoft Press on diskette and paper. (We all had email, of course, but email didn't support attachments at the time.) The edited chapters came back to me by mail decorated with proofreading marks and numerous sticky notes. I remember a page on which someone had drawn a thermometer indicating the increasing number of pages I was turning in with the caption "Temperature's Rising!"

Along the way, the focus of the book changed. Writing a book for "Programmers and Other Advanced Users" proved to be a flawed concept. I don't know who came up with the title *Programming Windows*.

The contract had a completion date of April, but I didn't finish until August and the book wasn't published until early 1988. The final page total was about 850. If these were normal book pages (that is, without program listings or diagrams) the word count would be about 400,000 rather than the 100,000 indicated in the contract.

The cover of the first edition of *Programming Windows* described it as "The Microsoft Guide to Programming for the MS-DOS Presentation Manager: Windows 2.0 and Windows/386." The reference to Presentation Manager reminds us of the days when Windows and the OS/2 Presentation Manager were supposed to peacefully coexist as similar environments for two different operating systems.

The first edition of *Programming Windows* went pretty much unnoticed by the programming community. When MS-DOS programmers gradually realized they needed to learn about the brave new environment of Windows, it was mostly the 2nd edition (published in 1990 and focusing on Windows 3) and the 3rd edition (1992, Windows 3.1) that helped out.

When the Windows API graduated from 16-bit to 32-bit, *Programming Windows* responded with the 4th edition (1996, Windows 95) and 5th edition (1998, Windows 98). Although the 5th edition is still in print, the email I receive from current readers indicates that the book is most popular in India and China.

From the 1st edition to the 5th, I used the C programming language. Sometime between the 3rd and 4th editions, my good friend Jeff Prosise said that he wanted to write *Programming Windows with MFC*, and that was fine by me. I didn't much care for the Microsoft Foundation Classes, which seemed to me a fairly light wrapper on the Windows API, and I wasn't that thrilled with C++ either.

As the years went by, *Programming Windows* acquired the reputation of being the book for programmers who needed to get close to the metal without any extraneous obstacles between their program code and the operating system.

But to me, the early editions of *Programming Windows* were nothing of the sort. In those days, getting close to the metal involved coding in assembly language, writing character output directly into video display memory, and resorting to MS-DOS only for file I/O. In contrast, programming for Windows involved a high-level language, completely unaccelerated graphics, and accessing hardware only through a heavy layer of APIs and device drivers.

This switch from MS-DOS to Windows represented a deliberate forfeit of speed and efficiency in return for other advantages. But what advantages? Many veteran programmers just couldn't see the point. Graphics? Pictures? Color? Fancy fonts? A mouse? That's not what computers are all about! The skeptics called it the WIMP (window-icon-menu-pointer) interface, which was not exactly a subtle implication about the people who chose to use such an environment or code for it.

Wait long enough, and a high-level language becomes a low-level language and multiple layers of interface seemingly shrink down (at least in lingo) to a native API. Some C and C++ programmers of today reject a managed language like C# on grounds of efficiency, and Windows has even sparked some energetic controversy once again. Windows 8 is easily the most revolutionary updating to

Windows since its very first release in 1985, but many old-time Windows users are wondering about the wisdom of bringing a touch-based interface tailored for smartphones and tablets to the mainstream desktop.

I suppose that *Programming Windows* could only be persuaded to emerge from semi-retirement with an exciting and controversial new user interface on Windows and an API and programming language suited to its modern aspirations.

Behind the Scenes

This book exists only because Ben Ryan and Devon Musgrave at Microsoft Press developed an interesting way to release early content to the developer community and get advances sales of the final book simultaneously. We are all quite eager to see the results of this experiment.

Part of the job duties of Devon and my technical reviewer Marc Young is to protect me from embarrassment by identifying blunders in my prose and code, and I thank them both for finding quite a few. Thanks also to Andrew Whitechapel for giving me feedback on the C++ sample code.

The errors that remain in these chapters are my own fault, of course. I'll try to identify the worst ones on my website at www.charlespetzold.com/pw6. And also give me feedback about pacing and the order that I cover material in these early chapters with an email to cp@charlespetzold.com.

Finally, I want to thank my wife Deirdre Sinnott for love and support and the necessary adjustments to our lives that writing a book inevitably entails.

Charles Petzold
New York City
May 1, 2012

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at oreilly.com. Search for the book at http://microsoftpress.oreilly.com, and then click the "View/Submit Errata" link. If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at mspinput@microsoft.com.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

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

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

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

Stay in Touch

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

Chapter 1

Markup and Code

Ever since the publication of Brian Kernighan and Dennis Ritchie's classic book *The C Programming Language* (Prentice Hall, 1978), it has been customary for programming tutorials to begin with a simple program that displays a short text string such as "hello, world." Let's create a few similar programs for Windows 8, and let's do it in what's referred to as "Metro style."

I'll assume you have the Windows 8 Consumer Preview installed with the development tools and software development kit, specifically Microsoft Visual Studio 11 Express Beta for Windows 8, which hereafter I'll simply refer to as Visual Studio.

Launch Visual Studio from the Windows 8 start screen, and let's get coding.

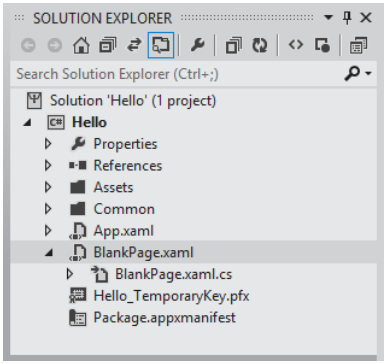
The First Project

On the opening screen in Visual Studio, the Get Started tab should already be selected. Over at the right you'll see a New Project option. Click that item, or select New Project from the File menu.

When the New Project dialog box comes up, select Templates in the left panel, then Visual C#, and Windows Metro Style. From the list of available templates in the central area, select Blank Application. Towards the bottom of the dialog box, type a project name in the Name field: **Hello**, for example. Let the Solution Name be the same. Use the Browse button to select a directory location for this program, and click OK. (I'll generally use mouse terminology such as "click" when referring to Visual Studio, but I'll switch to touch terminology such as "tap" for the applications you'll be creating. A version of Visual Studio that is optimized for touch is probably at least a few years away.)

Visual Studio creates a solution named Hello and a project within that solution named Hello, as well as a bunch of files in the Hello project. These files are listed in the Solution Explorer on the far right of the Visual Studio screen. Every Visual Studio solution has at least one project, but a solution might contain additional application projects and library projects.

The list of files for this project includes one called BlankPage.xaml, and if you click the little arrowhead next to that file, you'll see a file named BlankPage.xaml.cs indented underneath BlankPage.xaml:



You can view either of these two files by double-clicking the file name or by right-clicking the file name and choosing Open.

The BlankPage.xaml and BlankPage.xaml.cs files are linked in the Solution Explorer because they both contribute to the definition of a class named *BlankPage*. For a simple program like Hello, this *BlankPage* class defines all the visuals and user interface for the application. As the class name implies, the visuals are initially "blank," but they won't be for long.

Despite its funny file name, BlankPage.xaml.cs definitely has a .cs extension, which stands for "C Sharp." Stripped of all its comments, the skeleton BlankPage.xaml.cs file contains C# code that looks like this:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
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;

namespace Hello
{
    public sealed partial class BlankPage : Page
    {
        public BlankPage()
        {
            this.InitializeComponent();
        }
        protected override void OnNavigatedTo(NavigationEventArgs e)
        {
        }
    }
}
```

The file is dominated by *using* directives for all the namespaces that you are anticipated to need. You'll discover that most BlankPage.xaml.cs files don't require all these namespace names and many others require some additional namespaces.

These namespaces fall into two general categories based on the first word in the name:

- ***System.**** .NET for Metro style applications
- ***Windows.**** Windows Runtime (or WinRT)

As suggested by the list of *using* directives, namespaces that begin with *Windows.UI.Xaml* play a major role in the Windows Runtime.

Following the *using* directives, this BlankPage.xaml.cs file defines a namespace named *Hello* (the same as the project name) and a class named *BlankPage* that derives from *Page*, a class that is part of the Windows Runtime.

The documentation of the Windows 8 API is organized by namespace, so if you want to locate the documentation of the *Page* class, knowing the namespace where it's defined is useful. Let the mouse pointer hover over the name *Page* in the BlankPage.xaml.cs source code, and you'll discover that *Page* is in the *Windows.UI.Xaml.Controls* namespace.

The constructor of the *BlankPage* class calls an *InitializeComponent* method (which I'll discuss shortly), and the class also contains an override of a method named *OnNavigatedTo*. Metro style applications often have a page-navigation structure somewhat like a website, and hence they often consist of multiple classes that derive from *Page*. For navigational purposes, *Page* defines virtual methods named *OnNavigatingFrom*, *OnNavigatedFrom*, and *OnNavigatedTo*. The override of *OnNavigatedTo* is a convenient place to perform initialization when the page becomes active. But that's for later; most of the programs in the early chapters of this book will have only one page. I'll tend to refer to an application's "page" more than its "window." There is still a window underneath the application, but it doesn't play nearly as large a role as the page.

Notice the *partial* keyword on the *BlankPage* class definition. This keyword usually means that the class definition is continued in another C# source code file. In reality (as you'll see), that's exactly the case. Conceptually, however, the missing part of the *BlankPage* class is not another C# code file but the BlankPage.xaml file:

```
<Page
    x:Class="Hello.BlankPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Hello"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">

    </Grid>
```

```
</Page>
```

This file consists of markup conforming to the standard known as the eXtensible Application Markup Language, or XAML, pronounced “zammel.” As the name implies, XAML is based on eXtensible Markup Language, or XML.

Generally, you'll use the XAML file for defining all the visual elements of the page, while the C# file handles jobs that can't be performed in markup, such as number crunching and responding to user input. The C# file is often referred to as the "code-behind" file for the corresponding XAML file.

The root element of this XAML file is *Page*, which you already know is a class in the Windows Runtime. But notice the *x:Class* attribute:

```
<Page
    x:Class="Hello.BlankPage"
```

The *x:Class* attribute can appear only on the root element in a XAML file. This particular *x:Class* attribute translates as “a class *BlankPage* in the *Hello* namespace is defined as deriving from *Page*.” It means the same thing as the class definition in the C# file!

The *x:Class* attribute is followed by a bunch of XML namespace declarations. As usual, these URIs don’t actually reference interesting webpages but instead serve as unique identifiers maintained by particular companies or organizations. The first two are the most important:

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

The 2006 date harkens back to Microsoft's introduction of the Windows Presentation Foundation and the debut of XAML. WPF was part of the .NET Framework 3.0, which prior to its release was known as WinFX, hence the “winfx” in the URI. To a certain extent, XAML files are compatible between WPF, Silverlight, Windows Phone, and the Windows Runtime, but only if they use classes, properties, and features common to all the environments.

The first namespace declaration with no prefix refers to public classes, structures, and enumerations defined in the Windows Runtime, which includes all the controls and everything else that can appear in a XAML file, including the *Page* and *Grid* classes in this particular file. The word "presentation" in this URI refers to a visual user interface, and that distinguishes it from other types of applications that can use XAML. For example, if you were using XAML for the Windows Workflow Foundation (WF), you'd use a default namespace URI ending with the word "workflow".

The second namespace declaration associates an “x” prefix with elements and attributes that are intrinsic to XAML itself. Only nine of these are applicable in Windows Runtime applications, and obviously one of the most important is the *x:Class* attribute.

The third namespace declaration is interesting:

```
xmlns:local="using:Hello"
```

This associates an XML prefix of *local* with the *Hello* namespace of this particular application. You

might create custom classes in your application, and you'd use the *local* prefix to reference them in XAML. If you need to reference classes in code libraries, you'll define additional XML namespace declarations that refer to the assembly name and namespace name of these libraries. You'll see how to do this in chapters ahead.

The remaining namespace declarations are for Microsoft Expression Blend. Expression Blend might insert special markup of its own that should be ignored by the Visual Studio compiler, so that's the reason for the *Ignorable* attribute, which requires yet another namespace declaration. For any program in this book, these last three lines of the *Page* root element can be deleted.

The *Page* element has a child element named *Grid*, which is another class defined in the W*indows.UI.Xaml.Controls* namespace. The *Grid* will become extremely familiar. It is sometimes referred to as a "container" because it can contain other visual objects, but it's more formally classified as a "panel" because it derives from the *Panel* class. Classes that derive from *Panel* play a very important role in layout in Metro style applications. In the BlankPage.xaml file that Visual Studio creates for you, the *Grid* is assigned a background color (actually a *Brush* object) based on a predefined identifier using a syntax I'll discuss in Chapter 2, "XAML Syntax."

Generally, you'll divide a *Grid* into rows and columns to define individual cells (as I'll demonstrate in Chapter 5, "Control Interaction"), somewhat like a much improved version of an HTML table. A *Grid* without rows and columns is sometimes called a "single-cell *Grid*" and is still quite useful.

To display up to a paragraph of text in the Windows Runtime, you'll generally use a *TextBlock* (another class defined in the *Windows.UI.Xaml.Controls* namespace), so let's put a *TextBlock* in the single-cell *Grid* and assign a bunch of attributes. These attributes are actually properties defined by the *TextBlock* class:

Project: Hello | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <TextBlock Text="Hello, Windows 8!"
               FontFamily="Times New Roman"
               FontSize="96"
               FontStyle="Italic"
               Foreground="Yellow"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Note In this book, whenever a block of code or markup is preceded by a heading like this one, you'll find the code among this book's downloadable companion content. Generally I'll just show an excerpt of the total file, but with enough context so you know exactly where it is.

The order of these attributes doesn't matter, and of course the indentation doesn't matter, and all of them except the *Text* attribute can be skipped if you're in a hurry. As you type you'll notice that Visual Studio's Intellisense feature suggests attribute names and possible values for you. Often you can just select the one you want. As you finish typing the *TextBlock*, Visual Studio's design view gives you a

preview of the page's appearance.

You can also skip all the typing and simply drag a *TextBlock* from the Visual Studio Toolbox and then set the properties in a table, but I won't be doing that in this book. I'll instead describe the creation of these programs as if you and I actually type in the code and markup just like real programmers.

Press F5 to compile and run this program, or select Start Debugging from the Debug menu. Even for simple programs like this, it's best to run the program under the Visual Studio debugger. If all goes well, this is what you'll see:

The *HorizontalAlignment* and *VerticalAlignment* attributes on the *TextBlock* have caused the text to be centered, obviously without the need for you the programmer to explicitly determine the size of the video display and the size of the rendered text. You can alternatively set *HorizontalAlignment* to *Left* or *Right*, and *VerticalAlignment* to *Top* or *Bottom* to position the *TextBlock* in one of nine places in the *Grid*. As you'll see in Chapter 4, "Presentation with Panels," the Windows Runtime supports precise pixel placement of visual objects, but usually you'll want to rely on the built-in layout features.

The *TextBlock* has *Width* and *Height* properties, but generally you don't need to bother setting those. In fact, if you set the *Width* and *Height* properties on this particular *TextBlock*, you might end up cropping part of the text or interfering with the centering of the text on the page. The *TextBlock* knows better than you how large it should be.

You might be running this program on a device that responds to orientation changes, such as a tablet. If so, you'll notice that the page content dynamically conforms to the change in orientation and aspect ratio, apparently without any interaction from the program. The *Grid*, the *TextBlock*, and the Windows 8 layout system are doing most of the work.

To terminate the Hello program, press Shift+F5 in Visual Studio, or select Stop Debugging from the

Debug menu. You'll notice that the program hasn't merely been executed, but has actually been deployed to Windows 8 and is now executable from the start screen. The icon is not very pretty, but the program's icons are all stored in the Assets directory of the project so you can spruce them up if you want. You can run the program again outside of the Visual Studio debugger right from the Windows 8 start screen.

Graphical Greetings

Traditional "hello" programs display a greeting in text, but that's not the only way to do it. The HelloImage project accesses a bitmap from my website using a tiny piece of XAML:

Project: HelloImage | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <Image Source="http://www.charlespetzold.com/pw6/PetzoldJersey.jpg" />
</Grid>
```

The *Image* element is defined in *Windows.UI.Xaml.Controls* namespace, and it's the standard way to display bitmaps in a Windows Runtime program. By default, the bitmap is stretched to fit the space available for it while respecting the original aspect ratio:

If you make the page smaller—perhaps by changing the orientation or invoking a snap view—the image will change size to accommodate the new size of the page.

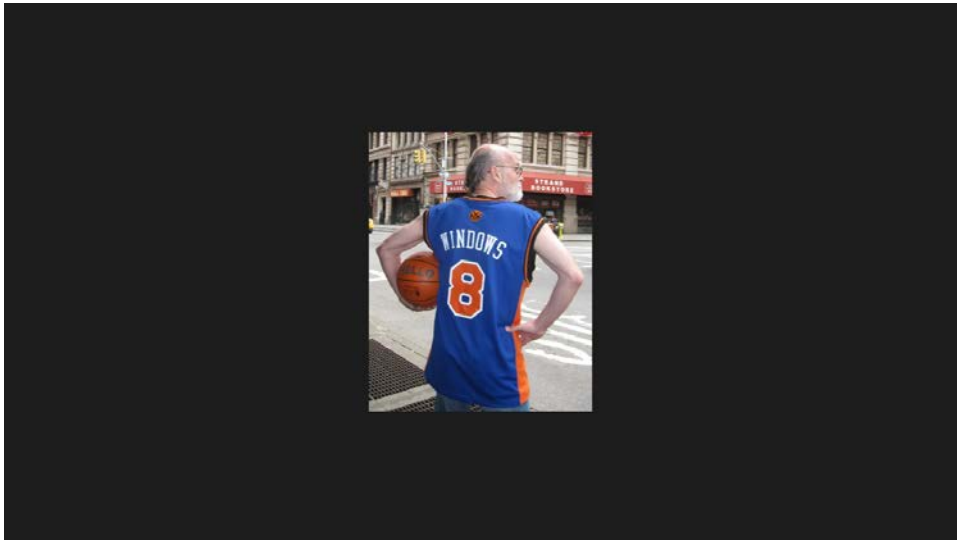
You can override the default display of this bitmap by using the *Stretch* property defined by *Image*. The default value is the enumeration member *Stretch.Uniform*. Try setting it to *Fill*:

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <Image Source="http://www.charlespetzold.com/pw6/PetzoldJersey.jpg"
```

```
            Stretch="Fill" />
</Grid>
```

Now the aspect ratio is ignored and the bitmap fills the container:

Set the *Stretch* property to *None* to display the image in its pixel dimensions (320 by 400):



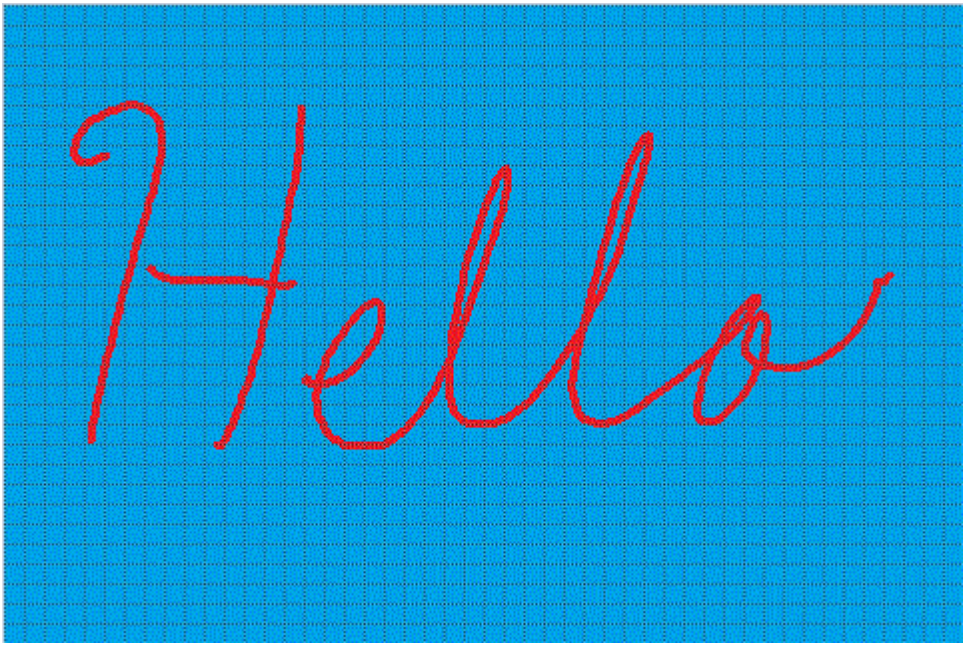
You can control where it appears on the page by using the same *HorizontalAlignment* and *VerticalAlignment* properties you use with *TextBlock*.

The fourth option for the *Stretch* property is *UniformToFill*, which respects the aspect ratio but fills the container regardless. It achieves this feat by the only way possible: clipping the image. Which part

of the image that gets clipped depends on the *HorizontalAlignment* and *VerticalAlignment* properties.

Accessing bitmaps over the Internet is dependent on a network connection and even then might require some time. A better guarantee of having an image immediately available is to bind the bitmap into the application itself.

You can create simple bitmaps right in Windows Paint. Let's run Paint and use the File Properties option to set a size of 480 by 320 (for example). Using a mouse, finger, or stylus, you can create your own personalized greeting:



The Windows Runtime supports the popular BMP, JPEG, PNG, and GIF formats, as well as a couple less common formats. For images such as the one above, PNG is common, so save it with a name like Greeting.png.

Now create a new project: HelloLocalImage, for example. It's common to store bitmaps used by a project in a directory named Images. In the Solution Explorer, right-click the project name and choose Add and New Folder. (Or, if the project is selected in the Solution Explorer, pick New Folder from the Project menu.) Give the folder a name such as Images.

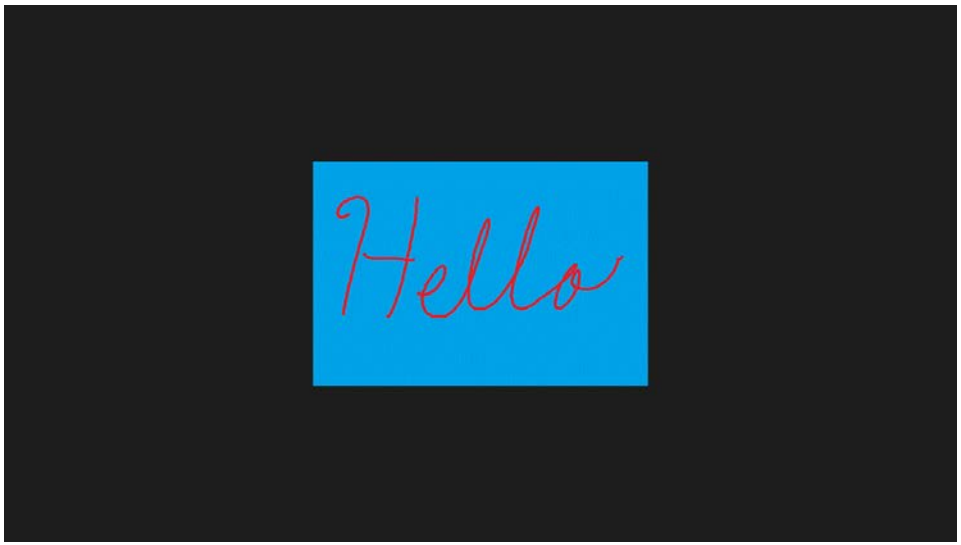
Now right-click the Images folder and choose Add and Existing Item. Navigate to the Greeting.png file you saved and click the Add button. Once the file is added to the project, you'll want to right-click the Greeting.png file name and select Properties. In the Properties panel, make sure the Build Action is set to Content. You want this image to become part of the content of the application.

The XAML file that references this image looks very much like one for accessing an image over the web:

Project: HelloLocalImage | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <Image Source="Images/Greeting.png"
           Stretch="None" />
</Grid>
```

Notice that the *Source* property is set to the folder and file name. Here's how it looks:



Sometimes programmers prefer giving a name of Assets to the folder that stores application bitmaps. You'll notice that the standard project already contains an Assets folder containing program icons. You can use that same folder for your other images instead of creating a separate folder.

Variations in Text

You might be tempted to refer to the *Grid*, *TextBlock*, and *Image* as "controls," perhaps based on the knowledge that these classes are in the *Windows.UI.Xaml.Controls* namespace. Strictly speaking, however, they are *not* controls. The Windows Runtime does define a class named *Control* but these three classes do not descend from *Control*. Here's a tiny piece of the Windows Runtime class hierarchy showing the classes encountered so far:

Object
    DependencyObject
        UIElement
            FrameworkElement
                TextBlock
                Image
                Panel

Grid

Control

UserControl

Page

Page derives from *Control* but *TextBlock* and *Image* do not. *TextBlock* and *Image* instead derive from *UIElement* and *FrameworkElement*. For that reason, *TextBlock* and *Image* are more correctly referred to as "elements," the same word often used to describe items that appear in XML files.

The distinction between an element and a control is not always obvious. Visually, controls are built from elements, and the visual appearance of the control can be customizable through a template. But the distinction is useful nonetheless. A *Grid* is also an element, but it's more often referred to as a "panel," and that (as you'll see) is a very useful distinction.

Try this: In the original Hello program move the *Foreground* attribute and all the font-related attributes from the *TextBlock* element to the *Page*. The entire BlankPage.xaml file now looks like this:

```
<Page
    x:Class="Hello.BlankPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:Hello"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    FontFamily="Times New Roman"
    FontSize="96"
    FontStyle="Italic"
    Foreground="Yellow">

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <TextBlock Text="Hello, Windows 8!"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Center" />
    </Grid>
</Page>
```

You'll discover that the result is exactly the same. When these attributes are set on the *Page* element, they apply to everything on that page.

Now try setting the *Foreground* property of the *TextBlock* to red:

```
<TextBlock Text="Hello, Windows 8!"
           Foreground="Red"
           HorizontalAlignment="Center"
           VerticalAlignment="Center" />
```

The local red setting overrides the yellow setting on the *Page*.

The *Page*, *Grid*, and *TextBlock* form what is called a "visual tree" of elements, except that in the XAML file the tree is upside-down. The *Page* is the trunk of the tree, and its descendants (*Grid* and

TextBlock) form branches. You might imagine that the values of the font properties and *Foreground* property defined on the *Page* are propagated down through the visual tree from parent to child. This is true except for a little peculiarity: These properties don't exist in *Grid*. These properties are defined by *TextBlock* and separately defined by *Control*, which means that the properties manage to propagate from the *Page* to the *TextBlock* despite an intervening element that has very different DNA.

If you begin examining the documentation of these properties in the *TextBlock* or *Page* class, you'll discover that they seem to appear twice under somewhat different names. In the documentation of *TextBlock* you'll see a *FontSize* property of type *double*:

```
public double FontSize { set; get; }
```

You'll also see a property named *FontSizeProperty* of type *DependencyProperty*:

```
public static DependencyProperty FontSizeProperty { get; }
```

Notice that this *FontSizeProperty* property is get-only and static as well.

FontSizeProperty is of type *DependencyProperty*, and a class with a similar name—*DependencyObject*—has a very prominent place in the class hierarchy I just showed you. These two types are related: A class that derives from *DependencyObject* often declares static get-only properties of type *DependencyProperty*. Both *DependencyObject* and *DependencyProperty* are defined in the *Windows.UI.Xaml* namespace, suggesting how fundamental they are to the whole system.

In a Metro style application, properties can be set in a variety of ways. For example, you've already seen that properties can be set directly on an object or inherited through the visual tree. As you'll see in Chapter 2, properties might also be set from a Style definition. In a future chapter you'll see properties set from animations. The *DependencyObject* and *DependencyProperty* classes are part of a system that help maintain order in such an environment by establishing priorities for the different ways in which the property might be set. I don't want to go too deeply into the mechanism just yet; it's something you'll experience more intimately when you begin defining your own controls.

The *FontSize* property is sometimes said to be "backed by" the dependency property named *FontSizeProperty*. But sometimes a semantic shortcut is used and *FontSize* itself is referred to as a dependency property. Usually this is not confusing.

Many of the properties defined by *UIElement* and its descendent classes are dependency properties, but only a few of these properties are propagated through the visual tree. *Foreground* and all the font-related properties are, as well as a few others that I'll be sure to call your attention to as we encounter them. Dependency properties also have an intrinsic default value. If you remove all the *TextBlock* and *Page* attributes except *Text*, you'll get white text displayed with an 11-pixel system font in the upper-left corner of the page.

The *FontSize* property is in units of pixels and refers to the design height of a font. This design height includes space for descenders and diacritical marks. As you might know, font sizes are often specified in *points*, which in electronic typography are units of 1/72 inch. The equivalence between pixels and points requires knowing the resolution of the video display in dots-per-inch (DPI). Without

that information, it's generally assumed that video displays have a resolution of 96 DPI, so a 96-pixel font is thus a 72-point font (one-inch high) and the default 11-pixel font is an 8¼-point font.

The user of Windows has the option of setting a desired screen resolution. A Metro style application can obtain the user setting from the *DisplayProperties* class, which pretty much dominates the *Windows.Graphics.Display* namespace. For most purposes, however, assuming a resolution of 96 DPI is fine, and you'll use this same assumption for the printer. In accordance with this assumption, I tend to use pixel dimensions that represent simple fractions of inches: 48 (1/2"), 24 (1/4"), 12 (1/8"), and 6 (1/16").

You've seen that if you remove the *Foreground* attribute, you get white text on a dark background. The background is not exactly black, but the predefined *ApplicationPageBackgroundBrush* identifier that the *Grid* references is close to it.

The Hello project also includes two other files that come in a pair: App.xaml and App.xaml.cs together define a class named *App* that derives from *Application*. Although an application can have multiple *Page* derivatives, it has only one *Application* derivative. This *App* class is responsible for settings or activities that affect the application as a whole.

Try this: In the root element of the App.xaml file, set the attribute *RequestedTheme* to *Light*.

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

The only options are *Light* and *Dark*. Now you get a light background, which means the color referenced by the *ApplicationPageBackgroundBrush* identifier is different. If the *Foreground* property on the *Page* or *TextBlock* is not explicitly set, you'll also get black text, which means that the *Foreground* property has a different default value with this theme.

In many of the sample programs in the remainder of this book, I'll be using the light theme without mentioning it. I think the screen shots look better on the page, and they won't consume as much ink if you decide to print pages from the book. However, keep in mind that many small devices and an increasing number of larger devices have displays built around organic light-emitting diode (OLED) technology and these displays consume less power if the screen isn't lit up like a billboard. Reduced power consumption is one reason why dark color schemes are becoming more popular.

Of course, you can completely specify your own colors by explicitly setting both the *Background* of the *Grid* and the *Foreground* of the *TextBlock*:

```
<Grid Background="Blue">
    <TextBlock Text="Hello, Windows 8!"
               Foreground="Yellow"
               … />
```

```
</Grid>
```

For these properties, Visual Studio's IntelliSense provides 140 standard color names, plus *Transparent*. These are actually static properties of the *Colors* class. Alternatively, you can specify red-green-blue (RGB) values directly in hexadecimal with values ranging from 00 to FF prefaced by a pound sign:

```
Foreground="#FF8000"
```

That's maximum red, half green, and no blue. An optional fourth byte at the beginning is the alpha channel, with values ranging from 00 for transparent and FF for opaque. Here's a half-transparent red:

```
Foreground="#80FF0000"
```

The *UIElement* class also defines an *Opacity* property that can be set to values between 0 (transparent) and 1 (opaque). In HelloImage, try setting the *Background* property of the *Grid* to a nonblack color (perhaps Blue) and set the *Opacity* property of the *Image* element to 0.5.

When you specify colors by using bytes, the values are in accordance with the familiar sRGB ("standard RGB") color space. This color space dates back to the era of cathode-ray tube displays where these bytes directly controlled the voltages illuminating the pixels. Very fortuitously, nonlinearities in pixel brightness and nonlinearities in the perception of brightness by the human eye roughly cancel each other out, so these byte values often seem perceptually linear, or nearly so.

An alternative is the scRGB color space, which uses values between 0 and 1 that are proportional to light intensity. Here's a value for medium gray:

```
Foreground="sc# 0.5 0.5 0.5"
```

Due to the logarithmic response of the human eye to light intensity, this gray will appear to be rather too light to be classified as medium.

If you need to display text characters that are not on your keyboard, you can specify them in Unicode by using standard XML character escaping. For example, if you want to display the text "This costs €55" and you're confined to an American keyboard, you can specify the Unicode Euro in decimal like this:

```
<TextBlock Text="This costs &#8364;55" …
```

Or perhaps you prefer hexadecimal:

```
<TextBlock Text="This costs &#x20AC;55" …
```

Or you can simply paste text into Visual Studio as I obviously did with a program later in this chapter.

As with standard XML, strings can contain special characters beginning with the ampersand:

- & is an ampersand
- ' is a single-quotation mark ("apostrophe")
- " is a double-quotation mark

- < is a left angle bracket (“less than”)
- > is a right angle bracket (“greater than”)

An alternative to setting the *Text* property of *TextBlock* requires separating the element into a start tag and end tag and specifying the text as content:

```
<TextBlock … >
    Hello, Windows 8!
</TextBlock>
```

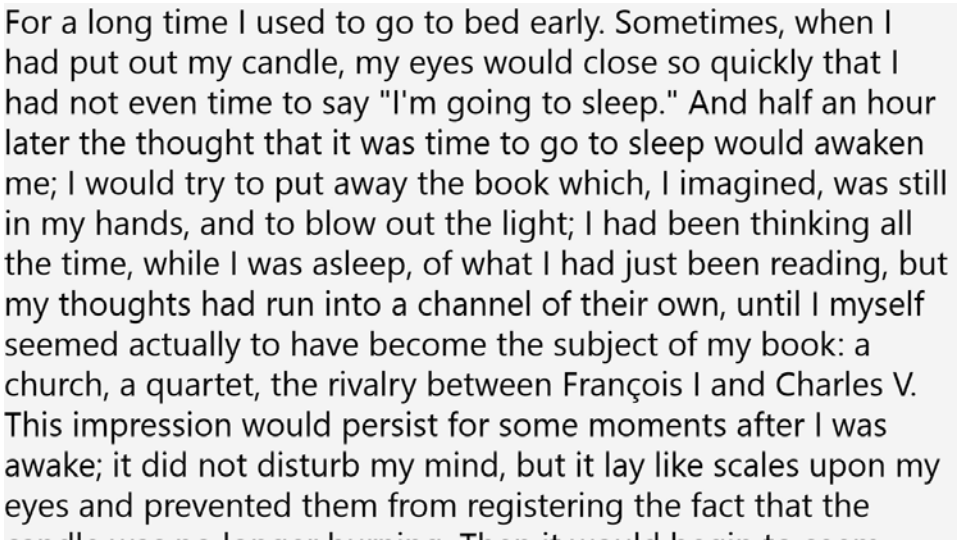
As I’ll discuss in Chapter 2, setting text as content of the *TextBlock* is not exactly equivalent to setting the *Text* property. It’s actually much more powerful. But even without taking advantage of additional features, specifying text as content is useful for displaying a larger quantity of text because you don’t have to worry about extraneous white space as much as when you’re dealing with quoted text. The *WrappedText* project displays a whole paragraph of text by specifying this text as content of the *TextBlock*:

Project: WrappedText | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <TextBlock FontSize="48"
               TextWrapping="Wrap">
        For a long time I used to go to bed early. Sometimes, when I had put out
        my candle, my eyes would close so quickly that I had not even time to
        say "I'm going to sleep." And half an hour later the thought that it was
        time to go to sleep would awaken me; I would try to put away the book
        which, I imagined, was still in my hands, and to blow out the light; I
        had been thinking all the time, while I was asleep, of what I had just
        been reading, but my thoughts had run into a channel of their own,
        until I myself seemed actually to have become the subject of my book:
        a church, a quartet, the rivalry between François I and Charles V. This
        impression would persist for some moments after I was awake; it did not
        disturb my mind, but it lay like scales upon my eyes and prevented them
        from registering the fact that the candle was no longer burning. Then
        it would begin to seem unintelligible, as the thoughts of a former
        existence must be to a reincarnate spirit; the subject of my book would
        separate itself from me, leaving me free to choose whether I would form
        part of it or no; and at the same time my sight would return and I
        would be astonished to find myself in a state of darkness, pleasant and
        restful enough for the eyes, and even more, perhaps, for my mind, to
        which it appeared incomprehensible, without a cause, a matter dark
        indeed.
    </TextBlock>
</Grid>
```

Notice the *TextWrapping* property. The default is the *TextWrapping.NoWrap* enumeration member; *Wrap* is the only alternative. You can also set the *TextAlignment* property to members of the *TextAlignment* enumeration: *Left*, *Right*, or *Center*. Although the *TextAlignment* enumeration also includes a *Justify* member, it is not supported under the current version of the Windows Runtime.

You can run this program in either portrait mode or landscape:



If your display responds to orientation changes, the text is automatically reformatted. The Windows Runtime breaks lines at spaces or hyphens, but it does not break lines at nonbreaking spaces (' ') or nonbreaking hyphens ('&#x2011'). Any soft hyphens ('&#x00AD') are ignored.

Not every element in XAML supports text content like *TextBlock*. You can't have text content in the *Page* or *Grid*, for example.

But the *Grid* can support multiple *TextBlock* children. The OverlappedStackedText project has two *TextBlock* elements in the *Grid* with different colors and font sizes:

Project: OverlappedStackText | File: BlankPage.xaml

```
<Grid Background="Yellow">
    <TextBlock Text="8"
               FontSize="864"
               FontWeight="Bold"
               Foreground="Red"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />

    <TextBlock Text="Windows"
               FontSize="192"
               FontStyle="Italic"
               Foreground="Blue"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Here's the result:

Notice that the second element is visually above the first. This is often referred to as "Z order" because in a three-dimensional coordinate space, an imaginary Z axis comes out of the screen. In Chapter 4 you'll see a way to override this behavior.

Of course, overlapping is not a generalized solution to displaying multiple items of text! In Chapter 5 you'll see how to define rows and columns in the *Grid* for layout purposes, but another approach to organizing multiple elements in a single-cell *Grid* is to use various values of *HorizontalAlignment* and *VerticalAlignment* to prevent them from overlapping. The InternationalHelloWorld program displays "hello, world" in nine different languages. (Thank you, Google Translate!)

Project: InternationalHelloWorld | File: BlankPage.xaml (excerpt)

```
<Page
    x:Class="InternationalHelloWorld.BlankPage"
    …
    FontSize="40">

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <!-- Chinese (simplified) -->
        <TextBlock Text="你好，世界"
                   HorizontalAlignment="Left"
                   VerticalAlignment="Top" />

        <!-- Urdu -->
        <TextBlock Text="ہیلو دنیا،"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Top" />


        <!-- Japanese -->
        <TextBlock Text="こんにちは、世界中のみなさん"
                   HorizontalAlignment="Right"
                   VerticalAlignment="Top" />
```

```
        <!-- Hebrew -->
        <TextBlock Text="שלום, עולם"
                   HorizontalAlignment="Left"
                   VerticalAlignment="Center" />

        <!-- Esperanto -->
        <TextBlock Text="Saluton, mondo"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Center" />

        <!-- Arabic -->
        <TextBlock Text="مرحبا، العالم"
                   HorizontalAlignment="Right"
                   VerticalAlignment="Center" />

        <!-- Korean -->
        <TextBlock Text="안녕하세요, 전 세계"
                   HorizontalAlignment="Left"
                   VerticalAlignment="Bottom" />

        <!-- Russian -->
        <TextBlock Text="Здравствуй, мир"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Bottom" />

        <!-- Hindi -->
        <TextBlock Text="नमस्ते दुनिया है,"
                   HorizontalAlignment="Right"
                   VerticalAlignment="Bottom" />
    </Grid>
</Page>
```

Notice the *FontSize* attribute set in the root element to apply to all nine *TextBlock* elements. Property inheritance is obviously one way to reduce repetition in XAML, and you'll see other approaches as well in the next chapter.

Media As Well

So far you've seen greetings in text and bitmaps. The HelloAudio project plays an audio greeting from a file on my website. I made the recording using the Windows 8 Sound Recorder application, which automatically saves in WMA format. The XAML file looks like this:

Project: HelloAudio | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <MediaPlayer Source="http://www.charlespetzold.com/pw6/AudioGreeting.wma"
                 VerticalAlignment="Center" />
</Grid>
```

The *MediaPlayer* class derives from *Control* and has its own built-in user interface that automatically fades out until you brush your mouse or finger across it. Alternatively you can use *MediaElement* for playing sounds. *MediaElement* is a *FrameworkElement* derivative that has no user interface of its own, although it provides enough information for you to build your own.

You can use MediaPlayer or MediaElement for playing movies. The HelloVideo program plays a video from my website:

Project: HelloVideo | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <MediaPlayer Source="http://www.charlespetzold.com/pw6/VideoGreeting.wmv" />
</Grid>
```

The Code Alternatives

It's not necessary to instantiate elements or controls in XAML. You can alternatively create them entirely in code. Indeed, very much of what can be done in XAML can be done in code instead. Code is particularly useful for creating many objects of the same type because there's no such thing as a *for* loop in XAML.

Let's create a new project named HelloCode, but let's visit the BlankPage.xaml file only long enough to give the *Grid* a name:

Project: HelloCode | File: BlankPage.xaml (excerpt)

```
<Grid Name="contentGrid"
      Background="{StaticResource ApplicationPageBackgroundBrush}">

</Grid>
```

Setting the *Name* attribute allows the *Grid* to be accessed from the code-behind file. Alternatively you can use *x:Name*:

```
<Grid x:Name="contentGrid"
      Background="{StaticResource ApplicationPageBackgroundBrush}">

</Grid>
```

There's really no practical difference between *Name* and *x:Name*. As the "x" prefix indicates, the *x:Name* attribute is intrinsic to XAML itself, and you can use it to identify any object in the XAML file. The *Name* attribute is more restrictive: *Name* is defined by *FrameworkElement*, so you can use it only with classes that derive from *FrameworkElement*. For a class not derived from *FrameworkElement*, you'll need to use *x:Name* instead. Some programmers prefer to be consistent by using *x:Name* throughout. I tend to use *Name* whenever I can and *x:Name* otherwise.

Whether you use *Name* or *x:Name*, the rules for the name you choose are the same as the rules for variable names. The name can't contain spaces or begin with a number, for example. All names within a particular XAML file must be unique.

In the BlankPage.xaml.cs file you'll want two additional *using* directives:

Project: HelloCode | File: BlankPage.xaml.cs (excerpt)

```
using Windows.UI;
using Windows.UI.Text;
```

The first is for the *Colors* class; the second is for a *FontStyle* enumeration. It's not strictly necessary that you insert these *using* directives manually. If you use the *Colors* class or *FontStyle* enumeration, Visual Studio will indicate with a red squiggly underline that it can't resolve the identifier, at which point you can right-click it and select Resolve from the context menu. The new *using* directive will be added to the others in correct alphabetical order (as long as the existing using directives are alphabetized). When you're all finished with the code file, you can right-click anywhere in the file and select Organize

Usings and Remove Unused Usings to clean up the list. (I've done that with this BlankPage.xaml.cs file.)

The constructor of the *Page* class is a handy place to create a *TextBlock*, assign properties, and then add it to the *Grid*:

Project: HelloCode | File: BlankPage.xaml.cs (excerpt)

```
public BlankPage()
{
    this.InitializeComponent();

    TextBlock txtblk = new TextBlock();
    txtblk.Text = "Hello, Windows 8!";
    txtblk.FontFamily = new FontFamily("Times New Roman");
    txtblk.FontSize = 96;
    txtblk.FontStyle = FontStyle.Italic;
    txtblk.Foreground = new SolidColorBrush(Colors.Yellow);
    txtblk.HorizontalAlignment = HorizontalAlignment.Center;
    txtblk.VerticalAlignment = VerticalAlignment.Center;

    contentGrid.Children.Add(txtblk);
}
```

Notice that the last line of code here references the *Grid* named *contentGrid* in the XAML file just as if it were a normal object, perhaps stored as a field. (As you'll see, it actually is a normal object and it is a field!) Although not evident in XAML, the *Grid* has a property named *Children* that it inherits from *Panel*. This *Children* property is of type *UIElementCollection*, which is a collection that implements the *IList<UIElement>* and *IEnumerable<UIElement>* interfaces. This is why the *Grid* can support multiple child elements.

Code often tends to be a little wordier than XAML partially because the XAML parser works behind the scenes to create additional objects and perform conversions. The code reveals that the *FontFamily* property requires that a *FontFamily* object be created and that *Foreground* is of type Brush and requires an instance of a Brush derivative, such as *SolidColorBrush*. *Colors* is a class that contains 141 static properties of type *Color*. You can create a *Color* object from ARGB bytes by using the static *Color.FromArgb* method.

The *FontStyle*, *HorizontalAlignment*, and *VerticalAlignment* properties are all enumeration types, where the enumeration is the same name as the property. Indeed, the *Text* and *FontSize* properties seem odd in that they are primitive types: a string and a double-precision floating-point number.

You can reduce the code bulk a little by using a style of property initialization introduced in C# 3.0:

```
TextBlock txtblk = new TextBlock
{
    Text = "Hello, Windows 8!",
    FontFamily = new FontFamily("Times New Roman"),
    FontSize = 96,
    FontStyle = FontStyle.Italic,
    Foreground = new SolidColorBrush(Colors.Yellow),
    HorizontalAlignment = HorizontalAlignment.Center,
```

```
    VerticalAlignment = VerticalAlignment.Center
};
```

Either way, you can now compile and run the HelloCode project and the result should look the same as the XAML version. It looks the same because it basically *is* the same.

You can alternatively create the *TextBlock* and add it to the *Children* collection of the *Grid* in the *OnNavigatedTo* override. Or you can create the *TextBlock* in the constructor, save it as a field, and add it to the *Grid* in *OnNavigatedTo*.

Notice that I put the code after the *InitializeComponent* call in the *Page* constructor. You can create the *TextBlock* prior to *InitializeComponent*, but you must add it to the *Grid* after *InitializeComponent* because the *Grid* does not exist prior to that call. The *InitializeComponent* method basically parses the XAML at run time and instantiates all the XAML objects and puts them all together in a tree. *InitializeComponent* is obviously an important method, which is why you might be puzzled when you can't find it in the documentation.

Here's the story: When Visual Studio compiles the application, it generates some intermediate files. You can find these files with Windows Explorer by navigating to the HelloCode solution, the HelloCode project, and then the obj and Debug directories. Among the list of files are BlankPage.g.cs and BlankPage.g.i.cs. The "g" stands for "generated." Both these files define *BlankPage* classes derived from *Page* with the *partial* keyword. The composite *BlankPage* class thus consists of the BlankPage.xaml.cs file under your control plus these two generated files, which you don't mess with. Although you don't edit these files, they are important to know about because they might pop up in Visual Studio if a run-time error occurs involving the XAML file.

The BlankPage.g.i.cs file is the more interesting of the two. Here you'll find the definition of the *InitializeComponent* method, which calls a static method named *Application.LoadComponent* to load the BlankPage.xaml file. Notice also that this partial class definition contains a private field named *contentGrid*, which is the name you've assigned to the *Grid* in the XAML file. The *InitializeComponent* method concludes by setting that field to the actual *Grid* object created by *Application.LoadComponent*.

The *contentGrid* field is thus accessible throughout the *BlankPage* class, but the value will be *null* until *InitializeComponent* is called.

In summary, parsing the XAML is a two-stage process. At compile time the XAML is parsed to extract all the element names (among other tasks) and generate the intermediate C# files in the obj directory. These generated C# files are compiled along with the C# files under your control. At run time the XAML file is parsed again to instantiate all the elements, assemble them in a visual tree, and obtain references to them.

Where is the standard *Main* method that serves as an entry point to any C# program? That's in App.g.i.cs, one of two files generated by Visual Studio based on App.xaml.

Let me show you something else that will serve as just a little preview of dependency properties:

As I mentioned earlier, many properties that we've been dealing with—*FontFamily*, *FontSize*, *FontStyle*, *Foreground*, *Text*, *HorizontalAlignment*, and *VerticalAlignment*—have corresponding static dependency properties, named *FontFamilyProperty*, *FontSizeProperty*, and so forth. You might amuse yourself by changing a normal statement like this:

```
txtblk.FontStyle = FontStyle.Italic;
```

to an alternative that might look quite peculiar:

```
txtblk.SetValue(TextBlock.FontStyleProperty, FontStyle.Italic);
```

What you're doing here is calling a method named *SetValue* defined by *DependencyObject* and inherited by *TextBlock*. You're calling this method on the *TextBlock* object but passing to it the static *FontStyleProperty* object of type *DependencyProperty* defined by *TextBlock* and the value you want for that property. There is no real difference between these two ways of setting the *FontStyle* property. Within *TextBlock*, the *FontStyle* property is very likely defined like this:

```
public FontStyle FontStyle
{
    set
    {
        SetValue(TextBlock.FontStyleProperty, value);
    }
    get
    {
        return (FontStyle)GetValue(TextBlock.FontStyleProperty);
    }
}
```

I say "very likely" because I'm not privy to the Windows Runtime source code, but if the *FontStyle* property is defined like all other properties backed by dependency properties, the *set* and *get* accessors simply call *SetValue* and *GetValue* with the *TextBlock.FontStyleProperty* dependency property. This is extremely standard code, and it's a pattern you'll come to be so familiar with that you'll generally define your own dependency properties without so much white space like this:

```
public FontStyle FontStyle
{
    set { SetValue(TextBlock.FontStyleProperty, value); }
    get { return (FontStyle)GetValue(TextBlock.FontStyleProperty); }
}
```

Earlier you saw how you can set the *Foreground* and font-related properties on the *Page* rather than the *TextBlock* and how these properties are inherited by the *TextBlock*. Of course you can do the same thing in code:

```
public BlankPage()
{
    this.InitializeComponent();

    this.FontFamily = new FontFamily("Times New Roman");
    this.FontSize = 96;
```

```
    this.FontStyle = FontStyle.Italic;
    this.Foreground = new SolidColorBrush(Colors.Yellow);

    TextBlock txtblk = new TextBlock();
    txtblk.Text = "Hello, Windows 8!";
    txtblk.HorizontalAlignment = HorizontalAlignment.Center;
    txtblk.VerticalAlignment = VerticalAlignment.Center;

    contentGrid.Children.Add(txtblk);
}
```

C# doesn't require the *this* prefix to access properties and methods of the class, but when you're editing the files in Visual Studio, typing the *this* prefix invokes Intellisense to give you a list of available methods, properties, and events.

Images in Code

Judging solely from the XAML files in the HelloImage and HelloLocalImage projects, you might have assumed that the *Source* property of *Image* is defined as a string or perhaps the *Uri* type. In XAML, that *Source* string is a shortcut for an object of type *ImageSource*, which encapsulates the actual image that the *Image* element is responsible for displaying. *ImageSource* doesn't define anything on its own and cannot be instantiated, but several important classes descend from *ImageSource*, as shown in this partial class hierarchy:

Object
    DependencyObject
        ImageSource
            BitmapSource
                BitmapImage
                WriteableBitmap

ImageSource is defined in the *Windows.UI.Xaml.Media* namespace, but the descendent classes are in *Windows.UI.Xaml.Media.Imaging*. A *BitmapSource* can't be instantiated either, but it defines public *PixelWidth* and *PixelHeight* properties as well as a *SetSource* method that lets you read in bitmap data from a file or network stream. *BitmapImage* inherits these members and also defines a *UriSource* property.

You can use *BitmapImage* for displaying a bitmap from code. Besides defining this *UriSource* property, *BitmapImage* also defines a constructor that accepts a *Uri* object. In the HelloImageCode project, the *Grid* has been given a name of "contentGrid" and a using directive for *Windows.UI.Xaml.Media.Imaging* has been added to the code-behind file. Here's the *BlankPage* constructor:

Project: HelloImageCode | File: BlankPage.xaml.cs (excerpt)

```
public BlankPage()
{
```

```
    this.InitializeComponent();

    Uri uri = new Uri("http://www.charlespetzold.com/pw6/PetzoldJersey.jpg");
    BitmapImage bitmap = new BitmapImage(uri);
    Image image = new Image();
    image.Source = bitmap;
    contentGrid.Children.Add(image);
}
```

Setting a *Name* of "contentGrid" on the *Grid* is not strictly necessary for accessing the *Grid* from code. The *Grid* is actually set to the *Content* property of the *Page*, so rather than accessing the *Grid* like so:

```
contentGrid.Children.Add(image);
```

you can do it like this:

```
Grid grid = this.Content as Grid;
grid.Children.Add(image);
```

In fact, the *Grid* isn't even necessary in such a simple program. You can effectively remove the *Grid* from the visual tree by setting the *Image* directly to the *Content* property of the *Page*:

```
this.Content = image;
```

The *Content* property that *Page* inherits from *UserControl* is of type *UIElement*, so it can support only one child. Generally the child of the *Page* is a *Panel* derivative that supports multiple children, but if you need only one child, you can use the *Content* property of the *Page* directly.

It's also possible to make a hybrid of the XAML and code approaches: to instantiate the *Image* element in XAML and create the *BitmapImage* in code, or to instantiate both the *Image* element and *BitmapImage* in XAML and then set the *UriSource* property of *BitmapImage* from code. I've used the first approach in the HelloLocalImageCode project, which has an Images directory with the Greeting.png file. The XAML file already contains the *Image* element, but it doesn't reference an actual bitmap:

Project: HelloLocalImageCode | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <Image Name="image"
           Stretch="None" />
</Grid>
```

The code-behind file sets the *Source* property of the *Image* element in a single line:

Project: HelloLocalImageCode | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    public BlankPage()
    {
        this.InitializeComponent();
        image.Source = new BitmapImage(new Uri("ms-appx:///Images/Greeting.png"));
    }
```

```
}
```

Look at that special URL for referencing the content bitmap file from code. In XAML, that special prefix is optional.

Are there general rules to determine when to use XAML and when to use code? Not really. I tend to use XAML whenever possible except when the repetition becomes ridiculous. My normal rule for code is "three or more: use a *for*," but I'll often allow somewhat more repetition in XAML before moving it into code. A lot depends on how concise and elegant you've managed to make the XAML and how much effort it would be to change something.

Not Even a Page

Insights into how a Windows Runtime program starts up can be obtained by examining the *OnLaunched* override in the standard App.xaml.cs file. You'll discover that it creates a *Frame* object, uses this *Frame* object to navigate to an instance of *BlankPage* (which is how *BlankPage* gets instantiated), and then sets this *Frame* object to a precreated *Window* object accessible through the *Window.Current* static property:

```
var rootFrame = new Frame();
rootFrame.Navigate(typeof(BlankPage));
Window.Current.Content = rootFrame;
Window.Current.Activate();
```

A Metro style application doesn't require a *Page*, a *Frame*, or even any XAML files at all. Let's conclude this chapter by creating a new project named StrippedDownHello and begin by deleting the App.xaml, App.xaml.cs, BlankPage.xaml, and BlankPage.xaml.cs files, as well as the entire Common folder. Yes, delete them all! Now the project has no code files and no XAML files. It's left with just an app manifest, assembly information, and some PNG files.

Right-click the project name and select Add and New Item. Select either a new class or code file and name it App.cs. Here's what you'll want it to look like:

Project: StrippedDownHello | File: App.cs

```
using Windows.ApplicationModel.Activation;
using Windows.UI;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media;

namespace TryStrippedDown
{
    public class App : Application
    {
        static void Main(string[] args)
        {
            Application.Start((p) => new App());
        }
```

```
        protected override void OnLaunched(LaunchActivatedEventArgs args)
        {
            TextBlock txtblk = new TextBlock
            {
                Text = "Stripped-Down Windows 8",
                FontFamily = new FontFamily("Lucida sans Typewriter"),
                FontSize = 96,
                Foreground = new SolidColorBrush(Colors.Red),
                HorizontalAlignment = Windows.UI.Xaml.HorizontalAlignment.Center,
                VerticalAlignment = Windows.UI.Xaml.VerticalAlignment.Center
            };

            Window.Current.Content = txtblk;
            Window.Current.Activate();
        }
    }
}
```

That's all you need (and obviously much less if you want default properties on the *TextBlock*). The static *Main* method is the entry point and that creates a new *App* object and starts it going, and the *OnLaunched* override creates a *TextBlock* and makes it the content of the application's default window.

I won't be pursuing this approach to creating Metro style applications in this book, but obviously it works.

Chapter 2

XAML Syntax

A Metro style application is divided into code and markup because each has its own strength. Despite the limitations of markup in performing complex logic or computational tasks, it's good to get as much of a program into markup as possible. Markup is easier to edit with tools and shows a clearer sense of the visual layout of a page. Of course, everything in markup is a string, so markup sometimes becomes cumbersome in representing complex objects. Because markup doesn't have the loop processing common in programming languages, it can also be prone to repetition.

These issues have been addressed in the syntax of XAML in several ways, the most important of which are explored in this chapter. But let me begin this vital subject with a topic that will at first appear to be completely unrelated: defining a gradient brush.

The Gradient Brush in Code

The *Background* property in *Grid* and the *Foreground* property of the *TextBlock* are both of type *Brush*. The programs shown so far have set these properties to a derivative of *Brush* called *SolidColorBrush*. As demonstrated in Chapter 1, "Markup and Code," you can create a *SolidColorBrush* in code and give it a *Color* value; in XAML this is done for you behind the scenes.

SolidColorBrush is only one of four available brushes, as shown in this class hierarchy:

Object
    DependencyObject
        Brush
            SolidColorBrush
            GradientBrush
                LinearGradientBrush
            TileBrush
                ImageBrush
                WebViewBrush

Only *SolidColorBrush*, *LinearGradientBrush*, *ImageBrush*, and *WebViewBrush* are instantiable. Like many other graphics-related classes, most of these brush classes are defined in the *Windows.UI.Xaml.Media* namespace, although *WebViewBrush* is defined in *Windows.UI.Xaml.Controls*.

The *LinearGradientBrush* creates a gradient between two or more colors. For example, suppose you want to display some text with blue at the left gradually turning to red at the right. While we're at it, let's set a similar gradient on the *Background* property of the *Grid* but going the other way.

In the GradientBrushCode program, a *TextBlock* is instantiated in XAML, and both the *Grid* and the *TextBlock* have names:

Project: GradientBrushCode | File: BlankPage.xaml (excerpt)

```
<Grid Name="contentGrid"
      Background="{StaticResource ApplicationPageBackgroundBrush}">

    <TextBlock Name="txtblk"
               Text="Hello, Windows 8!"
               FontSize="96"
               FontWeight="Bold"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

The constructor of the code-behind file creates two separate *LinearGradientBrush* objects to set to the *Background* property of the *Grid* and *Foreground* property of the *TextBlock*:

Project: GradientBrushCode | File: BlankPage.xaml.cs (excerpt)

```
public BlankPage()
{
    this.InitializeComponent();

    // Create the foreground brush for the TextBlock
    LinearGradientBrush foregroundBrush = new LinearGradientBrush();
    foregroundBrush.StartPoint = new Point(0, 0);
    foregroundBrush.EndPoint = new Point(1, 0);

    GradientStop gradientStop = new GradientStop();
    gradientStop.Offset = 0;
    gradientStop.Color = Colors.Blue;
    foregroundBrush.GradientStops.Add(gradientStop);

    gradientStop = new GradientStop();
    gradientStop.Offset = 1;
    gradientStop.Color = Colors.Red;
    foregroundBrush.GradientStops.Add(gradientStop);

    txtblk.Foreground = foregroundBrush;

    // Create the background brush for the Grid
    LinearGradientBrush backgroundBrush = new LinearGradientBrush
    {
         StartPoint = new Point(0, 0),
         EndPoint = new Point(1, 0)
    };
    backgroundBrush.GradientStops.Add(new GradientStop
    {
        Offset = 0,
        Color = Colors.Red
    });
    backgroundBrush.GradientStops.Add(new GradientStop
    {
```

```
        Offset = 1,
        Color = Colors.Blue
    });

    contentGrid.Background = backgroundBrush;
}
```

The two brushes are created with two different styles of property initialization, but otherwise they're basically the same. The *LinearGradientBrush* class defines two properties named *StartPoint* and *EndPoint* of type *Point*, which is a structure with *X* and *Y* properties representing a two-dimensional coordinate point. The *StartPoint* and *EndPoint* properties are relative to the object to which the brush is applied based on the standard windowing coordinate system: *X* values increase to the right and *Y* values increase going down. The relative point (0, 0) is the upper-left corner and (1, 0) is the upper-right corner, so the brush gradient extends along an imaginary line between these two points, and all lines parallel to that line. The *StartPoint* and *EndPoint* defaults are (0, 0) and (1, 1), which defines a gradient from the upper-left to the lower-right corners of the target object.

LinearGradientBrush also has a property named *GradientStops* that is a collection of *GradientStop* objects. Each *GradientStop* indicates an *Offset* relative to the gradient line and a *Color* at that offset. Generally the offsets range from 0 to 1, but for special purposes they can go beyond the range encompassed by the brush. *LinearGradientBrush* defines additional properties to indicate how the gradient is calculated and what happens beyond the smallest *Offset* and the largest *Offset*.

Here's the result:

If you now consider defining these same brushes in XAML, all of a sudden the limitations of markup become all too evident. XAML lets you define a *SolidColorBrush* by just specifying the color, but how on earth do you set a *Foreground* or *Background* property to a text string defining two points and two or more offsets and colors?

Fortunately, there is a way. As you've seen, you normally indicate that you want a *SolidColorBrush* in XAML simply by specifying the color of the brush:

```
<TextBlock Text="Hello, Windows 8!"
           Foreground="Blue"
           FontSize="96" />
```

The *SolidColorBrush* is created for you behind the scenes.

However, it's possible to use a variation of this syntax that gives you the option of being more explicit about the nature of this brush. Remove that *Foreground* property, and separate the *TextBlock* element into start and end tags:

```
<TextBlock Text="Hello, Windows 8!"
           FontSize="96">

</TextBlock>
```

Within those tags, insert additional start and end tags consisting of the element name, a period, and a property name:

```
<TextBlock Text="Hello, Windows 8!"
           FontSize="96">
    <TextBlock.Foreground>

    </TextBlock.Foreground>
</TextBlock>
```

And within those tags put the object you want to set to that property:

```
<TextBlock Text="Hello, Windows 8!"
           FontSize="96">
    <TextBlock.Foreground>
        <SolidColorBrush Color="Blue" />
    </TextBlock.Foreground>
</TextBlock>
```

Now it's explicit that *Foreground* is being set to an instance of a *SolidColorBrush*.

This is called *property-element syntax*, and it's an important feature of XAML. At first it might seem to you (as it did to me) that this syntax is an extension or aberration of standard XML, but it's definitely not. Periods are perfectly valid characters in XML element names.

With that last little snippet of XAML it is now possible to categorize three types of XAML syntax:

- The *TextBlock* and *SolidColorBrush* are both examples of "object elements" because they are XML elements that result in the creation of objects.
- The *Text*, *FontSize*, and *Color* settings are examples of "property attributes." They are XML

attributes that specify the settings of properties.

- The *TextBlock.Foreground* tag is a "property element." It is a property expressed as an XML element.

XAML poses a restriction on property element tags: Nothing else can go in the start tag. The object being set to the property must be content that goes between the start and end tags.

The following example uses a second set of property element tags for the *Color* property of the *SolidColorBrush*:

```
<TextBlock Text="Hello, Windows 8!"
           FontSize="96">
    <TextBlock.Foreground>
        <SolidColorBrush>
            <SolidColorBrush.Color>
                Blue
            </SolidColorBrush.Color>
        </SolidColorBrush>
    </TextBlock.Foreground>
</TextBlock>
```

If you want, you can set the other two properties of the *TextBlock* similarly:

```
<TextBlock>
    <TextBlock.Text>
        Hello, Windows 8
    </TextBlock.Text>

    <TextBlock.FontSize>
         96
    </TextBlock.FontSize>

    <TextBlock.Foreground>
        <SolidColorBrush>
            <SolidColorBrush.Color>
                Blue
            </SolidColorBrush.Color>
        </SolidColorBrush>
    </TextBlock.Foreground>
</TextBlock>
```

But there's really no point. For these simple properties, the property attribute syntax is shorter and clearer. Where property-element syntax comes to the rescue is in expressing more complex objects like *LinearGradientBrush*. Let's begin again with the property-element tags:

```
<TextBlock Text="Hello, Windows 8!"
           FontSize="96">
    <TextBlock.Foreground>

    </TextBlock.Foreground>
</TextBlock>
```

Put a *LinearGradientBrush* in there, separated into start tags and end tags. Set the *StartPoint* and *EndPoint* properties in this start tag:

```
<TextBlock Text="Hello, Windows 8!"
           FontSize="96">
    <TextBlock.Foreground>
        <LinearGradientBrush StartPoint="0 0" EndPoint="1 0">

        </LinearGradientBrush>
    </TextBlock.Foreground>
</TextBlock>
```

Notice that the two properties of type *Point* are specified with two numbers separated by a space. You can separate the two numbers with a comma if you choose.

The *LinearGradientBrush* has a *GradientStops* property that is a collection of *GradientStop* objects, so include the *GradientStops* property with another property element:

```
<TextBlock Text="Hello, Windows 8!"
           FontSize="96">
    <TextBlock.Foreground>
        <LinearGradientBrush StartPoint="0 0" EndPoint="1 0">
            <LinearGradientBrush.GradientStops>

            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </TextBlock.Foreground>
</TextBlock>
```

The *GradientStops* property is of type *GradientStopCollection*, so let's add that in as well:

```
<TextBlock Text="Hello, Windows 8!"
           FontSize="96">
    <TextBlock.Foreground>
        <LinearGradientBrush StartPoint="0 0" EndPoint="1 0">
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>

                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </TextBlock.Foreground>
</TextBlock>
```

Finally, add the two *GradientStop* objects to the collection:

```
<TextBlock Text="Hello, Windows 8!"
           FontSize="96">
    <TextBlock.Foreground>
        <LinearGradientBrush StartPoint="0 0" EndPoint="1 0">
            <LinearGradientBrush.GradientStops>
                <GradientStopCollection>
                    <GradientStop Offset="0" Color="Blue" />
                    <GradientStop Offset="1" Color="Red" />
```

```
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </TextBlock.Foreground>
</TextBlock>
```

And there we have it: a rather complex object expressed entirely in markup.

Content Properties

The syntax I've just shown you for instantiating and initializing the *LinearGradientBrush* is actually a bit more extravagant than what you actually need. You might be persuaded of this fact when you consider that all the XAML files we've seen so far have apparently been missing some properties and elements. Look at this little snippet of markup:

```
<Page … >
    <Grid … >
        <TextBlock … />
        <TextBlock … />
        <TextBlock … />
    </Grid>
</Page>
```

We know from working with the classes in code that the *TextBlock* elements are added to the *Children* collection of the *Grid*, and the *Grid* is set to the *Content* property of the *Page*. But where are those *Children* and *Content* properties in the markup?

Well, you can include them if you want. Here are the *Page.Content* and *Grid.Children* property elements as they are allowed to appear in a XAML file:

```
<Page … >
    <Page.Content>
        <Grid … >
            <Grid.Children>
                <TextBlock … />
                <TextBlock … />
                <TextBlock … />
            </Grid.Children>
        </Grid>
    </Page.Content>
</Page>
```

This markup is still missing the *UIElementCollection* object that is set to the *Children* property of the *Grid*. That cannot be explicitly included because only elements with parameterless public constructors can be instantiated in XAML files, and the *UIElementCollection* class is missing that constructor.

The real question is this: Why aren't the *Page.Content* and *Grid.Children* property elements required in the XAML file?

Simple: All classes referenced in XAML are allowed to have one (and only one) property that is designated as a "content" property. For this content property, and only this property, property-element tags are not required.

The content property for a particular class is specified as a .NET attribute. Somewhere in the actual class definition of the *Panel* class (from which *Grid* derives) is the following *ContentProperty* attribute:

```
[ContentProperty(Name="Children")]
public class Panel : FrameworkElement
{
    …
}
```

What this means is simple. Whenever the XAML parser encounters some markup like this:

```
<Grid … >
    <TextBlock … />
    <TextBlock … />
    <TextBlock … />
</Grid>
```

then it checks the *ContentProperty* attribute of the *Grid* and discovers that these *TextBlock* elements should be added to the *Children* property.

Similarly, the definition of the *UserControl* class (from which *Page* derives) defines the *Content* property as its content property (which might sound appropriately redundant if you say it out loud):

```
[ContentProperty(Name="Content")]
public class UserControl : Control
{
    …
}
```

You can define a *ContentProperty* attribute in your own classes. The *ContentPropertyAttribute* class required for this is in the *Windows.UI.Xaml.Markup* namespace.

Unfortunately, the current documentation for the Windows Runtime indicates only when a *ContentProperty* attribute has been set on a class—look in the Attributes section of the home page for the *Panel* class, for example—but not what that property actually is! You'll just have to learn by example and retain by habit.

Fortunately, many content properties are defined to be the most convenient property of the class. For *LinearGradientBrush*, the content property is *GradientStops*. Although *GradientStops* is of type *GradientStopCollection*, XAML does not require collection objects to be explicitly included. Here's the excessively wordy form of the *LinearGradientBrush* syntax:

```
<TextBlock Text="Hello, Windows 8!"
           FontSize="96">
    <TextBlock.Foreground>
        <LinearGradientBrush StartPoint="0 0" EndPoint="1 0">
            <LinearGradientBrush.GradientStops>
```

```
                <GradientStopCollection>
                    <GradientStop Offset="0" Color="Blue" />
                    <GradientStop Offset="1" Color="Red" />
                </GradientStopCollection>
            </LinearGradientBrush.GradientStops>
        </LinearGradientBrush>
    </TextBlock.Foreground>
</TextBlock>
```

Neither the *LinearGradientBrush.GradientStops* property elements nor the *GradientStopCollection* tags are required, so it simplifies to this:

```
<TextBlock Text="Hello, Windows 8!"
           FontSize="96">
    <TextBlock.Foreground>
        <LinearGradientBrush StartPoint="0 0" EndPoint="1 0">
            <GradientStop Offset="0" Color="Blue" />
            <GradientStop Offset="1" Color="Red" />
        </LinearGradientBrush>
    </TextBlock.Foreground>
</TextBlock>
```

Now it's difficult to imagine how it can get any simpler and still be valid XML.

It is now possible to rewrite the *GradientBrushCode* program so that everything is done in XAML:

Project: GradientBrushMarkup | File: BlankPage.xaml (excerpt)

```
<Grid>
    <Grid.Background>
        <LinearGradientBrush StartPoint="0 0" EndPoint="1 0">
            <GradientStop Offset="0" Color="Red" />
            <GradientStop Offset="1" Color="Blue" />
        </LinearGradientBrush>
    </Grid.Background>

    <TextBlock Name="txtblk"
               Text="Hello, Windows 8!"
               FontSize="96"
               FontWeight="Bold"
               HorizontalAlignment="Center"
               VerticalAlignment="Center">
        <TextBlock.Foreground>
            <LinearGradientBrush StartPoint="0 0" EndPoint="1 0">
                <GradientStop Offset="0" Color="Blue" />
                <GradientStop Offset="1" Color="Red" />
            </LinearGradientBrush>
        </TextBlock.Foreground>
    </TextBlock>
</Grid>
```

Even with the property element syntax, it's more readable than the code version. What code illustrates most clearly is how something is built. Markup shows the completed construction.

Here's something to watch out for—suppose you define a property element on a *Grid* with multiple

children:

```
<Grid>
    <Grid.Background>
        <SolidColorBrush Color="Blue" />
    </Grid.Background>

    <TextBlock Text="one" />
    <TextBlock Text="two" />
    <TextBlock Text="three" />
</Grid>
```

You can alternatively put the property element at the bottom:

```
<Grid>
    <TextBlock Text="one" />
    <TextBlock Text="two" />
    <TextBlock Text="three" />

    <Grid.Background>
        <SolidColorBrush Color="Blue" />
    </Grid.Background>
</Grid>
```

But you can't have some content before the property element and some content after it:

```
<!-- This doesn't work! -->
<Grid>
    <TextBlock Text="one" />

    <Grid.Background>
        <SolidColorBrush Color="Blue" />
    </Grid.Background>

    <TextBlock Text="two" />
    <TextBlock Text="three" />
</Grid>
```

Why the prohibition? The problem becomes very apparent when you include the property-element tags for the *Children* property:

```
<!-- This doesn't work! -->
<Grid>
    <Grid.Children>
        <TextBlock Text="one" />
    </Grid.Children>

    <Grid.Background>
        <SolidColorBrush Color="Blue" />
    </Grid.Background>

    <Grid.Children>
        <TextBlock Text="two" />
        <TextBlock Text="three" />
    </Grid.Children>
```

```
</Grid>
```

Now it's obvious that the *Children* property is defined twice with two separate collections, and that's not legal.

The *TextBlock* Content Property

As you saw in the WrappedText program in Chapter 1, *TextBlock* allows you to specify text as content. However, the content property of *TextBlock* is not the *Text* property. It is instead a property named *Inlines* of type *InlineCollection*, a collection of *Inline* objects, or more precisely, instances of *Inline* derivatives. The *Inline* class and its derivatives can all be found in the *Windows.UI.Xaml.Documents* namespace. Here's the hierarchy:

Object
- *DependencyObject*
  - *TextElement*
    - *Block*
      - *Paragraph*
    - *Inline*
      - *InlineUIContainer*
      - *LineBreak*
      - *Run* (defines *Text* property)
      - *Span* (defines *Inlines* property)
        - *Bold*
        - *Italic*
        - *Underline*

These classes allow you to specify varieties of formatted text in a single *TextBlock*. *TextElement* defines *Foreground* and all the font-related properties: *FontFamily*, *FontSize*, *FontStyle*, *FontWeight* (for setting bold), *FontStretch* (expanded and compressed for fonts that support it), and *CharacterSpacing*, and these are inherited by all the descendant classes.

The *Block* and *Paragraph* classes are mostly used in connection with a souped-up version of *TextBlock* called *RichTextBlock* that I'll discuss in a later chapter. The remainder of this discussion will focus entirely on classes that derive from *Inline*.

The *Run* element is the only class here that defines a *Text* property, and *Text* is also the content property of *Run*. Any text content in an *InlineCollection* is converted to a *Run*, except when that text is already content of a *Run*. You can also use *Run* objects explicitly to specify different font properties of the text strings.

Span defines an *Inlines* property just like *TextBlock*. This allows *Span* and its descendent classes to be nested. The three descendent classes of *Span* are shortcuts. For example, the *Bold* class is equivalent to *Span* with the *FontWeight* attribute set to *Bold*.

As an example, here's a *TextBlock* with a small *Inlines* collection using the shortcut classes with nesting:

```
<TextBlock>
    Text in <Bold>bold</Bold> and <Italic>italic</Italic> and
    <Bold><Italic>bold italic</Italic></Bold>
</TextBlock>
```

As this is parsed, all those pieces of loose text are converted to *Run* objects, so the *Inlines* collection of the *TextBlock* contains six items: instances of *Run*, *Bold*, *Run*, *Italic*, *Run*, and *Bold*. The *Inlines* collection of the first *Bold* item contains a single *Run* object as does the *Inlines* collection of the first *Italic* item. The *Inlines* collection of the second *Bold* item contains an *Italic* object, whose *Inlines* collection contains a *Run* object.

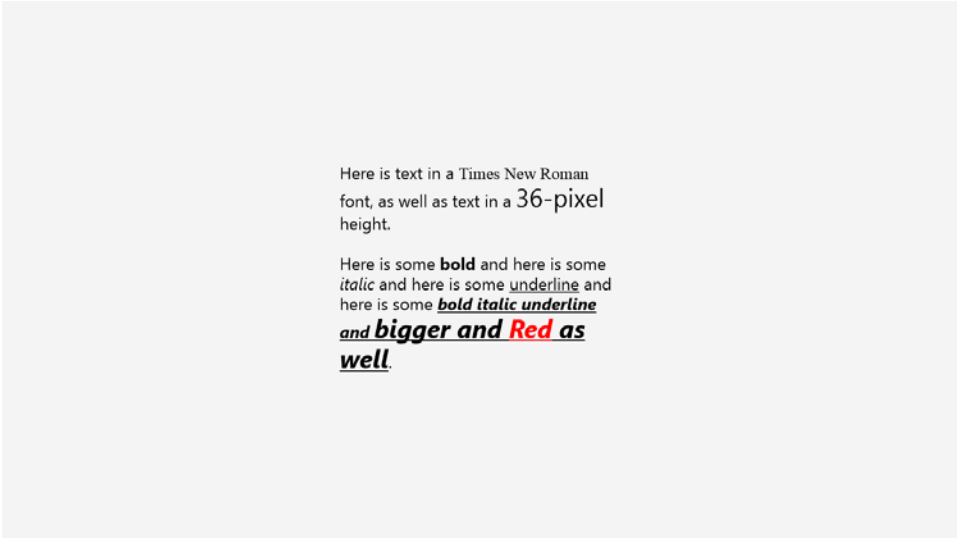
The use of *Bold* and *Italic* with a *TextBlock* demonstrates clearly how the syntax of XAML is based on the classes and properties that support these elements. It wouldn't be possible to nest an *Italic* tag in a *Bold* tag if *Bold* didn't have an *Inlines* collection.

Here's a somewhat more extensive *TextBlock* that shows off more formatting features:

Project: TextFormatting | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <TextBlock Width="400"
               FontSize="24"
               TextWrapping="Wrap"
               HorizontalAlignment="Center"
               VerticalAlignment="Center">
        Here is text in a
        <Run FontFamily="Times New Roman">Times New Roman</Run> font,
        as well as text in a
        <Run FontSize="36">36-pixel</Run> height.
        <LineBreak />
        <LineBreak />
        Here is some <Bold>bold</Bold> and here is some
        <Italic>italic</Italic> and here is some
        <Underline>underline</Underline> and here is some
        <Bold><Italic><Underline>bold italic underline and
        <Span FontSize="36">bigger and
        <Span Foreground="Red">Red</Span> as well</Span>
        </Underline></Italic></Bold>.
    </TextBlock>
</Grid>
```

The *TextBlock* is given an explicit 400-pixel width so that it doesn't sprawl too wide. Individual *Run* elements can always be used to format pieces of text as shown in the first several lines in this paragraph, but if you want nested formatting—and particularly in connection with the shortcut classes—you'll want to switch to *Span* and its shortcut derivatives:



As you can see, the *LineBreak* element can arbitrarily break lines. In theory, the *InlineUIContainer* class allows you to embed any *UIElement* in the text (for example, *Image* elements), but it is not implemented. Try to use it, and you'll get the error "Value does not fall within the expected range."

Sharing Brushes (and Other Resources)

Suppose you have multiple *TextBlock* elements on a page, and you want several of them to have the same brush. If this is a *SolidColorBrush*, the repetitive markup is not too bad. However, if it's a *LinearGradientBrush*, it gets messier. A *LinearGradientBrush* requires at least six tags, and all that repetitive markup becomes very painful, particularly if something needs to be changed.

The Windows Runtime has a feature called the "XAML resource" that lets you share objects among multiple elements. Sharing brushes is one common application of the XAML resource, but the most common is defining and sharing styles.

XAML resources are stored in a *ResourceDictionary*, a dictionary whose keys and values are both of type *object*. Very often, however, the keys are strings. Both *FrameworkElement* and *Application* define a property named *Resources* of type *ResourceDictionary*.

The SharedBrush project shows a typical way to share a *LinearGradientBrush* (and a couple other objects) among several elements on a page. Towards the top of the XAML file I've defined a *Resources* property element for the collection of resources for that page:

Project: SharedBrush | File: BlankPage.xaml (excerpt)

```
<Page … >

    <Page.Resources>
        <x:String x:Key="appName">Shared Brush App</x:String>
```

```
        <LinearGradientBrush x:Key="rainbowBrush">
            <GradientStop Offset="0" Color="Red" />
            <GradientStop Offset="0.17" Color="Orange" />
            <GradientStop Offset="0.33" Color="Yellow" />
            <GradientStop Offset="0.5" Color="Green" />
            <GradientStop Offset="0.67" Color="Blue" />
            <GradientStop Offset="0.83" Color="Indigo" />
            <GradientStop Offset="1" Color="Violet" />
        </LinearGradientBrush>

        <FontFamily x:Key="fontFamily">Times New Roman</FontFamily>

        <x:Double x:Key="fontSize">96</x:Double>
    </Page.Resources>
    …
</Page>
```

Often the definition of resources near the top of a XAML file is referred to as a "resources section." This particular *Resources* dictionary is initialized with four items of four different types: *String*, *LinearGradientBrush*, *FontFamily*, and *Double*. Notice the "x" prefix on *String* and *Double*. These are .NET primitive types, of course, but they are not Windows Runtime types, and hence they are not in the default XAML namespace. The *x:Boolean* and *x:Int32* types are also available.

Also notice that each of these objects has an *x:Key* attribute. The *x:Key* attribute is valid only in a *Resources* dictionary. As the name suggests, the *x:Key* attribute is the key for that item in the dictionary.

In the body of the XAML file, an element references the resource by using this key in some special markup called a XAML *markup extension*.

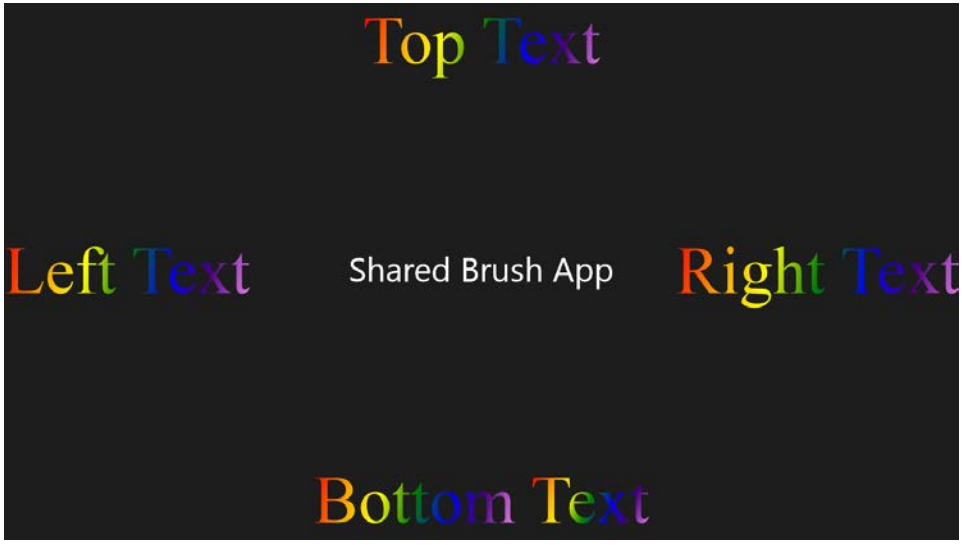
There are just a few XAML markup extensions, and you'll always recognize them by curly braces. The markup extension for referencing a resource consists of the keyword *StaticResource* and the key name. In fact, you've already seen the *StaticResource* markup extension numerous times: it provides the standard *Grid* with a background brush. The rest of this XAML file uses *StaticResource* to obtain items defined in the *Resources* dictionary:

Project: SharedBrush | File: BlankPage.xaml (excerpt)

```
<Page … >
    …
    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <TextBlock Text="{StaticResource appName}"
                   FontSize="48"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Center" />

        <TextBlock Text="Top Text"
                   Foreground="{StaticResource rainbowBrush}"
                   FontFamily="{StaticResource fontFamily}"
                   FontSize="{StaticResource fontSize}"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Top" />
```

```
        <TextBlock Text="Left Text"
                   Foreground="{StaticResource rainbowBrush}"
                   FontFamily="{StaticResource fontFamily}"
                   FontSize="{StaticResource fontSize}"
                   HorizontalAlignment="Left"
                   VerticalAlignment="Center" />

        <TextBlock Text="Right Text"
                   Foreground="{StaticResource rainbowBrush}"
                   FontFamily="{StaticResource fontFamily}"
                   FontSize="{StaticResource fontSize}"
                   HorizontalAlignment="Right"
                   VerticalAlignment="Center" />

        <TextBlock Text="Bottom Text"
                   Foreground="{StaticResource rainbowBrush}"
                   FontFamily="{StaticResource fontFamily}"
                   FontSize="{StaticResource fontSize}"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Bottom" />
    </Grid>
</Page>
```

Here's the result



A few notes:

Referencing the same three resources in four *TextBlock* elements cries out for a more efficient approach, namely a style, which I'll discuss later in this chapter.

Resources must be defined in a XAML file lexically preceding their use. This is why it's most common for the *Resources* dictionary to be near the top of a XAML file and most conveniently defined on the

root element.

However, every *FrameworkElement* descendant can support a *Resources* dictionary, so you might include them further down the visual tree. The keys must be unique within any *Resources* dictionary, but you can use duplicate keys in other *Resources* dictionaries. When the XAML parser encounters a *StaticResource* markup extension, it begins searching up the visual tree for a *Resources* dictionary with a matching key and it uses the first one it encounters. You can effectively override the values of *Resources* keys with those in more local dictionaries.

If the XAML parser cannot find a matching key by searching up the visual tree, it checks the *Resources* dictionary in the *Application* object. The App.xaml file is an ideal place for defining resources that are used throughout the application. To use a bunch of resources across multiple applications, you can define them in a separate XAML file with a root element of *ResourceDictionary*. Include that file in a project, reference it in the App.xaml file, and you can then use items in that dictionary.

Indeed, an example is already provided for you in the standard Visual Studio projects for Metro style applications. The Common folder contains a file named StandardStyles.xaml that has a root element of *ResourceDictionary*:

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

    …

</ResourceDictionary>
```

This file is referenced in the standard App.xaml file. In fact, referencing this resources collection is just about all that the standard App.xaml file does:

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

    <Application.Resources>
        <ResourceDictionary>
            <ResourceDictionary.MergedDictionaries>
                <ResourceDictionary Source="Common/StandardStyles.xaml"/>
            </ResourceDictionary.MergedDictionaries>
        </ResourceDictionary>
    </Application.Resources>
</Application>
```

You can include your own collections of resources by inserting additional *ResourceDictionary* tags in the *MergedDictionaries* collection. Or you can include your own resources directly in the *App* object's *Resources* dictionary.

You can also reference the *Resources* dictionary from code. Following the *InitializeComponent* call,

you can retrieve an item from the dictionary with an indexer:

```
FontFamily fntfam = this.Resources["fontFamily"] as FontFamily;
```

Now try this: Comment out the "fontFamily" entry in the BlankPage.xaml file, but add that item to the dictionary in the *BlankPage* constructor *prior* to the *InitializeComponent* call.

```
this.Resources.Add("fontFamily", new FontFamily("Times New Roman"));
```

When the XAML file is parsed by *InitializeComponent*, this object will be available within that XAML file.

At the time of this writing, the *ResourceDictionary* class does not define a public method that searches up the visual tree for dictionaries in ancestor classes. If you need something like that to search for resources in code, you can easily write it yourself by "climbing the visual tree" using the *Parent* property defined by *FrameworkElement* or the *VisualTreeHelper* class defined in the *Windows.UI.Xaml.Media* namespace. The *Application* object for the application is available from the static *Application.Current* property.

The predefined resources (such as the *ApplicationPageBackgroundBrush* referenced by the *Grid*) don't seem to be programmatically enumerable. Nor are they documented. However, in Visual Studio you can see a list of the predefined brushes by clicking the *Grid* in the BlankPage.xaml file and viewing the available *Background* brush identifiers in the Properties view in the lower-right corner of Visual Studio.

After *ApplicationPageBackgroundBrush*, the next most important predefined resource identifier is *ApplicationTextBrush*, which is black in the light theme, and white in the dark theme. If you need a color to properly contrast with the background (as I will shortly), this is it. The *ControlHighlightBrush* is also convenient for a splash of color that contrasts with both the background and foreground.

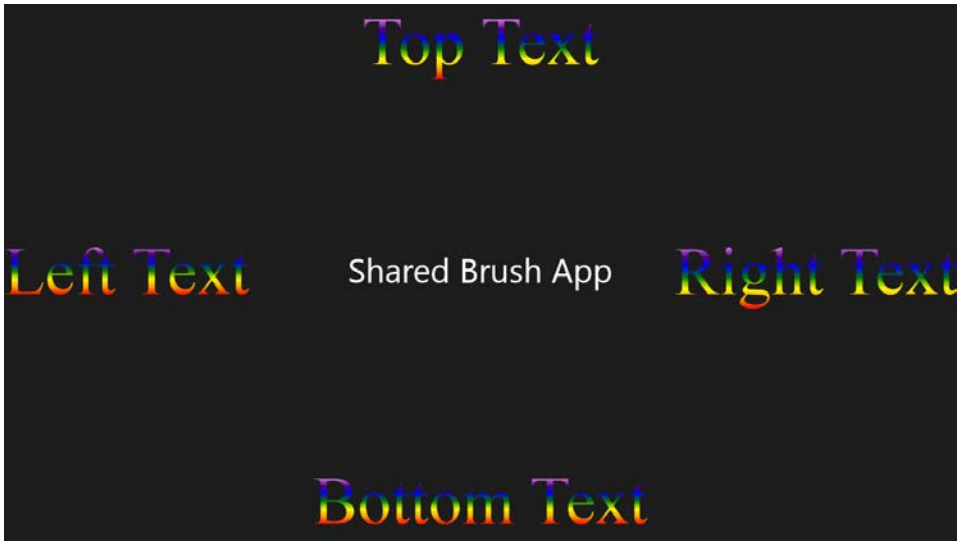
Resources Are Shared

Are resource objects truly shared among the elements that reference them? Or are separate instances created for each *StaticResource* reference?

Try inserting the following code after the *InitializeComponent* call in the SharedBrush.xaml.cs file:

```
TextBlock txtblk = (this.Content as Grid).Children[1] as TextBlock;
LinearGradientBrush brush = txtblk.Foreground as LinearGradientBrush;
brush.StartPoint = new Point(0, 1);
brush.EndPoint = new Point(0, 0);
```

This code references the *LinearGradientBrush* of the second *TextBlock* in the *Children* collection of the *Grid* and changes the *StartPoint* and *EndPoint* properties. Lo and behold, all the *TextBlock* elements referencing that *LinearGradientBrush* are affected:



Conclusion: resources are shared.

It's also easy to verify that even if a resource is not referenced by any element, it is still instantiated.

A Bit of Vector Graphics

As you've seen, displaying text and bitmaps in a Metro style application involves creating objects of type *TextBlock* and *Image* and attaching them to a visual tree. There's no concept of "drawing" or "painting," at least not on the application level. Internal to the Windows Runtime, the *TextBlock* and *Image* elements are rendering themselves.

Similarly, if you wish to display some vector graphics—lines, curves, and filled areas—you don't do it by calling methods like *DrawLine* and *DrawBezier*. These methods do not exist! Instead, you create elements of type *Line*, *Polyline*, *Polygon*, and *Path*. These classes derive from the *Shape* class (which itself derives from *FrameworkElement*) and can all be found in the *Windows.UI.Xaml.Shapes* namespace, which is sometimes referred to as the *Shapes* library.

A deep exploration of vector graphics awaits us in a future chapter. For now, let's just examine two of the most powerful members of the *Shapes* library: *Polyline* and *Path*.

Polyline renders a collection of connected straight lines, but its real purpose is to draw complex curves. All you need to do is keep the individual lines short and supply plenty of them. Don't hesitate to give *Polyline* thousands of lines. That's what it's there for.

Let's use *Polyline* to draw an Archimedean spiral. The XAML file for the Spiral program instantiates the *Polyline* object but doesn't include the points that define the figure:

Project: Spiral | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <Polyline Name="polyline"
              Stroke="{StaticResource ApplicationTextBrush}"
              StrokeThickness="3"
              HorizontalAlignment="Center"
              VerticalAlignment="Center" />
</Grid>
```

The *Stroke* property (inherited from *Shape*) is the brush used to draw the actual lines. Generally, this is a *SolidColorBrush*, but you'll see shortly that it doesn't have to be. I've used *StaticResource* with the predefined identifier that provides a white brush with a dark theme and a black brush with a light theme. *StrokeThickness* (also inherited from *Shape*) is the width of the lines in pixels, and you've seen *HorizontalAlignment* and *VerticalAlignment* before.

It might seem a little strange to specify *HorizontalAlignment* and *VerticalAlignment* for a chunk of vector graphics, so a little explanation might be in order.

Two-dimensional vector graphics involve the use of coordinate points in the form (*X*, *Y*) on a Cartesian coordinate system, where *X* is a position on the horizontal axis and *Y* is a position on the vertical axis. Vector graphics in the Windows Runtime use a coordinate convention commonly associated with windowing environments: values of *X* increase to the right (as is normal), but values of *Y* increase going down (which is opposite the mathematical convention).

When only positive values of *X* and *Y* are used, the origin—the point (0, 0)—is the upper-left corner of the graphical figure.

Negative coordinates can be used to indicate points to the left of the origin or above the origin. However, when the Windows Runtime calculates the dimensions of a vector graphics object for layout purposes, these negative coordinates are ignored. For example, suppose you draw a polyline with points that have *X* coordinates ranging from –100 to 300 and *Y* coordinates ranging from –200 to 400. This implies that the polyline has a dimension of 400 pixels wide and 600 pixels high, and that is certainly true. But for purposes of layout and alignment, the polyline is treated as if it were 300 pixels wide and 400 pixels tall.

For a vector graphics figure to be treated in a predictable manner in the Windows Runtime layout system, all that's required is that you regard the point (0, 0) as the upper-left corner. For purposes of layout, the maximum positive *X* coordinate becomes the element's width and the maximum positive *Y* coordinate become's the element's height.

For specifying a coordinate point, the *Windows.Foundation* namespace includes a *Point* structure that has two properties of type *double* named *X* and *Y*. In addition, the *Windows.UI.Xaml.Media* namespace includes a *PointCollection*, which is a collection of *Point* objects.

The only property that *Polyline* defines on its own is *Points* of type *PointCollection*. A collection of points can be assigned to the *Points* property in XAML, but for very many points calculated algorithmically, code is ideal. In the constructor of the *Spiral* class, a *for* loop goes from 0 to 3600

degrees, effectively spinning around a circle 10 times:

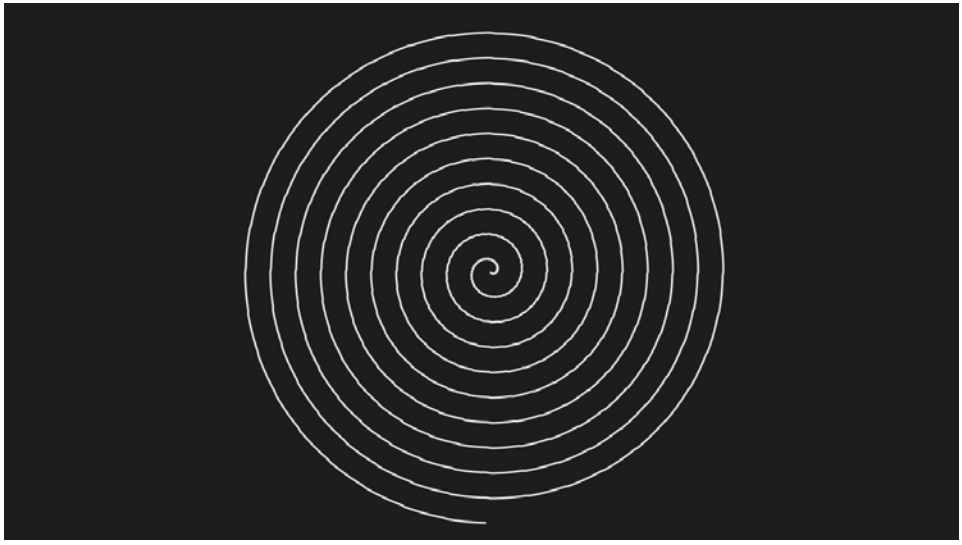
Project: Spiral | File: BlankPage.xaml.cs (excerpt)

```
public BlankPage()
{
    this.InitializeComponent();

    for (int angle = 0; angle < 3600; angle++)
    {
        double radians = Math.PI * angle / 180;
        double radius = angle / 10;
        double x = 360 + radius * Math.Sin(radians);
        double y = 360 + radius * Math.Cos(radians);
        polyline.Points.Add(new Point(x, y));
    }
}
```

The *radians* variable converts degrees to radians for the .NET trig functions, and *radius* is calculated to range from 0 through 360 depending on the *angle*, which means that the maximum radius will be 360 pixels. The values returned by the *Math.Sin* and *Math.Cos* static methods are multiplied by *radius*, which means these products will range between –360 and 360 pixels.

To shift this figure so that all pixels have positive values relative to an upper-left origin, 360 is added to both products. The spiral is thus centered at the point (360, 360) and extends not more than 360 pixels in all directions.

The loop concludes by instantiating a *Point* value and adding it to the *Points* collection of the *Polyline*. Here it is:



Without the *HorizontalAlignment* and *VerticalAlignment* settings, the figure would be aligned at the upper-left corner of the page. If the adjustment for the spiral's center is also removed from the

calculation, the center would be in the upper-left corner of the page and ¾ of the figure would not be visible. If you keep *HorizontalAlignment* and *VerticalAlignment* set to *Center* but remove the adjustment for the spiral's center, you'll see the figure positioned so that the lower-right quadrant is centered.

The spiral almost fills the screen, but that's only because the screen I'm using for these images has a height of 768 pixels. What if we wanted to ensure that the spiral filled the screen regardless of the screen's size?

One solution is to base the numbers going into the calculation of the spiral coordinates directly on the pixel size of the screen. You'll see how to do that in Chapter 3, "Basic Event Handling."

Another solution requires noticing that the *Shape* class defines a property named *Stretch* that you use in exactly the same way you use the *Stretch* property of *Image*. By default, the *Stretch* property for *Polyline* is the enumeration member *Stretch.None*, which means no stretching, but you can set it to *Uniform* so that the figure fills the container while maintaining its aspect ratio.

The StretchedSpiral project demonstrates this. The XAML file sets a larger stroke width as well:

Project: StretchedSpiral | File: BlankPage.xaml (excerpt)

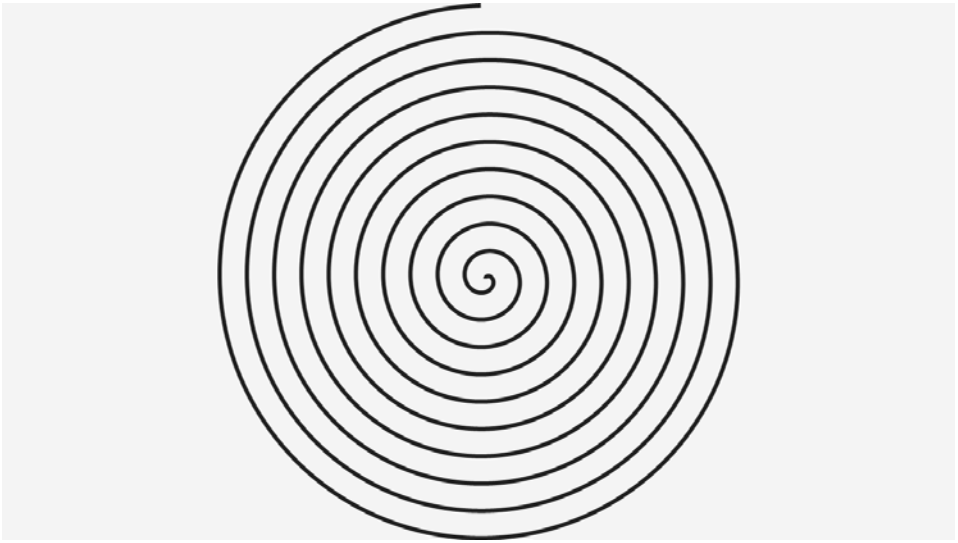
```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <Polyline Name="polyline"
              Stroke="{StaticResource ApplicationTextBrush}"
              StrokeThickness="6"
              Stretch="Uniform" />
</Grid>
```

The code-behind file calculates the coordinates of the spiral using arbitrary coordinates, which in this case I've chosen based on a radius of 1000:

Project: StretchedSpiral | File: BlankPage.xaml.cs (excerpt)

```
public BlankPage()
{
    this.InitializeComponent();

    for (int angle = 0; angle < 3600; angle++)
    {
        double radians = Math.PI * angle / 180;
        double radius = angle / 3.6;
        double x = 1000 + radius * Math.Sin(radians);
        double y = 1000 - radius * Math.Cos(radians);
        polyline.Points.Add(new Point(x, y));
    }
}
```

You might also notice that I changed a plus to a minus in the *y* calculation so that the spiral ends at the top rather than the bottom. The switch to the light theme demonstrates the convenience of using *ApplicationTextBrush* for the *Stroke* color:



Try setting the *Stretch* property to *Fill* to see this circular spiral be distorted into an elliptical spiral.

You'll recall how *LinearGradientBrush* adapts itself to the size of whatever element it's applied to. The same is true when using that brush with vector graphics. Let's instead try an *ImageBrush*, which is a brush created from a bitmap.

The code-behind file for ImageBrushedSpiral is the same as StretchedSpiral. The XAML file widens the stroke considerably and instantiates an *ImageBrush*:

Project: ImageBrushedSpiral | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <Polyline Name="polyline"
              StrokeThickness="25"
              Stretch="Uniform">
        <Polyline.Stroke>
            <ImageBrush ImageSource="http://www.charlespetzold.com/pw6/PetzoldJersey.jpg"
                        Stretch="UniformToFill"
                        AlignmentY="Top" />
        </Polyline.Stroke>
    </Polyline>
</Grid>
```

The *ImageSource* property of *ImageBrush* is of type *ImageSource*, just like the *Source* property of *Image*. In XAML you can just set it to a URL. *ImageBrush* has its own *Stretch* property, which by default is *Fill*. This means that the bitmap is stretched to fill the area without respecting the aspect ratio. For the image I'm using, that would make me look fat, so I switched to *UniformToFill*, which maintains the image's aspect ratio while filling the area. Doing so requires part of the image to be cropped. Use the *AlignmentX* and *AlignmentY* properties to indicate how the bitmap should be aligned with the graphical figure, and consequently, where the image should be cropped. For this bitmap, I prefer that the bottom be cropped rather than my head:

Notice that the alignment of the image seems to be based on the geometric line of the spiral rather than the line rendered with a width of 25 pixels. This causes areas at the top, left, and right sides to be shaved off. The problem can be fixed with the *Transform* property of *ImageBrush*, but that's a little too advanced for this chapter.

You may have noticed that *ImageBrush* derives from *TileBrush*. That heritage might suggest that you could repeat bitmap images horizontally and vertically to tile a surface, but doing so is not supported by the Windows Runtime.

Any curve that you can define with parametric formulas, you can render with *Polyline*. But if the complex curves you need are arcs (that is, curves on the circumference of an ellipse), cubic Bézier splines (the standard sort), or quadratic Bézier splines (which have only one control point), you don't need to use *Polyline*. These curves are all supported with the *Path* element.

Path defines just one property on its own called *Data*, of type *Geometry*, a class defined in *Windows.UI.Xaml.Media*. In the Windows Runtime, *Geometry* and related classes represent pure analytic geometry. The *Geometry* object defines lines and curves using coordinate points, and the *Path* renders those lines with a particular stroke brush and thickness.

The most powerful and flexible *Geometry* derivative is *PathGeometry*. The content property of *PathGeometry* is named *Figures*, which is a collection of *PathFigure* objects. Each *PathFigure* is a series of connected straight lines and curves. The content property of *PathFigure* is *Segments*, a collection of *PathSegment* objects. *PathSegment* is the parent class to *LineSegment*, *PolylineSegment*, *BezierSegment*, *PolyBezierSegment*, *QuadraticBezierSegment*, *PolyQuadraticBezierSegment*, and *ArcSegment*.

Let's display the word HELLO by using *Path* and *PathGeometry*:

Project: HelloVectorGraphics | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
```

```
    <Path Stroke="Red"
          StrokeThickness="12"
          StrokeLineJoin="Round"
          HorizontalAlignment="Center"
          VerticalAlignment="Center">
        <Path.Data>
            <PathGeometry>
                <!-- H -->
                <PathFigure StartPoint="0 0">
                    <LineSegment Point="0 100" />
                </PathFigure>
                <PathFigure StartPoint="0 50">
                    <LineSegment Point="50 50" />
                </PathFigure>
                <PathFigure StartPoint="50 0">
                    <LineSegment Point="50 100" />
                </PathFigure>

                <!-- E -->
                <PathFigure StartPoint="125 0">
                    <BezierSegment Point1="60 -10" Point2="60 60" Point3="125 50" />
                    <BezierSegment Point1="60 40" Point2="60 110" Point3="125 100" />
                </PathFigure>

                <!-- L -->
                <PathFigure StartPoint="150 0">
                    <LineSegment Point="150 100" />
                    <LineSegment Point="200 100" />
                </PathFigure>

                <!-- L -->
                <PathFigure StartPoint="225 0">
                    <LineSegment Point="225 100" />
                    <LineSegment Point="275 100" />
                </PathFigure>

                <!-- O -->
                <PathFigure StartPoint="300 50">
                    <ArcSegment Size="25 50" Point="300 49.9" IsLargeArc="True" />
                </PathFigure>
            </PathGeometry>
        </Path.Data>
    </Path>
</Grid>
```

Each letter is one or more *PathFigure* objects, which always specifies a starting point for a series of connected lines. The *PathSegment* derivatives continue the figure from that point. For example, to draw the "E," *BezierSegment* specifies two control points and an end point. The next *BezierSegment* then continues from the end of the previous segment. (In the *ArcSegment*, the end point for the arc can't be the same as the start point or nothing will be drawn. That why it's set to 1/10th pixel short.)

The result suggests that a pair of Bézier splines was perhaps not the best way to render a capital E:

Try setting the *Stretch* property of *Path* to *Fill* for a "really big hello":

Of course you can assemble the *PathFigure* and *PathSegment* objects in code, but let me show you an easier way to do it in XAML. A Path Markup Syntax is available that consists of single letters, coordinate points, an occasional size, and a couple Boolean values that reduce the markup considerably. The HelloVectorGraphicsPath project creates the same figure as HelloVectorGraphics:

Project: HelloVectorGraphicsPath | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <Path Stroke="Red"
          StrokeThickness="12"
          StrokeLineJoin="Round"
```

```
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Data="M 0 0 L 0 100 M 0 50 L 50 50 M 50 0 L 50 100
                  M 125 0 C 60 -10, 60 60, 125 50, 60 40, 60 110, 125 100
                  M 150 0 L 150 100, 200 100
                  M 225 0 L 225 100, 275 100
                  M 300 50 A 25 50 0 1 0 300 49.9" />
</Grid>
```

The *Data* property is now one big string, but I've separated it into five lines corresponding to the five letters. The M code is a "move" followed by *x* and *y* coordinate points. The L is a line (or, more precisely, a polyline) followed by one or more points; C is a cubic Bézier curve, followed by control points and an end point, but more than one can be included; and A is an arc. The arc is by far the most complex: The first two numbers indicate the horizontal and vertical radii of an ellipse, which is rotated a number of degrees given by the next argument. Following are two flags for the *IsLargeArc* property and sweep direction, followed by the end point.

Defining a complex geometry in terms of Path Markup Syntax is one example of something that can be done only in XAML. Whatever class performs this conversion is not publicly exposed in the Windows Runtime. It is available only to the XAML parser. To convert a string of Path Markup Syntax to a *Geometry* in code would require some way to convert XAML to an object in code.

Fortunately, something like that is available. It's a static method named *XamlReader.Load* in the *Windows.UI.Xaml.Markup* namespace. Pass it a string of XAML and get out an instance of the root element with all the other parts of the tree instantiated and assembled. *XamlReader.Load* has some restrictions—the XAML it parses can't refer to event handlers in external code, for example—but it is a very powerful facility. In Chapter 7, "Building an Application," I'll show you the source code for a tool called XamlCruncher that lets you interactively experiment with XAML.

Meanwhile, here's a *Path* with Path Markup Syntax created entirely in code:

Project: PathMarkupSyntaxCode | File: BlankPage.xaml.cs

```
using Windows.UI;                   // for Colors
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Markup;       // for XamlReader
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Shapes;       // for Path

namespace PathMarkupSyntaxCode
{
    public sealed partial class BlankPage : Page
    {
        public BlankPage()
        {
            this.InitializeComponent();

            Path path = new Path
            {
                Stroke = new SolidColorBrush(Colors.Red),
```

```
                StrokeThickness = 12,
                StrokeLineJoin = PenLineJoin.Round,
                HorizontalAlignment = HorizontalAlignment.Center,
                VerticalAlignment =  VerticalAlignment.Center,
                Data = PathMarkupToGeometry(
                    "M 0 0 L 0 100 M 0 50 L 50 50 M 50 0 L 50 100 " +
                    "M 125 0 C 60 -10, 60 60, 125 50, 60 40, 60 110, 125 100 " +
                    "M 150 0 L 150 100, 200 100 " +
                    "M 225 0 L 225 100, 275 100 " +
                    "M 300 50 A 25 50 0 1 0 300 49.9")
            };

            (this.Content as Grid).Children.Add(path);
        }

        Geometry PathMarkupToGeometry(string pathMarkup)
        {
            string xaml =
                "<Path " +
                "xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'>" +
                "<Path.Data>" + pathMarkup + "</Path.Data></Path>";

            Path path = XamlReader.Load(xaml) as Path;

            // Detach the PathGeometry from the Path
            Geometry geometry = path.Data;
            path.Data = null;
            return geometry;
        }
    }
}
```

Watch out when working with the *Path* class in code: the BlankPage.xaml.cs file that Visual Studio generates does not include a *using* directive for *Windows.UI.Xaml.Shape* where *Path* resides but does include a *using* directive for *System.IO*, which has a very different *Path* class for working with files and directories.

The magic method is down at the bottom. It assembles a tiny piece of legal XAML with *Path* as the root element and property-element syntax to enclose the string of Path Markup Syntax. Notice that the XAML must include the standard XML namespace declaration. If *XamlReader.Load* doesn't encounter any errors, it returns a *Path* with a *Data* property set to a *PathGeometry*. However, you can't use this *PathGeometry* for another *Path* unless you disconnect it from this *Path*, which requires setting the *Data* property of the returned *Path* to *null*.

Styles

You've seen how brushes can be defined as resources and shared among elements. By far the most common use of resources is to define styles, which are instances of the *Style* class. A style is basically a collection of property definitions that can be shared among multiple elements. The use of styles not

only reduces repetitive markup, but also allows easier global changes.

After this discussion, much of the StandardStyles.xaml file included in the Common folder of your Visual Studio projects will be comprehensible, except for large sections within *ControlTemplate* tags. That's coming up in a later chapter.

The SharedBrushWithStyle project is much the same as SharedBrush except that it uses a *Style* to consolidate several properties. Here's the new *Resources* section with the *Style* near the bottom:

Project: SharedBrushWithStyle | File: BlankPage.xaml (excerpt)

```
<Page.Resources>
    <x:String x:Key="appName">Shared Brush with Style</x:String>

    <LinearGradientBrush x:Key="rainbowBrush">
        <GradientStop Offset="0" Color="Red" />
        <GradientStop Offset="0.17" Color="Orange" />
        <GradientStop Offset="0.33" Color="Yellow" />
        <GradientStop Offset="0.5" Color="Green" />
        <GradientStop Offset="0.67" Color="Blue" />
        <GradientStop Offset="0.83" Color="Indigo" />
        <GradientStop Offset="1" Color="Violet" />
    </LinearGradientBrush>

    <Style x:Key="rainbowStyle" TargetType="TextBlock">
        <Setter Property="FontFamily" Value="Times New Roman" />
        <Setter Property="FontSize" Value="96" />
        <Setter Property="Foreground" Value="{StaticResource rainbowBrush}" />
    </Style>
</Page.Resources>
```

Like all resources, the start tag of the *Style* includes an *x:Key* attribute. *Style* also requires a *TargetType* attribute indicating either *FrameworkElement* or a class that derives from *FrameworkElement*. Styles can be applied only to *FrameworkElement* derivatives.

The body of the *Style* includes a bunch of *Setter* tags, each of which specifies *Property* and *Value* attributes. Notice that the last one has its *Value* attribute set to a *StaticResource* of the previously defined *LinearGradientBrush*. For this reference to work, this particular *Style* must be defined later in the XAML file than the brush, although it can be in a different *Resources* section deeper in the visual tree.

Like other resources, an element references a *Style* by using the *StaticResource* markup extension on its *Style* property:

Project: SharedBrushWithStyle | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <TextBlock Text="{StaticResource appName}"
               FontSize="48"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />

    <TextBlock Text="Top Text"
```

```
                Style="{StaticResource rainbowStyle}"
                HorizontalAlignment="Center"
                VerticalAlignment="Top" />

    <TextBlock Text="Left Text"
                Style="{StaticResource rainbowStyle}"
                HorizontalAlignment="Left"
                VerticalAlignment="Center" />

    <TextBlock Text="Right Text"
                Style="{StaticResource rainbowStyle}"
                HorizontalAlignment="Right"
                VerticalAlignment="Center" />

    <TextBlock Text="Bottom Text"
                Style="{StaticResource rainbowStyle}"
                HorizontalAlignment="Center"
                VerticalAlignment="Bottom" />
</Grid>
```

Except for the application name, the visuals are the same as the SharedBrush program.

There is an alternative way for this particular *Style* to incorporate the *LinearGradientBrush*. Just as you can use property-element syntax on elements to define an object with complex markup, you can use property-element syntax with the *Value* property of the *Setter* class:

```
<Style x:Key="rainbowStyle" TargetType="TextBlock">
    <Setter Property="FontFamily" Value="Times New Roman" />
    <Setter Property="FontSize" Value="96" />
    <Setter Property="Foreground">
        <Setter.Value>
            <LinearGradientBrush>
                <GradientStop Offset="0" Color="Red" />
                <GradientStop Offset="0.17" Color="Orange" />
                <GradientStop Offset="0.33" Color="Yellow" />
                <GradientStop Offset="0.5" Color="Green" />
                <GradientStop Offset="0.67" Color="Blue" />
                <GradientStop Offset="0.83" Color="Indigo" />
                <GradientStop Offset="1" Color="Violet" />
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
</Style>
```

I know it looks a little odd at first, but defining brushes within styles is very common. Notice that the *LinearGradientBrush* here has no *x:Key* of its own. Only items defined at the root level in a *Resources* collection can have *x:Key* attributes.

You can define a *Style* in code, for example, like so:

```
Style style = new Style(typeof(TextBlock));
style.Setters.Add(new Setter(TextBlock.FontSizeProperty, 96));
style.Setters.Add(new Setter(TextBlock.FontFamilyProperty,
                             new FontFamily("Times New Roman")));
```

You could then add this to the *Resources* collection of a *Page* prior to the *InitializeComponent* call so that it would be available to *TextBlock* elements defined in the XAML file. Or you could assign this *Style* object directly to the *Style* property of a *TextBlock*. This isn't common, however, because code offers other solutions for defining the same properties on several different elements, namely the *for* or *foreach* loop.

Take careful note of the first argument to the *Setter* constructor. It's defined as a *DependencyProperty*, and what you specify is a static dependency property defined by (or inherited by) the target class of the style. This is an excellent example of how dependency properties allow a property of a class to be specified independently of a particular instance of that class.

The code also makes clear that the properties targeted by a *Style* can only be dependency properties. I mentioned earlier that dependency properties impose a hierarchy on the way that properties can be set. For example, suppose you have the following markup in this program:

```
<TextBlock Text="Top Text"
           Style="{StaticResource rainbowStyle}"
           FontSize="24"
           HorizontalAlignment="Center"
           VerticalAlignment="Top" />
```

The *Style* defines a *FontSize* value, but the *FontSize* property is also set locally on the *TextBlock*. As you might hope and expect, the local setting takes precedence over the *Style* setting, and both take precedence over a *FontSize* value propagated through the visual tree.

Once a *Style* object is set to the *Style* property of an element, the *Style* can no longer be changed. You can later set a different *Style* object to the element, and you can change properties of objects referenced by the style (such as brushes), but you cannot set or remove *Setter* objects or change their *Value* properties.

Styles can inherit property settings from other styles by using a *Style* property called *BasedOn*, which is usually set to a *StaticResource* markup extension referencing a previously defined *Style* definition:

```
<Style x:Key="baseTextBlockStyle" TargetType="TextBlock">
    <Setter Property="FontFamily" Value="Times New Roman" />
    <Setter Property="FontSize" Value="24" />
</Style>

<Style x:Key="gradientStyle" TargetType="TextBlock"
       BasedOn="{StaticResource baseTextBlockStyle}">
    <Setter Property="FontSize" Value="96" />
    <Setter Property="Foreground">
       <Setter.Value>
          <LinearGradientBrush>
                <GradientStop Offset="0" Color="Red" />
                <GradientStop Offset="1" Color="Blue" />
           </LinearGradientBrush>
       </Setter.Value>
    </Setter>
```

```
</Style>
```

The *Style* with the key "gradientStyle" is based on the previous *Style* with the key "baseTextBlockStyle," which means that it inherits the *FontFamily* setting, overrides the *FontSize* setting, and defines a new *Foreground* setting.

Here's another example:

```
<Style x:Key="centeredStyle" TargetType="FrameworkElement">
    <Setter Property="HorizontalAlignment" Value="Center" />
    <Setter Property="VerticalAlignment" Value="Center" />
</Style>

<Style x:Key="rainbowStyle" TargetType="TextBlock"
       BasedOn="{StaticResource centeredStyle}">
    <Setter Property="FontSize" Value="96" />
    <Setter Property="Foreground">
        <Setter.Value>
            <LinearGradientBrush>
                <GradientStop Offset="0" Color="Red" />
                <GradientStop Offset="1" Color="Blue" />
            </LinearGradientBrush>
        </Setter.Value>
    </Setter>
</Style>
```

In this case, the first *Style* has a *TargetType* of *FrameworkElement*, which means that it can include only properties defined by *FrameworkElement* or inherited by *FrameworkElement*. You can still use this property for a *TextBlock* because *TextBlock* derives from *FrameworkElement*. The second *Style* is based on "centeredStyle" but has a *TargetType* of *TextBlock*, which means it can also include property settings specific to *TextBlock*. The *TargetType* must be the same as the *BasedOn* type or derived from the *BasedOn* type.

Despite all I've said about keys being required for resources, a *Style* is actually the only exception to this rule. A *Style* without an *x:Key* is a very special case called an *implicit style*. The *Resources* section of the ImplicitStyle project has an example:

Project: ImplicitStyle | File: BlankPage.xaml (excerpt)

```
<Page.Resources>
    <x:String x:Key="appName">Implicit Style App</x:String>

    <Style TargetType="TextBlock">
        <Setter Property="FontFamily" Value="Times New Roman" />
        <Setter Property="FontSize" Value="96" />
        <Setter Property="Foreground">
            <Setter.Value>
                <LinearGradientBrush>
                    <GradientStop Offset="0" Color="Red" />
                    <GradientStop Offset="0.17" Color="Orange" />
                    <GradientStop Offset="0.33" Color="Yellow" />
                    <GradientStop Offset="0.5" Color="Green" />
                    <GradientStop Offset="0.67" Color="Blue" />
```

```
                    <GradientStop Offset="0.83" Color="Indigo" />
                    <GradientStop Offset="1" Color="Violet" />
                </LinearGradientBrush>
            </Setter.Value>
        </Setter>
    </Style>
</Page.Resources>
```

A key is actually created behind the scenes. It's an object of type *RuntimeType* (which is not a public type) indicating the *TextBlock* type.

The implicit style is very powerful. Any *TextBlock* further down the visual tree that does not have its *Style* property set instead gets the implicit style. If you have a page full of *TextBlock* elements and you suddenly decide that you want them all to be styled the same way, the implicit style makes it very easy. Notice that none of these *TextBlock* elements have their *Style* properties set:

Project: ImplicitStyle | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">

    <TextBlock Text="{StaticResource appName}"
               FontFamily="Portable User Interface"
               FontSize="48"
               Foreground="{StaticResource ApplicationTextBrush}"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />

    <TextBlock Text="Top Text"
               HorizontalAlignment="Center"
               VerticalAlignment="Top" />

    <TextBlock Text="Left Text"
               HorizontalAlignment="Left"
               VerticalAlignment="Center" />

    <TextBlock Text="Right Text"
               HorizontalAlignment="Right"
               VerticalAlignment="Center" />

    <TextBlock Text="Bottom Text"
               HorizontalAlignment="Center"
               VerticalAlignment="Bottom" />
</Grid>
```

Although I obviously intended for the implicit style to apply to most of the *TextBlock* elements on the page, I didn't want it to apply to the first one, which appears in the center. If you want certain elements on the page to *not* have this implicit style, you must give those elements an explicit style or provide local settings that override the properties included in the *Style* object, or set the *Style* property to *null*. (I'll show you how to do that in XAML shortly.) In this example, I've overridden the implicit style in the first *TextBlock* by giving it the default *FontFamily* name, an explicit *FontSize*, and a *Foreground* based on a predefined resource.

You cannot derive a style from an implicit style. However, an implicit style can be based on a nonimplicit style. Simply provide *TargetType* and *BasedOn* attributes and leave out the *x:Key*.

The implicit style is very powerful, but remember: With great power comes…and you know the rest. In a large application, styles can be defined all over the place and visual trees can extend over multiple XAML files. It sometimes happens that a style is implicitly applied to an element, but it's very hard to determine where that style is actually defined!

At this point, you can begin using (or at least start looking at) the *TextBlock* styles defined in the StandardStyles.xaml file. These are called BasicTextStyle, BaselineTextStyle, HeaderTextStyle, SubheaderTextStyle, TitleTextStyle, ItemTextStyle, BodyTextStyle, and CaptionTextStyle, and obviously they are for more extensive text layout than I've been doing here.

A Taste of Data Binding

Another way to share objects in a XAML file is through data bindings. Basically, a data binding establishes a connection between two properties of different objects. As you'll see in Chapter 6, "WinRT and MVVM," data bindings find their greatest application in linking visual elements on a page with data sources, and they form a crucial part of implementing the popular Model-View-View Model (MVVM) architectural pattern. In MVVM, the target of the binding is a visual element in the View, and the source of the binding is a property in a corresponding View Model.

You can also use data bindings to link properties of two elements. Like *StaticResource*, *Binding* is generally expressed as a markup extension, which means that it appears between a pair of curly braces. However, *Binding* is more elaborate than *StaticResource* and can alternatively be expressed in property-element syntax.

Here's the *Resources* section from the SharedBrushWithBinding project:

Project: SharedBrushWithBinding | File: BlankPage.xaml (excerpt)

```
<Page.Resources>
    <x:String x:Key="appName">Shared Brush with Binding</x:String>

    <Style TargetType="TextBlock">
        <Setter Property="FontFamily" Value="Times New Roman" />
        <Setter Property="FontSize" Value="96" />
    </Style>
</Page.Resources>
```

The implicit style for the *TextBlock* no longer has a *Foreground* property. The *LinearGradientBrush* is defined on the first of the four *TextBlock* elements that use that brush, and the subsequent *TextBlock* elements reference that same brush through a binding:

Project: SharedBrushWithBinding | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <TextBlock Text="{StaticResource appName}"
```

```
            FontFamily="Portable User Interface"
            FontSize="48"
            HorizontalAlignment="Center"
            VerticalAlignment="Center" />

    <TextBlock Name="topTextBlock"
            Text="Top Text"
            HorizontalAlignment="Center"
            VerticalAlignment="Top">
        <TextBlock.Foreground>
            <LinearGradientBrush>
                <GradientStop Offset="0" Color="Red" />
                <GradientStop Offset="0.17" Color="Orange" />
                <GradientStop Offset="0.33" Color="Yellow" />
                <GradientStop Offset="0.5" Color="Green" />
                <GradientStop Offset="0.67" Color="Blue" />
                <GradientStop Offset="0.83" Color="Indigo" />
                <GradientStop Offset="1" Color="Violet" />
            </LinearGradientBrush>
        </TextBlock.Foreground>
    </TextBlock>

    <TextBlock Text="Left Text"
            HorizontalAlignment="Left"
            VerticalAlignment="Center"
            Foreground="{Binding ElementName=topTextBlock, Path=Foreground}" />

    <TextBlock Text="Right Text"
            HorizontalAlignment="Right"
            VerticalAlignment="Center"
            Foreground="{Binding ElementName=topTextBlock, Path=Foreground}" />

    <TextBlock Text="Bottom Text"
            HorizontalAlignment="Center"
            VerticalAlignment="Bottom">
        <TextBlock.Foreground>
            <Binding ElementName="topTextBlock" Path="Foreground" />
        </TextBlock.Foreground>
    </TextBlock>
</Grid>
```

Data bindings are said to have a *source* and a *target*. The target is always the property on which the binding is set, and the source is the property the binding references. The *TextBlock* with the name "topTextBlock" is considered the source of these data bindings; the three *TextBlock* elements that share the *Foreground* property are targets. Two of these targets show the more standard way of expressing the *Binding* object as a XAML markup extension:

```
Foreground="{Binding ElementName=topTextBlock, Path=Foreground}"
```

XAML markup extensions always appear in curly braces. In the markup extension for *Binding*, a couple properties and values usually need to be set. These properties are separated by commas. The *ElementName* property indicates the name of the element on which the desired property has been set;

the *Path* provides the name of the property.

When I'm typing a *Binding* markup extension, I always want to put quotation marks around the property values, but that's wrong. Quotation marks do not appear in a binding expression.

The final *TextBlock* shows the *Binding* expressed in less common property-element syntax:

```
<TextBlock.Foreground>
    <Binding ElementName="topTextBlock" Path="Foreground" />
</TextBlock.Foreground>
```

With this syntax, the quotation marks around the element name and path are required.

You can also create a *Binding* object in code and set it on a target property by using the *SetBinding* method defined by *FrameworkElement*. When doing this, you'll discover that the binding target must be a dependency property.

The *Path* property of the *Binding* class is called *Path* because it can actually be several property names separated by periods. For example, replace one of the *Text* settings in this project with the following:

```
Text="{Binding ElementName=topTextBlock, Path=FontFamily.Source}"
```

The first part of the *Path* indicates that we want something from the *FontFamily* property. That property is set to an object of type *FontFamily*, which has a property named *Source* indicating the font family name. The text displayed by this *TextBlock* is therefore "Times New Roman."

Try this on any *TextBlock* in this project:

```
Text="{Binding RelativeSource={RelativeSource Self}, Path=FontSize}"
```

That's a *RelativeSource* markup extension inside a *Binding* markup extension, and you use it to reference a property of the same element on which the binding is set.

With *StaticResource*, *Binding*, and *RelativeSource*, you've now seen 60 percent of the XAML markup extensions supported by the Windows Runtime. The *TemplateBinding* markup extension won't turn up until a later chapter.

The remaining markup extension is not used very often, but when you need it, it's indispensable. Suppose you've defined an implicit style for the *Grid* that includes a *Background* property, and it does exactly what you want except for one *Grid* where you want the *Background* property to be its default value of *null*. How do you specify *null* in markup? Like so:

```
Background="{x:Null}"
```

Or suppose you've defined an implicit style and there's one element where you don't want any part of the style to apply. Inhibit the implicit style like so:

```
Style="{x:Null}"
```

You have now seen nearly all the elements and attributes that appear with an "x" prefix in Windows

Runtime XAML files. These are the data types *x:Boolean*, *x:Double*, *x:Int32*, *x:String*, as well as the *x:Class*, *x:Name*, and *x:Key* attributes and the *x:Null* markup extension. The only one I haven't mentioned is *x:Uid*, which must be set to application-wide unique strings that reference resources for internationalization purposes. That's for a later chapter.

Chapter 3

Basic Event Handling

The previous chapters have demonstrated how you can instantiate and initialize elements and other objects in either XAML or code. The most common procedure is to use XAML to define the initial layout and appearance of elements on a page but then to change properties of these elements from code as the program is running.

As you've seen, assigning a *Name* or *x:Name* to an element in XAML causes a field to be defined in the page class that gives the code-behind file easy access to that element. This is one of the two major ways that code and XAML interact. The second is through events. An event is a general-purpose mechanism that allows one object to communicate something of interest to other objects. The event is said to be "fired" by the first object and "handled" by the other. In the Windows Runtime, one important application of events is to signal the presence of user input from touch, the mouse, a stylus, or the keyboard.

Following initialization, a Windows Runtime program generally sits dormant in memory waiting for something interesting to happen. Almost everything the program does thereafter is in response to an event, so the job of event handling is one that will occupy much of the rest of this book.

The *Tapped* Event

The *UIElement* class defines all the basic user-input events. These include eight events beginning with the word *Pointer* that consolidate input from touch, the mouse, and the stylus; five events beginning with the word *Manipulation* that combine input from multiple fingers; two *Key* events for keyboard input; as well as higher level events named *Tapped*, *DoubleTapped*, *RightTapped*, and *Holding*. (No, the *RightTapped* event is *not* generated by a finger on your right hand; it's mostly used to register right-button clicks on the mouse, but you can simulate a right tap with touch by holding your finger down for a moment and then lifting, a gesture that also generates *Holding* events. It's the application's responsibility to determine how it wants to handle these.)

A complete exploration of these user-input events awaits us in a future chapter. For now, let's focus on *Tapped* as a simple representative event. An element that derives from *UIElement* fires a *Tapped* event to indicate that the user has briefly touched the element with a finger, or clicked it with the mouse, or dinged it with the stylus. To qualify as a *Tapped* event, the finger (or mouse or stylus) cannot move very much and must be released in a short period of time.

All the user-input events have a similar pattern. *UIElement* defines the *Tapped* event like so:

```
public event TappedEventHandler Tapped;
```

The *TappedEventHandler* is defined in the *Windows.UI.Xaml.Input* namespace. It's a delegate type that defines the signature of the event handler:

```
public delegate void TappedEventHandler(object sender, TappedRoutedEventArgs e);
```

In the event handler, the first argument indicates the source of the event (which is always an instance of a class that derives from *UIElement*) and the second argument provides properties and methods specific to the *Tapped* event.

The XAML file for the TapTextBlock program defines a *TextBlock* with a *Name* attribute as well as a handler for the *Tapped* event:

Project: TapTextBlock | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <TextBlock Name="txtblk"
               Text="Tap Text!"
               FontSize="96"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               Tapped="txtblk_Tapped_1" />
</Grid>
```

As you type *TextBlock* attributes in XAML, IntelliSense suggests events as well as properties. These are distinguished with little icons: a wrench for properties and a lightning bolt for events. (You'll also see a few with pairs of curly brackets. These are attached properties that I'll describe in Chapter 4, "Presentation with Panels.") If you allow it, IntelliSense also suggests a name for the event handler, and I let it choose this one. Based solely on the XAML syntax, you really can't tell which attributes are properties and which are events.

The actual event handler is implemented in the code-behind file. If you allow Visual Studio to select a handler name for you, you'll discover that Visual Studio also creates a skeleton event handler in the BlankPage.xaml.cs file:

```
private void txtblk_Tapped_1(object sender, TappedRoutedEventArgs e)
{

}
```

This is the method that is called when the user taps the *TextBlock*. In future projects, I'll change the names of event handlers to make them more to my liking. I'll remove the *private* keyword (because that's the default), I'll change the name to eliminate underscores and preface it with the word *On* (for example *OnTextBlockTapped*), and I'll change the argument named *e* to *args*. In theory, you should be able to rename the method in the code file and then click a little global-rename icon to rename the method in the XAML file as well, but that doesn't seem to work in the current version of Visual Studio.

For this sample program, I decided I want to respond to the tap by setting the *TextBlock* to a random color. In preparation for that job, I defined fields for a *Random* object and a *byte* array for the red, green, and blue bytes:

Project: TapTextBlock | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    Random rand = new Random();
    byte[] rgb = new byte[3];

    public BlankPage()
    {
        this.InitializeComponent();
    }

    private void txtblk_Tapped_1(object sender, TappedRoutedEventArgs e)
    {
        rand.NextBytes(rgb);
        Color clr = Color.FromArgb(255, rgb[0], rgb[1], rgb[2]);
        txtblk.Foreground = new SolidColorBrush(clr);
    }
}
```

I've removed the *OnNavigatedTo* method, because it's not being used here. In the *Tapped* event handler, the *NextBytes* method of the *Random* object obtains three random bytes, and these are used to construct a *Color* value with the static *Color.FromArgb* method. The handler finishes by setting the *Foreground* property of the *TextBlock* to a *SolidColorBrush* based on that *Color* value.

When you run this program, you can tap the *TextBlock* with a finger, mouse, or stylus and it will change to a random color. If you tap on an area of the screen outside the *TextBlock*, nothing happens. If you're using a mouse or stylus, you might notice that you don't need to tap the actual strokes that comprise the letters. You can tap between and inside those strokes, and the *TextBlock* will still respond. It's as if the *TextBlock* has an invisible background that encompasses the full height of the font including diacritical marks and descenders, and that's precisely the case.

If you look inside the BlankPage.g.cs file generated by Visual Studio, you'll see a *Connect* method containing the code that attaches the event handler to the *Tapped* event of the *TextBlock*. You can do this yourself in code. Try eliminating the *Tapped* handler assigned in the XAML file and instead attach an event handler in the constructor of the code-behind file:

```
public BlankPage()
{
    this.InitializeComponent();
    txtblk.Tapped += txtblk_Tapped_1;
}
```

No real difference.

Several properties of *TextBlock* need to be set properly for the *Tapped* event to work. The *IsHitTestVisible* and *IsTapEnabled* properties must both be set to their default values of *true*. The *Visibility* property must be set to its default value of *Visibility.Visible*. If set to *Visibility.Collapsed*, the *TextBlock* will not be visible at all and will not respond to user input.

The first argument to the *txtblk_Tapped_1* event handler is the element that sent the event, in this

case the *TextBlock*. The second argument provides information about this particular event, including the coordinate point at which the tap occurred, and whether the tap came from a finger, mouse, or stylus. This information will be explored in more detail in future chapters.

Routed Event Handling

Because the first argument to the *Tapped* event handler is the element that generates the event, you don't need to give the *TextBlock* a name to access it from within the event handler. You can simply cast the *sender* argument to an object of type *TextBlock*. This is particularly useful for sharing an event handler among multiple elements, and I've done precisely that in the RoutedEvents0 project.

RoutedEvents0 is the first of several projects that demonstrate the concept of *routed event handling*, which is an important feature of the Windows Runtime. But this particular program doesn't show any features particular to routed events. Hence the suffix of zero. For this project I created the *Tapped* handler first with the proper signature and my preferred name:

Project: RoutedEvents0 | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    Random rand = new Random();
    byte[] rgb = new byte[3];

    public BlankPage()
    {
        this.InitializeComponent();
    }

    void OnTextBlockTapped(object sender, TappedRoutedEventArgs args)
    {
        TextBlock txtblk = sender as TextBlock;
        rand.NextBytes(rgb);
        Color clr = Color.FromArgb(255, rgb[0], rgb[1], rgb[2]);
        txtblk.Foreground = new SolidColorBrush(clr);
    }
}
```

Notice the first line of the event handler casts the *sender* argument to *TextBlock*.

Because this event handler already exists in the code-behind file, Visual Studio suggests that name when you type the name of the event in the XAML file. This was handy because I added nine *TextBlock* elements to the *Grid*:

Project: RoutedEvents0 | File: BlankPage.xaml (excerpt)

```
<Page
    x:Class="RoutedEvents0.BlankPage"
    …
    FontSize="48">
```

```
    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <TextBlock Text="Left / Top"
                   HorizontalAlignment="Left"
                   VerticalAlignment="Top"
                   Tapped="OnTextBlockTapped" />

        …

        <TextBlock Text="Right / Bottom"
                   HorizontalAlignment="Right"
                   VerticalAlignment="Bottom"
                   Tapped="OnTextBlockTapped" />
    </Grid>
</Page>
```

I'm sure you don't need to see them all to get the general idea. Notice that *FontSize* is set for the *Page* so that it is inherited by all the *TextBlock* elements. When you run the program, you can tap the individual elements and each one changes its color independently of the others:

If you tap anywhere between the elements, nothing happens.

You might consider it a nuisance to set the same event handler on nine different elements in the XAML file. If so, you'll probably appreciate the following variation to the program. The RoutedEvents1 program uses *routed input handling*, a term used to describe how input events such as *Tapped* are fired by the element on which the event occurs but the events are then routed up the visual tree. Rather than set a *Tapped* handler for the individual *TextBlock* elements, you can instead set it on the parent of one of these elements (for example, the *Grid*). Here's an excerpt from the XAML file for the RoutedEvents1 program:

Project: RoutedEvents1 | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}"
```

```
      Tapped="OnGridTapped">

    <TextBlock Text="Left / Top"
               HorizontalAlignment="Left"
               VerticalAlignment="Top" />

    …

    <TextBlock Text="Right / Bottom"
               HorizontalAlignment="Right"
               VerticalAlignment="Bottom" />
</Grid>
```

In the process of moving the *Tapped* handler from the individual *TextBlock* elements to the *Grid*, I've also renamed it to more accurately describe the source of the event.

The event handler must also be modified. The previous *Tapped* handler cast the *sender* argument to a *TextBlock*. It could perform this cast with confidence because the event handler was set only on elements of type *TextBlock*. However, when the event handler is set on the *Grid* as it is here, the *sender* argument to the event handler will be the *Grid*. How can we determine which *TextBlock* was tapped?

Easy: the *TappedRoutedEventArgs* class—an instance of which appears as the second argument to the event handler—has a property named *OriginalSource*, and that indicates the source of the event. In this example, *OriginalSource* can be either a *TextBlock* (if you tap the text) or the *Grid* (if you tap between the text), so the new event handler must perform a check before casting:

Project: RoutedEvents1 | File: BlankPage.xaml.cs (excerpt)

```
void OnGridTapped(object sender, TappedRoutedEventArgs args)
{
    if (args.OriginalSource is TextBlock)
    {
        TextBlock txtblk = args.OriginalSource as TextBlock;
        rand.NextBytes(rgb);
        Color clr = Color.FromArgb(255, rgb[0], rgb[1], rgb[2]);
        txtblk.Foreground = new SolidColorBrush(clr);
    }
}
```

TappedRoutedEventArgs derives from *RoutedEventArgs*, which defines *OriginalSource* and no other properties. Obviously, the *OriginalSource* property is a central concept of routed event handling. The property allows ancestor elements to process events that originate with their descendants and to know the source of these events.

Alternatively, you can set the *Tapped* handler on the *Page* rather than the *Grid*. But with the *Page* there's an easier way. I mentioned earlier that *UIElement* defines all the user-input events. These events are inherited by all descendant classes, but the *Control* class adds its own event interface consisting of a whole collection of virtual methods corresponding to these events. For the *Tapped* event defined by *UIElement*, the *Control* class defines a virtual method named *OnTapped*. These virtual methods always begin with the word *On* followed by the name of the event, so they are sometimes referred to as "*On*

methods." *Page* derives from *Control* via *UserControl*, so these methods are inherited on the *Page* class.

Here's an excerpt from the XAML file for RoutedEvents2 showing that the XAML file defines no event handlers:

Project: RoutedEvents2 | File: BlankPage.xaml (excerpt)

```
<Page
    x:Class="RoutedEvents2.BlankPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:RoutedEvents2"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    FontSize="48">

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <TextBlock Text="Left / Top"
                   HorizontalAlignment="Left"
                   VerticalAlignment="Top" />

        …

        <TextBlock Text="Right / Bottom"
                   HorizontalAlignment="Right"
                   VerticalAlignment="Bottom" />
    </Grid>
</Page>
```

Instead, the code-behind file has an override of the *OnTapped* method:

Project: RoutedEvents2 | File: BlankPage.xaml.cs (excerpt)

```
protected override void OnTapped(TappedRoutedEventArgs args)
{
    if (args.OriginalSource is TextBlock)
    {
        TextBlock txtblk = args.OriginalSource as TextBlock;
        rand.NextBytes(rgb);
        Color clr = Color.FromArgb(255, rgb[0], rgb[1], rgb[2]);
        txtblk.Foreground = new SolidColorBrush(clr);
    }
    base.OnTapped(args);
}
```

When you're typing in Visual Studio and you want to override a virtual method like *OnTapped*, simply type the keyword *override* and press the space bar, and Visual Studio will provide a list of all the virtual methods defined for that class. When you select one, Visual Studio creates a skeleton method with a call to the base method. A call to the base method isn't really required here, but including it is a good habit to develop when overriding virtual methods

The *On* methods are basically the same as the event handlers, but they have no *sender* argument because it's no longer needed. In this context, *sender* would be the same as *this*, the instance of the

Page that is processing the event.

The next project is RoutedEvents3. I decided to give the *Grid* a random background color if that's the element being tapped. The XAML file looks the same, but the revised *OnTapped* method looks like this:

Project: RoutedEvents3 | File: BlankPage.xaml.cs (excerpt)

```
protected override void OnTapped(TappedRoutedEventArgs args)
{
    rand.NextBytes(rgb);
    Color clr = Color.FromArgb(255, rgb[0], rgb[1], rgb[2]);
    SolidColorBrush brush = new SolidColorBrush(clr);

    if (args.OriginalSource is TextBlock)
        (args.OriginalSource as TextBlock).Foreground = brush;

    else if (args.OriginalSource is Grid)
        (args.OriginalSource as Grid).Background = brush;

    base.OnTapped(args);
}
```

Now when you tap a *TextBlock* element, it changes color, but when you tap anywhere else on the screen, the *Grid* changes color.

Now suppose for one reason or another, you decide you want to go back to the original scheme of explicitly defining an event handler separately for each *TextBlock* element to change the text colors, but you also want to retain the *OnTapped* override for changing the *Grid* background color. In the RoutedEvents4 project, the XAML file has the *Tapped* events restored for *TextBlock* elements and the *Grid* has been given a name:

Project: RoutedEvents4 | File: BlankPage.xaml (excerpt)

```
<Grid Name="contentGrid"
      Background="{StaticResource ApplicationPageBackgroundBrush}">

    <TextBlock Text="Left / Top"
               HorizontalAlignment="Left"
               VerticalAlignment="Top"
               Tapped="OnTextBlockTapped" />

    …

    <TextBlock Text="Right / Bottom"
               HorizontalAlignment="Right"
               VerticalAlignment="Bottom"
               Tapped="OnTextBlockTapped" />
</Grid>
```

One advantage is that the methods to set the *TextBlock* and *Grid* colors are now separate and distinct, so there's no need for *if-else* blocks. The *Tapped* handler for the *TextBlock* elements can cast the *sender* argument with impunity, and the *OnTapped* override can simply access the *Grid* by name:

Project: RoutedEvents4 | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    Random rand = new Random();
    byte[] rgb = new byte[3];

    public BlankPage()
    {
        this.InitializeComponent();
    }

    void OnTextBlockTapped(object sender, TappedRoutedEventArgs args)
    {
        TextBlock txtblk = sender as TextBlock;
        txtblk.Foreground = GetRandomBrush();
    }

    protected override void OnTapped(TappedRoutedEventArgs args)
    {
        contentGrid.Background = GetRandomBrush();
        base.OnTapped(args);
    }

    Brush GetRandomBrush()
    {
        rand.NextBytes(rgb);
        Color clr = Color.FromArgb(255, rgb[0], rgb[1], rgb[2]);
        return new SolidColorBrush(clr);
    }
}
```

However, the code might not do exactly what you want. When you tap a *TextBlock*, not only does the *TextBlock* change color, but the event continues to go up the visual tree where it's processed by the *OnTapped* override, and the *Grid* changes color as well! If that's what you want, you're in luck. If not, then I'm sure you'll be interested to know that the *TappedRoutedEventArgs* has a property specifically to prevent this. If the *OnTextBlockTapped* handler sets the *Handled* property of the event arguments to *true*, the event is effectively inhibited from further processing higher in the visual tree.

This is demonstrated in the RoutedEvents5 project, which is the same as RoutedEvents4 except for a single statement in the *OnTextBlockTapped* method:

Project: RoutedEvents5 | File: BlankPage.xaml.cs (excerpt)

```
void OnTextBlockTapped(object sender, TappedRoutedEventArgs args)
{
    TextBlock txtblk = sender as TextBlock;
    txtblk.Foreground = GetRandomBrush();
    args.Handled = true;
}
```

Overriding the *Handled* Setting

You've just seen that when an element handles an event such as *Tapped* and concludes its event processing by setting the *Handled* property of the event arguments to *true*, the routing of the event effectively stops. The event isn't visible to elements higher in the visual tree.

In some cases, this behavior might be undesirable. Suppose you're working with an element that sets the *Handled* property to *true* in its event handler but you still want to see that event higher in the visual tree. One solution is to simply change the code, but that option might not be available. The element might be implemented in a dynamic-link library, and you might not have access to the source code.

In RoutedEvents6, the XAML file is the same as in RoutedEvents5: each *TextBlock* has a handler set for its *Tapped* event. The *Tapped* handler sets the *Handled* property to *true*. The class also defines a separate *OnPageTapped* handler that sets the background color of the *Grid*:

Project: RoutedEvents6 | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    Random rand = new Random();
    byte[] rgb = new byte[3];

    public BlankPage()
    {
        this.InitializeComponent();

        this.AddHandler(UIElement.TappedEvent,
                        new TappedEventHandler(OnPageTapped),
                        true);
    }

    void OnTextBlockTapped(object sender, TappedRoutedEventArgs args)
    {
        TextBlock txtblk = sender as TextBlock;
        txtblk.Foreground = GetRandomBrush();
        args.Handled = true;
    }

    void OnPageTapped(object sender, TappedRoutedEventArgs args)
    {
        contentGrid.Background = GetRandomBrush();
    }

    Brush GetRandomBrush()
    {
        rand.NextBytes(rgb);
        Color clr = Color.FromArgb(255, rgb[0], rgb[1], rgb[2]);
        return new SolidColorBrush(clr);
    }
}
```

Look at the interesting way that the constructor sets a *Tapped* handler for the *Page*. Normally it would attach the event handler like so:

```
this.Tapped += OnPageTapped;
```

In that case the *OnPageTapped* handler would not get a *Tapped* event originating with the *TextBlock* because the *TextBlock* handler sets *Handled* to true. Instead, it attaches the handler with a method named *AddHandler*:

```
this.AddHandler(UIElement.TappedEvent,
                new TappedEventHandler(OnPageTapped),
                true);
```

AddHandler is defined by *UIElement*, which also defines the static *UIElement.TappedEvent property*. This property is of type *RoutedEvent*.

Just as a property like *FontSize* is backed by a static property named *FontSizeProperty* of type *DependencyProperty*, a routed event such as *Tapped* is backed by a static property named *TappedEvent* of type *RoutedEvent*. *RoutedEvent* defines nothing public on its own; it mainly exists to allow an event to be referenced in code without requiring an instance of an element.

The *AddHandler* method attaches a handler to that event. The second argument of *AddHandler* is defined as just an *object*, so creating a delegate object is required to reference the event handler. And here's the magic: set the last argument to *true* if you want this handler to also receive routed events that have been flagged as *Handled*.

The *AddHandler* method isn't used often, but when you need it, it can be very useful.

Input, Alignment, and Backgrounds

I have just one more, very short program in the RoutedEvents series to make a couple important points about input events.

The XAML file for RoutedEvents7 has just one *TextBlock* and no event handlers defined:

Project: RoutedEvents7 | File: BlankPage.xaml (excerpt)

```
<Page …
    FontSize="48">

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
         <TextBlock Text="Hello, Windows 8!"
                    Foreground="Red" />
    </Grid>
</Page>
```

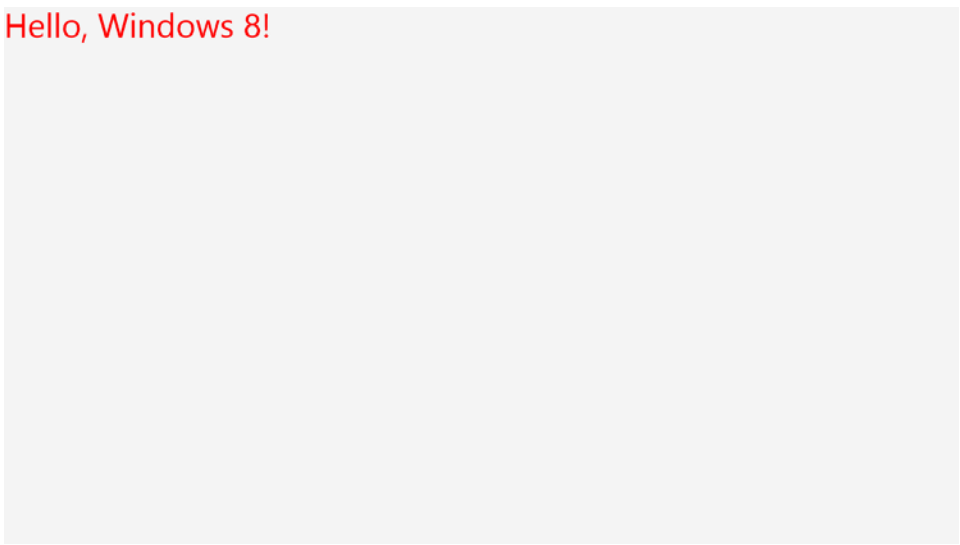
The absence of *HorizontalAlignment* and *VerticalAlignment* settings on the *TextBlock* cause it to appear in the upper-left corner of the *Grid*.

Like RoutedEvents3, the code-behind file contains separate processing for an event originating from the *TextBlock* and an event coming from the *Grid*:

Project: RoutedEvents7 | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    Random rand = new Random();
    byte[] rgb = new byte[3];

    public BlankPage()
    {
        this.InitializeComponent();
    }

    protected override void OnTapped(TappedRoutedEventArgs args)
    {
        rand.NextBytes(rgb);
        Color clr = Color.FromArgb(255, rgb[0], rgb[1], rgb[2]);
        SolidColorBrush brush = new SolidColorBrush(clr);

        if (args.OriginalSource is TextBlock)
            (args.OriginalSource as TextBlock).Foreground = brush;

        else if (args.OriginalSource is Grid)
            (args.OriginalSource as Grid).Background = brush;

        base.OnTapped(args);
    }
}
```

Here it is:



As you tap the *TextBlock*, it changes to a random color like normal, but when you tap outside the

TextBlock, the *Grid* doesn't change color like it did earlier. Instead, the *TextBlock* changes color! It's as if...yes, it's as if the *TextBlock* is now occupying the entire page and snagging all the *Tapped* events for itself.

And that's precisely the case. This *TextBlock* has default values of *HorizontalAlignment* and *VerticalAlignment*, but those default values are not *Left* and *Top* like the visuals may suggest. The default values are named *Stretch*, and that means that the *TextBlock* is stretched to the size of its parent, the *Grid*. It's hard to tell because the text still has a 48-pixel font, but the *TextBlock* has a transparent background that now fills the entire page.

In fact, throughout the Windows Runtime, all elements have default *HorizontalAlignment* and *VerticalAlignment* values of *Stretch*, and it's an important part of the Windows Runtime layout system. More details are coming in Chapter 4.

Let's put *HorizontalAlignment* and *VerticalAlignment* values in this *TextBlock*:

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
     <TextBlock Text="Hello, Windows 8!"
                HorizontalAlignment="Left"
                VerticalAlignment="Top"
                Foreground="Red" />
</Grid>
```

Now the *TextBlock* is only occupying a small area in the upper-left corner of the page, and when you tap outside the *TextBlock*, the *Grid* changes color.

Now change *HorizontalAlignment* to *TextAlignment*:

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <TextBlock Text="Hello, Windows 8!"
               TextAlignment="Left"
               VerticalAlignment="Top"
               Foreground="Red" />
</Grid>
```

The program looks the same. The text is still positioned at the upper-left corner. But now when you tap to the right of the *TextBlock*, the *TextBlock* changes color rather than the *Grid*. The *TextBlock* has its default *HorizontalAlignment* property of *Stretch*, so it is now occupying the entire width of the screen, but within the total width that the *TextBlock* occupies, the text is aligned to the left.

The lesson: *HorizontalAlignment* and *TextAlignment* are not equivalent, although they might seem to be if you judge solely from the visuals.

Now try another experiment by restoring the *HorizontalAlignment* setting and removing the *Background* property of the *Grid*:

```
<Grid>
     <TextBlock Text="Hello, Windows 8!"
                HorizontalAlignment="Left"
                VerticalAlignment="Top"
                Foreground="Red" />
```

```
</Grid>
```

With a light theme, the *Grid* has an off-white background. When the *Background* property is removed, the background of the page changes to black. But you'll also experience a change in the behavior of the program: the *TextBlock* still changes color when you tap it, but when you tap outside the *TextBlock*, the *Grid* doesn't change color at all.

The default value of the *Background* property defined by *Panel* (and inherited by *Grid*) is *null*, and with a *null* background, the *Grid* doesn't trap touch events. They just fall right through.

One way to fix this without altering the visual appearance is to give the *Grid* a *Background* property of *Transparent*:

```
<Grid Background="Transparent">
     <TextBlock Text="Hello, Windows 8!"
                HorizontalAlignment="Left"
                VerticalAlignment="Top"
                Foreground="Red" />
</Grid>
```

It looks the same as *null*, but now you'll get *Tapped* events with an *OriginalSource* of *Grid*.

The lessons here are important: Looks can be deceiving. An element with default settings of *HorizontalAlignment* and *VerticalAlignment* might look the same as one with settings of *Left* and *Top*, but it is actually occupying the entire area of its container and might block events from reaching underlying elements. A *Panel* derivative with a default *Background* property of *null* might look the same as one with a setting of *Transparent*, but it does not respond to touch events.

I can almost guarantee that sometime in the future, one of these two issues will cause a bug in one of your programs that will drive you crazy for the good part of a day, and that this will happen even after many years of working with the XAML layout system.

I speak from experience.

Size and Orientation Changes

The very first Windows program to be described in a magazine article was called WHATSIZE (all capital letters, of course), and it appeared in the December 1986 issue of *Microsoft Systems Journal*, the predecessor to *MSDN Magazine*. The program did little more than display the current size of the program's window, but as the size of the window changed, the displayed size also changed.

Obviously the original WHATSIZE program was written for the Windows APIs of that era, so it redrew the display in response to a WM_PAINT message. In the original Windows API, this message occurred whenever the contents of part of a program's window became "invalid" and needed redrawing. A program could define its window so that the entire window was invalidated whenever its size changed.

The Windows Runtime has no equivalent of the WM_PAINT message, and indeed, the entire graphics paradigm is quite different. Previous versions of Windows implemented a "direct mode" graphics system in which applications drew to the actual video memory. Of course, this occurred through a software lawyer (the Graphics Device Interface) and a device driver, but at some point in the actual drawing functions, code was writing into video display memory.

The Windows Runtime is quite different. In its public programming interface, it doesn't even have a concept of drawing or painting. Instead, a Metro style application creates elements—that is, objects instantiated from classes that derive from *FrameworkElement*—and adds them to the application's visual tree. These elements are responsible for rendering themselves. When a Metro style application wants to display text, it doesn't draw text but instead creates a *TextBlock*. When the application wants to display a bitmap, it creates an *Image* element. Instead of drawing lines and Bézier splines and ellipses, the program creates *Polyline* and *Path* elements.

The Windows Runtime implements a "retained mode" graphics system. Between your application and the video display is a composition layer on which all the rendered output is assembled before it is presented to the user. Perhaps the most important benefit of retained mode graphics is flicker-free animation, as you'll witness for yourself towards the end of this chapter and in much of the remainder of this book.

Although the graphics system in the Windows Runtime is very different from earlier versions of Windows, in another sense a Metro style application is similar to its earlier brethren. Once a program is loaded into memory and starts running, it spends most of its time generally sitting dormant in memory, waiting for something interesting to happen. These notifications take the form of events and callbacks. Often these events signal user input, but there may be other interesting activity as well. One such callback is the *OnNavigatedTo* method. In a simple single-page program, this method is called soon after the constructor returns.

Another event that might be of interest to a Metro style application—particularly one that does what the old WHATSIZE program did—is named *SizeChanged*. Here's the XAML file for the Metro Style WhatSize program. Notice that the root element defines a handler for the *SizeChanged* event:

Project: WhatSize | File: BlankPage.xaml (excerpt)

```
<Page
    x:Class="WhatSize.BlankPage"
    …
    FontSize="36"
    SizeChanged="OnPageSizeChanged">

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <TextBlock HorizontalAlignment="Center"
                   VerticalAlignment="Top">
            &#x21A4; <Run x:Name="widthText" /> pixels &#x21A6;
        </TextBlock>

        <TextBlock HorizontalAlignment="Center"
                   VerticalAlignment="Center"
```

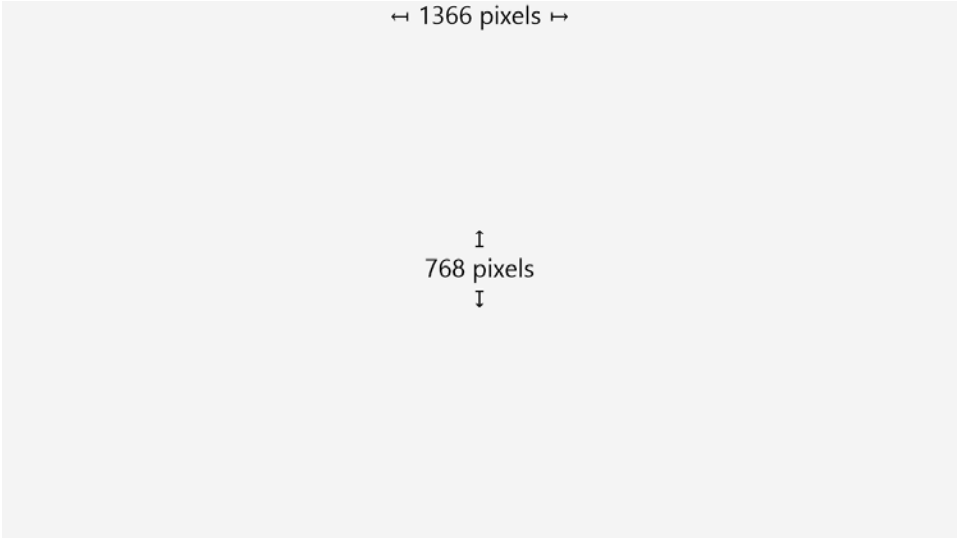
```
                TextAlignment="Center">
            &#x21A5;
            <LineBreak />
            <Run x:Name="heightText" /> pixels
            <LineBreak />
            &#x21A7;
        </TextBlock>
    </Grid>
</Page>
```

The remainder of the XAML file defines two *TextBlock* elements containing some *Run* objects surrounded by arrow characters. (You'll see what they look like soon.) It might seem excessive to set three properties to *Center* in the second *TextBlock*, but they're all necessary. The first two center the *TextBlock* in the page; setting *TextAlignment* to *Center* results in the two arrows being centered relative to the text. The two *Run* elements are given *x:Name* attributes so that the *Text* properties can be set in code. This happens in the *SizeChanged* event handler:

Project: WhatSize | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    public BlankPage()
    {
        this.InitializeComponent();
    }

    void OnPageSizeChanged(object sender, SizeChangedEventArgs args)
    {
        widthText.Text = args.NewSize.Width.ToString();
        heightText.Text = args.NewSize.Height.ToString();
    }
}
```

Very conveniently, the event arguments supply the new size in the form of a *Size* structure, and the handler simply converts the *Width* and *Height* properties to strings and sets them to the *Text* properties of the two *Run* elements:



If you're running the program on a device that responds to orientation changes, you can try flipping the screen and observe how the numbers change. You can also sweep your finger from the left of the screen to invoke the snapped views and then divide the screen between this program and another to see how the width value changes.

You don't need to set the *SizeChanged* event handler in XAML. You can set it in code, perhaps during the *Page* constructor:

```
this.SizeChanged += OnPageSizeChanged;
```

SizeChanged is defined by *FrameworkElement* and inherited by all descendent classes. Despite the fact that *SizeChangedEventArgs* derives from *RoutedEventArgs*, this is not a routed event. You can tell it's not a routed event because the *OriginalSource* property of the event arguments is always *null*; there is no *SizeChangedEvent* property; and whatever element you set this event on, that's the element's size you get. But you can set *SizeChanged* handlers on any element. Generally, the order the events are fired proceeds down the visual tree: *Page* first (in this example), and then *Grid* and *TextBlock*.

If you need the rendered size of an element other than in the context of a *SizeChanged* handler, that information is available from the *ActualWidth* and *ActualHeight* properties defined by *FrameworkElement*. Indeed, the *SizeChanged* handler in WhatSize is actually a little shorter when accessing those properties:

```
void OnPageSizeChanged(object sender, SizeChangedEventArgs args)
{
    widthText.Text = this.ActualWidth.ToString();
    heightText.Text = this.ActualHeight.ToString();
}
```

What you probably do *not* want are the *Width* and *Height* properties. Those properties are also defined by *FrameworkElement*, but they have default values of "not a number" or NaN. A program can

set *Width* and *Height* to explicit values (such as in the TextFormatting project in Chapter 2, "XAML Syntax"), but usually these properties remain at their default values and they are of no use in determining how large an element actually is. *FrameworkElement* also defines *MinWidth*, *MaxWidth*, *MinHeight*, and *MaxHeight* properties, but these aren't used very often.

If you access the *ActualWidth* and *ActualHeight* properties in the page's constructor, however, you'll find they have values of zero. Despite the fact that *InitializeComponent* has constructed the visual tree, that visual tree has not yet gone through a layout process. After the constructor finishes, the page gets several events in sequence:

- *OnNavigatedTo*
- *SizeChanged*
- *LayoutUpdated*
- *Loaded*

If the page later changes size, additional *SizeChanged* events and *LayoutUpdated* events are fired. *LayoutUpdated* can also be fired if elements are added to or removed from the visual tree or if an element is changed so as to affect layout.

If you need a place to perform initialization after initial layout when all the elements in the visual tree have nonzero sizes, the event you want is *Loaded*. It is very common for a *Page* class to attach a handler for the *Loaded* event. Generally, the *Loaded* event occurs only once during the lifetime of a *Page* object. I say "generally" because if the *Page* object is detached from its parent (a *Frame*) and reattached, the *Loaded* event will occur again. But this won't happen unless you deliberately make it happen. Also, the *Unloaded* event can let you know if the page has been detached from the visual tree.

Every *FrameworkElement* derivative has a *Loaded* event. As a visual tree is built, the *Loaded* events occur in a sequence going up the visual tree, ending with *Page*. When *Page* gets a *Loaded* event, it can assume that all its children have fired their own *Loaded* events and everything has been correctly sized.

Handling a *Loaded* event in a *Page* class is so common that some programmers perform *Loaded* processing right in the constructor using an anonymous handler:

```
public BlankPage()
{
    this.InitializeComponent();

    Loaded += (sender, args) =>
        {
            …
        };
}
```

Sometimes Metro style applications need to know when the orientation of the screen changes. In Chapter 1, "Markup and Code," I showed an InternationalHelloWorld program that looks fine in landscape mode but probably results in overlapping text if switched to portrait mode. For that reason,

the code-behind file changes the page's *FontSize* property to 24 in portrait mode:

Project: InternationalHelloWorld | File: BlankPage.xaml.cs

```
public sealed partial class BlankPage : Page
{
    public BlankPage()
    {
        this.InitializeComponent();
        SetFont();
        DisplayProperties.OrientationChanged += OnDisplayPropertiesOrientationChanged;
    }

    void OnDisplayPropertiesOrientationChanged(object sender)
    {
        SetFont();
    }

    void SetFont()
    {
        bool isLandscape =
            DisplayProperties.CurrentOrientation == DisplayOrientations.Landscape ||
            DisplayProperties.CurrentOrientation == DisplayOrientations.LandscapeFlipped;

        this.FontSize = isLandscape ? 40 : 24;
    }
}
```

The *DisplayProperties* class and *DisplayOrientations* enumeration are defined in the *Windows.Graphics.Display* namespace. *DisplayProperties.OrientationChanged* is a static event, and when that event is fired, the static *DisplayProperties.CurrentOrientation* property provides the current orientation.

Somewhat more information, including snapped states, is provided by the *ViewStateChanged* event of the *AppicationView* class in the *Windows.UI.ViewManagement* namespace, but working with this event must await a future chapter.

Bindings to *Run*?

In Chapter 2 I discussed data bindings. Data bindings can link properties of two elements so that when a source property changes, the target property also changes. Data bindings are particularly satisfying when they eliminate the need for event handlers.

Is it possible to rewrite WhatSize to use data bindings rather than a *SizeChanged* handler? It's worth a try.

In the WhatSize project, remove the *OnPageSizeChanged* handler from the BlankPage.xaml.cs file (or just comment it out if you don't want to do *too* much damage to the file). In the root tag of the BlankPage.xaml file, remove the *SizeChanged* attribute and give the *Page* a name of "page." Then set

Binding markup extensions on the two *Run* objects referencing the *ActualWidth* and *ActualHeight* properties of the page:

```
<Page …
    FontSize="36"
    Name="page">

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <TextBlock HorizontalAlignment="Center"
                   VerticalAlignment="Top">
            &#x21A4;
            <Run Text="{Binding ElementName=page, Path=ActualWidth}" />
            pixels &#x21A6;
        </TextBlock>

        <TextBlock HorizontalAlignment="Center"
                   VerticalAlignment="Center"
                   TextAlignment="Center">
            &#x21A5;
            <LineBreak />
            <Run Text="{Binding ElementName=page, Path=ActualHeight}" /> pixels
            <LineBreak />
            &#x21A7;
        </TextBlock>
    </Grid>
</Page>
```

The program compiles fine, and it runs smoothly without any run-time exceptions. The only problem is: where the numbers should appear is nothing.

This is likely to seem odd, particularly when you set the same bindings on the *Text* property of *TextBlock* instead of *Run*:

```
<Page …
    FontSize="36"
    Name="page">

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <TextBlock HorizontalAlignment="Center"
                   VerticalAlignment="Top"
                   Text="{Binding ElementName=page, Path=ActualWidth}" />

        <TextBlock HorizontalAlignment="Center"
                   VerticalAlignment="Center"
                   TextAlignment="Center"
                   Text="{Binding ElementName=page, Path=ActualHeight}" />
    </Grid>
</Page>
```

This works. It works so well that the size is displayed to the nearest millionth of a pixel:

As you change the orientation or size of the page, the numbers are updated. This is what makes data bindings so great. Internally, a data binding is notified when a source property changes so that it can change the target property, but the application source code appears to have no event handlers and no moving parts.

Unfortunately, by giving up on the bindings to *Run* we've also lost the informative arrows. So why do the data bindings work on the *Text* property of *TextBlock* but not on the *Text* property of *Run*?

It's very simple. The target of a data binding must be a dependency property. This fact is obvious when you define a data binding in code by using the *SetBinding* method. That's the difference: The *Text* property of *TextBlock* is backed by the *TextProperty* dependency property, but the *Text* property of *Run* is not. It's a plain old property that cannot serve as a target for a data binding. The XAML parser probably shouldn't allow a binding to be set on the *Text* property of *Run*, but it does.

In Chapter 4 I'll show you how to use a *StackPanel* to get the arrows back in a version of WhatSize that uses data bindings.

Timers and Animation

Sometimes a Metro style application needs to receive periodic events at a fixed interval. A clock application, for example, probably needs to update its display every second. The ideal class for this job is *DispatcherTimer*. Set a timer interval, set a handler for the *Tick* event, and go.

Here's the XAML file for a digital clock application. It's just a big *TextBlock*:

Project: DigitalClock | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <TextBlock Name="txtblk"
```

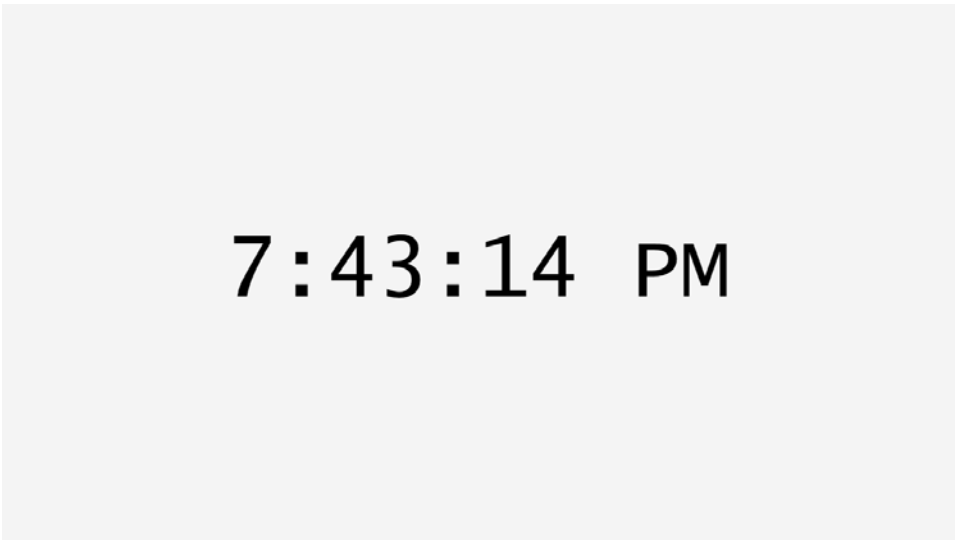
```
                FontFamily="Lucida Console"
                FontSize="120"
                HorizontalAlignment="Center"
                VerticalAlignment="Center" />
</Grid>
```

The code-behind file creates the *DispatcherTimer* with a 1-second interval and sets the *Text* property of the *TextBlock* in the event handler:

Project: DigitalClock | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    public BlankPage()
    {
        this.InitializeComponent();

        DispatcherTimer timer = new DispatcherTimer();
        timer.Interval = TimeSpan.FromSeconds(1);
        timer.Tick += OnTimerTick;
        timer.Start();
    }

    void OnTimerTick(object sender, object e)
    {
        txtblk.Text = DateTime.Now.ToString("h:mm:ss tt");
    }
}
```

And here it is:



Calls to the *Tick* handler occur in the same execution thread as the rest of the user interface, so if the program is busy doing something in that thread, the calls won't interrupt that work and might become somewhat irregular and even skip a few beats. In a multipage application, you might want to

start the timer in the *OnNavigatedTo* override and stop it in *OnNavigatedFrom* to avoid the program wasting time doing work when the page is not visible.

This is a good illustration of the difference in how a desktop Windows application and a Metro style application updates the video display. Both types of applications use a timer for implementing a clock, but rather than drawing and redrawing text every second by invalidating the contents of the window, the Metro style application changes the visual appearance of an existing element simply by changing one of its properties.

You can set the *DispatcherTimer* for an interval as low as you want, but you're not going to get calls to the *Tick* handler faster than the frame rate of the video display, which is probably 60 Hz or about a 17-millisecond period. Of course, it doesn't make sense to update the video display faster than the frame rate. Updating the display precisely at the frame rate gives you as smooth an animation as possible. If you want to perform an animation in this way, don't use *DispatcherTimer*. A better choice is the static *CompositionTarget.Rendering* event, which is specifically designed to be called prior to a screen refresh.

Even better than *CompositionTarget.Rendering* are all the animation classes provided as part of the Windows Runtime. These classes let you define animations in XAML or code, they have lots of options, and some of them are performed in background threads.

But until I cover the animation classes—and perhaps even after I do—the *CompositionTarget.Rendering* event is well suited for performing animations. These are sometimes called "manual" animations because the program itself has to carry out some calculations based on elapsed time.

Here's a little project called ExpandingText that changes the *FontSize* of a *TextBlock* in the *CompositionTarget.Rendering* event handler, making the text larger and smaller. The XAML file simply instantiates a *TextBlock*:

Project: ExpandingText | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <TextBlock Name="txtblk"
               Text="Hello, Windows 8!"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

In the code-behind file, the constructor starts a *CompositionTarget.Rendering* event simply by setting an event handler. The second argument to that handler is defined as type *object*, but it is actually of type *RenderingEventArgs*, which has a property named *RenderingTime* of type *TimeSpan*, giving you an elapsed time since the app was started:

Project: ExpandingText | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    public BlankPage()
    {
```

```
        this.InitializeComponent();
        CompositionTarget.Rendering += OnCompositionTargetRendering;
    }

    void OnCompositionTargetRendering(object sender, object args)
    {
        RenderingEventArgs renderArgs = args as RenderingEventArgs;
        double t = (0.25 * renderArgs.RenderingTime.TotalSeconds) % 1;
        double scale = t < 0.5 ? 2 * t : 2 - 2 * t;
        txtblk.FontSize = 1 + scale * 143;
    }
}
```

I've attempted to generalize this code slightly. The calculation of *t* causes it to repeatedly increase from 0 to 1 over the course of 4 seconds. During those same 4 seconds, the value of *scale* goes from 0 to 1 and back to 0, so *FontSize* ranges from 1 to 144 and back to 1. (The code ensures that the *FontSize* is never set to zero, which would raise an exception.) When you run this program, you might see a little jerkiness at first because fonts need to be rasterized at a bunch of different sizes. But after it settles into a rhythm, it's fairly smooth and there is definitely no flickering.

It's also possible to animate color, and I'll show you two different ways to do it. The second way is better than the first, but I want to make a point here, so here's the XAML file for the ManualBrushAnimation project:

Project: ManualBrushAnimation | File: BlankPage.xaml (excerpt)

```
<Grid Name="contentGrid">
    <TextBlock Name="txtblk"
               Text="Hello, Windows 8!"
               FontFamily="Times New Roman"
               FontSize="96"
               FontWeight="Bold"
               HorizontalAlignment="Center"
               VerticalAlignment="Center" />
</Grid>
```

Neither the *Grid* nor the *TextBlock* have explicit brushes defined. Creating those brushes based on animated colors is the job of the *CompositionTarget.Rendering* event handler:

Project: ManualBrushAnimation | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    public BlankPage()
    {
        this.InitializeComponent();
        CompositionTarget.Rendering += OnCompositionTargetRendering;
    }

    void OnCompositionTargetRendering(object sender, object args)
    {
        RenderingEventArgs renderingArgs = args as RenderingEventArgs;
        double t = (0.25 * renderingArgs.RenderingTime.TotalSeconds) % 1;
        t = t < 0.5 ? 2 * t : 2 - 2 * t;
```

```
        // Background
        byte gray = (byte)(255 * t);
        Color clr = Color.FromArgb(255, gray, gray, gray);
        contentGrid.Background = new SolidColorBrush(clr);

        // Foreground
        gray = (byte)(255 - gray);
        clr = Color.FromArgb(255, gray, gray, gray);
        txtblk.Foreground = new SolidColorBrush(clr);
    }
}
```

As the background color of the *Grid* goes from black to white and back, the foreground color of the *TextBlock* goes from white to black and back, meeting halfway through.

The effect is nice, but notice that that two *SolidColorBrush* objects are being created at the frame rate of the video display (which is probably about 60 times a second) and these objects are just as quickly discarded. This is not necessary. A much better approach is to create two *SolidColorBrush* objects initially in the XAML file:

Project: ManualColorAnimation | File: BlankPage.xaml (excerpt)

```
<Grid>
    <Grid.Background>
        <SolidColorBrush x:Name="gridBrush" />
    </Grid.Background>

    <TextBlock Text="Hello, Windows 8!"
               FontFamily="Times New Roman"
               FontSize="96"
               FontWeight="Bold"
               HorizontalAlignment="Center"
               VerticalAlignment="Center">
        <TextBlock.Foreground>
            <SolidColorBrush x:Name="txtblkBrush" />
        </TextBlock.Foreground>
    </TextBlock>
</Grid>
```

These *SolidColorBrush* objects exist for the entire duration of the program, and they are given names for easy access from the *CompositionTarget.Rendering* handler:

Project: ManualColorAnimation | File: BlankPage.xaml.cs (excerpt)

```
void OnCompositionTargetRendering(object sender, object args)
{
    RenderingEventArgs renderingArgs = args as RenderingEventArgs;
    double t = (0.25 * renderingArgs.RenderingTime.TotalSeconds) % 1;
    t = t < 0.5 ? 2 * t : 2 - 2 * t;

    // Background
    byte gray = (byte)(255 * t);
    gridBrush.Color = Color.FromArgb(255, gray, gray, gray);
```

```
    // Foreground
    gray = (byte)(255 - gray);
    txtblkBrush.Color = Color.FromArgb(255, gray, gray, gray);
}
```

At first this might not seem a whole lot different because two *Color* objects are being created and discarded at the video frame rate. But it's wrong to speak of *objects* here because *Color* is a structure rather than a class. It is more correct to speak of *values* of *Color*s. These *Color* values are stored on the stack rather than requiring a memory allocation from the heap.

It's best to avoid frequent allocations from the heap whenever possible, and particularly when they're happening 60 times per second. But what I like most about this example is the idea of *SolidColorBrush* objects remaining alive in the Windows Runtime composition system. This program is effectively reaching down into that composition layer and changing a property of the brush so that it renders differently.

This program also illustrates part of the wonders of dependency properties. Dependency properties are built to respond to changes in a very structured manner. As you'll discover, the built-in animation facilities of the Windows Runtime can target *only* dependency properties, and "manual" animations using *CompositionTarget.Rendering* have pretty much the same limitation. Fortunately, the *Foreground* property of *TextBlock* and the *Background* property of *Grid* are both dependency properties of type *Brush*, and the *Color* property of the *SolidColorBrush* is also a dependency property.

Indeed, whenever you encounter a dependency property, you might ask yourself, "How can I animate that?" For example, the *Offset* property in the *GradientStop* class is a dependency property, and you can animate it for some interesting effects.

Here's the XAML file for the RainbowEight project:

Project: RainbowEight | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <TextBlock Name="txtblk"
               Text="8"
               FontFamily="CooperBlack"
               FontSize="1"
               HorizontalAlignment="Center">
        <TextBlock.Foreground>
            <LinearGradientBrush x:Name="gradientBrush">
                <GradientStop Offset="0.00" Color="Red" />
                <GradientStop Offset="0.14" Color="Orange" />
                <GradientStop Offset="0.28" Color="Yellow" />
                <GradientStop Offset="0.43" Color="Green" />
                <GradientStop Offset="0.57" Color="Blue" />
                <GradientStop Offset="0.71" Color="Indigo" />
                <GradientStop Offset="0.86" Color="Violet" />
                <GradientStop Offset="1.00" Color="Red" />
                <GradientStop Offset="1.14" Color="Orange" />
                <GradientStop Offset="1.28" Color="Yellow" />
                <GradientStop Offset="1.43" Color="Green" />
                <GradientStop Offset="1.57" Color="Blue" />
```

```
                <GradientStop Offset="1.71" Color="Indigo" />
                <GradientStop Offset="1.86" Color="Violet" />
                <GradientStop Offset="2.00" Color="Red" />
            </LinearGradientBrush>
        </TextBlock.Foreground>
    </TextBlock>
</Grid>
```

A bunch of those *GradientStop* objects have *Offset* values above 1, so they're not going to be visible. Moreover, the *TextBlock* itself won't be very obvious because it has a *FontSize* of 1. However, during its *Loaded* event, the *Page* class obtains the *ActualHeight* of that tiny *TextBlock* and saves it in a field. It then starts a *CompositionTarget.Rendering* event going:

Project: RainbowEight | File: BlankPage.xaml (excerpt)

```
public sealed partial class BlankPage : Page
{
    double txtblkBaseSize;  // ie, for 1-pixel FontSize

    public BlankPage()
    {
        this.InitializeComponent();
        Loaded += OnPageLoaded;
    }

    void OnPageLoaded(object sender, RoutedEventArgs args)
    {
        txtblkBaseSize = txtblk.ActualHeight;
        CompositionTarget.Rendering += OnCompositionTargetRendering;
    }

    void OnCompositionTargetRendering(object sender, object args)
    {
        // Set FontSize as large as it can be
        txtblk.FontSize = this.ActualHeight / txtblkBaseSize;

        // Calculate t from 0 to 1 repetitively
        RenderingEventArgs renderingArgs = args as RenderingEventArgs;
        double t = (0.25 * renderingArgs.RenderingTime.TotalSeconds) % 1;

        // Loop through GradientStop objects
        for (int index = 0; index < gradientBrush.GradientStops.Count; index++)
            gradientBrush.GradientStops[index].Offset = index / 7.0 - t;
    }
}
```

In the *CompositionTarget.Rendering* handler, the *FontSize* of the *TextBlock* is increased based on the *ActualHeight* property of the *Page*. It won't be the full height of the page because the *ActualHeight* of the *TextBlock* includes space for descenders and diacriticals, but it will be as large as is convenient to make it, and it will change when the display switches orientation.

Moreover, the *CompositionTarget.Rendering* handler goes on to change all the *Offset* properties of the *LinearGradientBrush* for an animated rainbow effect that I'm afraid can't quite be rendered on the

static page of this book:

You might wonder: Isn't it inefficient to change the *FontSize* property of the *TextBlock* at the frame rate of the video display? Wouldn't it make more sense to set a *SizeChanged* handler for the *Page* and do it then?

Perhaps a little. But it is another feature of dependency properties that the object doesn't register a change unless the property really changes. If the property is being set to the value it already is, nothing happens, as you can verify by attaching a *SizeChanged* handler on the *TextBlock* itself.

Chapter 4

Presentation with Panels

A Windows Runtime program generally consists of one of more classes that derive from *Page*. Each page contains a visual tree of elements connected in a parent-child hierarchy. A *Page* object can have only one child set to its *Content* property, but in most cases this child is an instance of a class that derives from *Panel*. *Panel* defines a property named *Children* that is of type *UIElementCollection*—a collection of *UIElement* derivatives, including other panels.

These *Panel* derivatives form the core of the Windows Runtime dynamic layout system. As the size or orientation of a page changes, panels can reorganize their children to optimally fill the available space. Each type of panel arranges its children differently. The *Grid*, for example, arranges its children in rows and columns. The *StackPanel* stacks its children either horizontally or vertically. The *VariableSizedWrapGrid* also stacks its children horizontally or vertically but then uses additional rows or columns if necessary. much like the Windows 8 start screen. The *Canvas* allows its children to be positioned at specific pixel locations.

What makes a layout system complex is balancing the conflicting needs of parents and children. In part, a layout system needs to be "child-driven" in that each child should be allowed to determine how large it needs to be, and to obtain sufficient screen space for itself. But the layout system also needs to be "parent-driven." At any time, the page is fixed in size and cannot give its descendants in the visual tree more space than it has available.

For example, a simple HTML page has a width that is parent-driven because it's constrained by the width of the video display or the browser window. However, the height of a page is child-driven because it depends on the content of the page. If that height exceeds the height of the browser window, scrollbars are required.

The Windows 8 start screen is the other way around: The number of application tiles that can fit vertically is parent-driven because it’s based on the height of the screen. The width of this tile display is child-driven. If tiles extend off the screen horizontally, they must be moved into view by scrolling.

The *Border* Element

Two of the most important properties connected with layout are *HorizontalAlignment* and *VerticalAlignment*. These properties are defined by *FrameworkElement* and set to members of enumerations with identical names: *HorizontalAlignment* and *VerticalAlignment*.

As you saw in Chapter 3, “Basic Event Handling,” the default values of *HorizontalAlignment* and *VerticalAlignment* are not *Left* and *Top*. They are instead *HorizontalAlignment.Stretch* and

VerticalAlignment.Stretch. These default *Stretch* settings imply parent-driven layout: elements automatically stretch to become as large as their parents. This is not always visually apparent, but in the last chapter you saw how a *TextBlock* stretched to the size of its parent gets all the *Tapped* events anywhere within that parent.

When the *HorizontalAlignment* or *VerticalAlignment* properties are set to values other than *Stretch*, the element sets its own width or height based on its content and layout becomes more child-driven.

The important role of *HorizontalAlignment* and *VerticalAlignment* also becomes apparent when you start adding more parents and children to the page. For example, suppose you want to display a *TextBlock* with a border around it. You might discover (perhaps with some dismay) that the *TextBlock* has no properties that relate to a border. However, the *Windows.UI.Xaml.Controls* namespace contains a *Border* element with a property named *Child*. So you put the *TextBlock* in a *Border* and put the *Border* in the *Grid*, like so:

Project: NaiveBorderedText | File: BlankPage.xaml (excerpt)

```
<Page … >

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">

        <Border BorderBrush="Red"
                BorderThickness="12"
                CornerRadius="24"
                Background="Yellow">

            <TextBlock Text="Hello Windows 8!"
                       FontSize="96"
                       Foreground="Blue"
                       HorizontalAlignment="Center"
                       VerticalAlignment="Center" />
        </Border>

    </Grid>
</Page>
```

The *BorderThickness* property defined by *Border* can be set to different values for the four sides. Just specify four different values in the order left, top, right, and bottom. If you specify only two values, the first applies to the left and right and the second applies to the top and bottom. The *CornerRadius* property defines the curvature of the corners. You can set it a uniform value or four different values in the order upper-left, upper-right, lower-right, and lower-left.

Notice the *HorizontalAlignment* and *VerticalAlignment* properties set on the *TextBlock*. The markup looks reasonable, but the result is probably not what you want:

Because *Border* derives from *FrameworkElement*, it also has *HorizontalAlignment* and *VerticalAlignment* properties, and their default values are *Stretch*, which causes the size of the *Border* to be stretched to the size of its parent. To get the effect you probably want, you need to move the *HorizontalAlignment* and *VerticalAlignment* settings from the *TextBlock* to the *Border*:

Project: BetterBorderedText | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">

    <Border BorderBrush="Red"
            BorderThickness="12"
            CornerRadius="24"
            Background="Yellow"
            HorizontalAlignment="Center"
            VerticalAlignment="Center">

        <TextBlock Text="Hello Windows 8!"
                   FontSize="96"
                   Foreground="Blue"
                   Margin="24" />
    </Border>

</Grid>
```

I've also added a quarter-inch margin to the *TextBlock* by setting its *Margin* property. This causes the *Border* to be a quarter-inch larger than the size of the text on all four sides:

The *Margin* property is defined by *FrameworkElement*, so it is available on every element. The property is of type *Thickness* (the same as the type of the *BorderThickness* property)—a structure with four properties named *Left*, *Top*, *Right*, and *Bottom*. *Margin* is exceptionally useful for defining a little breathing room around elements so that they don't butt up against each other, and it appears a lot in real-life XAML. Like *BorderThickness*, *Margin* can potentially have four different values. In XAML, they appear in the order left, top, right, and bottom. Specify just two values and the first applies to the left and right, and the second to the top and bottom.

In addition, *Border* defines a *Padding* property, which is similar to *Margin* except that it applies to the inside of the element rather than the outside. Try removing the *Margin* property from *TextBlock* and instead set *Padding* on the *Border*:

```
<Border BorderBrush="Red"
        BorderThickness="12"
        CornerRadius="24"
        Background="Yellow"
        HorizontalAlignment="Center"
        VerticalAlignment="Center"
        Padding="24">

    <TextBlock Text="Hello Windows 8!"
               FontSize="96"
               Foreground="Blue" />
</Border>
```

The result is the same. In either case, any *HorizontalAlignment* or *VerticalAlignment* settings on the *TextBlock* are now irrelevant.

For layout purposes, *Margin* is considered to be part of the size of the element, but otherwise it is entirely out of the element's control. The element cannot control the background color of its margin, for example. That color depends on the element's parent. Nor does an element get user input from the

margin area. If you tap in an element's margin area, the element's *parent* gets the *Tapped* event.

The *Padding* property is also of type *Thickness*, but only a few classes define a *Padding* property: *Control*, *Border*, *TextBlock*, *RichTextBlock*, and *RichTextBlockOverflow*. The *Padding* property defines an area *inside* the element. This area is considered to be part of the element for all purposes, including user input.

If you want a *TextBlock* to respond to taps not only on the text itself but also within a 100-pixel area surrounding the text, set the *Padding* property of the *TextBlock* to 100 rather than the *Margin* property.

Rectangle and Ellipse

As you saw in Chapter 2, "XAML Syntax," the *Windows.UI.Xaml.Shapes* namespace contains classes used to render vector graphics: lines, and curves, and filled areas. The *Shape* class itself derives from *FrameworkElement* and defines various properties, including *Stroke* (for specifying the brush used to render straight lines and curves), *StrokeThickness*, and *Fill* (for specifying the brush used to render enclosed areas).

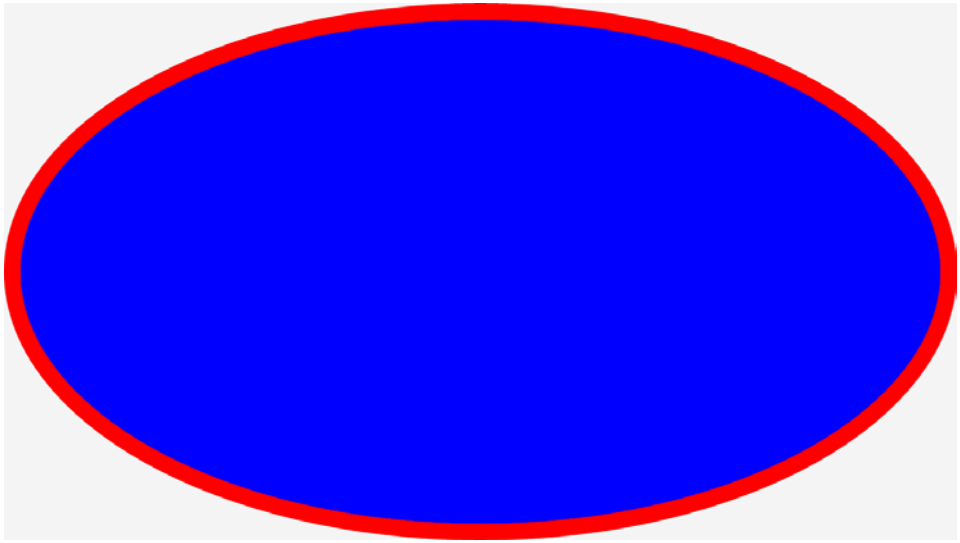
Six classes derive from *Shape*. *Line*, *Polyline*, and *Polygon* render straight lines based on coordinate points, and *Path* uses a series of classes in *Windows.UI.Xaml.Media* for rendering a series of straight lines, arcs, and Bezier curves.

The remaining two classes that derive from *Shape* are *Rectangle* and *Ellipse*. Despite the innocent names, these elements are real oddities in that they define figures without the use of coordinate points. Here, for example, is a tiny piece of XAML to render an ellipse:

Project: SimpleEllipse | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <Ellipse Stroke="Red"
             StrokeThickness="24"
             Fill="Blue" />
</Grid>
```

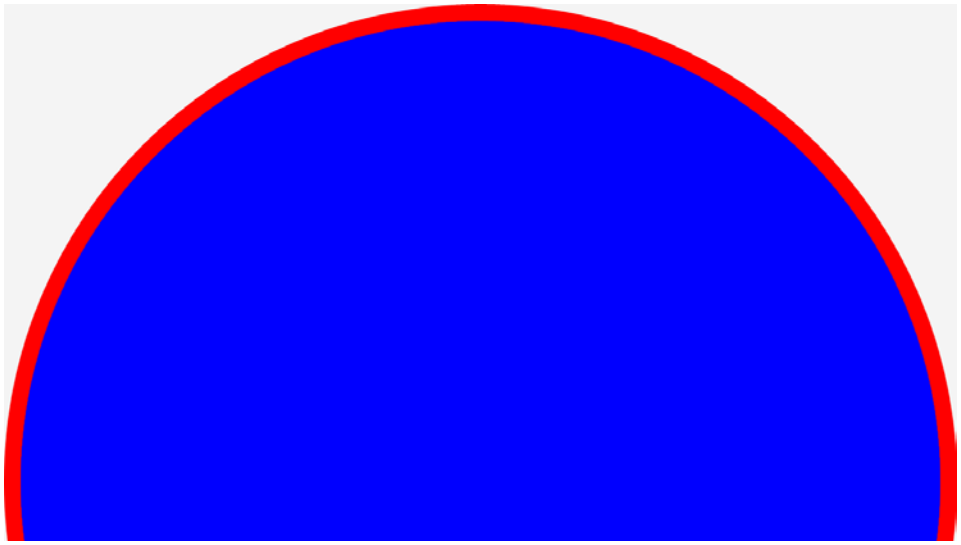
Notice how the ellipse fills its container:



Like all other *FrameworkElement* derivatives, *Ellipse* has default *HorizontalAlignment* and *VerticalAlignment* settings of *Stretch*, but *Ellipse* most decisively reveals the implications of these settings.

What happens if you set a nondefault *HorizontalAlignment* or *VerticalAlignment* on this *Ellipse* element? Try it! The ellipse shrinks down to nothing. It disappears. In fact, it's hard to imagine how it can legitimately have any other behavior. If you do not want the *Ellipse* or *Rectangle* element to fill its container, your only real alternative is to set explicit *Height* and *Width* values on it.

The *Shape* class also defines a *Stretch* property, which is similar to the *Stretch* property defined by *Image*. For example, in the SimpleEllipse program, if you set the *Stretch* property to *Uniform*, you'll get a special case of an ellipse that has equal horizontal and vertical radii. This is a circle, and its diameter is set to the minimum of the container's width and height. Setting the *Stretch* property to *UniformToFill* also gets you a circle, but now the diameter is the *maximum* of the container's width and height, so part of the circle is cropped:



You can control what part is cropped with the *HorizontalAlignment* and *VerticalAlignment* properties.

Rectangle is very similar to *Ellipse* and also shares several characteristics with *Border*, although the properties have different names:

Border	**Rectangle**
BorderBrush	Stroke
BorderThickness	StrokeThickness
Background	Fill
CornerRadius	RadiusX / RadiusY

The big difference between *Border* and *Rectangle* is that *Border* has a *Child* property and *Rectangle* does not.

The *StackPanel*

Panel and its derivative classes form the core of the Windows Runtime layout system. *Panel* defines just a few properties on its own, but one of them is *Children*, and that's crucial. A *Panel* derivative is the only type of element that supports multiple children.

This class hierarchy shows *Panel* and some of its derivatives:

Object
DependencyObject
UIElement
FrameworkElement
Panel
Canvas
Grid

StackPanel
VariableSizedWrapGrid

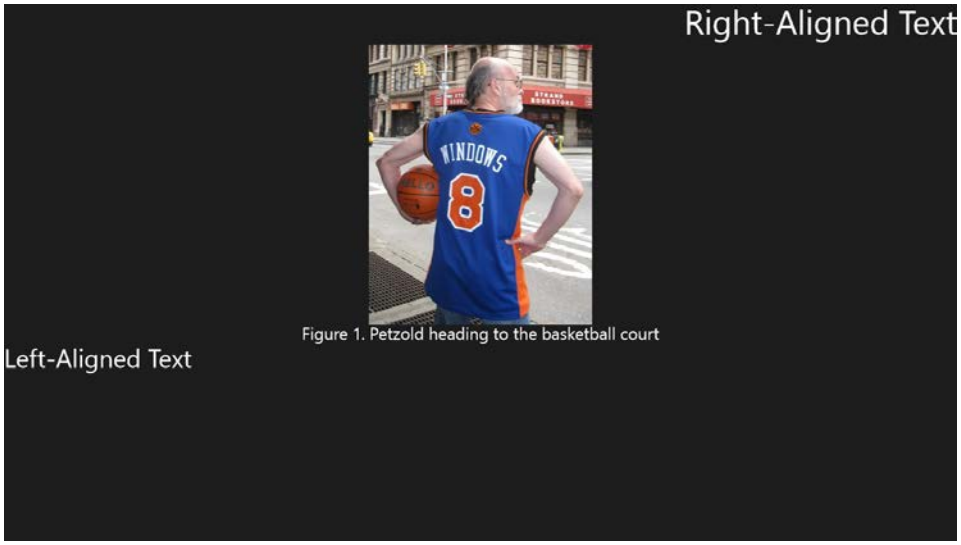
There are others, but they have restrictions that prevent them from being used except in controls of type *ItemsControl* (which I'll discuss in a future chapter). I'll save the *Grid* for Chapter 5, "Control Interaction," and I'll cover the other three here.

Of these standard panels, the *StackPanel* is certainly the easiest to use. Like the name suggests, it stacks its children, by default vertically. The children can be different heights, but each child gets only as much height as it needs. The SimpleVerticalStack program shows how it's done:

Project: SimpleVerticalStack | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <StackPanel>
        <TextBlock Text="Right-Aligned Text"
                   FontSize="48"
                   HorizontalAlignment="Right" />

        <Image Source="http://www.charlespetzold.com/pw6/PetzoldJersey.jpg"
               Stretch="None" />

        <TextBlock Text="Figure 1. Petzold heading to the basketball court"
                   FontSize="24"
                   HorizontalAlignment="Center" />

        <Ellipse Stroke="Red"
                 StrokeThickness="12"
                 Fill="Blue" />

        <TextBlock Text="Left-Aligned Text"
                   FontSize="36"
                   HorizontalAlignment="Left" />
    </StackPanel>
</Grid>
```

In XAML the children of the *StackPanel* are simply listed in order, and that's how they appear on the screen:



Notice that I made this *StackPanel* a child of the *Grid*. Panels can be nested, and they very often *are* nested. In this particular case I could have replaced the *Grid* with *StackPanel* and set that same *Background* property on it.

Each element in the *StackPanel* gets only as much height as it needs but can stretch to the panel's full width, as demonstrated by the first and last *TextBlock* aligned to the right and left. In a vertical *StackPanel*, any *VerticalAlignment* settings on the children are irrelevant and are basically ignored.

Notice that the *Stretch* property of the *Image* element is set to *None* to display the bitmap in its pixel dimensions. If left at its default value of *Uniform*, the *Image* is stretched to the width of the *StackPanel* (which is the same as the width of the *Page*) and its vertical dimension increases proportionally. This might cause all the elements below the *Image* to be pushed right off the bottom and into the bit bucket.

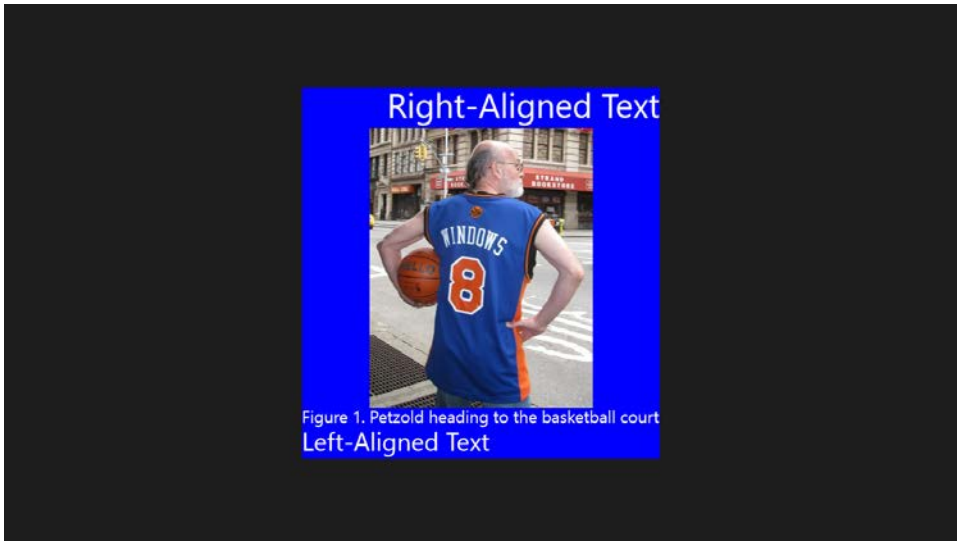
The XAML also includes an *Ellipse*. What happened to it? Like all the other children of the *StackPanel*, the *Ellipse* is given only as much vertical space as it needs, and it really doesn't need any, so it shrinks to nothing. If you want the *Ellipse* to be visible, give it at least a nonzero *Height*, for example, 48:

If you also set the *Stretch* property of the *Ellipse* to *Uniform*, you'll get a circle rather than a very wide ellipse.

This *StackPanel* occupies the entire page. How do I know this? When experimenting with panels, one very useful technique is to give each panel a unique *Background* so that you can see the real estate that the panel occupies on the screen. For example:

```
<StackPanel Background="Blue">
```

Like all other *FrameworkElement* derivatives, *StackPanel* also has *HorizontalAlignment* and *VerticalAlignment* properties. When set to nondefault values, these properties cause the *StackPanel* to tightly hug its contents, and the change can be dramatic. Here's what it looks like with the *StackPanel* getting a *Background* of *Blue* and *HorizontalAlignment* and *VerticalAlignment* values of *Center*:



Horizontal Stacks

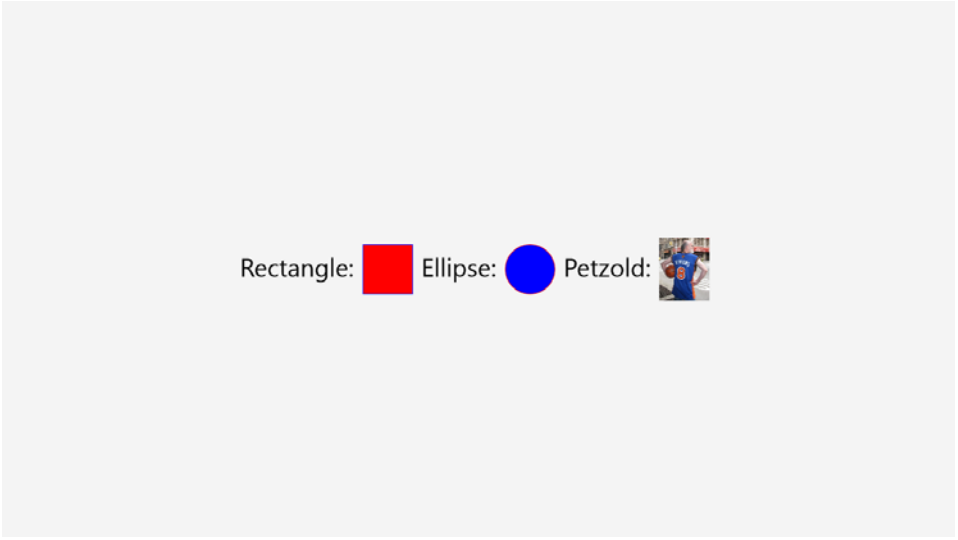
It is also possible to use *StackPanel* to stack elements horizontally by setting its *Orientation* property to *Horizontal*. The SimpleHorizontalStack program shows an example:

Project: SimpleHorizontalStack | File: BlankPage.xaml (excerpt)

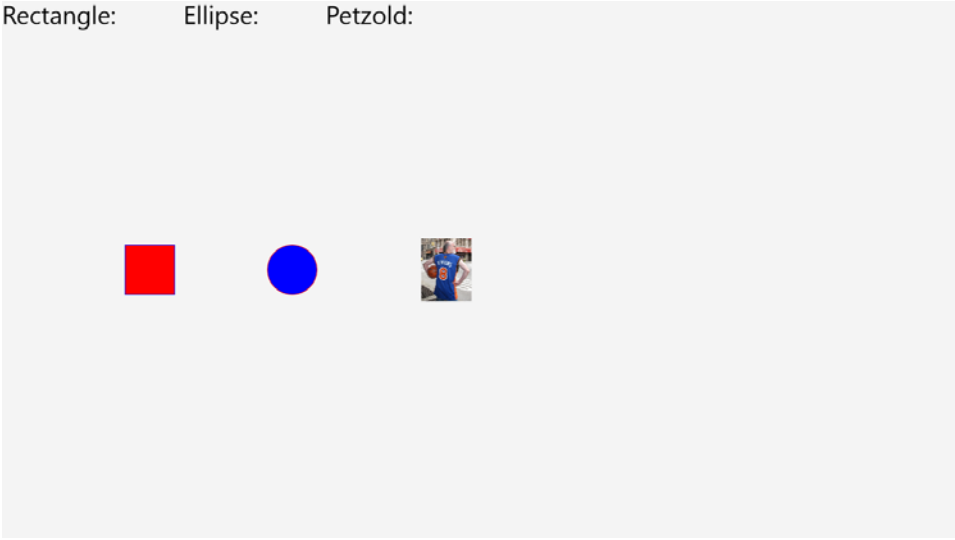
```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">

    <StackPanel Orientation="Horizontal"
                VerticalAlignment="Center"
                HorizontalAlignment="Center">

        <TextBlock Text="Rectangle: "
                   VerticalAlignment="Center" />

        <Rectangle Stroke="Blue"
                   Fill="Red"
                   Width="72"
                   Height="72"
                   Margin="12 0"
                   VerticalAlignment="Center" />

        <TextBlock Text="Ellipse: "
                   VerticalAlignment="Center" />

        <Ellipse Stroke="Red"
                 Fill="Blue"
                 Width="72"
                 Height="72"
                 Margin="12 0"
```

```
                VerticalAlignment="Center" />

        <TextBlock Text="Petzold: "
                   VerticalAlignment="Center" />

        <Image Source="http://www.charlespetzold.com/pw6/PetzoldJersey.jpg"
               Stretch="Uniform"
               Width="72"
               Margin="12 0"
               VerticalAlignment="Center" />

    </StackPanel>
</Grid>
```

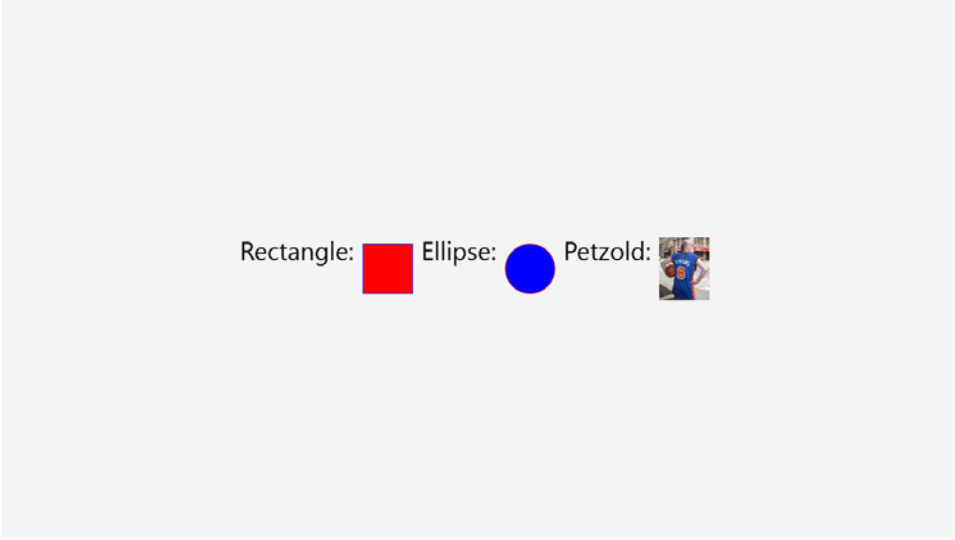
Here it is:



You might question the apparently excessive number of alignment settings. Try removing all the *VerticalAlignment* and *HorizontalAlignment* settings, and the result looks like this:



The *StackPanel* is now occupying the entire page, and each of the individual elements is occupying the full height of the *StackPanel*. *TextBlock* aligns itself at the top, and the other elements are in the center. Setting the *HorizontalAlignment* and *VerticalAlignment* settings of the *Panel* to *Center* tightens up the space the panel occupies and moves it to the center of the display, like this:



The height of the *StackPanel* is now governed by the height of its tallest element, but all the elements are stretched to that height. To center all the elements relative to each other, the easiest approach is to give them all *VerticalAlignment* settings of *Center*.

WhatSize with Bindings (and a Converter)

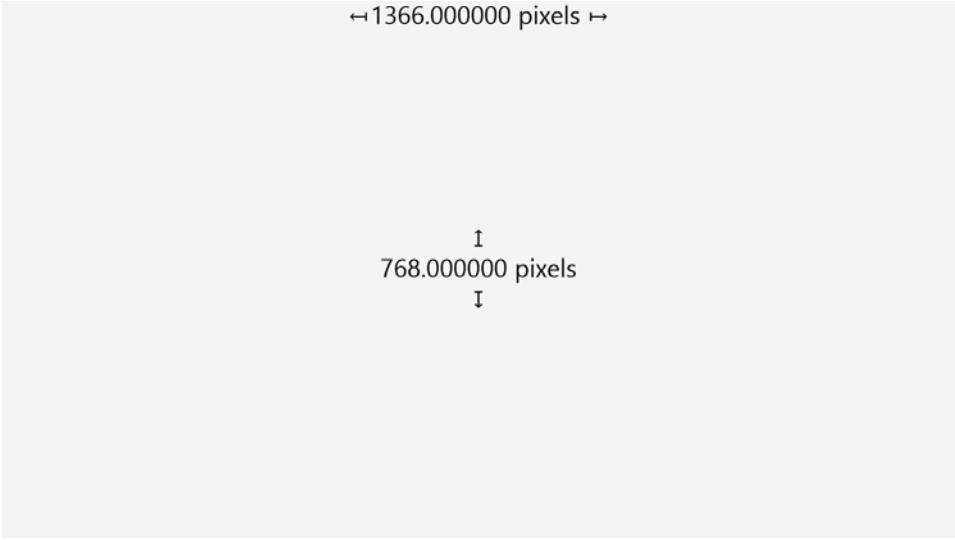
In Chapter 3 I discussed how the WhatSize program couldn't accommodate a data binding because the *Text* property in the *Run* class isn't a dependency property. Only dependency properties can be targets of data bindings.

Fortunately, for single lines of text, you can mimic multiple *Run* objects with multiple *TextBlock* elements in a horizontal *StackPanel*. Here's WhatSizeWithBindings:

Project: WhatSizeWithBindings | File: BlankPage.xaml (excerpt)

```
<Page
    x:Class="WhatSizeWithBindings.BlankPage"
    …
    FontSize="36"
    Name="page">

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <StackPanel Orientation="Horizontal"
                    HorizontalAlignment="Center"
                    VerticalAlignment="Top">
            <TextBlock Text="&#x21A4; " />
            <TextBlock Text="{Binding ElementName=page, Path=ActualWidth}" />
            <TextBlock Text=" pixels &#x21A6;" />
        </StackPanel>

        <StackPanel HorizontalAlignment="Center"
                    VerticalAlignment="Center">
            <TextBlock Text="&#x21A5;" TextAlignment="Center" />

            <StackPanel Orientation="Horizontal"
                        HorizontalAlignment="Center">
                <TextBlock Text="{Binding ElementName=page, Path=ActualHeight}" />
                <TextBlock Text=" pixels" />
            </StackPanel>

            <TextBlock Text="&#x21A7;" TextAlignment="Center" />
        </StackPanel>
    </Grid>
</Page>
```

Notice that the root element is now given a name of *page*, which is referenced in the two data bindings to obtain the *ActualWidth* and *ActualHeight* properties. The big advantage over the previous version is that there's no longer any need for an event handler in the code-behind file. And here it is:



No? You don't like that it's accurate to a millionth of a pixel?

The problem, of course, is that *ActualWidth* and *ActualHeight* are *double* values, and when these values are converted to strings for the *Text* property of *TextBlock*, this is what sometimes happens.

In cases like this, it is possible to supply a little piece of code to the *Binding* object so that it performs the data conversion in exactly the way you want. The *Binding* class has a property named *Converter* of type *IValueConverter*, an interface with two methods named *Convert* (to convert from a binding source to a binding target) and *ConvertBack* (for a conversion from the target back to the source in a two-way binding).

To create your own custom converter, you'll need to derive a class from *IValueConverter* and to fill in the two methods. Here's an example that shows these methods doing nothing:

```
public class NothingConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, string language)
    {
        return value;
    }

    public object ConvertBack(object value, Type targetType, object parameter, string language)
    {
        return value;
    }
}
```

If you'll be using the binding only in a one-way mode, you can ignore the *ConvertBack* method. In the *Convert* method, the *value* argument is the value coming from the source. In the WhatSize example, this is a *double*. The *TargetType* is the type of the target—in the WhatSize example, a *string*.

If you're writing a binding converter specifically for WhatSize to convert floating-point numbers to strings with no decimal points, the *Convert* method can be as simple as this:

```
public object Convert(object value, Type targetType, object parameter, string language)
{
    return ((double)value).ToString("F0");
}
```

But it's more common to generalize binding converters. For example, it might be useful for the converter to handle *value* arguments of any type that implements the *IFormattable* interface, which includes *double* as well as all the other numeric types and *DateTime*. The *IFormattable* interface defines a *ToString* method with two arguments: a formatting string and an object that implements *IFormatProvider*, which is generally a *CultureInfo* object.

Besides *value* and *targetType*, the *Convert* method also has *parameter* and *language* arguments. These come from two properties of the *Binding* class named *ConverterParameter* and *ConverterLanguage*, which are generally set right in the XAML file. This means that the formatting specification for *ToString* can be provided by the *parameter* argument to *Convert*, and a *CultureInfo* object could be created from the *language* argument. Here's one possibility:

Project: WhatSizeWithBindingConverter | File: FormattedStringConverter.cs

```
using System;
using System.Globalization;
using Windows.UI.Xaml.Data;

namespace WhatSizeWithBindingConverter
{
    public class FormattedStringConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, string language)
        {
            if (value is IFormattable &&
                parameter is string &&
                !String.IsNullOrEmpty(parameter as string) &&
                targetType == typeof(string))
            {
                if (String.IsNullOrEmpty(language))
                    return (value as IFormattable).ToString(parameter as string, null);

                return (value as IFormattable).ToString(parameter as string,
                                                        new CultureInfo(language));
            }

            return value;
        }
        public object ConvertBack(object value, Type targetType, object parameter, string language)
        {
            return value;
        }
    }
}
```

The *Convert* method uses *ToString* only if several conditions are met. If the conditions are not met, the fallback is simply to return the incoming *value* argument.

In the XAML file, the binding converter is generally defined as a resource so that it can be shared among multiple bindings:

Project: WhatSizeWithBindingConverter | File: BlankPage.xaml (excerpt)

```
<Page
    x:Class="WhatSizeWithBindingConverter.BlankPage"
    …
    FontSize="36"
    Name="page">

    <Page.Resources>
        <local:FormattedStringConverter x:Key="stringConverter" />
    </Page.Resources>

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <StackPanel Orientation="Horizontal"
                    HorizontalAlignment="Center"
                    VerticalAlignment="Top">
            <TextBlock Text="&#x21A4; " />
            <TextBlock Text="{Binding ElementName=page,
                                     Path=ActualWidth,
                                     Converter={StaticResource stringConverter},
                                     ConverterParameter=F0}" />
            <TextBlock Text=" pixels &#x21A6;" />
        </StackPanel>

        <StackPanel HorizontalAlignment="Center"
                    VerticalAlignment="Center">
            <TextBlock Text="&#x21A5;" TextAlignment="Center" />

            <StackPanel Orientation="Horizontal"
                        HorizontalAlignment="Center">
                <TextBlock Text="{Binding ElementName=page,
                                         Path=ActualHeight,
                                         Converter={StaticResource stringConverter},
                                         ConverterParameter=F0}" />
                <TextBlock Text=" pixels" />
            </StackPanel>

            <TextBlock Text="&#x21A7;" TextAlignment="Center" />
        </StackPanel>
    </Grid>
</Page>
```

The display looks just like the original WhatSize program in Chapter 3. Take careful note of the syntax here:

```
<TextBlock Text="{Binding ElementName=page,
                         Path=ActualWidth,
                         Converter={StaticResource stringConverter},
```

```
                ConverterParameter=F0}" />
```

The *Binding* markup is spread out over four lines for purposes of clarity (and to stay within the margins of the book page), but notice that the *Binding* markup extension contains an embedded markup extension of *StaticResource* for referencing the binding converter resource. No quotation marks appear within either markup extension.

The Common folder in the standard Visual Studio project contains two binding converters. *BooleanToVisibilityConverter* is useful for controlling the *Visibility* property, which takes on values of *Visibility.Visible* and *Visibility.Collapsed*. The *BooleanNegationConverter* changes *true* to *false* and *false* to *true*.

The *ScrollViewer* Solution

What happens if there are too many elements for *StackPanel* to display on the screen? In real life, that situation occurs quite often and it's why a *StackPanel* with more than just a few elements is almost always put inside a *ScrollViewer*.

The *ScrollViewer* has a property named *Content* that you can set to anything that might be too large to display in the space allowed for it—a single large *Image*, for example. *ScrollViewer* provides scrollbars for the mouse-users among us. Otherwise, you can just scroll it with your fingers. By default, *ScrollViewer* also adds a pinch interface so that you can use two fingers to make the content larger or smaller. This can be disabled if you want by setting the *ZoomMode* property to *Disabled*.

ScrollViewer defines a couple other crucial properties. Most often you'll be using *ScrollViewer* for vertical scrolling, such as with a vertical *StackPanel*. Consequently, the default value of the *VerticalScrollBarVisibility* property is the enumeration member *ScrollBarVisibility.Visible*. This setting doesn't mean that the scrollbar is actually visible all the time. For mouse users, the scrollbar appears only when the mouse is moved to the right side of the *ScrollViewer*, and then it fades from view if the mouse is moved away. A much thinner slider appears when you scroll using your finger.

Horizontal scrolling is different: the default value of *HorizontalScrollBarVisibility* property is *Disabled*, so you'll want to change that to enable horizontal scrolling. The other two options are *Hidden*, which allows scrolling with your fingers but not the mouse, and *Auto*, which is the same as *Visible* if the content requires scrolling and *Disabled* otherwise.

The XAML file for the StackPanelWithScrolling program contains a *StackPanel* in a *ScrollViewer*. Notice that the *FontSize* property is set in the root tag based on a predefined identifier:

Project: StackPanelWithScrolling | File: BlankPage.xaml (excerpt)

```
<Page
    x:Class="StackPanelWithScrolling.BlankPage"
    …
    FontSize="{StaticResource HeaderMediumFontSize}">
```

```
    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <ScrollViewer>
            <StackPanel Name="stackPanel" />
        </ScrollViewer>
    </Grid>
</Page>
```

Now all that's necessary in the code-behind file is to generate so many items for the *StackPanel* that they can't all be visible at once. Where do we get so many items? One convenient solution is to use .NET reflection to obtain all 141 static *Color* properties defined in the *Colors* class:

Project: StackPanelWithScrolling | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    public BlankPage()
    {
        this.InitializeComponent();

        IEnumerable<PropertyInfo> properties =
                                typeof(Colors).GetTypeInfo().DeclaredProperties;

        foreach (PropertyInfo property in properties)
        {
            Color clr = (Color)property.GetValue(null);
            TextBlock txtblk = new TextBlock();
            txtblk.Text = String.Format("{0} \x2014 {1:X2}-{2:X2}-{3:X2}-{4:X2}",
                                        property.Name, clr.A, clr.R, clr.G, clr.B);
            stackPanel.Children.Add(txtblk);
        }
    }
}
```

Windows 8 reflection works a little differently from .NET reflection. Generally, to get anything interesting from the *Type* object, you need to call a Windows 8 extension method *GetTypeInfo*. The returned *TypeInfo* object makes available additional information about the *Type*. In this program, the *DeclaredProperties* property of *TypeInfo* obtains all the properties of the *Colors* class in the form of *PropertyInfo* objects. Because all the properties in the *Colors* class are static, the value of these static properties can be obtained by calling *GetValue* on each *PropertyInfo* object with a *null* parameter. Each *TextBlock* gets the name of the color, an em-dash (Unicode 0x2014), and the hexadecimal color bytes. The display looks like this:

```
GreenYellow — FF-AD-FF-2F
Honeydew — FF-F0-FF-F0
HotPink — FF-FF-69-B4
IndianRed — FF-CD-5C-5C
Indigo — FF-4B-00-82
Ivory — FF-FF-FF-F0
Khaki — FF-F0-E6-8C
Lavender — FF-E6-E6-FA
LavenderBlush — FF-FF-F0-F5
LawnGreen — FF-7C-FC-00
LemonChiffon — FF-FF-FA-CD
LightBlue — FF-AD-D8-E6
LightCoral — FF-F0-80-80
LightCyan — FF-E0-FF-FF
LightGoldenrodYellow — FF-FA-FA-D2
LightGray — FF-D3-D3-D3
LightGreen — FF-90-EE-90
LightPink — FF-FF-B6-C1
LightSalmon — FF-FF-A0-7A
LightSeaGreen — FF-20-B2-AA
LightSkyBlue — FF-87-CE-FA
LightSlateGray — FF-77-88-99
LightSteelBlue — FF-B0-C4-DE
LightYellow — FF-FF-FF-E0
Lime — FF-00-FF-00
```

And, of course, you can scroll it with your finger or the mouse.

As you play around with the program, you'll discover that the *ScrollViewer* incorporates a nice fluid response to your finger movements, including inertia and bounce. You'll want to use *ScrollViewer* for virtually all your scrolling needs. You'll discover that many controls that incorporate scrolling—such as the *ListBox* and *GridView* coming up in a future chapter—have this same *ScrollViewer* built right in. I wouldn't be surprised if this same *ScrollViewer* is used in the Windows 8 start screen.

Wouldn't it be nice to see the actual colors as well as their names and values? That enhancement is coming up soon!

Several times already in this book I've shown you partial class hierarchies. You may have discovered that the documentation for each class shows only an ancestor class hierarchy but not derived classes, so you might have wondered how I assembled the class hierarchies for these pages. They came from a program I wrote called DependencyObjectClassHierarchy, which uses a *ScrollViewer* and *StackPanel* to show all the classes that derive from *DependencyObject*.

The XAML file is similar to the previous one except I've specified a smaller font:

Project: DependencyObjectClassHierarchy | File: BlankPage.xaml (excerpt)

```
<Page
    x:Class="DependencyObjectClassHierarchy.BlankPage"
    …
    FontSize="{StaticResource ContentFontSize}">

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <ScrollViewer>
            <StackPanel Name="stackPanel" />
        </ScrollViewer>
    </Grid>
</Page>
```

The program builds a tree of classes and their descendant classes. Each node is a particular class and a collection of its immediate descendent classes, so I added another code file to the project for a class that represents this node:

Project: DependencyObjectClassHierarchy | File: ClassAndSubclasses.cs

```
using System;
using System.Collections.Generic;

namespace DependencyObjectClassHierarchy
{
    class ClassAndSubclasses
    {
        public ClassAndSubclasses(Type parent)
        {
            this.Type = parent;
            this.Subclasses = new List<ClassAndSubclasses>();
        }

        public Type Type { protected set; get; }
        public List<ClassAndSubclasses> Subclasses { protected set; get; }
    }
}
```

Just as it's possible to use reflection to get all the properties defined by a class, you can use reflection to get all public classes defined in an assembly. These classes are available from the *ExportedTypes* property of the *Assembly* object. That's the simple part. The hard part is getting all the *Assembly* objects you'll need. Conceptually, each of the namespaces in the Windows Runtime is associated with an assembly of the same name. If you know a class defined in a particular assembly, the assembly in which that class is defined is available from the *Assembly* property of the *TypeInfo* object for that class.

To write this program, I had to figure out which namespaces contain classes that derive from *DependencyObject* and then pick a sample class from each of those namespaces. That's the purpose of the long list of *AddToClassList* calls here (and I can't guarantee I got them all):

Project: DependencyObjectClassHierarchy | File: BlankPage.xaml.cs (excerpt)

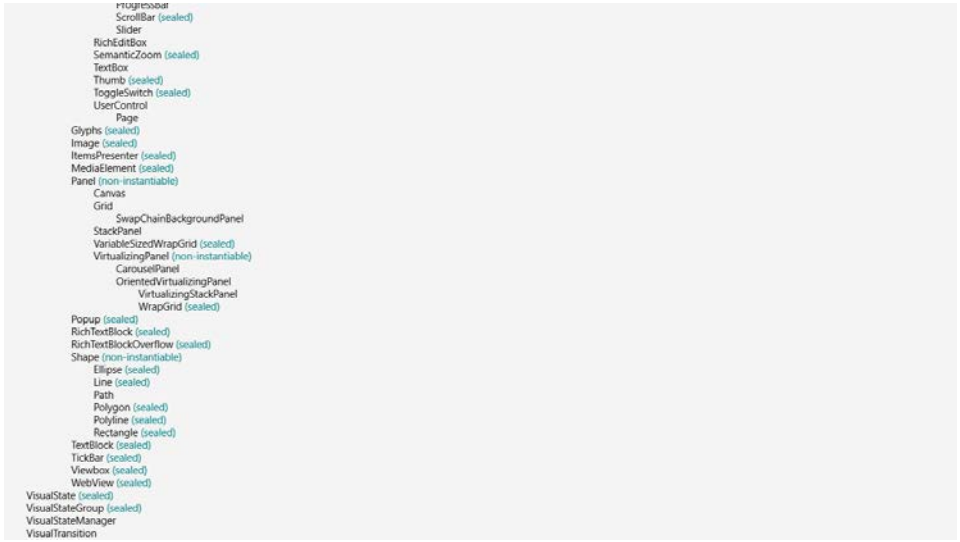
```
public sealed partial class BlankPage : Page
{
    Type rootType = typeof(DependencyObject);
    TypeInfo rootTypeInfo = typeof(DependencyObject).GetTypeInfo();
    List<Type> classes = new List<Type>();
    Brush highlightBrush;

    public BlankPage()
    {
        this.InitializeComponent();
        highlightBrush = this.Resources["ControlHighlightBrush"] as Brush;

        // Accumulate all the classes that derive from DependencyObject
        AddToClassList(typeof(Windows.UI.Xaml.DependencyObject));
        AddToClassList(typeof(Windows.UI.Xaml.Automation.Peers.AppBarAutomationPeer));
```

```
    AddToClassList(typeof(Windows.UI.Xaml.Automation.Provider.IRawElementProviderSimple));
    AddToClassList(typeof(Windows.UI.Xaml.Controls.Button));
    AddToClassList(typeof(Windows.UI.Xaml.Controls.Primitives.ButtonBase));
    AddToClassList(typeof(Windows.UI.Xaml.Data.Binding));
    AddToClassList(typeof(Windows.UI.Xaml.Documents.Block));
    AddToClassList(typeof(Windows.UI.Xaml.Input.FocusManager));
    AddToClassList(typeof(Windows.UI.Xaml.Media.Brush));
    AddToClassList(typeof(Windows.UI.Xaml.Media.Animation.BackEase));
    AddToClassList(typeof(Windows.UI.Xaml.Media.Imaging.BitmapImage));
    AddToClassList(typeof(Windows.UI.Xaml.Printing.PrintDocument));
    AddToClassList(typeof(Windows.UI.Xaml.Shapes.Ellipse));

    // Sort them alphabetically by name
    classes.Sort((t1, t2) =>
        {
            return String.Compare(t1.GetTypeInfo().Name, t2.GetTypeInfo().Name);
        });

    // Put all these sorted classes into a tree structure
    ClassAndSubclasses rootClass = new ClassAndSubclasses(rootType);
    AddToTree(rootClass, classes);

    // Display the tree using TextBlock's added to StackPanel
    Display(rootClass, 0);
}

void AddToClassList(Type sampleType)
{
    Assembly assembly = sampleType.GetTypeInfo().Assembly;

    foreach (Type type in assembly.ExportedTypes)
    {
        TypeInfo typeInfo = type.GetTypeInfo();

        if (typeInfo.IsPublic && rootTypeInfo.IsAssignableFrom(typeInfo))
            classes.Add(type);
    }
}

void AddToTree(ClassAndSubclasses parentClass, List<Type> classes)
{
    foreach (Type type in classes)
    {
        Type baseType = type.GetTypeInfo().BaseType;

        if (baseType == parentClass.Type)
        {
            ClassAndSubclasses subClass = new ClassAndSubclasses(type);
            parentClass.Subclasses.Add(subClass);
            AddToTree(subClass, classes);
        }
    }
}
```

```
    void Display(ClassAndSubclasses parentClass, int indent)
    {
        TypeInfo typeInfo = parentClass.Type.GetTypeInfo();

        // Create TextBlock with type name
        TextBlock txtblk = new TextBlock();
        txtblk.Inlines.Add(new Run { Text = new string(' ', 8 * indent) });
        txtblk.Inlines.Add(new Run { Text = typeInfo.Name });

        // Indicate if the class is sealed
        if (typeInfo.IsSealed)
            txtblk.Inlines.Add(new Run
                {
                    Text = " (sealed)",
                    Foreground = highlightBrush
                });

        // Indicate if the class can't be instantiated
        IEnumerable<ConstructorInfo> constructorInfos = typeInfo.DeclaredConstructors;
        int publicConstructorCount = 0;

        foreach (ConstructorInfo constructorInfo in constructorInfos)
            if (constructorInfo.IsPublic)
                publicConstructorCount += 1;

        if (publicConstructorCount == 0)
            txtblk.Inlines.Add(new Run
            {
                Text = " (non-instantiable)",
                Foreground = highlightBrush
            });

        // Add to the StackPanel
        stackPanel.Children.Add(txtblk);

        // Call this method recursively for all subclasses
        foreach (ClassAndSubclasses subclass in parentClass.Subclasses)
            Display(subclass, indent + 1);
    }
}
```

Notice how the *TextBlock* for each class is constructed by adding *Run* items to its *Inlines* collection. It's sometimes useful for a class hierarchy to display additional information, so the program also checks whether the class is marked as *sealed* and whether it can be instantiated. In the Windows Presentation Foundation and Silverlight, classes that can't be instantiated are generally defined as *abstract*. In the Windows Runtime, they have protected constructors instead.

Here's the section of the class hierarchy with *Panel* derivatives:



Layout Weirdness or Normalcy?

Suppose you have a *StackPanel* and you decide that one of the items in this *StackPanel* should be a *ScrollViewer* with another *StackPanel*. To determine what might happen in such a situation, you might experiment with the StackPanelWithScrolling project and change the XAML file like so:

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <StackPanel>
        <ScrollViewer>
            <StackPanel Name="stackPanel" />
        </ScrollViewer>
    </StackPanel>
</Grid>
```

When you try it out, you'll discover it doesn't work. You can't scroll. What happened?

Becoming acquainted with the mechanics of layout is an important part of being a crafty Windows Runtime developer, and the best way to make this acquaintance is to write your own *Panel* derivatives. That job awaits us in a future chapter.

The conflict here results from the different ways in which *StackPanel* and *ScrollViewer* calculate their desired heights. The *StackPanel* calculates a desired height based on the total height of all its children. In the vertical dimension (by default), *StackPanel* is entirely child-driven. To calculate a total height, it offers to each of its children an *infinite* height. (When you write your own *Panel* derivatives, you'll see that I'm not speaking metaphorically or abstractly. A *Double.PositiveInfinity* value actually comes into play!) The children respond by calculating a desired height based on their natural size. The *StackPanel* adds these heights to calculate its own desired height.

The height of the *ScrollViewer*, however, is parent-driven. Its height is only what its parent offers to it, and in our simple example this has been the height of the *Grid*, which is the height of the *Page*, which is the height of the window. The *ScrollViewer* is able to determine how to scroll its content because it knows the difference between the height of its child (often a *StackPanel*) and its own height.

Now put a vertically-scrolling *ScrollViewer* as a child of a vertical *StackPanel*. To determine the desired size of this *ScrollViewer* child, the *StackPanel* offers it an infinite height. How tall does the *ScrollViewer* really want to be? The height of the *ScrollViewer* is now child-driven rather than parent-driven, and its desired height is the height of its child, which is the total height of the inner *StackPanel*, which is the total accumulated height of all the children in that *StackPanel*.

From the perspective of the *ScrollViewer*, its height is the same as the height of its content, which means that there's nothing to scroll.

In other words, when a vertically-scrolling *ScrollViewer* is put in a vertical *StackPanel*, losing the ability to scroll is totally expected behavior!

Here's another seeming layout oddity that is actually quite normal: Try giving a *TextBlock* a very long chunk of text to display, and set the *TextWrapping* property to *Wrap*. In most cases, the text wraps as we might expect. Now put that *TextBlock* in a *StackPanel* with an *Orientation* property set to *Horizontal*. To determine how wide the *TextBlock* needs to be, the *StackPanel* offers it an infinite width, and in response to that infinite width, the *TextBlock* stops wrapping the text.

In the WhatSizeWithBindings and WhatSizeWithBindingConverter you saw how a horizontal *StackPanel* can effectively concatenate *TextBlock* elements, one of which has a binding on its *Text* property. But you can't use this same technique with a paragraph of wrapped text, because the text will never wrap in the horizontal *StackPanel*. If you need to concatenate different text strings in a paragraph, you'll need to use a single *TextBlock* with an *Inlines* collection. If one piece of text needs to be set to a variable data item, you can't use a binding because the *Text* property of *Run* is not backed by a dependency property. You'll need to set that item from code.

Because a vertical *StackPanel* has a finite width, it's an ideal host for *TextBlock* elements that wrap text, as you'll see next.

Making an E-Book

A *TextBlock* item that goes into a vertical *StackPanel* can have its *TextWrapping* property set to *Wrap*, which means that it can actually be a whole paragraph rather than just a word or two. *Image* elements can also go into this same *StackPanel*, and the result can be a rudimentary illustrated e-book.

On the famous Project Gutenberg website, I found an illustrated version of Beatrix Potter's classic children's book *The Tale of Tom Kitten* (http://www.gutenberg.org/ebooks/14837), so I created a Visual Studio project named TheTaleOfTomKitten and I made a folder called Images. From Project Gutenberg's HTML version of the book, it was easy to download all the illustrations in the form of JPEG

files. These have names such as tomxx.jpg, where xx is the original page number of the book where that illustration appeared. From within the Visual Studio project, I then added all 28 of these JPEG files to the Images folder.

Most of the rest of the work involved the BlankPage.xaml file. Each paragraph of the book became a *TextBlock*, and these I interspersed with *Image* elements referencing the JPEG files in the Images folder.

However, I felt it necessary to deviate somewhat from the ordering of the text and images in Project Gutenberg's HTML file. A PDF of the original edition of *The Tale of Tom Kitten* on the Internet Archive site (http://archive.org/details/taleoftomkitten00pottuoft) reveals how Miss Potter's illustrations are associated with the text of the book. There are two patterns:

1. Text appears on the verso (left-hand, even-numbered) page with an accompanying illustration on the recto (right-hand, odd-numbered) page.

2. Text appears on the recto page with an accompanying illustration on the verso page.

Adapting this paginated book to a continuous format required altering the order of the text and image in this second case so that the text appears *before* the accompanying illustration. That's why you'll see some page swaps in the XAML file.

Given the very many *TextBlock* and *Image* elements, styles seemed almost mandatory:

Project: TheTaleOfTomKitten | File: BlankPage.xaml (excerpt)

```
<Page.Resources>
    <Style x:Key="commonTextStyle" TargetType="TextBlock">
        <Setter Property="FontFamily" Value="Century Schoolbook" />
        <Setter Property="FontSize" Value="36" />
        <Setter Property="Foreground" Value="Black" />
        <Setter Property="Margin" Value="0 12" />
    </Style>

    <Style x:Key="paragraphTextStyle" TargetType="TextBlock"
           BasedOn="{StaticResource commonTextStyle}">
        <Setter Property="TextWrapping" Value="Wrap" />
    </Style>

    <Style x:Key="frontMatterTextStyle" TargetType="TextBlock"
           BasedOn="{StaticResource commonTextStyle}">
        <Setter Property="TextAlignment" Value="Center" />
    </Style>

    <Style x:Key="imageStyle" TargetType="Image">
        <Setter Property="Stretch" Value="None" />
        <Setter Property="HorizontalAlignment" Value="Center" />
    </Style>
</Page.Resources>
```

Notice the *Margin* value that provides a little spacing between the paragraphs. Each *TextBlock* element references either *paragraphTextStyle* (for the actual paragraphs of the book) or *frontMatterTextStyle* (for all the titles and other information that appears in the front of the book). I

could have made the style for the *Image* element an implicit style by simply removing the *x:Key* attribute and removing the *Style* attributes from the *Image* elements.

Many of the *TextBlock* elements that comprise the front matter have various local *FontSize* settings. Books generally are printed with black ink on white pages, so I hard-coded the *Foreground* of the *TextBlock* to black and set the *Background* of the *Grid* to white. To restrict the text to reasonable line lengths, the *StackPanel* is given a *MaxWidth* of 640 and centered within the *ScrollViewer*. Here's a little excerpt of the alternating *TextBlock* elements and *Image* elements:

Project: TheTaleOfTomKitten | File: BlankPage.xaml (excerpt)

```
<Grid Background="White">
    <ScrollViewer>
        <StackPanel MaxWidth="640"
                    HorizontalAlignment="Center">
            …
            <!-- pg. 38 -->
            <TextBlock Style="{StaticResource paragraphTextStyle}">
                  Mittens laughed so that she fell off the
                wall. Moppet and Tom descended after her; the pinafores
                and all the rest of Tom's clothes came off on the way down.
            </TextBlock>

            <TextBlock Style="{StaticResource paragraphTextStyle}">
                  “Come! Mr. Drake Puddle-Duck,” said Moppet
                – “Come and help us to dress him! Come and button up Tom!”
            </TextBlock>

            <Image Source="Images/tom39.jpg" Style="{StaticResource imageStyle}" />

            <!-- pg. 41 -->
            <TextBlock Style="{StaticResource paragraphTextStyle}">
                  Mr. Drake Puddle-Duck advanced in a slow
                sideways manner, and picked up the various articles.
            </TextBlock>

            <Image Source="Images/tom40.jpg" Style="{StaticResource imageStyle}" />
            …
        </StackPanel>
    </ScrollViewer>
</Grid>
```

The two   characters at the beginning of each paragraph are em-spaces. These provide a first-line indentation, which, unfortunately, is something not provided by the *TextBlock* property.

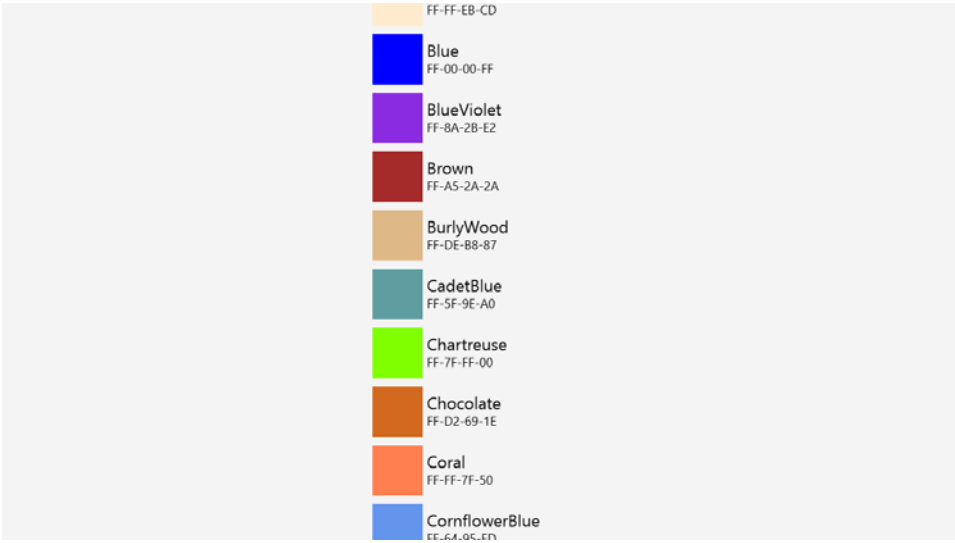
You can read this book in either landscape or portrait mode:

Mrs. Tabitha dressed Moppet and Mittens in clean pinafores and tuckers; and then she took all sorts of elegant uncomfortable clothes out of a chest of drawers, in order to dress up her son Thomas.

Tom Kitten was very fat, and he had grown; several buttons burst off. His mother sewed them on again.

Fancier *StackPanel* Items

I mentioned earlier I'd be showing you a program that displays all 141 available Windows Runtime colors with the colors as well as their names and RGB values. My first example is called ColorList1, but let's begin with the screen shot of the completed program so that you can see the goal:



This program contains a total of 283 *StackPanel* elements. Each of the 141 colors gets a pair: a vertical *StackPanel* is parent to the two *TextBlock* elements, and a horizontal *StackPanel* is parent to a *Rectangle* and the vertical *StackPanel*. All the horizontal *StackPanel* elements are then children of the main vertical *StackPanel* in a *ScrollViewer*. The XAML file is responsible for centering that *StackPanel*:

Project: ColorList1 | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <ScrollViewer>
        <StackPanel Name="stackPanel"
                    HorizontalAlignment="Center" />
    </ScrollViewer>
</Grid>
```

While enumerating through the static properties of the *Colors* class, the constructor in the code-behind file builds the nested *StackPanel* elements for each item:

Project: ColorList1 | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    public BlankPage()
    {
        this.InitializeComponent();

        IEnumerable<PropertyInfo> properties = typeof(Colors).GetTypeInfo().DeclaredProperties;

        foreach (PropertyInfo property in properties)
        {
            Color clr = (Color)property.GetValue(null);

            StackPanel vertStackPanel = new StackPanel
            {
                VerticalAlignment = VerticalAlignment.Center
```

```
            };

            TextBlock txtblkName = new TextBlock
            {
                Text = property.Name,
                FontSize = 24
            };
            vertStackPanel.Children.Add(txtblkName);

            TextBlock txtblkRgb = new TextBlock
            {
                Text = String.Format("{0:X2}-{1:X2}-{2:X2}-{3:X2}",
                                     clr.A, clr.R, clr.G, clr.B),
                FontSize = 18
            };
            vertStackPanel.Children.Add(txtblkRgb);

            StackPanel horzStackPanel = new StackPanel
            {
                Orientation = Orientation.Horizontal
            };

            Rectangle rectangle = new Rectangle
            {
                Width = 72,
                Height = 72,
                Fill = new SolidColorBrush(clr),
                Margin = new Thickness(6)
            };
            horzStackPanel.Children.Add(rectangle);
            horzStackPanel.Children.Add(vertStackPanel);
            stackPanel.Children.Add(horzStackPanel);
        }
    }
}
```

Now, there's nothing really wrong with this code, except that there are numerous ways to do it better, and by "better" I don't mean faster or more efficient but cleaner and more elegant and—most importantly—easier to maintain and modify.

Let's look at a better solution, but at the same time be aware that I won't be finished with this example until a future chapter, where you'll see not only a better way of doing it, but the *best* way of doing it.

The key to making this program better is expressing those color items—the nested *StackPanel* and *TextBlock* and *Rectangle*—in XAML. Just offhand, this doesn't seem possible. We can't put this XAML in the BlankPage.xaml file because we can't tell XAML to make 141 instances of the item unless we actually paste in 141 copies, and I suspect we're all agreed that would be the *worst* way to do it.

The ColorList2 program shows how to do it. After creating the project, I right-clicked the project name in the Solution Explorer and selected Add and New Item. In the Add New Item dialog box, I chose User Control and gave it a name of ColorItem.xaml. This process creates a pair of files:

ColorItem.xaml accompanied by a code-behind file ColorItem.xaml.cs.

The ColorItem.xaml.cs file created by Visual Studio defines a *ColorItem* class in the *ColorList2* namespace that derives from *UserControl*:

```
namespace ColorList2
{
    public sealed partial class ColorItem : UserControl
    {
        public ColorItem()
        {
            this.InitializeComponent();
        }
    }
}
```

The ColorItem.xaml file created by Visual Studio says the same thing in XAML:

```
<UserControl
    x:Class="ColorList2.ColorItem"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:ColorList2"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d"
    d:DesignHeight="300"
    d:DesignWidth="400">

    <Grid>

    </Grid>
</UserControl>
```

You've actually already seen the *UserControl* class before because *Page* derives from *UserControl*. The "user" refers not to the end-user of your application but to *you*, the programmer. Deriving from *UserControl* is the easiest way for you (the programmer) to make a custom control because you can define the visuals of the control in this XAML file. *UserControl* defines a property named *Content*, which is also the class's content property, so anything you add within the *UserControl* tags is set to this *Content* property.

Don't worry about the *d:DesignHeight* and *d:DesignWidth* properties in the ColorItem.xaml file. Those are for Microsoft Expression Blend. The actual size of this control depends on its contents.

The next step is to define the visuals of the color item in this ColorItem.xaml file:

Project: ColorList2 | File: ColorItem.xaml (excerpt)

```
<UserControl
    x:Class="ColorList2.ColorItem" … >

    <Grid>
        <StackPanel Orientation="Horizontal">
```

```
            <Rectangle Name="rectangle"
                       Width="72"
                       Height="72"
                       Margin="6" />

            <StackPanel VerticalAlignment="Center">

                <TextBlock Name="txtblkName"
                           FontSize="24" />

                <TextBlock Name="txtblkRgb"
                           FontSize="18" />
            </StackPanel>
        </StackPanel>
    </Grid>
</UserControl>
```

It's the same element hierarchy as defined in code in ColorList1, but now it's actually easily readable. The *Rectangle* and the two *TextBlock* elements all have names, so they can be referenced in the code-behind file:

Project: ColorList2 | File: ColorItem.xaml.cs (excerpt)

```
public sealed partial class ColorItem : UserControl
{
    public ColorItem(string name, Color clr)
    {
        this.InitializeComponent();

        rectangle.Fill = new SolidColorBrush(clr);
        txtblkName.Text = name;
        txtblkRgb.Text = String.Format("{0:X2}-{1:X2}-{2:X2}-{3:X2}",
                                       clr.A, clr.R, clr.G, clr.B);
    }
}
```

The code-behind file defines a constructor that accepts a color name and a *Color* value as arguments. It uses those arguments to set the appropriate properties of the *Rectangle* and two *TextBlock* elements.

Let me warn you that defining a parameterized constructor in a *UserControl* derivative is *extremely* unorthodox. A much better approach is to define properties instead, but I don't want to do that right now because these properties should really be dependency properties, and that's too involved at the moment.

Without a parameterless constructor, this *ColorItem* class cannot be instantiated in XAML. But that's OK for this program because I'm not going to try instantiating it in XAML. The BlankPage.xaml file for the ColorList2 project looks the same as the one for ColorList1. What's different is the simplicity of the code-behind file:

Project: ColorList2 | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
```

```
    public BlankPage()
    {
        this.InitializeComponent();

        IEnumerable<PropertyInfo> properties = typeof(Colors).GetTypeInfo().DeclaredProperties;

        foreach (PropertyInfo property in properties)
        {
            Color clr = (Color)property.GetValue(null);
            ColorItem clrItem = new ColorItem(property.Name, clr);
            stackPanel.Children.Add(clrItem);
        }
    }
}
```

Each *ColorItem* is instantiated with a name and *Color* and then added to the *StackPanel*.

Creating Windows Runtime Libraries

Let's create another version of this program, but this time the *ColorItem* class will be in a library that can be shared with other projects.

You can create a Visual Studio solution containing only a library project, but this is rarely done for a new library. As you're developing the code in the library, you want to test it, and it really helps to have an application project in the same solution for that purpose. It's much more common to develop libraries in conjunction with an application and then share those libraries later if desired.

So let's create a new application project named ColorList3. In the Solution Explorer, add a library project to the solution by right-clicking the solution name and selecting Add and New Project. (Or pick Add New Project from the File menu.) In the Add New Project dialog box, select Visual C# and Windows Metro Style at the left and Class Library among the available templates.

Generally. a library has a multilevel name separated by periods. This name also becomes the default namespace for that project. The library name usually begins with a company name (or its equivalent), so for this example I chose a library name of Petzold.Windows8.Controls.

In a new library, Visual Studio automatically creates a file named Class1.cs, but you can delete that. Now right-click the library project name and select Add and New Item, and in the Add New Item dialog box, select User Control and give it a name of *ColorItem*. I decided to enhance the visuals of this *ColorItem* a little beyond the one you've already seen:

Solution: ColorList3 | Project: Petzold.Windows8.Controls | File: ColorItem.xaml (excerpt)

```
<UserControl … >
    <Grid>
        <Border BorderBrush="{StaticResource ApplicationTextBrush}"
                BorderThickness="1"
                Width="336"
                Margin="6">
```

```
            <StackPanel Orientation="Horizontal">
                <Rectangle Name="rectangle"
                           Width="72"
                           Height="72"
                           Margin="6" />

                <StackPanel VerticalAlignment="Center">

                    <TextBlock Name="txtblkName"
                               FontSize="24" />

                    <TextBlock Name="txtblkRgb"
                               FontSize="18" />
                </StackPanel>
            </StackPanel>
        </Border>
    </Grid>
</UserControl>
```

Notice that I've given it a *Border* with an explicit *Width* property and a *Margin*. I chose this width empirically based on the longest color name (*LightGoldenrodYellow*). Notice also that the *BorderBrush* is set to a predefined identifier, which will be black with a light theme and white with a dark theme. Themes are set on applications rather than libraries—indeed, a library has no *App* class to set a theme—so this brush will be based on the theme of the application that uses *ColorItem*.

We still haven't touched the ColorList3 application project. Despite the fact that they're in the same solution, this application project will need a reference to the library, so right-click the References item under the ColorList3 project and select Add Reference. In the Reference Manager dialog box, at the left select Solution (indicating you want an assembly in the same solution), click Petzold.Windows8.Controls, and click OK.

There is a distinct advantage to having both these projects in the same solution: whenever you build ColorList3, Visual Studio will also rebuild the Petzold.Windows8.Controls library if it's not up to date.

The BlankPage.xaml file in ColorList3 is the same as in the previous two projects. The code-behind file needs a *using* directive for the library, but otherwise it's the same as ColorList2:

Project: ColorList3 | File: BlankPage.xaml.cs

```
using System.Collections.Generic;
using System.Reflection;
using Windows.UI;
using Windows.UI.Xaml.Controls;
using Petzold.Windows8.Controls;

namespace ColorList3
{
    public sealed partial class BlankPage : Page
    {
        public BlankPage()
        {
            this.InitializeComponent();
```

```
            IEnumerable<PropertyInfo> properties =
                            typeof(Colors).GetTypeInfo().DeclaredProperties;

            foreach (PropertyInfo property in properties)
            {
                Color clr = (Color)property.GetValue(null);
                ColorItem clrItem = new ColorItem(property.Name, clr);
                stackPanel.Children.Add(clrItem);
            }
        }
    }
}
```

Here's the result:

The Wrap Alternative

Now let's use that library in another project. There are three ways to do it:

Method 1: Add a new application project to the same solution as the existing library: the ColorList3 solution, in this example. This is the easiest approach, and it certainly makes sense if the two applications are related some way.

Instead, I'm going to use one of the other two methods. These two methods both involve creating a new solution and application project, which I'll call ColorWrap. This project needs a reference to the Petzold.Windows8.Control library.

Method 2: Right-click the References item in the ColorWrap project, and select Add Reference. In

the left column of the Reference Manager, select Browse, and then click the Browse button in the lower right corner. This will allow you to browse to the directory location where the Petzold.Windows8.Controls.dll file is located (which is the bin/Debug directory of the Petzold.Windows8.Controls project in the ColorList3 solution), and you can select that DLL.

The disadvantage to this method is that you're assuming that the library is complete and finished and that you won't need to make any changes. You're referencing a DLL rather than the project with its source code. However, in my experience the *really* big disadvantage to this method is that it doesn't work quite right with the current release of Windows 8 when there are XAML files involved.

That leaves us with:

Method 3: In the ColorWrap solution, right-click the solution name and select Add and Existing Project. The existing project you want to add is the library. In the Add Existing Project dialog box, navigate to the Petzold.Windows8.Controls.csproj file. This is the C# project file maintained by Visual Studio in the ColorList3 solution. Select that. The library project is not copied! Instead, only a reference is created to that library project. Regardless, Visual Studio can still determine if the library needs to be rebuilt, and it performs that rebuild if necessary.

Now the Petzold.Windows8.Controls project is part of the ColorWrap solution, but the ColorWrap application project still needs a reference to the library. Right-click the References section under the ColorWrap project and select the library from the solution, just as you did in ColorList3.

It could be that you have two instances of Visual Studio running, perhaps with the ColorList3 and ColorWrap solutions loaded, both of which let you make changes to the Petzold.Windows8.Controls library. That's generally OK as long as you save or compile after making changes. If the same file is open in both instances of Visual Studio and you make changes to that file, the other instance of Visual Studio will notify you of changes when that file is saved to disk.

With those preliminaries out of the way, let's focus on the ColorWrap program, which demonstrates how to display these colors with a *VariableSizedWrapGrid* panel. Despite the name of this panel, it *really* wants all the items to be the same size, and that's why I added the explicit *Width* to the *Border* in *ColorItem*.

Like *StackPanel*, *VariableSizedWrapGrid* has an *Orientation* property and the default is *Vertical*. The first items in the *Children* collection are displayed in a column. The difference is that *VariableSizedWrapGrid* will use multiple columns, just like the Windows 8 start screen. This means that the default VariableSizedWrapGrid must be horizontally scrolled, so *ScrollViewer* properties must be set accordingly. Here's the XAML file:

Project: ColorWrap | File: BlankPage.xaml (excerpt)

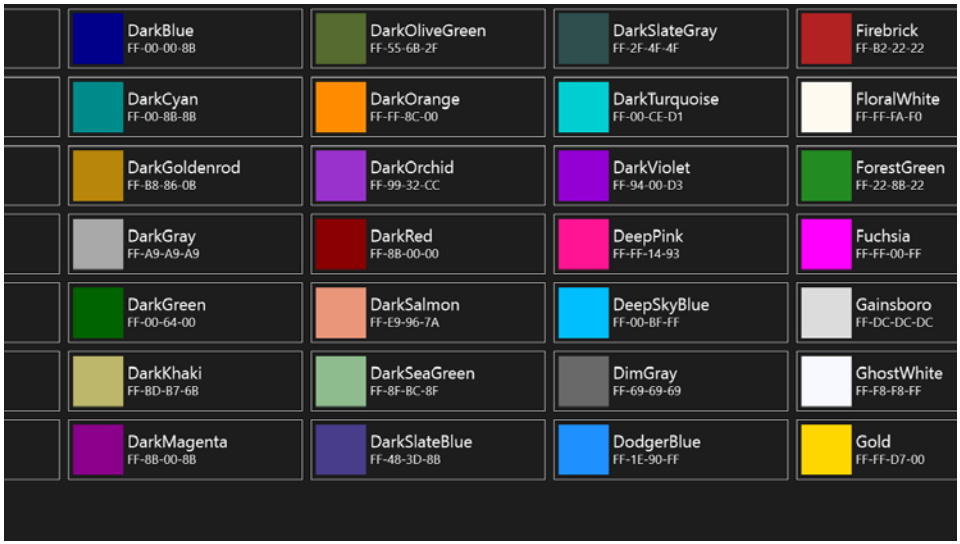
```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <ScrollViewer HorizontalScrollBarVisibility="Visible"
                  VerticalScrollBarVisibility="Disabled">
        <VariableSizedWrapGrid Name="wrapPanel" />
    </ScrollViewer>
</Grid>
```

The code-behind file is similar to the previous program except that now it puts the items into *wrapPanel*:

Project: ColorWrap | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    public BlankPage()
    {
        this.InitializeComponent();

        IEnumerable<PropertyInfo> properties = typeof(Colors).GetTypeInfo().DeclaredProperties;

        foreach (PropertyInfo property in properties)
        {
            Color clr = (Color)property.GetValue(null);
            ColorItem clrItem = new ColorItem(property.Name, clr);
            wrapPanel.Children.Add(clrItem);
        }
    }
}
```

And here it is:



The *Canvas* and Attached Properties

The final *Panel* derivative I'll discuss in this chapter is the *Canvas*. In one sense, *Canvas* is the most "traditional" type of panel because it allows you to position elements at precise pixel locations. However, if you've scoured the properties defined by *UIElement* and *FrameworkElement* searching for a property named *Location* or *Position* or *X* or *Y*, you haven't found one. Such a property does not exist

because it doesn't have a generalized applicability. We've managed to make it this far without specifying pixel locations for positioning elements, and the only time one is needed is when the element is a child of a *Canvas*.

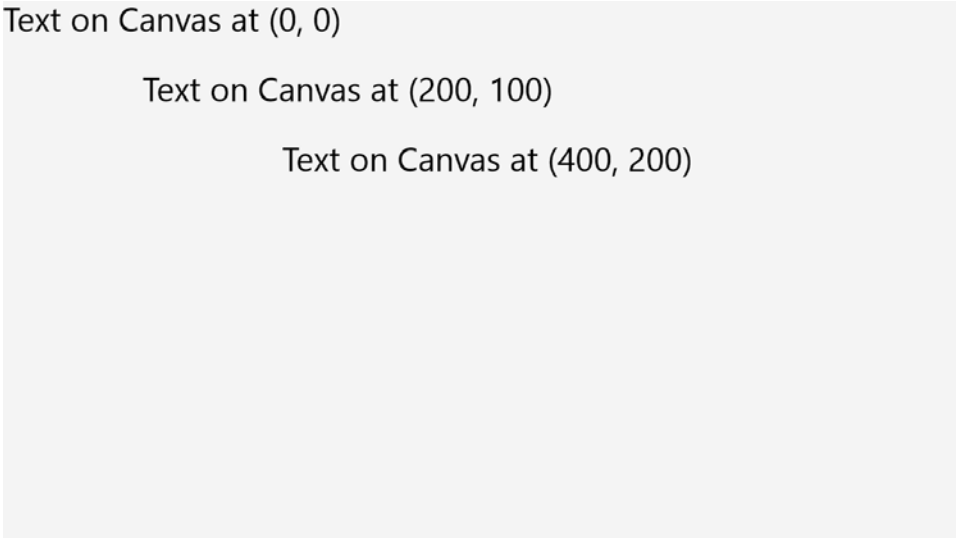
For that reason, *Canvas* itself defines the properties used to position elements relative to itself. These are a very special type of properties known as *attached properties*, and they are a subset of dependency properties. The attached properties defined by one class (*Canvas* in this example) are actually set on instances of other classes (children of the *Canvas*, in this case). The objects on which you set an attached property don't need to know what that property does or where it came from.

Let's see how this works. The TextOnCanvas project has a XAML file that contains a *Canvas* within the standard *Grid*. (You can alternatively replace the *Grid* with the *Canvas*.) The *Canvas* contains three *TextBlock* children:

Project: TextOnCanvas | File: BlankPage.xaml (excerpt)

```
<Page
    x:Class="TextOnCanvas.BlankPage"
    …
    FontSize="48">

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <Canvas>
            <TextBlock Text="Text on Canvas at (0, 0)"
                       Canvas.Left="0"
                       Canvas.Top="0" />

            <TextBlock Text="Text on Canvas at (200, 100)"
                       Canvas.Left="200"
                       Canvas.Top="100" />

            <TextBlock Text="Text on Canvas at (400, 200)"
                       Canvas.Left="400"
                       Canvas.Top="200" />
        </Canvas>
    </Grid>
</Page>
```

Here's the (rather unexciting) result:



Look at that markup again, and take special note of the strange syntax:

```
<TextBlock Text="Text on Canvas at (200, 100)"
           Canvas.Left="200"
           Canvas.Top="100" />
```

Judging from their names, the *Canvas.Left* and *Canvas.Top* attributes appear to be defined by the *Canvas* class, and yet they are set on the children of the *Canvas* to indicate their positions. Attributes with class and property names like this are always attached properties.

The funny thing is, *Canvas* actually doesn't define any properties named *Left* and *Top*! It defines properties and methods with similar names but not those names exactly.

The nature of these attached properties might become a little clearer by examining how they are set in code. The XAML file for the TapAndShowPoint program contains only a named *Canvas* in the standard *Grid*:

Project: TapAndShowPoint | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <Canvas Name="canvas" />
</Grid>
```

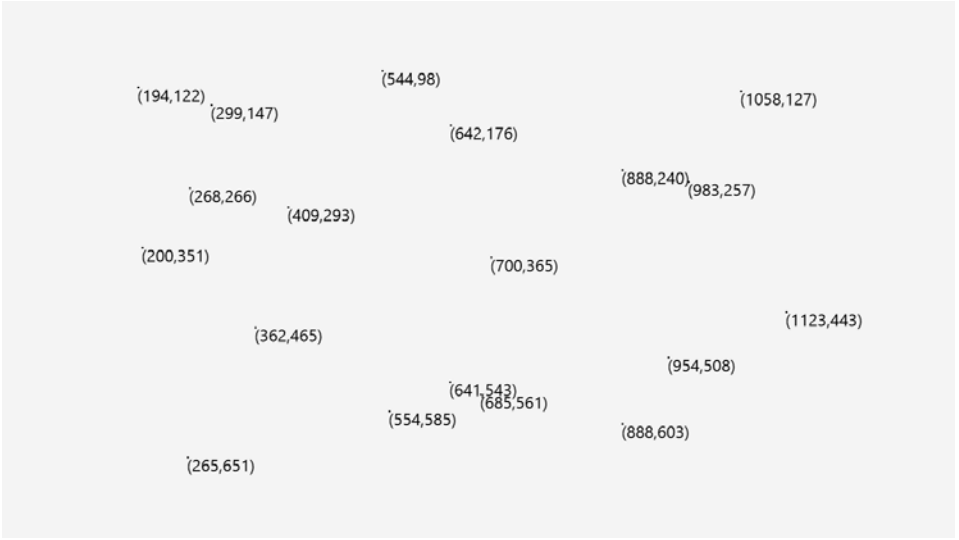
Everything else is the responsibility of the code-behind file. It overrides the *OnTapped* method to create a dot (an *Ellipse* element actually) and a *TextBlock*, both of which it adds to the *Canvas* at the point where the screen was tapped:

Project: TapAndShowPoint | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    public BlankPage()
    {
        this.InitializeComponent();
```

```
    }

    protected override void OnTapped(TappedRoutedEventArgs args)
    {
        Point pt = args.GetPosition(this);

        // Create dot
        Ellipse ellipse = new Ellipse
        {
            Width = 3,
            Height = 3,
            Fill = this.Foreground
        };

        Canvas.SetLeft(ellipse, pt.X);
        Canvas.SetTop(ellipse, pt.Y);
        canvas.Children.Add(ellipse);

        // Create text
        TextBlock txtblk = new TextBlock
        {
            Text = String.Format("({0})", pt),
            FontSize = 24,
        };

        Canvas.SetLeft(txtblk, pt.X);
        Canvas.SetTop(txtblk, pt.Y);
        canvas.Children.Add(txtblk);

        args.Handled = true;
        base.OnTapped(args);
    }
}
```

As you tap on the screen, the dots and text appear at the tap point:



Here's how the position of the dot is specified in code before it's added to the *Children* collection of the *Canvas*:

```
Canvas.SetLeft(ellipse, pt.X);
Canvas.SetTop(ellipse, pt.Y);
canvas.Children.Add(ellipse);
```

The order doesn't matter: you could add the element to the *Canvas* first, and then set its position. The *Canvas.SetLeft* and *Canvas.SetTop* static methods play the same role here as the *Canvas.Left* and *Canvas.Top* attributes in XAML. They let you specify a coordinate point where a particular element is to be positioned.

If you make the *Ellipse* a little larger, you'll see a little flaw in the approach I've used. The *Canvas.SetLeft* and *Canvas.SetTop* methods position the upper-left corner of the *Ellipse* at the specified point rather than its center. If you want the center of the *Ellipse* at the point *pt*, you'll want to subtract half its width from *pt.X* and half its height from *pt.Y*.

I mentioned that *Canvas* doesn't define *Left* and *Top* properties specifically. Instead, Canvas defines static *SetLeft* and *SetTop* methods as well as static properties of type *DependencyProperty*:

```
public static DependencyProperty LeftProperty { get; }
public static DependencyProperty TopProperty { get; }
```

As you'll see in a later chapter, these are special types of dependency properties in that they can be set on elements other than *Canvas*.

Instead of setting the position of an element by calling *Canvas.SetLeft* and *Canvas.SetTop*, you can instead set the position by calling *SetValue* on the child element and referencing the static *DependencyProperty* objects:

```
ellipse.SetValue(Canvas.LeftProperty, pt.X);
```

```
ellipse.SetValue(Canvas.TopProperty, pt.Y);
```

These statements are exactly equivalent to the *Canvas.SetLeft* and *Canvas.SetTop* calls. In fact, although I have never seen the internal source code of the *Canvas* class, I can practically guarantee you that the *SetLeft* and *SetTop* static methods in *Canvas* are defined like this:

```
public static void SetLeft(DependencyObject element, double value)
{
    element.SetValue(LeftProperty, value);
}
public static void SetTop(DependencyObject element, double value)
{
    element.SetValue(TopProperty, value);
}
```

These methods show very clearly how the dependency property is actually being set on the element rather than the *Canvas*.

Canvas also defines *GetLeft* and *GetTop* methods:

```
public static double GetLeft(DependencyObject element)
{
    return (double)element.GetValue(LeftProperty);
}
public static double GetTop(DependencyObject element)
{
    return (double)element.GetValue(TopProperty);
}
```

The *Canvas* class uses these methods internally to obtain the left and top settings on each of its children so that it can position them during the layout process.

You will recall that the *SetValue* and *GetValue* methods are defined by *DependencyObject*, which is a very basic class in the Windows Runtime. A property like *FontSize* is actually defined in terms of the static dependency property:

```
public double FontSize
{
    set { SetValue(FontSizeProperty, value); }
    get { return (double)GetValue(FontSizeProperty); }
}
```

The static *SetLeft*, *SetTop*, *GetLeft*, and *GetTop* methods suggest that the dependency property system involves a dictionary of sorts. The *SetValue* method allows an attached property like *Canvas.LeftProperty* to be stored in an element that has no knowledge of this property or its purpose. *Canvas* can later retrieve this property to determine where the child should appear relative to itself.

The Z-Index

Canvas has a third attached property that you can set in XAML with the attribute *Canvas.ZIndex*. The

"Z" in *ZIndex* refers to a three-dimensional coordinate system, where the Z axis extends out of the screen towards the user.

When sibling elements overlap, they are normally displayed in the order they appear in the visual tree, which means that elements early in a panel's *Children* collection can be covered by elements later in the *Children* collection. For example, consider the following:

```
<Grid>
    <TextBlock Text="Blue Text" Foreground="Blue" FontSize="96" />
    <TextBlock Text="Red Text" Foreground="Red" FontSize="96" />
</Grid>
```

The red text obscures part of the blue text.

You can override that behavior with the *Canvas.ZIndex* attached property, and the weird thing is this: it works with all panels, and not just *Canvas*. To make the blue text appear on top of the red text, give it a higher z-index:

```
<Grid>
    <TextBlock Text="Blue Text" Foreground="Blue" FontSize="96" Canvas.ZIndex="1" />
    <TextBlock Text="Red Text" Foreground="Red" FontSize="96" Canvas.ZIndex="0" />
</Grid>
```

Canvas Weirdness

Much of what I've described about layout doesn't apply to the *Canvas*. Layout within a *Canvas* is always child-driven. The *Canvas* always offers its children an infinite size, which means that each child sets a natural size for itself and that's the only space the child occupies. *HorizontalAlignment* and *VerticalAlignment* settings have no effect on a child of a *Canvas*. Likewise, the *Stretch* property of *Image* has no effect when the *Image* is a child of a *Canvas*: *Image* always displays the bitmap in its pixel size. *Rectangle* and *Ellipse* shrink to nothing in a *Canvas* unless given an explicit width and height.

Although *HorizontalAlignment* and *VerticalAlignment* have no effect on a child of the *Canvas*, they do have an effect when set on the *Canvas* itself. With other panels, when you set the alignment properties to something other than *Stretch*, the panel becomes as small as possible while still encompassing its children. The *Canvas*, however, is different. Set *HorizontalAlignment* and *VerticalAlignment* to values other than *Stretch*, and the *Canvas* shrinks to nothing regardless of its children.

Even when the *Canvas* shrinks down to a zero size, the display of its children is not affected. Conceptually, the *Canvas* is more like a reference point than a container.

You can use this characteristic of the *Canvas* to your advantage. For example, suppose you try to display a *TextBlock* in a *Grid* that is obviously too small for it:

```
<Grid Width="200" Height="100">
    <TextBlock Text="Text in a Small Grid" FontSize="144" />
```

```
</Grid>
```

The *TextBlock* is clipped to the dimensions of the *Grid*. You could make the *Grid* larger of course, but you might be stuck with this *Grid* size, perhaps because of other child elements. Still, you want the *TextBlock* to be aligned with these other elements without being clipped to the *Grid*.

The extremely simple solution is to put a *Canvas* in the *Grid* and put the *TextBlock* in that *Canvas*:

```
<Grid Width="200" Height="100">
    <Canvas>
        <TextBlock Text="Text in a Small Grid" FontSize="144" />
    </Canvas>
</Grid>
```

Even though the *Canvas* is now clipped to the size of the *Grid*, the *TextBlock* is not. The *TextBlock* is still where you want it—aligned with the upper-left corner of the *Grid*—but it's now displayed without any clipping.

It's a very simple technique that can be very useful when you need it.

Chapter 5

Control Interaction

Early on in this book I made a distinction between classes that derive from *FrameworkElement* and those that derive from *Control*. I've tended to refer to *FrameworkElement* derivatives (such as *TextBlock* and *Image*) as "elements" to preserve this distinction, but a deeper explication is now required.

The title of this chapter might suggest that elements are for presentation and controls are for interaction, but that's not necessarily so. *UIElement* defines all the user input events for touch, mouse, stylus, and keyboard, which means that elements as well as controls can interact with the user in very sophisticated ways.

Nor are elements deficient in layout, styling, or data binding capabilities. It's the *FrameworkElement* class that defines layout properties such as *Width*, *Height*, *HorizontalAlignment*, *VerticalAlignment*, and *Margin*, as well as the *Style* property and the *SetBinding* method.

The *Control* Difference

Visually and functionally, *FrameworkElement* derivatives are primitives—atoms, so to speak—while *Control* derivatives are assemblages of these primitives, or molecules in this analogy. A *Button* is actually constructed from a *Border* and a *TextBlock* (in many cases). A *Slider* consists of a couple of *Rectangle* elements with a *Thumb*, which itself is a *Control* probably built from a *Rectangle*. Anything that has visual content beyond text, a bitmap, or vector graphics is almost certainly a *Control* derivative.

Consequently, one of the most important properties defined by *Control* is called *Template*. As I'll demonstrate in a future chapter, this property allows you to completely redefine the appearance of a control by defining a visual tree of your own invention. It makes sense to visually redefine a *Button* because (for example) you might want it to be round rather than rectangular so that it looks right in an application bar. It makes no sense to visually redefine a *TextBlock* or *Image* because there's nothing you can do with it beyond the text or bitmap itself. If you want to *add* something to a *TextBlock* or *Image*, you're defining a *Control* because you're constructing a visual tree that includes the element primitive.

Although you can derive a custom class from *FrameworkElement*, there is little you can do with the result. You can't give it any visuals. But when you derive from *Control*, you give your class a default visual appearance by defining a visual tree in XAML.

For use by derived classes, *Control* defines a bunch of properties that the *Control* class itself does not need. These are properties mostly associated with *TextBlock* (*CharacterSpacing*, *FontFamily*,

FontSize, *FontStretch*, *FontStyle*, *FontWeight*, and *Foreground*) and *Border* (*Background*, *BorderBrush*, *BorderThickness*, and *Padding*). Not every *Control* derivative has text or a border, but if you need those properties when creating a new control or creating a new template for an existing control, they are conveniently provided. *Control* also provides two new properties named *HorizontalContentAlignment* and *VerticalContentAlignment* for purposes of defining control visuals.

A *Control* derivative often defines a few of its own properties and its own events. Commonly, a *Control* derivative will process user-input events from the pointer, mouse, stylus, and keyboard and will convert that input into a higher-level event. For example, the *ButtonBase* class (from which all the buttons derive) defines a *Click* event. The *Slider* defines a *ValueChanged* event indicating when its *Value* property changes. The *TextBox* defines a *TextChanged* event indicating when its *Text* property changes.

It turns out that in real life, *Control* derivatives really *do* interact more with users, so the title of this chapter is accurate. For the convenience of working with user input, *Control* provides protected virtual methods corresponding to all the user-input events defined by *UIElement*. For example, *UIElement* defines the *Tapped* event, but *Control* defines the protected virtual method *OnTapped*. *Control* also defines an *IsEnabled* property so that controls can avoid user input if input is not currently applicable, and it defines an *IsEnabledChanged* event that is fired when the property changes. This is the only public event actually defined by *Control*.

The idea of a control having "input focus" is still applicable in Windows 8. When a control has the input focus, the user expects that particular control to get most keyboard events. (Of course, some keyboard events, such as the Windows key, transcend input focus.) For this purpose, *Control* defines a *Focus* method, as well as *OnGotFocus* and *OnLostFocus* virtual methods.

In connection with keyboard focus is the idea of being able to navigate among controls by using the keyboard Tab key. *Control* provides for this by defining *IsTabStop*, *TabIndex*, and *TabNavigation* properties.

Many *Control* derivatives are in the *Windows.UI.Xaml.Controls* namespace, but a few are in the *Windows.UI.Xaml.Controls.Primitives* namespace. The latter namespace is generally reserved for those controls that usually appear only as parts of other controls, but that's a suggestion rather than a restriction.

Most *Control* derivatives derive directly from *Control*, but four important classes derive from *Control* to define their own subcategories of controls. Here they are:

Object
DependencyObject
UIElement
FrameworkElement
Control
ContentControl
ItemsControl

RangeBase
UserControl

ContentControl—from which important classes like *Button*, *ScrollViewer*, and *AppBar* derive—seemingly does little more than define a property named *Content* of type *object*. For a *Button*, for example, this *Content* property is what you use to set whatever you want to appear inside the *Button*. Most often this is text or a bitmap, but you can also use a panel that contains other content.

It is interesting that the *Content* property of *ContentControl* is of type *object* rather than *UIElement*. There's a good reason for that. You can actually put pretty much any type of object you want as the content of a *Button*, and you can supply a template (in the form of a visual tree) that tells the *Button* how to display this content. This feature is not so much used for *Button*, but it's used a great deal for items in *ItemsControl* derivatives. I'll show you how to define a content template in a future chapter.

ItemsControl is the parent class to a bunch of controls that display collections of items. Here you'll find the familiar *ListBox* and *ComboBox* as well as the new Windows 8 controls *GridView* and *ListView*. This is such an important category of controls that a whole future chapter will be devoted to it.

There are a couple ways to create custom controls. The really simple way is by defining a *Style* for the control, but more extensive visual changes require a template. In some cases you can derive from an existing control to add some features to it, or you can derive from *ContentControl* or *ItemsControl* if these controls provide features you need.

But one of the most common ways to create a custom control is by deriving from *UserControl*. This is not the approach you'll use if you want to market a custom control library, but it's great for controls that you use yourself within the context of an application.

The *Slider* for Ranges

The final important parent class that derives from *Control* is *RangeBase*, which has three derivatives: *ProgressBar*, *ScrollBar*, and *Slider*.

Which of these is not like the others? Obviously *ProgressBar*, which exists in this hierarchy mainly to inherit several properties from *RangeBase*: *Minimum*, *Maximum*, *SmallChange*, *LargeChange*, and *Value*. In every *RangeBase* control, the *Value* property takes on values of type *double* ranging from *Minimum* through *Maximum*. With the *ScrollBar* and *Slider*, the *Value* property changes when the user manipulates the control; with *ProgressBar*, the *Value* property is set programmatically to indicate the progress of a lengthy operation.

ProgressBar has an indeterminate mode to display a row of dots that skirt across the screen, but also available is *ProgressRing*, which displays a series of dots that parade around in a circle.

In the quarter-century evolution of Windows, the *ScrollBar* has slipped from its high perch in the control hierarchy, and it's commonly seen today only in a *ScrollViewer* control. Try to instantiate the Windows Runtime version of *ScrollBar*, and you won't even see it. If you want to use *ScrollBar*, you'll

have to supply a template for it. Like *RangeBase*, *ScrollBar* is defined in the *Windows.UI.Xaml.Controls.Primitives* namespace, indicating that it's not something application programmers normally use.

For virtually all applications involving choosing from a range of values, *ScrollBar* has been replaced with *Slider*, and with touch interfaces, *Slider* has become simpler than ever. In its default manifestation, *Slider* has no arrows. It simply jumps to the value corresponding to the point where you touch the *Slider* or drag your finger or mouse.

The *Value* property of the *Slider* can change either programmatically or through user manipulation. To obtain a notification when the *Value* property changes, attach an event handler for the *ValueChanged* event, such as shown in the SliderEvents project:

Project: SliderEvents | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <StackPanel>
        <Slider ValueChanged="OnSliderValueChanged" />

        <TextBlock HorizontalAlignment="Center"
                   FontSize="48" />

        <Slider ValueChanged="OnSliderValueChanged" />

        <TextBlock HorizontalAlignment="Center"
                   FontSize="48" />
    </StackPanel>
</Grid>
```

Both *Slider* controls here share the same event handler. The idea behind this simple program is that the current *Value* of each *Slider* is displayed by the *TextBlock* below it. This might be considered somewhat challenging when you notice that nothing in this XAML file is assigned a name. However, the event handler makes a few assumptions. It assumes that the parent to the *Slider* is a *Panel*, and the next child in this *Panel* is a *TextBlock*:

Project: SliderEvents | File: BlankPage.xaml.cs (excerpt)

```
void OnSliderValueChanged(object sender, RangeBaseValueChangedEventArgs args)
{
    Slider slider = sender as Slider;
    Panel parentPanel = slider.Parent as Panel;
    int childIndex = parentPanel.Children.IndexOf(slider);
    TextBlock txtblk = parentPanel.Children[childIndex + 1] as TextBlock;
    txtblk.Text = args.NewValue.ToString();
}
```

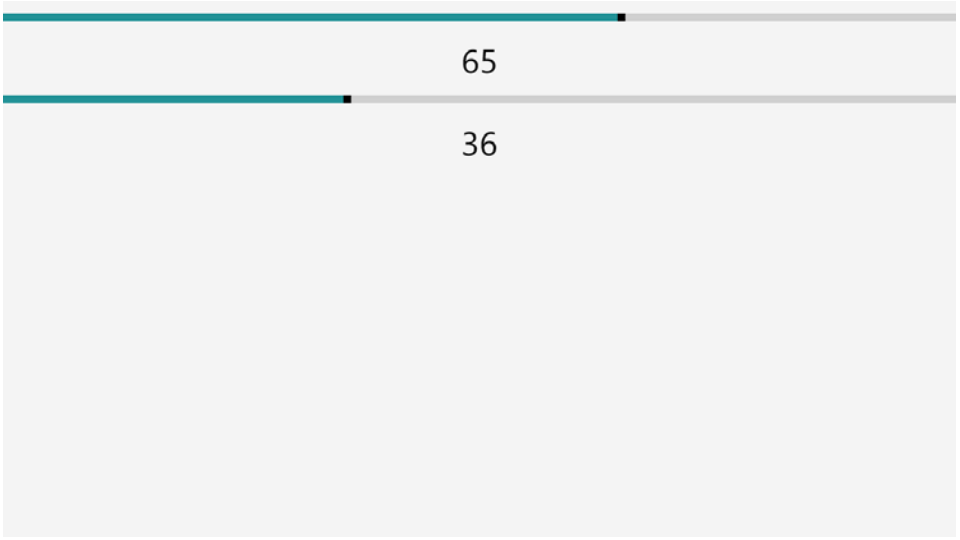
This little bit of "trickery" is merely to demonstrate that there's more than one way to access elements in the visual tree. In the final step, the *Text* property of the *TextBlock* is assigned the *NewValue* argument from the event arguments, converted to a string. Equally valid would be using the *Value* property of the *Slider*:

```
txtblk.Text = slider.Value.ToString();
```

Although *RangeBaseValueChangedEventArgs* derives from *RoutedEvent*, this is not a routed event. The event does not travel up the visual tree. The *sender* argument is always the *Slider*, and the *OriginalSource* property of the event arguments is always *null*.

When you run the program, you'll notice that the *TextBlock* elements initially display nothing. The *ValueChanged* event is not fired until *Value* actually changes from its default value of zero.

As you touch a *Slider* or click it with a mouse, the value jumps to that position. You can then sweep your finger or mouse pointer back and forth to change the value. As you manipulate the *Slider* controls, you'll see that they let you select values from 0 to 100, inclusive:



This default range is a result of the default values of the *Minimum* and *Maximum* properties, which are 0 and 100, respectively.. Although the *Value* property is a *double*, it takes on integral values as a result of the default *StepFrequency* property, which is 1.

By default the *Slider* is oriented horizontally, but you can switch to vertical with the *Orientation* property. The height of the slider area cannot be changed (except if you redefine the visuals with a template). The total height of the control in layout includes a bit more space. In layout, the default height of a horizontal *Slider* is 60 pixels; the default width of a vertical *Slider* is 45 pixels. In use, these dimensions are adequate for touch purposes.

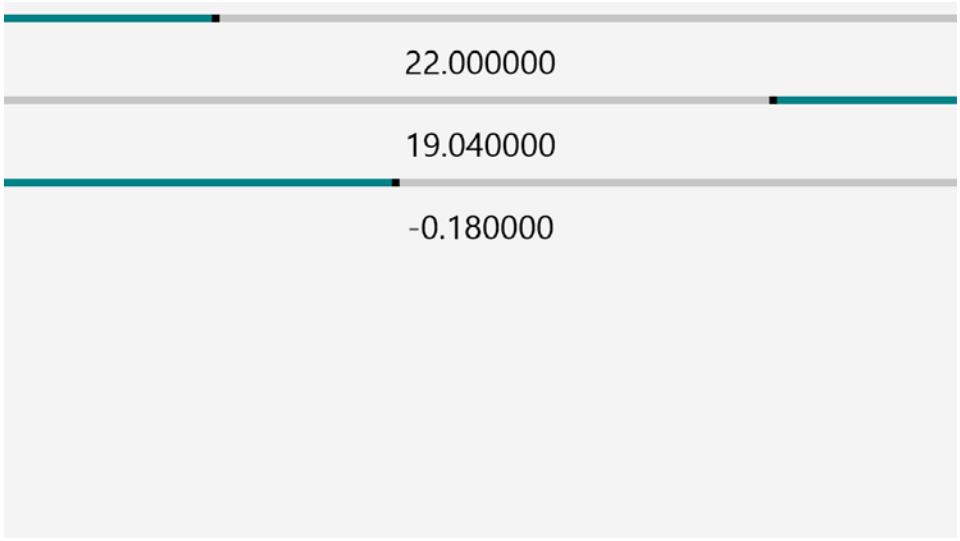
If you repeatedly press the Tab key while this program is running, you can change the keyboard input focus from one *Slider* to another and then use the keyboard arrow keys to make the value go up or down. Pressing Home and End shoots to the minimum and maximum values.

Some other variations are illustrated in the SliderBindings project, where all the updating logic is incorporated right in the XAML file. Three *Slider* controls are instantiated in a *StackPanel* and alternated with *TextBlock* elements with bindings to the *Value* properties of each *Slider*. An implicit style for the *TextBlock* is defined to reduce markup:

Project: SliderBindings | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <Grid.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="FontSize" Value="48" />
            <Setter Property="HorizontalAlignment" Value="Center" />
        </Style>
    </Grid.Resources>

    <StackPanel>
        <Slider Name="slider1" />

        <TextBlock Text="{Binding ElementName=slider1, Path=Value}" />

        <Slider Name="slider2"
                IsDirectionReversed="True"
                StepFrequency="0.01" />

        <TextBlock Text="{Binding ElementName=slider2, Path=Value}" />

        <Slider Name="slider3"
                Minimum="-1"
                Maximum="1"
                StepFrequency="0.01"
                SmallChange="0.01"
                LargeChange="0.1" />

        <TextBlock Text="{Binding ElementName=slider3, Path=Value}" />
    </StackPanel>
</Grid>
```

Bindings obtain initial values and don't wait for the first *ValueChanged* event to be fired. Obviously, the binding is using a somewhat different approach to converting the *double* values to strings:



The markup for the second *Slider* sets the *StepFrequency* property to 0.01 and also sets *IsDirectionReversed* to *true* so that the minimum value of 0 occurs when the thumb is positioned to the far right. It's rather rare to set *IsDirectionReversed* to *true* for horizontal sliders but more common for vertical sliders. The default vertical slider has a minimum value when the slider is all the way down, and for some purposes that should be a maximum value.

For that second *Slider*, however, the keyboard arrow keys change the value in increments of 1 rather than the *StepFrequency* of 0.01. The keyboard interface is governed by the *SmallChange* property, which by default is 1.

The third *Slider* has a range from –1 to 1. When the *Slider* is first displayed, the thumb is in the center at the default *Value* of 0. I've set both *StepFrequency* and *SmallChange* to 0.01, and *LargeChange* to 0.1, but I've found no way to trigger the *LargeChange* jump with either the mouse or keyboard.

The *Slider* class defines *TickFrequency* and *TickPlacement* properties to display tick marks adjacent to the *Slider*, but it is my experience that with the current release of Windows 8, these tick marks are barely visible. Something is wrong with the way they're handling color.

If the *Background* and *Foreground* properties of the *Slider* are set, the *Slider* uses *Foreground* for the slider area associated with the minimum value and *Background* for the area associated with the maximum value, but it switches to default colors when the *Slider* is being manipulated or when the mouse hovers overhead.

As we begin creating more *Slider* controls, it becomes necessary to find a better way to lay them out on the page. It's time to get familiar with the *Grid*.

The Grid

The *Grid* probably seems like a familiar friend at this point because it's been in almost every program in this book, but obviously we haven't gotten to know it in any depth. Many of the programs in the remainder of this book will use the *Grid* not in its single-cell mode but with actual rows and columns.

The *Grid* has a superficial resemblance to the HTML *table*, but it's quite different. The *Grid* doesn't have any facility to define borders or margins for individual cells. It is strictly for layout purposes. Any sprucing up for presentation must occur on the parent or children elements. For example, the *Grid* can be in a *Border*, and *Border* elements can adorn the contents of the individual *Grid* cells.

The number of rows and columns in a *Grid* must be explicitly indicated; the *Grid* cannot determine this information by the number of children. Children of the *Grid* generally go in a particular cell, which is an intersection of a row and column, but children can also span multiple rows and columns.

Although the numbers of rows and columns can be changed programmatically at run time, it's not often done. Most common is to fix the number of desired rows and columns in the XAML file. This is

accomplished with objects of type *RowDefinition* and *ColumnDefinition* added to two collections defined by *Grid* called *RowDefinitions* and *ColumnDefinitions*.

The size of each row and column can be defined in one of three ways:

- An explicit row height or column width in pixels
- *Auto*, meaning based on the size of the children
- Asterisk (or star), which allocates remaining space proportionally

In XAML, property element syntax is used to fill the *RowDefinitions* and *ColumnDefinitions* collections, so a typical *Grid* looks like this:

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="55" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="10*" />
        <ColumnDefinition Width="20*" />
        <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>

    <!-- Children go here -->

</Grid>
```

Notice that the *Grid* collection properties are named *RowDefinitions* and *ColumnDefinitions* (plural) but they contain objects of type *RowDefinition* and *ColumnDefinition* (singular). You can omit the *RowDefinitions* or *ColumnDefinitions* for a *Grid* with only one row or one column.

This particular *Grid* has three rows and four columns, and it shows the various ways that the size of the rows and columns can be defined. A number by itself indicates a width (or height) in pixels. Explicit row heights and column widths are not generally used as much as the other two options.

The word *Auto* means to let the child decide. The calculated height of the row (or width of the column) is based on the maximum height (or width) of the children in that row (or column).

As in HTML, the asterisk (pronounced "star") directs the *Grid* to allocate the available space. In this *Grid*, the height of the third row is calculated by subtracting the height of the first and second rows from the total height of the *Grid*. For the columns, the second and third columns are allocated the remaining space calculated by subtracting the widths of the first and fourth columns from the total width of the *Grid*. The numbers before the asterisks indicates proportions, and here they mean that the third column gets twice the width of the second column.

The star values are applicable only when the size of the *Grid* is parent-driven! For example, suppose

that this *Grid* is a child of a *StackPanel* with a vertical orientation. The *StackPanel* offers to the *Grid* an unconstrained infinite height. How can the *Grid* allocate that infinite height to its middle row? It cannot. The asterisk specification degenerates to *Auto*.

Similarly, if a *Grid* is a child of a *Canvas* and the *Grid* is not given an explicit *Height* and *Width*, all the star specifications degenerate to *Auto*. The same thing happens to a *Grid* that does not have default *Stretch* values of *HorizontalAlignment* and *VerticalAlignment*. In the *Grid* example shown above, the second column may actually become wider than the third if that's what the sizes of the children in those columns dictate.

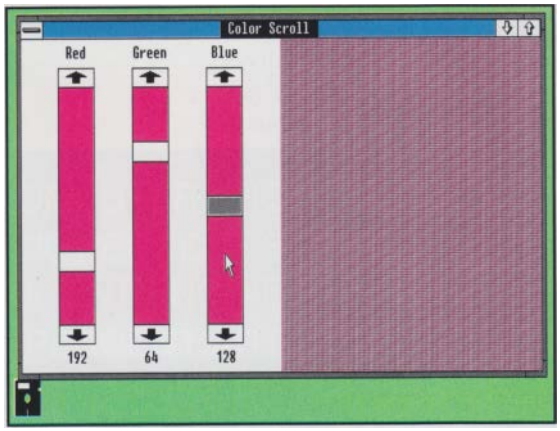
However, if you have no *RowDefinition* objects with a star specification, the height of the *Grid* is child-driven. The *Grid* can go in a vertical *StackPanel* or *Canvas* or be given a nondefault *VerticalAlignment* without weirdness happening.

The *Height* property of *RowDefinition* and the *Width* property of *ColumnDefinition* are both of type *GridLength*, a structure defined in *Windows.UI.Xaml* that lets you specify *Auto* or star sizes from code. *RowDefinition* also defines *MinHeight* and *MaxHeight* properties, and *ColumnDefinition* defines *MinWidth* and *MaxWidth*. These are all of type *double* and indicate minimum and maximum sizes in pixels. You can obtain the actual sizes with the *ActualHeight* property of *RowDefinition* and the *ActualWidth* property of *ColumnDefinition*.

Grid also defines four attached properties that you set on the children of a *Grid*: *Grid.Row* and *Grid.Column* have default values of 0, and *Grid.RowSpan* and *Grid.ColumnSpan* have default values of 1. This is how you indicate the cell in which a particular child resides and how many rows and columns it spans. A cell can contain more than one element.

You can nest a *Grid* within a *Grid* or put other panels in *Grid* cells, but the nesting of panels could degrade layout performance, so watch out if a deeply nested element is changing size based on an animation or if children are frequently being added to or removed from *Children* collections. You don't want the layout of your page being recalculated at the video frame rate!

In Chapter 3, "Basic Event Handling," I presented a Windows 8 version of WHATSIZE, the first program to appear in a magazine article about Windows programming. The third article about Windows Programming was in the May 1987 issue of *Microsoft Systems Journal* and featured a program called COLORSCR ("color scroll"). Here it is as it appeared in that article running under a beta version of Windows 2:



Manipulate the scrollbars to mix red, green, and blue values, and you'd see the result at the right. (In those days, most graphics displays didn't have full ranges of color, so dithering was used to approximate colors not renderable by the device.) The value of each scrollbar is also displayed beneath the scrollbar. The program performed a rather crude (and heavily arithmetic) attempt at dynamic layout, even changing the width of the scrollbars when the window size changed.

This seems like an ideal program to demonstrate a simple *Grid*. Considering the six instances of *TextBlock* and three instances of *Slider* required, the XAML file in the SimpleColorScroll project starts off with two implicit styles:

Project: SimpleColorScroll | File: BlankPage.xaml (excerpt)

```
<Page.Resources>
    <Style TargetType="TextBlock">
        <Setter Property="Text" Value="00" />
        <Setter Property="FontSize" Value="24" />
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="Margin" Value="0 12" />
    </Style>

    <Style TargetType="Slider">
        <Setter Property="Orientation" Value="Vertical" />
        <Setter Property="IsDirectionReversed" Value="True" />
        <Setter Property="Maximum" Value="255" />
        <Setter Property="HorizontalAlignment" Value="Center" />
    </Style>
</Page.Resources>
```

I've decided to display the current value of each *Slider* in hexadecimal, so the *Style* for the *TextBlock* initializes sets the *Text* property to "00", which is the hexadecimal value corresponding to the minimum *Slider* position.

The *Grid* begins by defining three rows (for each *Slider* and two accompanying *TextBlock* labels) and four columns. Notice that the first three columns are all the same width but the fourth column is three times as wide:

Project: SimpleColorScroll | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="3*" />
    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    …

</Grid>
```

The remainder of the XAML file instantiates 10 children of the *Grid*. Each one has both *Grid.Row* and *Grid.Column* attached properties set, although these aren't necessary for values of 0. I tend to put these attached properties early among the attributes but after at least one attribute (such as a *Name* or *Text*) that provides a quick visual identification of the element:

Project: SimpleColorScroll | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">

    …

    <!-- Red -->
    <TextBlock Text="Red"
               Grid.Column="0"
               Grid.Row="0"
               Foreground="Red" />

    <Slider Name="redSlider"
            Grid.Column="0"
            Grid.Row="1"
            Orientation="Vertical"
            Foreground="Red"
            ValueChanged="OnSliderValueChanged" />

    <TextBlock Name="redValue"
               Grid.Column="0"
               Grid.Row="2"
               Foreground="Red" />

    <!-- Green -->
    <TextBlock Text="Green"
               Grid.Column="1"
               Grid.Row="0"
               Foreground="Green" />
```

```
    <Slider Name="greenSlider"
            Grid.Column="1"
            Grid.Row="1"
            Orientation="Vertical"
            Foreground="Green"
            ValueChanged="OnSliderValueChanged" />

    <TextBlock Name="greenValue"
               Grid.Column="1"
               Grid.Row="2"
               Foreground="Green" />

    <!-- Blue -->
    <TextBlock Text="Blue"
               Grid.Column="2"
               Grid.Row="0"
               Foreground="Blue" />

    <Slider Name="blueSlider"
            Grid.Column="2"
            Grid.Row="1"
            Orientation="Vertical"
            Foreground="Blue"
            ValueChanged="OnSliderValueChanged" />

    <TextBlock Name="blueValue"
               Grid.Column="2"
               Grid.Row="2"
               Foreground="Blue" />

    <!-- Result -->
    <Rectangle Grid.Column="3"
               Grid.Row="0"
               Grid.RowSpan="3">
        <Rectangle.Fill>
            <SolidColorBrush x:Name="brushResult"
                             Color="Black" />
        </Rectangle.Fill>
    </Rectangle>
</Grid>
```

Notice that all the *TextBlock* and *Slider* elements are given *Foreground* property assignments based on what color they represent.

The *Rectangle* at the bottom has the *Grid.RowSpan* attached property set to 3, indicating that it spans all three rows. The *SolidColorBrush* is set to *Black*, so that's consistent with the three initial *Slider* values. If you can't get everything initialized correctly in the XAML file, the constructor of the code-behind file is the place to do it.

You might wonder why the *Orientation* property is set on each *Slider* when that property is also set in the implicit *Style*. It turns out that in the version of Windows 8 I'm using to write this chapter, the property setting in the *Style* doesn't "take." It looks good in the Visual Studio preview but not when it

actually runs.

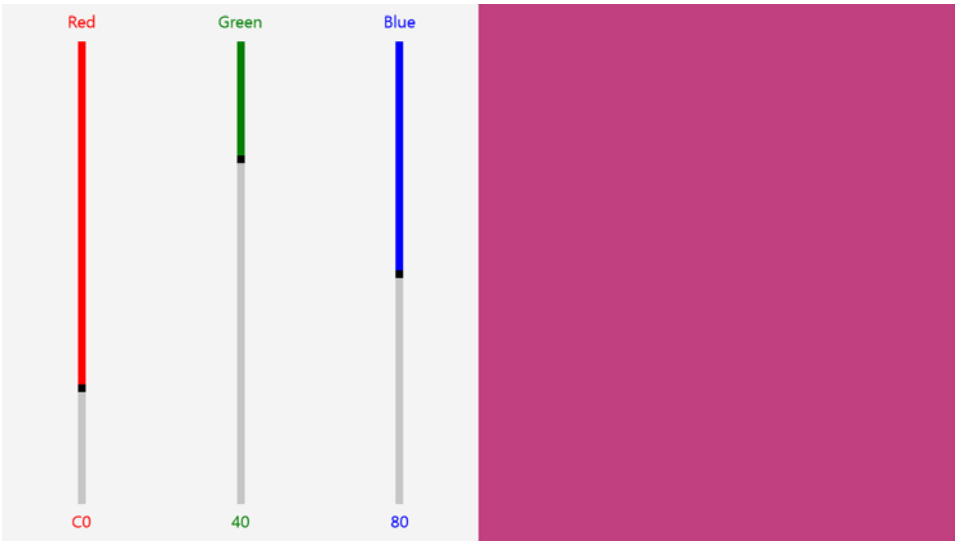
All three *Slider* controls have the same handler for the *ValueChanged* event. That's in the code-behind file:

Project: SimpleColorScroll | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    public BlankPage()
    {
        this.InitializeComponent();
    }

    void OnSliderValueChanged(object sender, RangeBaseValueChangedEventArgs args)
    {
        byte r = (byte)redSlider.Value;
        byte g = (byte)greenSlider.Value;
        byte b = (byte)blueSlider.Value;

        redValue.Text = r.ToString("X2");
        greenValue.Text = g.ToString("X2");
        blueValue.Text = b.ToString("X2");

        brushResult.Color = Color.FromArgb(255, r, g, b);
    }
}
```

The event handler could obtain the actual *Slider* firing the event with the *sender* argument and get the new value from the *RangeBaseValueChangedEventArgs* object. But regardless of which *Slider* actually changes value, the event handler needs to create a whole new *Color* value, and that requires all three values. The only somewhat wasteful part of this code is setting all three text values when only one is changing, but fixing that would require accessing the *TextBlock* associated with the particular *Slider* firing the event.

Here's one of 16,777,216 possible results:



Orientation and Aspect Ratios

If you run SimpleColorScroll on a tablet and rotate it into portrait mode, the layout starts to look a little funny, and even if you run it in landscape mode, a snap view might cause some of the text labels to overlap. It might make sense to add some logic in the code-behind file that adjusts the layout based on the orientation or aspect ratio of the display.

Adjusting the layout with this particular program becomes much easier if the single *Grid* is split in two, one nested in the other. The inner *Grid* has three rows and three columns for the *TextBlock* elements and *Slider* controls. The outer *Grid* has just two children: the inner *Grid* and the *Rectangle*. In landscape mode, the outer *Grid* has two columns; in portrait mode, it has two rows.

The XAML file for the OrientableColorScroll project has the same *Style* definitions as SimpleColorScroll. The outer *Grid* is shown here:

Project: OrientableColorScroll | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}"
      SizeChanged="OnGridSizeChanged">

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition x:Name="secondColDef" Width="*" />
    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition x:Name="secondRowDef" Height="0" />
    </Grid.RowDefinitions>
```

```
    <Grid Grid.Row="0"
          Grid.Column="0">

    …

    </Grid>

    <!-- Result -->
    <Rectangle Name="rectangleResult"
               Grid.Column="1"
               Grid.Row="0">
        <Rectangle.Fill>
            <SolidColorBrush x:Name="brushResult"
                             Color="Black" />
        </Rectangle.Fill>
    </Rectangle>
</Grid>
```

The outer *Grid* has its *RowDefinitions* and *ColumnDefinitions* collections initialized for either contingency: two columns or two rows. In each collection, the second item has been given a name so that it can be accessed from code. The second row has a height of zero, so the initial configuration assumes a landscape mode.

The inner *Grid* (containing the *TextBlock* elements and *Slider* controls) is always in either the first column or first row:

```
<Grid Grid.Row="0"
      Grid.Column="0">

…

</Grid>
```

Setting *Grid.Row* and *Grid.Column* attributes on a *Grid* always looks a little peculiar to me. They refer not to the rows and columns of this *Grid* but to the rows and columns of the parent *Grid*. The default values of these attached properties are both zero, so these particular attribute settings aren't actually required.

The *Rectangle* is initially in the second column and first row:

```
<Rectangle Name="rectangleResult"
           Grid.Column="1"
           Grid.Row="0">
    …
</Rectangle>
```

In this version of the program the *Rectangle* has a name, so these attached properties can be changed from the code-behind file. This is done in the *SizeChanged* event handler set on the outer *Grid*:

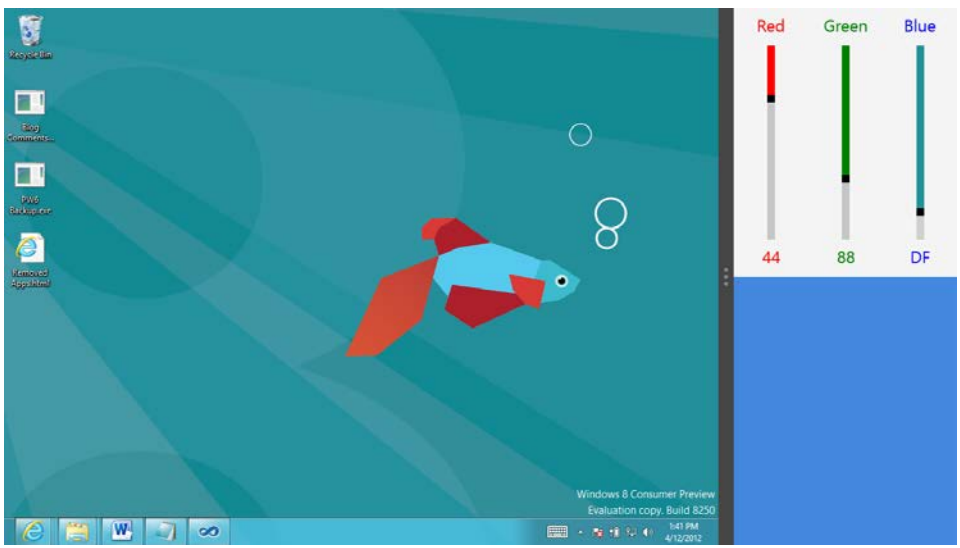
Project: OrientableColorScroll | File: BlankPage.xaml.cs (excerpt)

```
void OnGridSizeChanged(object sender, SizeChangedEventArgs args)
```

```
{
    // Landscape mode
    if (args.NewSize.Width > args.NewSize.Height)
    {
        secondColDef.Width = new GridLength(1, GridUnitType.Star);
        secondRowDef.Height = new GridLength(0);

        Grid.SetColumn(rectangleResult, 1);
        Grid.SetRow(rectangleResult, 0);
    }
    // Portrait mode
    else
    {
        secondColDef.Width = new GridLength(0);
        secondRowDef.Height = new GridLength(1, GridUnitType.Star);

        Grid.SetColumn(rectangleResult, 0);
        Grid.SetRow(rectangleResult, 1);
    }
}
```

This code changes the second *RowDefinition* and *ColumnDefinition* in the outer *Grid*. These both apply to the *Rectangle*, which has its column and row attached properties changed so that it finds itself in the second column (for portrait mode) or second row (for landscape mode).

Here's the program running in a snap mode:



When changing sizes and orientation, sometimes the *Slider* controls don't seem to update themselves properly, but I'm sure that won't be a problem in the release version of Windows 8.

Slider and the Formatted String Converter

In both ColorScroll programs so far, the *TextBlock* labels at the bottom show the current values of the *Slider* in hexadecimal. It's not necessary to provide these values from the code-behind file. It could be done with a data binding from the *Slider* to the *TextBlock*. The only thing that's required is a binding converter that can convert a double into a two-digit hexadecimal string.

It's disturbing to discover that the *FormattedStringConverter* class I described in Chapter 4, "Presentation with Panels," in connection with the WhatSizeWithBindingConverter project will *not* work in this case. You're welcome to try it out, but you'll discover (if you don't already know) that a hexadecimal formatting specification of "X2" can be used only with integral types and the *Value* property of the *Slider* is a *double*.

However, in this case it might make more sense to write a very short ad hoc binding converter, particularly when you realize it can be used for two purposes, as I'll discuss next.

Tooltips and Conversions

As you manipulate the *Slider* controls in either ColorScroll program, you've probably noticed something peculiar: the *Slider* has a built-in tooltip that shows the current value in a little box. That's a nice feature except that this tooltip shows the value in decimal but the program insists on displaying the current value in hexadecimal.

If you think it's great that the *Slider* value is displayed in both decimal and hexadecimal, skip to the next section. If you'd prefer that the two values be consistent—and that they both display the value in hexadecimal—you'll be pleased to know that the *Slider* defines a *ThumbToolTipValueConverter* property that lets you supply a class that performs the formatting you want. This class must implement the *IValueConverter* interface, which is the same interface you implement to write binding converters.

However, a converter class for the *ThumbToolTipValueConverter* property can't be as sophisticated as a converter class for a data binding because you don't have the option of supplying a parameter for the conversion. On the plus side, the converter class can be very simple and do only what is required for the particular case.

The ColorScrollWithValueConverter project defines a converter dedicated to converting a *double* to a two-character string indicating the value in hexadecimal. The name of this class is almost longer than the actual code:

Project: ColorScrollWithValueConverter | File: DoubleToStringHexByteConverter.cs

```
using System;
using Windows.UI.Xaml.Data;

namespace ColorScrollWithValueConverter
{
```

```
    public class DoubleToStringHexByteConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, string language)
        {
            return ((int)(double)value).ToString("X2");
        }
        public object ConvertBack(object value, Type targetType, object parameter, string language)
        {
            return value;
        }
    }
}
```

This converter is suitable not only for formatting the tooltip value, but also for a binding converter to display the value of the *Slider* in the *TextBlock*. The following variation of the ColorScroll program shows how it's done. (To keep things simple, this version doesn't adjust for aspect ratio.) The XAML file instantiates the converter in the *Resources* section:

Project: ColorScrollWithValueConverter | File: BlankPage.xaml (excerpt)

```
<Page.Resources>
    <local:DoubleToStringHexByteConverter x:Key="hexConverter" />
    …
</Page.Resources>
```

Here's the first set of *TextBlock* labels and *Slider*. The *hexConverter* resource is referenced by a simple *StaticResource* markup extension by the *Slider*. The *Binding* on the *TextBlock* is broken into three lines for easy readability:

Project: ColorScrollWithValueConverter | File: BlankPage.xaml (excerpt)

```
<!-- Red -->
<TextBlock Text="Red"
           Grid.Column="0"
           Grid.Row="0"
           Foreground="Red" />

<Slider Name="redSlider"
        Grid.Column="0"
        Grid.Row="1"
        ThumbToolTipValueConverter="{StaticResource hexConverter}"
        Orientation="Vertical"
        Foreground="Red"
        ValueChanged="OnSliderValueChanged" />

<TextBlock Text="{Binding ElementName=redSlider,
                         Path=Value,
                         Converter={StaticResource hexConverter}}"
           Grid.Column="0"
           Grid.Row="2"
           Foreground="Red" />
```

Because the *ValueChanged* handler no longer needs to update the *TextBlock* labels, that code has been removed:

Project: ColorScrollWithValueConverter | File: BlankPage.xaml.cs (excerpt)

```
void OnSliderValueChanged(object sender, RangeBaseValueChangedEventArgs args)
{
    byte r = (byte)redSlider.Value;
    byte g = (byte)greenSlider.Value;
    byte b = (byte)blueSlider.Value;

    brushResult.Color = Color.FromArgb(255, r, g, b);
}
```

Is it possible to go another step and define sufficient data bindings so that the *ValueChanged* handler could be entirely eliminated? That would surely be feasible if it were possible to establish bindings on the individual properties of *Color*, like so:

```
<!-- Doesn't work! -->
<Rectangle Grid.Column="3"
           Grid.Row="0"
           Grid.RowSpan="3">
    <Rectangle.Fill>
        <SolidColorBrush>
            <SolidColorBrush.Color>
                <Color A="255"
                       R="{Binding ElementName=redSlider, Path=Value}"
                       G="{Binding ElementName=greenSlider, Path=Value}"
                       B="{Binding ElementName=blueSlider, Path=Value}" />
            </SolidColorBrush.Color>
        </SolidColorBrush>
    </Rectangle.Fill>
</Rectangle>
```

The big problem with this markup is that binding targets need to be backed by dependency properties, and the properties of *Color* are not. They can't be, because dependency properties can be implemented only in a class that derives from *DependencyObject* and *Color* isn't a class at all. It's a structure.

The *Color* property of *SolidColorBrush* is backed by a dependency property, and that could be the target of a data binding. However, in this program the *Color* property needs three values to be computed, and the Windows Runtime does not support data bindings with multiple sources.

The solution is to have a separate class devoted to the job of creating a *Color* object from red, green, and blue values, and I'll show you how to do it in Chapter 6, "WinRT and MVVM."

Sketching with Sliders

I'm not going to show you a screen shot of the next program. It's called SliderSketch, and it's a *Slider* version of a popular toy invented about 50 years ago. The user of SliderSketch must skillfully manipulate a horizontal *Slider* and a vertical *Slider* in tandem to control a conceptual stylus that progressively extends a continuous polyline. I'm not going to show you a screen shot because the

program is very difficult to use, and I'm pretty much at the baby stage with it at the moment.

The XAML file defines a 2-by-2 *Grid*, but the screen is dominated by one cell containing a large *Border* and a *Polyline*. A vertical *Slider* is at the far left, and a horizontal *Slider* sits at the bottom. The cell in the lower-left corner is empty:

Project: SliderSketch | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <Slider Name="ySlider"
            Grid.Row="0"
            Grid.Column="0"
            Orientation="Vertical"
            IsDirectionReversed="True"
            Margin="0 18"
            ValueChanged="OnSliderValueChanged" />

    <Slider Name="xSlider"
            Grid.Row="1"
            Grid.Column="1"
            Margin="18 0"
            ValueChanged="OnSliderValueChanged" />

    <Border Grid.Row="0"
            Grid.Column="1"
            BorderBrush="{StaticResource ApplicationTextBrush}"
            BorderThickness="3 0 0 3"
            Background="#C0C0C0"
            Padding="24"
            SizeChanged="OnBorderSizeChanged">

        <Polyline Name="polyline"
                  Stroke="#404040"
                  StrokeThickness="3"
                  Points="0 0" />
    </Border>
</Grid>
```

It is very common for a *Grid* to define rows and columns at the edges using *Auto* and then make the whole interior as large as possible with a star specification. The content at the edges is effectively docked. Windows 8 has no *DockPanel*, but it's easy to mimic with *Grid*.

The *Margin* properties on the *Slider* controls were developed based on experimentation. For the

program to work intuitively, the range of *Slider* values should be set equal to the number of pixels between the minimum and maximum positions, and the *Slider* thumbs should be approximately even with the pixel for that value. The calculation of the *Minimum* and *Maximum* values for each *Slider* occurs when the size of the display area changes:

Project: SliderSketch | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    public BlankPage()
    {
        this.InitializeComponent();
    }

    void OnBorderSizeChanged(object sender, SizeChangedEventArgs args)
    {
        Border border = sender as Border;
        xSlider.Maximum = args.NewSize.Width - border.Padding.Left
                                             - border.Padding.Right
                                             - polyline.StrokeThickness;

        ySlider.Maximum = args.NewSize.Height - border.Padding.Top
                                              - border.Padding.Bottom
                                              - polyline.StrokeThickness;
    }

    void OnSliderValueChanged(object sender, RangeBaseValueChangedEventArgs args)
    {
        polyline.Points.Add(new Point(xSlider.Value, ySlider.Value));
    }
}
```

After all that, it's really astonishing to see the actual "drawing" method down at the bottom: just a single line of code that adds a new *Point* to a *Polyline*.

But don't try turning your tablet upside down and shaking it to start anew. I haven't defined an erase function just yet.

The Varieties of Button Experience

The Windows Runtime supports several buttons that derive from the *ButtonBase* class:

Object
DependencyObject
UIElement
FrameworkElement
Control
ContentControl
ButtonBase

Button
HyperlinkButton
RepeatButton
ToggleButton
CheckBox
RadioButton

The ButtonVarieties program demonstrates the default appearances and functionality of all these buttons:

Project: ButtonVarieties | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <StackPanel>
        <Button Content="Just a plain old Button" />
        <HyperlinkButton Content="HyperlinkButton" />
        <RepeatButton Content="RepeatButton" />
        <ToggleButton Content="ToggleButton" />
        <CheckBox Content="CheckBox" />

        <RadioButton Content="RadioButton #1" />
        <RadioButton>RadioButton #2</RadioButton>
        <RadioButton>
            <RadioButton.Content>
                RadioButton #3
            </RadioButton.Content>
        </RadioButton>
        <RadioButton>
            <RadioButton.Content>
                <TextBlock Text="RadioButton #4" />
            </RadioButton.Content>
        </RadioButton>

        <ToggleSwitch />
    </StackPanel>
</Grid>
```

I've included four *RadioButton* instances, all with different approaches to setting the *Content* property, and they're all basically equivalent:

If you don't like the look of any of these, keep in mind that you can entirely redesign them with a *ControlTemplate* that I'll explore in a future chapter.

Like all *FrameworkElement* derivatives, the default values of the *HorizontalAlignment* and *VerticalAlignment* properties are *Stretch*. However, by the time the button is loaded, the *HorizontalAlignment* property has been set to *Left*, the *VerticalAlignment* is *Center*, and a nonzero *Padding* has also been set. Although the *Margin* property is zero, the visuals contain a little built-in margin that surrounds the *Border*.

ButtonBase defines the *Click* event, which is fired when a finger, mouse, or stylus presses the control and then releases, but that behavior can be altered with the *ClickMode* property. Alternatively, a program can be notified that the button has been clicked through a command interface that I'll discuss in Chapter 6.

The classic button is *Button*. There's nothing really special about *HyperlinkButton* except that it looks different as a result of a different template. *RepeatButton* generates a series of *Click* events if held down for a moment; this is mostly intended for the repeat behavior of the *ScrollBar*.

Each click of the *ToggleButton* toggles it on and off. The screen shot shows the on state. *CheckBox* defines nothing public on its own; it simply inherits all the functionality of *ToggleButton* and achieves a different look with a template.

ToggleButton defines an *IsChecked* property to indicate the current state, as well as *Checked* and *Unchecked* events to signal when changing to the on or off state. In general, you'll want to install handlers for both these events, but you can share one handler for the job.

The *IsChecked* property of *ToggleButton* is not a *bool*. It is a *Nullable<bool>*, which means that it can have a value of *null*. This oddity is to accommodate toggle buttons that have a third "indeterminate" state. The classic example is a *CheckBox* labeled "Bold" in a word-processing program: If the selected

text is bold, the box should be checked. If the selected text is not bold, it should be unchecked. If the selected text contains some bold and some nonbold, however, the *CheckBox* should show an indeterminate state. You'll need to set the *IsThreeState* property to *true* to enable this feature, and you'll want to install a handler for the *Indeterminate* event. *ToggleButton* does not have a unique appearance for the indeterminate state; *CheckBox* displays a little box rather than a checkmark.

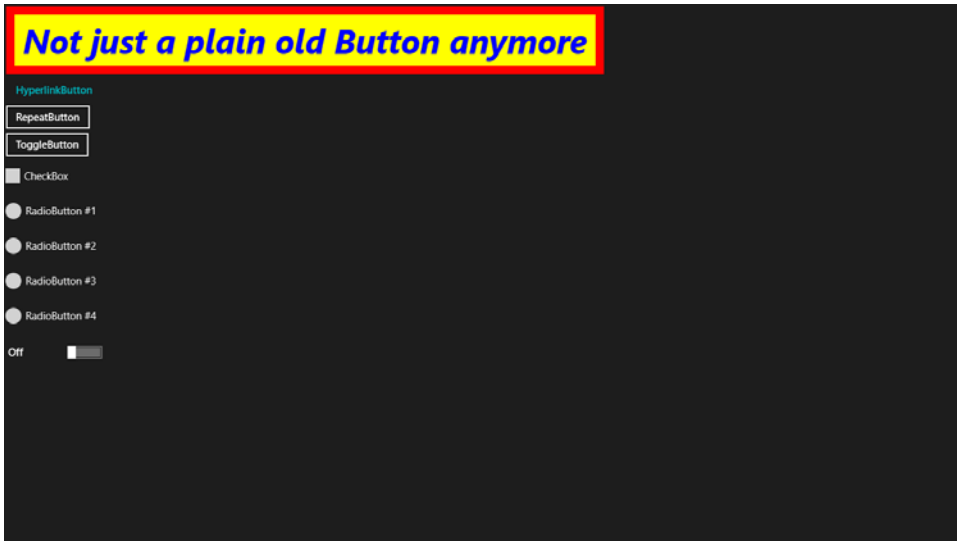
With all that said, you might want to gravitate towards the *ToggleSwitch* control for your toggling needs because it's specifically designed for touch in Metro style applications. Although *ToggleSwitch* does not derive from *ButtonBase*, I've included one anyway at the bottom of the list. As you can see, it provides default labels of "Off" and "On" but you can change those. A header is also available, as you'll discover in Chapter 7, "Building an Application."

The *RadioButton* is a special form of *ToggleButton* for selecting one item from a collection of mutually exclusive options. The name of the control comes from old car radios with buttons for preselected stations: press a button, and the previously pressed button pops out. Similarly, when a *RadioButton* control is checked, it unchecks all other sibling *RadioButton* controls. The only thing you need to do is make them all children of the same panel. (Watch out: if you put a *RadioButton* in a *Border*, it is no longer a sibling with any other *RadioButton*.) If you prefer to separate the *RadioButton* controls into multiple mutually exclusive groups within the same panel, a *GroupName* property is provided for that purpose.

The *Control* class defines a *Foreground* property, many font-related properties, and several properties associated with *Border*, and setting these properties will change button appearance. For example, suppose you initialize a *Button* like so:

```
<Button Content="Not just a plain old Button anymore"
        Background="Yellow"
        BorderBrush="Red"
        BorderThickness="12"
        Foreground="Blue"
        FontSize="48"
        FontStyle="Italic" />
```

Now it looks like this:



However, certain visual characteristics are still governed by the template. For example, when you pass the mouse over this button or press it, the yellow background momentarily disappears and the button background changes to standard colors. Also, although you can change the *Border* color and thickness, you can't give it rounded corners.

ButtonBase derives from *ContentControl*, which defines a property named *Content*. Although the *Content* property is commonly set to text, it can be set to an *Image* or a panel. This is obviously very powerful. For example, here's how a *Button* can contain a bitmap and a caption for the bitmap:

```
<Button>
    <StackPanel>
        <Image Source="http://www.charlespetzold.com/pw6/PetzoldJersey.jpg"
               Width="100" />
        <TextBlock Text="Figure 1"
                   HorizontalAlignment="Center" />
    </StackPanel>
</Button>
```

In a future chapter I'll show you how the *Content* property can be set to virtually any object and how you can supply a template to display that object in a desirable way.

Let's make a simple telephone-like keypad. The keys are *Button* controls, and the result is displayed in a *TextBlock*.

In the following XAML file, the keypad is enclosed in a *Grid* that is given a *HorizontalAlignment* and *VerticalAlignment* of *Center* so that it sits in the center of the screen. Regardless of the size of this keypad and the contents of the buttons, it should have 12 buttons of exactly the same size. I handled the width and the height of these buttons in two different ways. A width of 288 (that is, 3 inches) is imposed on the keyboard *Grid* itself. I wanted a specific width because I realized that a user could type many numbers, and I didn't want the width of the keypad to expand to accommodate an extra-wide

TextBlock. The *Height* of each *Button*, however, is specified in an implicit style:

Project: SimpleKeypad | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">

    <Grid HorizontalAlignment="Center"
          VerticalAlignment="Center"
          Width="288">

        <Grid.Resources>
            <Style TargetType="Button">
                <Setter Property="ClickMode" Value="Press" />
                <Setter Property="HorizontalAlignment" Value="Stretch" />
                <Setter Property="Height" Value="72" />
                <Setter Property="FontSize" Value="36" />
            </Style>
        </Grid.Resources>

        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Grid Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="3">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="Auto" />
            </Grid.ColumnDefinitions>

            <Border Grid.Column="0"
                    HorizontalAlignment="Left">

                <TextBlock Name="resultText"
                           HorizontalAlignment="Right"
                           VerticalAlignment="Center"
                           FontSize="24" />
            </Border>

            <Button Name="deleteButton"
                    Content="&#x21E6;"
                    Grid.Column="1"
                    IsEnabled="False"
                    FontFamily="Segoe Symbol"
                    HorizontalAlignment="Left"
                    Padding="0"
```

```
                BorderThickness="0"
                Click="OnDeleteButtonClick" />
        </Grid>

        <Button Content="1"
                Grid.Row="1" Grid.Column="0"
                Click="OnCharButtonClick" />

        <Button Content="2"
                Grid.Row="1" Grid.Column="1"
                Click="OnCharButtonClick" />

        <Button Content="3"
                Grid.Row="1" Grid.Column="2"
                Click="OnCharButtonClick" />

        <Button Content="4"
                Grid.Row="2" Grid.Column="0"
                Click="OnCharButtonClick" />

        <Button Content="5"
                Grid.Row="2" Grid.Column="1"
                Click="OnCharButtonClick" />

        <Button Content="6"
                Grid.Row="2" Grid.Column="2"
                Click="OnCharButtonClick" />

        <Button Content="7"
                Grid.Row="3" Grid.Column="0"
                Click="OnCharButtonClick" />

        <Button Content="8"
                Grid.Row="3" Grid.Column="1"
                Click="OnCharButtonClick" />

        <Button Content="9"
                Grid.Row="3" Grid.Column="2"
                Click="OnCharButtonClick" />

        <Button Content="*"
                Grid.Row="4" Grid.Column="0"
                Click="OnCharButtonClick" />

        <Button Content="0"
                Grid.Row="4" Grid.Column="1"
                Click="OnCharButtonClick" />

        <Button Content="#"
                Grid.Row="4" Grid.Column="2"
                Click="OnCharButtonClick" />
    </Grid>
</Grid>
```

The hard part is the first row. This must accommodate a *TextBlock* to show the typed result as well as a delete button. I didn't want a very large delete button, so I made the whole first row of the *Grid* a separate *Grid* just for these two items. The attributes of the delete button override many of the properties set in the implicit style. Notice that the delete button is initially disabled. It should be enabled only when there are characters to delete.

The *TextBlock* was a little tricky. I wanted it to be left-justified during normal typing, but if the string got too long to be displayed, I wanted the *TextBlock* to be clipped at the left, not at the right. My solution was to enclose the *TextBlock* in a *Border*:

```
<Border Grid.Column="0"
        HorizontalAlignment="Left">

    <TextBlock Name="resultText"
               HorizontalAlignment="Right"
               VerticalAlignment="Center"
               FontSize="24" />
</Border>
```

The *Border* has a fixed limit to its width: it cannot get wider that the width of the overall *Grid* minus the width of the delete button. But within that area the *Border* is aligned to the left, and the *TextBlock* fits entirely within that *Border*, also aligned at the left. As more characters are typed, the *TextBlock* gets wider until it becomes wider than the *Border*. At that point, the *HorizontalAlignment* setting of *Right* comes into play and the left part of *TextBlock* is what gets clipped.

After that top row, everything else is smooth sailing. The implicit style helps keep the markup for each of the nine buttons as small as possible.

The code-behind file handles the *Click* event from the delete button and has a shared handler for the other 12 buttons:

Project: SimpleKeypad | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    string inputString = "";
    char[] specialChars = { '*', '#' };

    public BlankPage()
    {
        this.InitializeComponent();
    }

    void OnCharButtonClick(object sender, RoutedEventArgs args)
    {
        Button btn = sender as Button;
        inputString += btn.Content as string;
        FormatText();
    }

    void OnDeleteButtonClick(object sender, RoutedEventArgs args)
    {
```

```
        inputString = inputString.Substring(0, inputString.Length - 1);
        FormatText();
    }

    void FormatText()
    {
        bool hasNonNumbers = inputString.IndexOfAny(specialChars) != -1;

        if (hasNonNumbers || inputString.Length < 4 || inputString.Length > 10)
            resultText.Text = inputString;

        else if (inputString.Length < 8)
            resultText.Text = String.Format("{0}-{1}", inputString.Substring(0, 3),
                                                       inputString.Substring(3));

        else
            resultText.Text = String.Format("({0}) {1}-{2}", inputString.Substring(0, 3),
                                                             inputString.Substring(3, 3),
                                                             inputString.Substring(6));

        deleteButton.IsEnabled = inputString.Length > 0;
    }
}
```

The handler for the delete button removes a character from the *inputString* field, and the other handler adds a character. Each handler then calls *FormatText*, which attempts to format the string. At the end of the method, the delete button is enabled only if the input string contains characters.

The *OnCharButtonClick* event handler uses the *Content* property of the button being pressed to determine what character to add to the string. Such an easy equivalence between the *Content* visuals of the button and the functionality of the button isn't always available. Sometimes sharing an event handler among multiple controls requires that the handler extract more information from the button

being clicked. *FrameworkElement* defines a *Tag* property of type *object* specifically for this purpose. You can set *Tag* to an identifying string or object in the XAML file and check it in the event handler.

Dependency Properties

Perhaps you're writing an application where you want all the *Button* controls to display text with a gradient brush. Of course, you can simply define the *Foreground* property of each *Button* to be a *LinearGradientBrush*, but the markup might start becoming a bit overwhelming. You could then try a *Style* with the *Foreground* property set to a *LinearGradientBrush*, but then each *Button* shares the same *LinearGradientBrush* with the same gradient colors, and perhaps you want more flexibility than that.

What you really want is a *Button* with two properties named *Color1* and *Color2* that you can set to the gradient colors. That sounds like a custom control. It's a class that derives from *Button* that creates a *LinearGradientBrush* in its constructor and defines *Color1* and *Color2* properties to control this gradient.

Can these *Color1* and *Color2* properties be just plain old .NET properties with *set* and *get* accessors? Yes, they can. However, defining the properties like that will limit them in some crucial ways. Such properties cannot be the targets of styles, bindings, or animations. Only dependency properties can do all that.

Dependency properties have a bit more overhead than regular properties, but learning how to define dependency properties in your own classes is an important skill. In a new project, begin by adding a new item to the project and select Class from the list. Give it a name of *GradientButton*, and in the file, make the class public and derived from *Button*:

```
public class GradientButton : Button
{

}
```

Now let's fill up that class. You will need to add some *using* directives along the way.

The two new properties are named *Color1* and *Color2* of type *Color*. These two properties require two dependency properties of type *DependencyProperty* named *Color1Property* and *Color2Property*. They must be public and static but settable only from within the class:

```
public static DependencyProperty Color1Property { private set; get; }
public static DependencyProperty Color2Property { private set; get; }
```

These *DependencyProperty* objects must be created in the static constructor. The *DependencyProperty* class defines a static method named *Register* for the job of creating DependencyProperty objects:

```
static GradientButton()
{
    Color1Property =
```

```
        DependencyProperty.Register("Color1",
            typeof(Color),
            typeof(GradientButton),
            new PropertyMetadata(Colors.White, OnColorChanged));

    Color2Property =
        DependencyProperty.Register("Color2",
            typeof(Color),
            typeof(GradientButton),
            new PropertyMetadata(Colors.Black, OnColorChanged));
}
```

A slightly different static method named *DependencyProperty.RegisterAttached* is used to create attached properties.

The first argument to *DependencyProperty.Register* is the text name of the property. This is used sometimes by the XAML parsers. The second argument is the type of the property. The third argument is the type of the class that is registering this dependency property.

The fourth argument is an object of type *PropertyMetadata*. The constructor comes in two versions. In one version, all you need to specify is a default value of the property. In the other, you also specify a method that is called when the property changes. This method will not be called if the property happens to be set to the same value it already has.

The default value you specify as the first argument to the *PropertyMetadata* constructor must match the type indicated in the second argument or a run-time exception will result. This is not as easy as it sounds. For example, it is very common for programmers to supply a default value of 0 for a property of type *double*. During compilation, the 0 is assumed to be an integer, so at run time a type mismatch is discovered and an exception is thrown. If you're defining a dependency property of type *double*, give it a default value of 0.0 so that the compiler knows the correct data type of this argument.

The *GradientButton* class also needs regular .NET property definitions of *Color1* and *Color2*, but these are always of a very specific form:

```
public Color Color1
{
    set { SetValue(Color1Property, value); }
    get { return (Color)GetValue(Color1Property); }
}

public Color Color2
{
    set { SetValue(Color2Property, value); }
    get { return (Color)GetValue(Color2Property); }
}
```

The *set* accessor always calls *SetValue* (inherited from the *DependencyObject* class), referencing the dependency property object, and the *get* accessor always calls *GetValue* and casts the return value to the proper type for the property. You can make the *set* accessor *protected* or *private* if you don't want the property being set from outside the class.

In my *GradientButton* control, I want the *Foreground* property to be a *LinearGradientBrush* and I want the *Color1* and *Color2* properties to be the colors of the two *GradientStop* objects. Two *GradientStop* objects are thus defined as fields:

```
GradientStop gradientStop1, gradientStop2;
```

The regular instance constructor of the class creates those objects as well as the *LinearGradientBrush* to set it to the *Foreground* property:

```
public GradientButton()
{
    gradientStop1 = new GradientStop
    {
        Offset = 0,
        Color = this.Color1
    };

    gradientStop2 = new GradientStop
    {
        Offset = 1,
        Color = this.Color2
    };

    LinearGradientBrush brush = new LinearGradientBrush();
    brush.GradientStops.Add(gradientStop1);
    brush.GradientStops.Add(gradientStop2);

    this.Foreground = brush;
}
```

Notice how the property initializers for the two *GradientStop* objects access the *Color1* and *Color2* properties. This is how the colors in the *LinearGradientBrush* are set to the default colors defined for the two dependency properties.

You'll recall that in the definition of the two dependency properties, a method named *OnColorChanged* was specified as the method to be called whenever either the *Color1* or *Color2* property changes value. Because this property-changed method is referenced in a static constructor, the method itself must also be static:

```
static void OnColorChanged(DependencyObject obj, DependencyPropertyChangedEventArgs args)
{

}
```

Now this is kind of weird, because the whole point of defining this *GradientButton* class is to use it multiple times in an application, and now we're defining a static property that is called whenever the *Color1* or *Color2* property in an instance of this class changes. How do you know to what instance this method call applies?

Easy: it's the first argument. That first argument to this *OnColorChanged* method is always a *GradientButton* object, and you can safely cast it to a *GradientButton* and then access fields and

properties in the particular *GradientButton* instance.

What I like to do in the static property-changed method is call an instance method of the same name, passing to it the second argument:

```
static void OnColorChanged(DependencyObject obj, DependencyPropertyChangedEventArgs args)
{
    (obj as GradientButton).OnColorChanged(args);
}
void OnColorChanged(DependencyPropertyChangedEventArgs args)
{

}
```

This second method then does all the work accessing instance fields and properties of the class.

The *DependencyPropertyChangedEventArgs* object contains some useful information. The *Property* property is of type *DependencyProperty* and indicates the changed property. In this example, the *Property* property will be either *Color1Property* or *Color2Property*. *DependencyPropertyChangedEventArgs* also has properties named *OldValue* and *NewValue* of type *object*.

In *GradientButton*, the property-changed handler sets the *Color* property of the appropriate *GradientStop* object from *NewValue*:

```
void OnColorChanged(DependencyPropertyChangedEventArgs args)
{
    if (args.Property == Color1Property)
        gradientStop1.Color = (Color)args.NewValue;

    if (args.Property == Color2Property)
        gradientStop2.Color = (Color)args.NewValue;
}
```

And that's it for *GradientButton*. The only job left to do is arrange all these pieces in the class in a way that makes sense to you. I like to put all fields at the top, static constructor next, static properties next, and then the instance constructor, instance properties, and all methods. Here's the complete *GradientButton* class from the DependencyProperties project:

Project: DependencyProperties | File: GradientButton.cs

```
using Windows.UI;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media;

namespace DependencyProperties
{
    public class GradientButton : Button
    {
        GradientStop gradientStop1, gradientStop2;

        static GradientButton()
```

```
{
    Color1Property =
        DependencyProperty.Register("Color1",
            typeof(Color),
            typeof(GradientButton),
            new PropertyMetadata(Colors.White, OnColorChanged));

    Color2Property =
        DependencyProperty.Register("Color2",
            typeof(Color),
            typeof(GradientButton),
            new PropertyMetadata(Colors.Black, OnColorChanged));
}

public static DependencyProperty Color1Property { private set; get; }

public static DependencyProperty Color2Property { private set; get; }

public GradientButton()
{
    gradientStop1 = new GradientStop
    {
        Offset = 0,
        Color = this.Color1
    };

    gradientStop2 = new GradientStop
    {
        Offset = 1,
        Color = this.Color2
    };

    LinearGradientBrush brush = new LinearGradientBrush();
    brush.GradientStops.Add(gradientStop1);
    brush.GradientStops.Add(gradientStop2);

    this.Foreground = brush;
}

public Color Color1
{
    set { SetValue(Color1Property, value); }
    get { return (Color)GetValue(Color1Property); }
}

public Color Color2
{
    set { SetValue(Color2Property, value); }
    get { return (Color)GetValue(Color2Property); }
}

static void OnColorChanged(DependencyObject obj,
                           DependencyPropertyChangedEventArgs args)
{
```

```
            (obj as GradientButton).OnColorChanged(args);
        }

        void OnColorChanged(DependencyPropertyChangedEventArgs args)
        {
            if (args.Property == Color1Property)
                gradientStop1.Color = (Color)args.NewValue;

            if (args.Property == Color2Property)
                gradientStop2.Color = (Color)args.NewValue;
        }
    }
}
```

There are some alternate ways of writing the property-changed handler. If you specify separate handlers for each property, you don't need to look at the *Property* property of the event arguments. Rather than access the *NewValue* property, you can just get the value of the property from the class, for example:

```
gradientStop1.Color = this.Color1;
```

The *Color1* property has already been set to the new value by the time the property-changed handler is called.

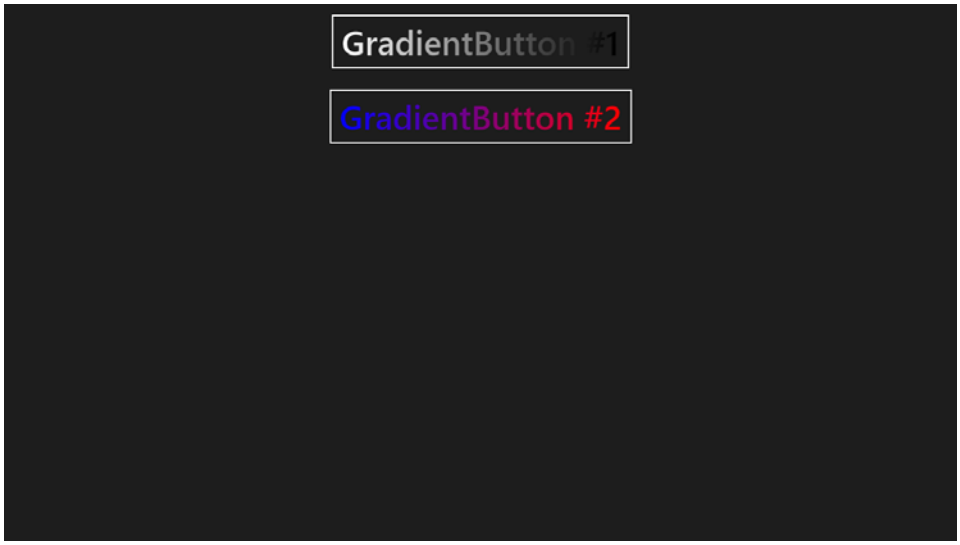
The XAML file in this project defines a couple styles, one with *Setter* elements for *Color1* and *Color2*, and applies these styles to two instances of *GradientButton*. Any reference to *GradientButton* in this XAML file must be preceded by the *local* XML namespace that is associated with the *DependencyProperties* namespace in which *GradientButton* is defined. Notice the *local* prefix in both the *TargetType* of the *Style* and when the buttons are instantiated:

Project: DependencyProperties | File: BlankPage.xaml (excerpt)

```
<Page …
    xmlns:local="using:DependencyProperties"
    … >

    <Page.Resources>
        <Style x:Key="baseButtonStyle" TargetType="local:GradientButton">
            <Setter Property="FontSize" Value="48" />
            <Setter Property="HorizontalAlignment" Value="Center" />
            <Setter Property="Margin" Value="0 12" />
        </Style>

        <Style x:Key="blueRedButtonStyle"
               TargetType="local:GradientButton"
               BasedOn="{StaticResource baseButtonStyle}">
            <Setter Property="Color1" Value="Blue" />
            <Setter Property="Color2" Value="Red" />
        </Style>
    </Page.Resources>

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <StackPanel>
```

```
            <local:GradientButton Content="GradientButton #1"
                                  Style="{StaticResource baseButtonStyle}" />

            <local:GradientButton Content="GradientButton #2"
                                  Style="{StaticResource blueRedButtonStyle}" />
        </StackPanel>
    </Grid>
</Page>
```

The first one gets the default settings of *Color1* and *Color2*, and the second one gets the settings defined in the *Style*. Here it is:



What I was *not* able to do was set *Color1* and Color2 locally in the *GradientButton* tag, but I assume this is just some oddity that will go away before the Windows 8 release.

I want to show you an alternative way to create the *GradientButton* class that lets you define the *LinearGradientBrush* in XAML and eliminate the property-changed handlers. Interested?

In a separate project, rather than adding a new item and picking Class from the list, add a new item, pick User Control from the list, and give it a name of *GradientButton*. As usual you'll get a pair of files: GradientButton.xaml and GradientButton.xaml.cs. The *GradientButton* class derives from *UserControl*. Here's the class definition in the GradientButton.xaml.cs file:

```
public sealed partial class GradientButton : UserControl
{
    public GradientButton()
    {
        this.InitializeComponent();
    }
}
```

Change the base class from *UserControl* to *Button*:

```
public sealed partial class GradientButton : Button
{
    public GradientButton()
    {
        this.InitializeComponent();
    }
}
```

The body of this class will be very much like the first *GradientButton* class except the instance constructor doesn't do anything except call *InitializeComponent*. There are no property-changed handlers. Here's how it looks in the DependencyPropertiesWithBindings project:

Project: DependencyPropertiesWithBindings | File: GradientButton.xaml.cs

```
public sealed partial class GradientButton : Button
{
    static GradientButton()
    {
        Color1Property =
            DependencyProperty.Register("Color1",
                typeof(Color),
                typeof(GradientButton),
                new PropertyMetadata(Colors.White));

        Color2Property =
            DependencyProperty.Register("Color2",
                typeof(Color),
                typeof(GradientButton),
                new PropertyMetadata(Colors.Black));
    }

    public static DependencyProperty Color1Property { private set; get; }

    public static DependencyProperty Color2Property { private set; get; }

    public GradientButton()
    {
        this.InitializeComponent();
    }

    public Color Color1
    {
        set { SetValue(Color1Property, value); }
        get { return (Color)GetValue(Color1Property); }
    }

    public Color Color2
    {
        set { SetValue(Color2Property, value); }
        get { return (Color)GetValue(Color2Property); }
    }
}
```

When first created, the GradientButton.xaml file has a root element that indicates the class derives

from *UserControl*:

```
<UserControl
    x:Class="DependencyPropertiesWithBindings.GradientButton" … >
    …
</UserControl>
```

Change that to *Button* as well:

```
<Button
    x:Class="DependencyPropertiesWithBindings.GradientButton" … >
    …
</Button>
```

Normally when you put stuff between the root tags of a XAML file, you're implicitly setting the *Content* property. But in this case we don't want to set the *Content* property of the *Button*. We want to set the *Foreground* property of *GradientButton* to a *LinearGradientBrush*. This requires property-element tags of *Button.Foreground*. Here's the complete XAML file:

Project: DependencyPropertiesWithBindings | File: GradientButton.xaml

```
<Button
    x:Class="DependencyPropertiesWithBindings.GradientButton"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Name="root">

    <Button.Foreground>
        <LinearGradientBrush>
            <GradientStop Offset="0"
                          Color="{Binding ElementName=root,
                                          Path=Color1}" />
            <GradientStop Offset="1"
                          Color="{Binding ElementName=root,
                                          Path=Color2}" />
        </LinearGradientBrush>
    </Button.Foreground>
</Button>
```

Notice the cool way that the *Color* properties of the *GradientStop* objects are set: the root element is given a name of *"root"* so that it can be the source of two data bindings referencing the custom dependency properties.

The BlankPage.xaml file for this project is the same as the previous project, and the result is also the same.

RadioButton Tags

A group of *RadioButton* controls allows a user to choose between one of several mutually exclusive items. From the program's perspective, often it is convenient that each *RadioButton* in a particular

group corresponds with a member of an enumeration and that the enumeration value be identifiable from the *RadioButton* object. This allows all the buttons in a group to share the same event handler.

The *Tag* property is ideal for this purpose. For example, suppose you want to write a program that lets you experiment with the *StrokeStartLineCap*, *StrokeEndLineCap*, and *StrokeLineJoin* properties defined by the *Shape* class. When rendering thick lines, these properties govern the shape of the ends of the line and the shape where two lines join. The first two properties are set to members of the *PenLineCap* enumeration type and the third is set to members of the *PenLineJoin* enumeration.

For example, one of the members of the *PenLineJoin* enumeration is *Bevel*. You might define a *RadioButton* to represent this option like so:

```
<RadioButton Content="Bevel join"
             Tag="Bevel"
             … />
```

The problem is that *"Bevel"* is interpreted by the XAML parser as a string, so in the event handler in the code-behind file, you need to use *switch* and *case* to differentiate between the different strings or *Enum.TryParse* to convert the string into an actual *PenLineJoin.Bevel* value.

A better way of defining the *Tag* property involves breaking it out as a property element and explicitly indicating that it's being set to a value of type *PenLineJoin*:

```
<RadioButton Content="Bevel join"
             … >
    <RadioButton.Tag>
        <PenLineJoin>Bevel</PenLineJoin>
    </RadioButton.Tag>
</RadioButton>
```

Of course, this is a bit cumbersome. Nevertheless, I've used this approach in the LineCapsAndJoins project. The XAML file defines three groups of *RadioButton* controls for the three *Shape* properties. Each group contains three or four controls corresponding to the appropriate enumeration members.

Project: LineCapsAndJoins | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>

    <StackPanel Name="startLineCapPanel"
                Grid.Row="0" Grid.Column="0"
                Margin="24">
```

```
        <RadioButton Content="Flat start"
                     Checked="OnStartLineCapRadioButtonChecked">
            <RadioButton.Tag>
                <PenLineCap>Flat</PenLineCap>
            </RadioButton.Tag>
        </RadioButton>

        <RadioButton Content="Round start"
                     Checked="OnStartLineCapRadioButtonChecked">
            <RadioButton.Tag>
                <PenLineCap>Round</PenLineCap>
            </RadioButton.Tag>
        </RadioButton>

        <RadioButton Content="Square start"
                     Checked="OnStartLineCapRadioButtonChecked">
            <RadioButton.Tag>
                <PenLineCap>Square</PenLineCap>
            </RadioButton.Tag>
        </RadioButton>

        <RadioButton Content="Triangle start"
                     Checked="OnStartLineCapRadioButtonChecked">
            <RadioButton.Tag>
                <PenLineCap>Triangle</PenLineCap>
            </RadioButton.Tag>
        </RadioButton>
    </StackPanel>

    <StackPanel Name="endLineCapPanel"
                Grid.Row="0" Grid.Column="2"
                Margin="24">
        <RadioButton Content="Flat end"
                     Checked="OnEndLineCapRadioButtonChecked">
            <RadioButton.Tag>
                <PenLineCap>Flat</PenLineCap>
            </RadioButton.Tag>
        </RadioButton>

        <RadioButton Content="Round end"
                     Checked="OnEndLineCapRadioButtonChecked">
            <RadioButton.Tag>
                <PenLineCap>Round</PenLineCap>
            </RadioButton.Tag>
        </RadioButton>

        <RadioButton Content="Square end"
                     Checked="OnEndLineCapRadioButtonChecked">
            <RadioButton.Tag>
                <PenLineCap>Square</PenLineCap>
            </RadioButton.Tag>
        </RadioButton>

        <RadioButton Content="Triangle End"
```

```
                    Checked="OnEndLineCapRadioButtonChecked">
            <RadioButton.Tag>
                <PenLineCap>Triangle</PenLineCap>
            </RadioButton.Tag>
        </RadioButton>
    </StackPanel>

    <StackPanel Name="lineJoinPanel"
                Grid.Row="1" Grid.Column="1"
                HorizontalAlignment="Center"
                Margin="24">

        <RadioButton Content="Bevel join"
                    Checked="OnLineJoinRadioButtonChecked">
            <RadioButton.Tag>
                <PenLineJoin>Bevel</PenLineJoin>
            </RadioButton.Tag>
        </RadioButton>

        <RadioButton Content="Miter join"
                    Checked="OnLineJoinRadioButtonChecked">
            <RadioButton.Tag>
                <PenLineJoin>Miter</PenLineJoin>
            </RadioButton.Tag>
        </RadioButton>

        <RadioButton Content="Round join"
                    Checked="OnLineJoinRadioButtonChecked">
            <RadioButton.Tag>
                <PenLineJoin>Round</PenLineJoin>
            </RadioButton.Tag>
        </RadioButton>
    </StackPanel>

    <Polyline Name="polyline"
              Grid.Row="0"
              Grid.Column="1"
              Points="0 0, 500 1000, 1000 0"
              Stroke="{StaticResource ApplicationTextBrush}"
              StrokeThickness="100"
              Stretch="Fill"
              Margin="24" />
</Grid>
```

Each of the three groups of *RadioButton* controls is in its own *StackPanel*, and all the controls within each *StackPanel* share the same handler for the *Checked* event.

The markup doesn't put any *RadioButton* in its checked state. This is the responsibility of the constructor in the code-behind file. At the bottom of the markup is a thick *Polyline* waiting for its *StrokeStartLineCap*, *StrokeEndLineCap*, and *StrokeLineJoin* properties to be set. This happens in the three *Checked* event handlers also in the code-behind file:

Project: LineCapsAndJoins | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    public BlankPage()
    {
        this.InitializeComponent();

        foreach (UIElement child in startLineCapPanel.Children)
            (child as RadioButton).IsChecked =
                (PenLineCap)(child as RadioButton).Tag == polyline.StrokeStartLineCap;

        foreach (UIElement child in endLineCapPanel.Children)
            (child as RadioButton).IsChecked =
                (PenLineCap)(child as RadioButton).Tag == polyline.StrokeStartLineCap;

        foreach (UIElement child in lineJoinPanel.Children)
            (child as RadioButton).IsChecked =
                (PenLineJoin)(child as RadioButton).Tag == polyline.StrokeLineJoin;
    }

    void OnStartLineCapRadioButtonChecked(object sender, RoutedEventArgs args)
    {
        polyline.StrokeStartLineCap = (PenLineCap)(sender as RadioButton).Tag;
    }

    void OnEndLineCapRadioButtonChecked(object sender, RoutedEventArgs args)
    {
        polyline.StrokeEndLineCap = (PenLineCap)(sender as RadioButton).Tag;
    }

    void OnLineJoinRadioButtonChecked(object sender, RoutedEventArgs args)
    {
        polyline.StrokeLineJoin = (PenLineJoin)(sender as RadioButton).Tag;
    }
}
```

The constructor loops through all the *RadioButton* controls in each group, setting the *IsChecked* property to *true* if the *Tag* value matches the corresponding property of the *Polyline*. Any further *RadioButton* checking occurs under the user's control. The event handlers simply need to set a property of the *Polyline* based on the *Tag* property of the checked *RadioButton*. Here's the result:

It seems that the Round join is not implemented in the version of Windows 8 I'm using.

Although the markup is very explicit about setting the *Tag* property to a member of the *PenLineCap* or *PenLineJoin* enumeration, the XAML parser actually assigns the *Tag* an integer corresponding to the underlying enumeration value. This integer can easily be cast into the correct enumeration member, but it's definitely not the enumeration member itself.

Much of the awkward markup in the LineCapsAndJoins can be eliminated by defining a couple simple custom controls. These custom controls don't need to have dependency properties; they can have just a very simple regular .NET property for a tag corresponding to a particular type.

The LineCapsAndJoinsWithCustomClass shows how this works. Here's a *RadioButton* derivative specifically for representing a *PenLineCap* value:

Project: LineCapsAndJoinsWithCustomClass | File: LineCapRadioButton.cs

```
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media;

namespace LineCapsAndJoinsWithCustomClass
{
    public class LineCapRadioButton : RadioButton
    {
        public PenLineCap LineCapTag { set; get; }
    }
}
```

Similarly, here's one for PenLineJoin values:

Project: LineCapsAndJoinsWithCustomClass | File: LineJoinRadioButton.cs

```
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media;
```

```
namespace LineCapsAndJoinsWithCustomClass
{
    public class LineJoinRadioButton : RadioButton
    {
        public PenLineJoin LineJoinTag { set; get; }
    }
}
```

Let me show you just a little piece of the XAML (the last group of three *RadioButton* controls) to demonstrate how the property-element syntax has been eliminated:

Project: LineCapsAndJoinsWithCustomClass | File: BlankPage.xaml (excerpt)

```
<StackPanel Name="lineJoinPanel"
            Grid.Row="1" Grid.Column="1"
            HorizontalAlignment="Center"
            Margin="24">

    <local:LineJoinRadioButton Content="Bevel join"
                               LineJoinTag="Bevel"
                               Checked="OnLineJoinRadioButtonChecked" />

    <local:LineJoinRadioButton Content="Miter join"
                               LineJoinTag="Miter"
                               Checked="OnLineJoinRadioButtonChecked" />

    <local:LineJoinRadioButton Content="Round join"
                               LineJoinTag="Round"
                               Checked="OnLineJoinRadioButtonChecked" />
</StackPanel>
```

You'll notice that as you type this markup, IntelliSense correctly recognizes the *LineCapTag* and *LineJoinTag* properties to be an enumeration type and gives you an option of typing in one of the enumeration members. Nice!

This switch to custom *RadioButton* derivatives mostly affects the XAML file. The code-behind file is pretty much the same except for somewhat less casting:

Project: LineCapsAndJoinsWithCustomClass | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    public BlankPage()
    {
        this.InitializeComponent();

        foreach (UIElement child in startLineCapPanel.Children)
            (child as LineCapRadioButton).IsChecked =
                (child as LineCapRadioButton).LineCapTag == polyline.StrokeStartLineCap;

        foreach (UIElement child in endLineCapPanel.Children)
            (child as LineCapRadioButton).IsChecked =
                (child as LineCapRadioButton).LineCapTag == polyline.StrokeStartLineCap;

        foreach (UIElement child in lineJoinPanel.Children)
```

```
            (child as LineJoinRadioButton).IsChecked =
                (child as LineJoinRadioButton).LineJoinTag == polyline.StrokeLineJoin;
    }

    void OnStartLineCapRadioButtonChecked(object sender, RoutedEventArgs args)
    {
        polyline.StrokeStartLineCap = (sender as LineCapRadioButton).LineCapTag;
    }

    void OnEndLineCapRadioButtonChecked(object sender, RoutedEventArgs args)
    {
        polyline.StrokeEndLineCap = (sender as LineCapRadioButton).LineCapTag;
   }

    void OnLineJoinRadioButtonChecked(object sender, RoutedEventArgs args)
    {
        polyline.StrokeLineJoin = (sender as LineJoinRadioButton).LineJoinTag;
    }
}
```

Keyboard Input and *TextBox*

Keyboard input in Metro style applications is complicated somewhat by the on-screen touch keyboard that allows the user to enter text by tapping on the screen. Although the touch keyboard is important for tablets and other devices that don't have real keyboards attached, it can also be invoked as a supplement to a real keyboard.

It is vital that the touch keyboard not pop up and disappear in an annoying fashion. For this reason, many controls—including custom controls—do not automatically receive keyboard input. If they did, the system would need to invoke the touch keyboard whenever these controls received input focus. Consequently, if you create a custom control and install event handlers for the *KeyUp* and *KeyDown* events (or override the *OnKeyUp* and *OnKeyDown* methods), you'll discover that nothing comes through.

If you are interested in getting keyboard input from the physical keyboard only and you don't care about the touch keyboard—perhaps for a program intended only for yourself or for testing purposes—there is a fairly easy way to do it. In the constructor of your page, obtain your application's *CoreWindow* object:

```
CoreWindow coreWindow = Window.Current.CoreWindow;
```

This class is defined in the *Windows.UI.Core* namespace. You can then install event handlers on this object for *KeyDown* and *KeyUp* (which indicate keys on the keyboard) as well as *CharacterReceived* (which translates keys to text characters).

If you need to create a custom control that obtains keyboard input from both the physical keyboard and the touch keyboard, the process is rather more involved. You need to derive a class from *FrameworkElementAutomationPeer* that implements the *ITextProvider* and *IValueProvider* interfaces

and return this class in an override of the *OnCreateAutomationPeer* method of your custom control.

Obviously this is a nontrivial task, and I'll provide full details in a forthcoming chapter.

Meanwhile, if your program needs text input, the best approach is to use one of the controls specifically provided for this purpose:

- *TextBox* features single-line or multiline input with a uniform font, much like the traditional Windows Notepad program.
- *RichEditBox* features formatted text, much like the traditional Windows WordPad program.
- *PasswordBox* allows a single line of masked input.

I'll be focusing on *TextBox* in this brief discussion, and I'll provide more examples in the chapters ahead.

TextBox defines a *Text* property that sets the text in the *TextBox* or obtains the current text. The *SelectedText* property is the text that's selected (if any), and the *SelectionStart* and *SelectionLength* properties indicate the offset and length of the selection. If *SelectionLength* is 0, *SelectionStart* is the position of the cursor. Setting the *IsReadOnly* property to *true* inhibits typed input but allows text to be selected and copied to the Clipboard. All cut, copy, and paste interaction occurs through context menus. The *TextBox* defines both *TextChanged* and *SelectionChanged* events.

By default, a *TextBox* allows only a single line of input. Two properties can change that behavior. Normally the *TextBox* ignores the Return key, but setting *AcceptsReturn* to *true* causes the *TextBox* to begin a new line when Return is pressed. The default setting of the *TextWrapping* property is *NoWrap*. Setting that to *Wrap* causes the *TextBox* to generate a new line when the user types beyond the end of the current line. These properties can be set independently. Either will cause a *TextBox* to grow vertically as additional lines are added. *TextBox* has a built-in *ScrollViewer*. If you don't want the *TextBox* to grow indefinitely, set the *MaxLength* property.

There is not one touch keyboard but several, and some are more suitable for entering numbers or email addresses or URIs. A *TextBox* specifies what type of keyboard it wants with the *InputScope* property. With the current version of Windows 8, setting this property in XAML is particularly clumsy (involving two layers of property elements), which accounts for much of the bulk in the following XAML file for the TextBoxInputScopes program. This program lets you experiment with these different keyboard layouts, as well as different modes of multiline *TextBox* instances and (as a bonus) *PasswordBox*:

Project: TextBoxInputScopes | File: BlankPage.xaml (excerpt)

```
<Page
    x:Class="TextBoxInputScopes.BlankPage" … >

    <Page.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="FontSize" Value="24" />
            <Setter Property="VerticalAlignment" Value="Center" />
            <Setter Property="Margin" Value="6" />
```

```
    </Style>

    <Style TargetType="TextBox">
        <Setter Property="Width" Value="320" />
        <Setter Property="VerticalAlignment" Value="Center" />
        <Setter Property="Margin" Value="0 6" />
    </Style>
</Page.Resources>

<Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
    <Grid HorizontalAlignment="Center">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>

        <!-- Multiline with Return, no wrapping -->
        <TextBlock Text="Multiline (accepts Return, no wrap):"
                   Grid.Row="0" Grid.Column="0" />

        <TextBox AcceptsReturn="True"
                 Grid.Row="0" Grid.Column="1" />

        <!-- Multiline with no Return, wrapping -->
        <TextBlock Text="Multiline (ignores Return, wraps):"
                   Grid.Row="1" Grid.Column="0" />

        <TextBox TextWrapping="Wrap"
                 Grid.Row="1" Grid.Column="1" />

        <!-- Multiline with Return and wrapping -->
        <TextBlock Text="Multiline (accepts Return, wraps):"
                   Grid.Row="2" Grid.Column="0" />

        <TextBox AcceptsReturn="True"
                 TextWrapping="Wrap"
                 Grid.Row="2" Grid.Column="1" />

        <!-- Default input scope -->
        <TextBlock Text="Default input scope:"
                   Grid.Row="3" Grid.Column="0" />
```

```
<TextBox Grid.Row="3" Grid.Column="1">
    <TextBox.InputScope>
        <InputScope>
            <InputScope.Names>
                <InputScopeName NameValue="Default" />
            </InputScope.Names>
        </InputScope>
    </TextBox.InputScope>
</TextBox>

<!-- Email address input scope -->
<TextBlock Text="Email address input scope:"
           Grid.Row="4" Grid.Column="0" />

<TextBox Grid.Row="4" Grid.Column="1">
    <TextBox.InputScope>
        <InputScope>
            <InputScope.Names>
                <InputScopeName NameValue="EmailSmtpAddress" />
            </InputScope.Names>
        </InputScope>
    </TextBox.InputScope>
</TextBox>

<!-- Number input scope -->
<TextBlock Text="Number input scope:"
           Grid.Row="5" Grid.Column="0" />

<TextBox Grid.Row="5" Grid.Column="1">
    <TextBox.InputScope>
        <InputScope>
            <InputScope.Names>
                <InputScopeName NameValue="Number" />
            </InputScope.Names>
        </InputScope>
    </TextBox.InputScope>
</TextBox>

<!-- Search input scope -->
<TextBlock Text="Search input scope:"
           Grid.Row="6" Grid.Column="0" />

<TextBox Grid.Row="6" Grid.Column="1">
    <TextBox.InputScope>
        <InputScope>
            <InputScope.Names>
                <InputScopeName NameValue="Search" />
            </InputScope.Names>
        </InputScope>
    </TextBox.InputScope>
</TextBox>

<!-- Telephone number input scope -->
```

```
            <TextBlock Text="Telephone number input scope:"
                       Grid.Row="7" Grid.Column="0" />

            <TextBox Grid.Row="7" Grid.Column="1">
                <TextBox.InputScope>
                    <InputScope>
                        <InputScope.Names>
                            <InputScopeName NameValue="TelephoneNumber" />
                        </InputScope.Names>
                    </InputScope>
                </TextBox.InputScope>
            </TextBox>

            <!-- URL input scope -->
            <TextBlock Text="URL input scope:"
                       Grid.Row="8" Grid.Column="0" />

            <TextBox Grid.Row="8" Grid.Column="1">
                <TextBox.InputScope>
                    <InputScope>
                        <InputScope.Names>
                            <InputScopeName NameValue="Url" />
                        </InputScope.Names>
                    </InputScope>
                </TextBox.InputScope>
            </TextBox>

            <!-- PasswordBox -->
            <TextBlock Text="PasswordBox:"
                       Grid.Row="9" Grid.Column="0" />

            <PasswordBox Grid.Row="9" Grid.Column="1" />

        </Grid>
    </Grid>
</Page>
```

This is a program you'll want to experiment with before choosing a multiline mode or an *InputScope* value.

Touch and *Thumb*

In a future chapter I'll discuss touch input and how you can use it to manipulate objects on the screen. Meanwhile, a modest control called *Thumb* provides some rudimentary touch functionality. *Thumb* is defined in the *Windows.UI.Xaml.Controls.Primitives* namespace, and it is primarily intended as a building block for the *Slider* and *Scrollbar*. In Chapter 7, I'll use it in a custom control.

The *Thumb* control generates three events based on mouse, stylus, or touch movement relative to itself: *DragStarted*, *DragDelta*, and *DragCompleted*. The *DragStarted* event occurs when you put your finger on a *Thumb* control or move the mouse to its surface and click. Thereafter, *DragDelta* events

indicate how the finger or mouse is moving. You can use these events to move the *Thumb* (and anything else), most conveniently on a *Canvas*.

In the AlphabetBlocks program, a series of buttons labeled with letters, numbers, and some punctuation surround the perimeter. Click one, and an alphabet block appears that you can drag with your finger or the mouse. I know that you'll want to send this alphabet block scurrying across the screen with a flick of your finger, but it won't respond in that way. The *Thumb* does not incorporate touch inertia. For inertia, you'll have to tap into the actual touch events.

For the alphabet blocks themselves, a *UserControl* derivative named *Block* has a XAML file that defines a 144-pixel square image with a *Thumb*, some graphics and a *TextBlock*:

Project: AlphabetBlocks | File: Block.xaml

```
<UserControl
    x:Class="AlphabetBlocks.Block"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:AlphabetBlocks"
    Width="144"
    Height="144"
    Name="root">

    <Grid>
        <Thumb DragStarted="OnThumbDragStarted"
               DragDelta="OnThumbDragDelta"
               Margin="18 18 6 6" />

        <!-- Left -->
        <Polygon Points="0 6, 12 18, 12 138, 0 126"
                 Fill="#E0C080" />

        <!-- Top -->
        <Polygon Points="6 0, 18 12, 138 12, 126 0"
                 Fill="#F0D090" />

        <!-- Edge -->
        <Polygon Points="6 0, 18 12, 12 18, 0 6"
                 Fill="#E8C888" />

        <Border BorderBrush="{Binding ElementName=root, Path=Foreground}"
                BorderThickness="12"
                Background="#FFE0A0"
                CornerRadius="6"
                Margin="12 12 0 0"
                IsHitTestVisible="False" />

        <TextBlock FontFamily="Courier New"
                   FontSize="156"
                   FontWeight="Bold"
                   Text="{Binding ElementName=root, Path=Text}"
                   HorizontalAlignment="Center"
                   VerticalAlignment="Center"
```

```
                Margin="12 18 0 0"
                IsHitTestVisible="False" />
    </Grid>
</UserControl>
```

The *Polygon* is similar to *Polyline* except that it automatically closes the figure and then fills the figure with the brush referenced by the *Fill* property.

The *Thumb* has *DragStarted* and *DragDelta* event handlers installed. The two elements that sit on top of the *Thumb*—the *Border* and *TextBlock*—visually hide the *Thumb* but have their *IsHitTestVisible* properties set to *false* so that they don't block touch input from reaching the *Thumb*.

The *BorderBrush* property of the *Border* has a binding to the *Foreground* property of the root element. *Foreground*, you'll recall, is defined by the *Control* class and inherited by *UserControl* and propagated through the visual tree. The *Foreground* property of the *TextBlock* automatically gets this same brush. The *Text* property of the *TextBlock* element is bound to the *Text* property of the control. *UserControl* doesn't have a *Text* property, which strongly suggests that *Block* defines it.

The code-behind file confirms that supposition. Much of this class is devoted to defining a *Text* property backed by a dependency property:

Project: AlphabetBlocks | File: Block.xaml.cs

```
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;

namespace AlphabetBlocks
{
    public sealed partial class Block : UserControl
    {
        static int zindex;

        static Block()
        {
            TextProperty = DependencyProperty.Register("Text",
                typeof(string),
                typeof(Block),
                new PropertyMetadata("?"));
        }

        public static DependencyProperty TextProperty { private set; get; }

        public static int ZIndex
        {
            get { return ++zindex; }
        }

        public Block()
        {
            this.InitializeComponent();
        }
```

```
        public string Text
        {
            set { SetValue(TextProperty, value); }
            get { return (string)GetValue(TextProperty); }
        }

        void OnThumbDragStarted(object sender, DragStartedEventArgs args)
        {
            Canvas.SetZIndex(this, ZIndex);
        }

        void OnThumbDragDelta(object sender, DragDeltaEventArgs args)
        {
            Canvas.SetLeft(this, Canvas.GetLeft(this) + args.HorizontalChange);
            Canvas.SetTop(this, Canvas.GetTop(this) + args.VerticalChange);
        }
    }
}
```

This *Block* class also defines a static *ZIndex* property that requires an explanation. As you click buttons in this program and *Block* objects are created and added to a *Canvas*, each subsequent *Block* appears on top of the previous *Block* objects because of the way they're ordered in the collection. However, when you later put your finger on a *Block*, you want that object to pop to the top of the pile, which means that it should have a z-index higher than every other *Block*.

The static *ZIndex* property defined here helps achieve that. Notice that the value is incremented each time it's called. Whenever a *DragStarted* event occurs, which means that the user has touched one of these controls, the *Canvas.SetZIndex* method gives the *Block* a z-index higher than all the others. Of course, this process will break down eventually when the *ZIndex* property reaches its maximum value, but it's highly unlikely that will happen.

The *DragDelta* event of the *Thumb* reports how touch or the mouse has moved relative to itself in the form of *HorizontalChange* and *VerticalChange* properties. These are simply used to increment the *Canvas.Left* and *Canvas.Top* attached properties.

The BlankPage.xaml file is very bare. The XAML is dominated by some text that displays the name of the program in the center of the page:

Project: AlphabetBlocks | File: BlankPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundBrush}"
      SizeChanged="OnGridSizeChanged">

    <TextBlock Text="Alphabet Blocks"
               FontStyle="Italic"
               FontWeight="Bold"
               FontSize="96"
               TextWrapping="Wrap"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               TextAlignment="Center"
               Opacity="0.1" />
```

```
    <Canvas Name="buttonCanvas" />
    <Canvas Name="blockcanvas" />
</Grid>
```

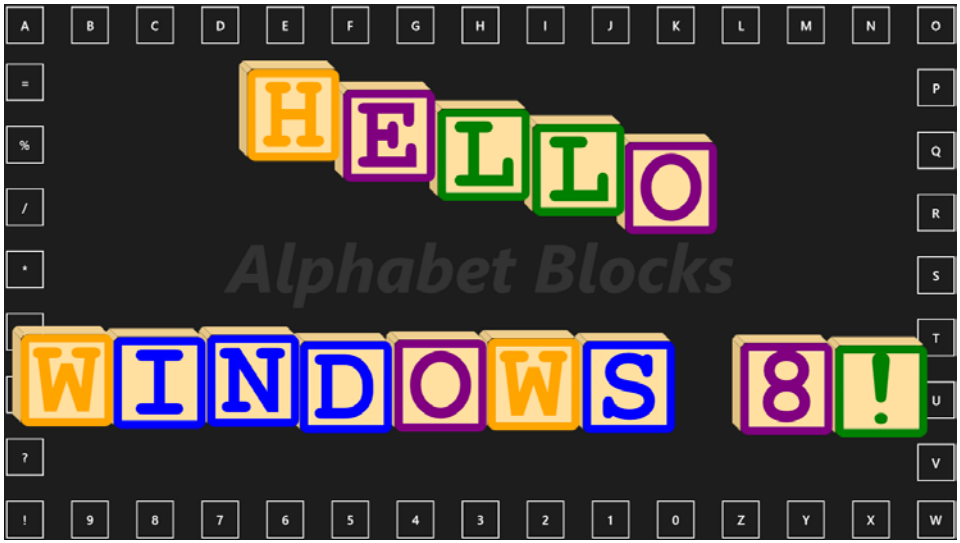
Notice the *SizeChanged* handler on the *Grid*. Whenever the size of the page changes, the handler is responsible for re-creating all the *Button* objects and distributing them equally around the perimeter of the page. That code dominates that code-behind file:

Project: AlphabetBlocks | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    const double BUTTON_SIZE = 60;
    const double BUTTON_FONT = 18;
    string blockChars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!?-+*/%=";
    Color[] colors = { Colors.Red, Colors.Green, Colors.Orange, Colors.Blue, Colors.Purple };
    Random rand = new Random();

    public BlankPage()
    {
        this.InitializeComponent();
    }

    void OnGridSizeChanged(object sender, SizeChangedEventArgs args)
    {
        buttonCanvas.Children.Clear();

        double widthFraction = args.NewSize.Width /
                        (args.NewSize.Width + args.NewSize.Height);
        int horzCount = (int)(widthFraction * blockChars.Length / 2);
        int vertCount = (int)(blockChars.Length / 2 - horzCount);
        int index = 0;

        double slotWidth = (args.NewSize.Width - BUTTON_SIZE) / horzCount;
        double slotHeight = (args.NewSize.Height - BUTTON_SIZE) / vertCount + 1;

        // Across top
        for (int i = 0; i < horzCount; i++)
        {
            Button button = MakeButton(index++);
            Canvas.SetLeft(button, i * slotWidth);
            Canvas.SetTop(button, 0);
            buttonCanvas.Children.Add(button);
        }

        // Down right side
        for (int i = 0; i < vertCount; i++)
        {
            Button button = MakeButton(index++);
            Canvas.SetLeft(button, this.ActualWidth - BUTTON_SIZE);
            Canvas.SetTop(button, i * slotHeight);
            buttonCanvas.Children.Add(button);
        }
```

```
            // Across bottom from right
            for (int i = 0; i < horzCount; i++)
            {
                Button button = MakeButton(index++);
                Canvas.SetLeft(button, this.ActualWidth - i * slotWidth - BUTTON_SIZE);
                Canvas.SetTop(button, this.ActualHeight - BUTTON_SIZE);
                buttonCanvas.Children.Add(button);
            }

            // Up left side
            for (int i = 0; i < vertCount; i++)
            {
                Button button = MakeButton(index++);
                Canvas.SetLeft(button, 0);
                Canvas.SetTop(button, this.ActualHeight - i * slotHeight - BUTTON_SIZE);
                buttonCanvas.Children.Add(button);
            }
        }

        Button MakeButton(int index)
        {
            Button button = new Button
            {
                Content = blockChars[index].ToString(),
                Width = BUTTON_SIZE,
                Height = BUTTON_SIZE,
                FontSize = BUTTON_FONT,
                Tag = new SolidColorBrush(colors[index % colors.Length]),
            };
            button.Click += OnButtonClick;
            return button;
        }

        void OnButtonClick(object sender, RoutedEventArgs e)
        {
            Button button = sender as Button;

            Block block = new Block
            {
                Text = button.Content as string,
                Foreground = button.Tag as Brush
            };
            Canvas.SetLeft(block, this.ActualWidth / 2 - 144 * rand.NextDouble());
            Canvas.SetTop(block, this.ActualHeight / 2 - 144 * rand.NextDouble());
            Canvas.SetZIndex(block, Block.ZIndex);
            blockcanvas.Children.Add(block);
        }
    }
```

A *Block* is created in the *Click* handler for the *Button* and given a random location somewhere close to the center of the screen. It's the responsibility of the user to then move the blocks to discover yet another way to say Hello to Windows 8:


HELLO
Alphabet Blocks
WINDOWS 8!

Chapter 6

WinRT and MVVM

In structuring software, one of the main guiding rules is the separation of concerns. A large application is best developed, debugged, and maintained by being separated into specialized layers. In highly interactive graphical environments, one obvious separation is between presentation and content. The presentation layer is the part of the program that displays controls and other graphics and interacts with the user. Underlying this presentation layer is business logic and data providers.

To help programmers conceptualize and implement separations of concerns, architectural patterns are developed. In XAML-based programming environments, one pattern that has become extremely popular is Model-View-ViewModel, or MVVM. MVVM is particularly suited for implementing a presentation layer in XAML and linking to the underlying business logic through data bindings and commands.

Unfortunately, books such as this one tend to contain very small programs to illustrate particular features and concepts. Very small programs often become much larger when they are made to fit an architectural pattern! MVVM is overkill for a small application and may very well obfuscate rather than clarify.

Nevertheless, data binding and commanding are an important part of the Windows Runtime, and you should see how they help implement an MVVM architecture.

MVVM (Brief and Simplified)

As the name suggests, an application using the Model-View-ViewModel pattern is split into three layers:

- The Model is the layer that deals with data and raw content. It is often involved with obtaining and maintaining data from files or web services.
- The View is the presentation layer of controls and graphics, generally implemented in XAML.
- The View Model sits between the Model and View. In the general case, it is responsible for making the data or content from the Model more conducive to the View.

It's not uncommon for the Model layer to be unnecessary and therefore absent, and that's the case for the programs shown in this chapter.

If all the interaction between these three layers occurs through procedural method calls, a calling hierarchy would be imposed:

View → View Model → Model

Calls in the other direction are not allowed except for events. The Model can define an event that the View Model handles, and the View Model can define an event that the View handles. Events allow the View Model (for example) to signal to the View that updated data is available. The View can then call into the View Model to obtain that updated data.

Most often, the View and View Model interact through data bindings and commands. Consequently, many of the method calls and event handling actually occurs under the covers. These data bindings and commands serve to allow three types of interactions:

- The View can transfer user input to the View Model.
- The View Model can notify the View when updated data is available.
- The View can obtain and display updated data from the View Model.

One of the goals inherent in MVVM is to minimize the code-behind file—at least on the page or window level. MVVM mavens are happiest when all the connections between the View and View Model are accomplished through bindings in the XAML file.

Data Binding Notifications

In Chapter 5, "Control Interaction," you saw data bindings that looked like this:

```
<TextBlock Text="{Binding ElementName=slider, Path=Value}" />
```

This is a binding between two *FrameworkElement* derivatives. The target of this data binding is the *Text* property of the *TextBlock*. The binding source is the *Value* property of a *Slider* identified by the name *slider*. Both the target and source properties are backed by dependency properties. This is a requirement for the binding target but not (as you'll see) for the source.

Whenever the *Value* property of the *Slider* changes, the text displayed by the *TextBlock* changes accordingly. How does this work? When the binding source is a dependency property, the actual mechanism is internal to the Windows Runtime. Undoubtedly an event is involved. The *Binding* object installs a handler for an event that provides a notification when the *Value* property of the *Slider* changes, and the *Binding* object sets that changed value to the *Text* property of the *TextBlock*. This shouldn't be very mysterious, considering that *Slider* has a public *ValueChanged* event that is also fired when the *Value* property changes.

When implementing a View Model, the data bindings are a little different: the binding targets are still elements in the XAML file, but the binding sources are properties in the View Model class. This is the basic way that the View Model and the View (the XAML file) transfer data back and forth.

A binding source is not required to be backed by a dependency property. But in order for the binding to work properly, the binding source must implement some other kind of notification

mechanism to signal to the *Binding* object when a property has changed. This notification does not happen automatically; it must be implemented through an event.

The standard way for a View Model to serve as a binding source is by implementing the *INotifyPropertyChanged* interface defined in the *System.ComponentModel* namespace. This interface has an exceptionally simple definition:

```
public interface INotifyPropertyChanged
{
    event PropertyChangedEventHandler PropertyChanged;
}
```

The *PropertyChangedEventHandler* delegate is associated with the *PropertyChangedEventArgs* class, which defines one property: *PropertyName* of type *string*. When a class implements *INotifyPropertyChanged*, it fires a *PropertyChanged* event whenever one of its properties changes.

Here's a simple example of a class that implements *INotifyPropertyChanged*. The single property named *TotalScore* fires the *PropertyChanged* event when the property changes:

```
public class SimpleViewModel : INotifyPropertyChanged
{
    double totalScore;

    public event PropertyChangedEventHandler PropertyChanged;

    public double TotalScore
    {
        set
        {
            if (totalScore != value)
            {
                totalScore = value;

                if (PropertyChanged != null)
                    PropertyChanged(this, new PropertyChangedEventArgs("TotalScore"));
            }
        }
        get
        {
            return totalScore;
        }
    }
}
```

The *TotalScore* property is backed by the *totalScore* field. Notice that the *TotalScore* property checks the value coming into the *set* accessor against the *totalScore* field and fires the *PropertyChanged* event only when the property actually changes. Do not skimp on this step just to make these *set* accessors a little shorter! The event is called *PropertyChanged* and not *PropertySetAndPerhapsChangedOrMaybeNot*.

Also notice that it's possible for a class to legally implement *INotifyPropertyChanged* and not

actually fire any *PropertyChanged* events, but that would be considered *very* bad behavior.

When a class has more than a couple properties, it starts making sense to define a protected method named *OnPropertyChanged* and let that method do the actual event firing. It's also possible to automate part of this class, as you'll see shortly.

As you design a View and View Model, it helps to start thinking of controls as visual manifestations of data types. The controls in the View are bound to properties of these types in the View Model. For example, a *Slider* is a *double*, a *TextBox* is a *string*, a *CheckBox* or *ToggleSwitch* is a *bool*, and a group of *RadioButton* controls is an enumeration.

A View Model for ColorScroll

The ColorScroll programs in Chapter 5 showed how to use data bindings to update a *TextBlock* from the value property of a *Slider*. However, defining a data binding to change the color based on the three *Slider* values proved much more elusive. Is it possible at all?

The solution is to have a separate class devoted to the job of creating a *Color* object from the values of *Red*, *Green*, and *Blue* properties. Any change to one of these three properties triggers a recalculation of the *Color* property. In the XAML file, bindings connect the *Slider* controls with the *Red*, *Green*, and *Blue* properties and the *SolidColorBrush* with the *Color* property. Even if we don't call this class a View Model, that's what it is.

Here's an *RgbViewModel* class that implements the *INotifyPropertyChanged* interface to fire *PropertyChanged* events whenever its *Red*, *Green*, *Blue*, or *Color* properties change:

Project: ColorScrollWithViewModel | File: RgbViewModel.cs

```
using System.ComponentModel;        // for INotifyPropertyChanged
using Windows.UI;                   // for Color

namespace ColorScrollWithViewModel
{
    public class RgbViewModel : INotifyPropertyChanged
    {
        double red, green, blue;
        Color color = Color.FromArgb(255, 0, 0, 0);

        public event PropertyChangedEventHandler PropertyChanged;

        public double Red
        {
            set
            {
                if (red != value)
                {
                    red = value;
                    OnPropertyChanged("Red");
                    Calculate();
```

```
        }
    }
    get
    {
        return red;
    }
}

public double Green
{
    set
    {
        if (green != value)
        {
            green = value;
            OnPropertyChanged("Green");
            Calculate();
        }
    }
    get
    {
        return green;
    }
}

public double Blue
{
    set
    {
        if (blue != value)
        {
            blue = value;
            OnPropertyChanged("Blue");
            Calculate();
        }
    }
    get
    {
        return blue;
    }
}

public Color Color
{
    protected set
    {
        if (color != value)
        {
            color = value;
            OnPropertyChanged("Color");
        }
    }
    get
    {
```

```
                return color;
            }
        }

        void Calculate()
        {
            this.Color = Color.FromArgb(255, (byte)this.Red, (byte)this.Green, (byte)this.Blue);
        }

        protected void OnPropertyChanged(string propertyName)
        {
            if (PropertyChanged != null)
                PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

The *OnPropertyChanged* method at the bottom of the class has the job of actually firing the *PropertyChanged* event with the name of the property.

I've defined the *Red*, *Green*, and *Blue* properties as *double* to facilitate data bindings. These properties are basically input to the View Model, and they'll probably come from controls such as *Slider*, so the *double* type is the most generalized.

Each of the *Red*, *Green*, and *Blue* property *set* accessors fires a *PropertyChanged* event and then calls *Calculate*, which sets a new *Color* value, which causes another *PropertyChanged* event to be fired for the *Color* property. The *Color* property itself has a protected *set* accessor, indicating that this class isn't designed to calculate *Red*, *Green*, and *Blue* values from a new *Color* value. (I'll discuss this issue shortly.)

The *RgbViewModel* class is part of the ColorScrollWithViewModel project. The BlankPage.xaml file instantiates the *RgbViewModel* in its *Resources* section.

Project: ColorScrollWithViewModel | File: BlankPage.xaml (excerpt)

```
<Page.Resources>
    <local:RgbViewModel x:Key="rgbViewModel" />
    …
</Page.Resources>
```

Notice the namespace prefix of *local*.

Defining the View Model as a resource is one of two basic ways that a XAML file can get access to the object. As was demonstrated in Chapter 2, "XAML Syntax," a class included in a *Resources* section is instantiated only once and shared among all *StaticResource* references. This behavior is essential for an application such as this, in which all the bindings need to reference the same object.

Each of the *Slider* controls is similar. Only one is shown here:

Project: ColorScrollWithViewModel | File: BlankPage.xaml (excerpt)

```
<!-- Red -->
<TextBlock Text="Red"
           Grid.Column="0"
```

```
            Grid.Row="0"
            Foreground="Red" />

<Slider Grid.Column="0"
        Grid.Row="1"
        Value="{Binding Source={StaticResource rgbViewModel},
                        Path=Red,
                        Mode=TwoWay}"
        ThumbToolTipValueConverter="{StaticResource hexConverter}"
        Orientation="Vertical"
        Foreground="Red" />

<TextBlock Text="{Binding Source={StaticResource rgbViewModel},
                          Path=Red,
                          Converter={StaticResource hexConverter}}"
           Grid.Column="0"
           Grid.Row="2"
           Foreground="Red" />
```

Notice that the *Slider* element no longer has a *Name* attribute because no other element in the XAML file refers to this element, and neither does the code-behind file. There's no *ValueChanged* event handler because that's not needed either. The code-behind file contains nothing except a call to *InitializeComponent*.

Take careful note of the binding on the *Slider*:

```
<Slider …
        Value="{Binding Source={StaticResource rgbViewModel},
                        Path=Red,
                        Mode=TwoWay}" … />
```

This binding is a little long, so I've broken it into three lines. It does not specify an *ElementName* because it's not referencing another element in the XAML file. Instead, it's referencing an object instantiated as a XAML resource, so it must use *Source* with *StaticResource*. The syntax of this binding implies that the binding target is the *Value* property of the *Slider* and the binding source is the *Red* property of the *RgbViewModel* instance.

RgbViewModel must be a binding source rather than a target. It can't be a binding target because it has no dependency properties. Despite the syntax implying that *Value* is the binding target, in reality we want the *Slider* to provide a value to the *Red* property. For this reason, the *Mode* property of *Binding* must be set to *TwoWay*, which means both that an updated source value causes a change to the target property (the normal case) and that an updated target value causes a change to the source property (which is actually the essential transfer here).

The default *Mode* setting is *OneWay*. The only other option is *OneTime*, which means that the target is updated from the source property only when the binding is established. With *OneTime*, no updating occurs when the source property later changes. You can use *OneTime* if the source has no notification mechanism.

Also notice that the *TextBlock* showing the current value now has a binding to the *RgbViewModel*

object:

```
<TextBlock Text="{Binding Source={StaticResource rgbViewModel},
                          Path=Red,
                          Converter={StaticResource hexConverter}}" … />
```

This binding could instead refer directly to the *Slider* as in the previous project, but I thought it would be better that it also refer to the *RgbViewModel* instance. The default *OneWay* mode is fine here because data only needs to go from the source to the target.

The *OneWay* mode is also good for the binding on the *Color* property of the *SolidColorBrush*:

Project: ColorScrollWithViewModel | File: BlankPage.xaml (excerpt)

```
<Rectangle Grid.Column="3"
           Grid.Row="0"
           Grid.RowSpan="3">
    <Rectangle.Fill>
        <SolidColorBrush Color="{Binding Source={StaticResource rgbViewModel},
                                         Path=Color}" />
    </Rectangle.Fill>
</Rectangle>
```

The *SolidColorBrush* no longer has an *x:Name* attribute because there's nothing in the code-behind file that refers to it.

Of course, the code in the *RgbViewModel* class is much longer than the *ValueChanged* event handler we've managed to remove from the code-behind file. I warned you at the outset that MVVM is overkill for small programs. Even in larger applications, often there's an initial price to pay for cleaner architecture, but the separation of presentation and business logic certainly has long-term advantages.

In the *RgbViewModel* class I made the *set* accessor of *Color* protected so that it can be accessed only from within the class. Is this really necessary? Perhaps the *Color* property can be defined so that an external change to the property causes new values of the *Red*, *Green*, and *Blue* properties to be calculated:

```
public Color Color
{
    set
    {
        if (color != value)
        {
            color = value;
            OnPropertyChanged("Color");
            this.Red = color.R;
            this.Green = color.G;
            this.Blue = color.B;
        }
    }
    get
    {
        return color;
    }
```

```
}
```

At first this might seem like asking for trouble because it causes recursive property changes and recursive calls to *OnPropertyChanged*. But that doesn't happen because the *set* accessors do nothing if the property is not actually changing, so this should be safe.

But it's actually flawed. Suppose the *Color* property is currently the RGB value (0, 0, 0) and it's set to value (255, 128, 0). When the *Red* property is set to 255 in the code, a *PropertyChanged* event is fired, but now *Color* (and *color*) is set to (255, 0, 0), so the code here continues with *Green* and *Blue* being set to the new *color* values of 0.

This version works OK, however, even though it causes a flurry of *PropertyChanged* events:

```
public Color Color
{
    set
    {
        if (color != value)
        {
            color = value;
            OnPropertyChanged("Color");
            this.Red = value.R;
            this.Green = value.G;
            this.Blue = value.B;
        }
    }
    get
    {
        return color;
    }
}
```

I'll make the *set* accessor of *Color* property public in the next version of the program.

Deriving from *BindableBase*

You might have concluded from the *RgbViewModel* code that implementing *INotifyPropertyChanged* is a bit of a hassle, and that's true. To make it somewhat easier, Visual Studio creates a *BindableBase* class in the Common folder of your projects. (Don't confuse this class with the *BindingBase* class from which *Binding* derives.)

The *BindableBase* class is defined in a namespace that consists of the project name followed by a period and the word *Common*. Stripped of comments and attributes, here's what it looks like:

```
public abstract class BindableBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected bool SetProperty<T>(ref T storage, T value,
                                  [CallerMemberName] String propertyName = null)
```

```
    {
        if (object.Equals(storage, value)) return false;

        storage = value;
        this.OnPropertyChanged(propertyName);
        return true;
    }

    protected void OnPropertyChanged([CallerMemberName] string propertyName = null)
    {
        var eventHandler = this.PropertyChanged;
        if (eventHandler != null)
        {
            eventHandler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

A class that derives from *BindableBase* calls *SetProperty* in the *set* accessor of its property definitions. The signature for the *SetProperty* method looks a little hairy, but it's very easy to use. For a property named *Red* of type *double*, for example, you would have a backing field defined like this:

```
double red;
```

You call *SetProperty* in the *set* accessor like so:

```
SetProperty<double>(ref red, value, "Red");
```

Notice the use of *CallerMemberName* in *BindableBase*. This is an attribute added to .NET 4.5 that C# 5.0 can use to obtain information about code that's calling a particular property or method, which means that you can call *SetProperty* without that last argument. If you're calling *SetProperty* from the *set* access of the *Red* property, the name will be automatically provided:

```
SetProperty<double>(ref red, value);
```

The return value from *SetProperty* is *true* if the property is actually changing. You'll probably want to use the return in logic that does something with the new value. For the next project, called ColorScrollWithDataContext, I've created an alternate version of *RgbViewModel* that derives from *BindableBase*, and I've given *Color* a public *set* accessor:

Project: ColorScrollWithDataContext | File: RgbViewModel.cs

```
using Windows.UI;
using ColorScrollWithDataContext.Common;

namespace ColorScrollWithDataContext
{
    public class RgbViewModel :  BindableBase
    {
        double red, green, blue;
        Color color = Color.FromArgb(255, 0, 0, 0);

        public double Red
```

```
{
    set
    {
        if (SetProperty<double>(ref red, value))
            Calculate();
    }
    get
    {
        return red;
    }
}

public double Green
{
    set
    {
        if (SetProperty<double>(ref green, value))
            Calculate();
    }
    get
    {
        return green;
    }
}

public double Blue
{
    set
    {
        if (SetProperty<double>(ref blue, value))
            Calculate();
    }
    get
    {
        return blue;
    }
}

public Color Color
{
    set
    {
        if (SetProperty<Color>(ref color, value))
        {
            this.Red = value.R;
            this.Green = value.G;
            this.Blue = value.B;
        }
    }
    get
    {
        return color;
    }
}
```

```
        void Calculate()
        {
            this.Color = Color.FromArgb(255, (byte)this.Red, (byte)this.Green, (byte)this.Blue);
        }
    }
}
```

This form of the *INotifyPropertyChanged* implementation is somewhat cleaner and certainly sleeker. I'll use this version in the ColorScrollWithDataContext project in the next section.

The *DataContext* Property

So far you've seen three ways to specify a source object in a binding. *ElementName* is ideal for referencing a named element in XAML, and *RelativeSource* allows a binding to reference a property in the target object. (*RelativeSource* actually has a more important but also more esoteric use that you'll discover in a future chapter.) The third option is the *Source* property, which is generally used with *StaticResource* for accessing an object in the *Resources* collection.

There's a fourth way to specify a binding source: if *ElementName*, *RelativeSource*, and *Source* are all *null*, the *Binding* object checks the *DataContext* property of the binding target.

The *DataContext* property is defined by *FrameworkElement*, and it has the wonderful (and essential) characteristic of propagating down through the visual tree. Not many properties propagate through the visual tree in this way. *Foreground* and all the font-related properties do so, but not many others. *DataContext* is one of the big exceptions to the rule.

The constructor of a code-behind file can instantiate a View Model and set that instance to the *DataContext* of the page. Here's how it's done in the BlankPage.xaml.cs file of the ColorScrollWithDataContext project:

Project: ColorScrollWithDataContext | File: BlankPage.xaml.cs

```
public BlankPage()
{
    this.InitializeComponent();
    this.DataContext = new RgbViewModel();

    // Initialize to highlight color
    (this.DataContext as RgbViewModel).Color =
        (this.Resources["ControlHighlightBrush"] as SolidColorBrush).Color;
}
```

Instantiating the View Model in code might be necessary or desirable for one reason or another. Perhaps the View Model has a constructor that requires an argument. That's something XAML can't do.

Notice that I've also taken the opportunity to test the settability of the *Color* property by initializing it to the system highlight color.

One big advantage to the *DataContext* approach is the simplification of the data bindings. Since they no longer require *Source* settings, they can look like this:

```
<Slider … Value="{Binding Path=Red, Mode=TwoWay}" … />
```

Moreover, if the Path item is the first item in the binding markup, the *Path*= part can be removed:

```
<Slider … Value="{Binding Red, Mode=TwoWay}" … />
```

Now that's a simple *Binding* syntax!

You can remove the *Path*= part of any binding specification regardless of the source, but only if *Path* is the first item. Whenever I use *Source* or *ElementName*, I prefer for that part of the *Binding* specification to appear first, so I'll drop *Path*= only when the *DataContext* comes into play.

Here's an excerpt from the XAML file showing the new bindings. They've become so short that I've stopped breaking them into multiple lines:

Project: ColorScrollWithDataContext | File: BlankPage.xaml (excerpt)

```
<!-- Red -->
<TextBlock Text="Red"
           Grid.Column="0"
           Grid.Row="0"
           Foreground="Red" />

<Slider Grid.Column="0"
        Grid.Row="1"
        Value="{Binding Red, Mode=TwoWay}"
        ThumbToolTipValueConverter="{StaticResource hexConverter}"
        Orientation="Vertical"
        Foreground="Red" />

<TextBlock Text="{Binding Red, Converter={StaticResource hexConverter}}"
           Grid.Column="0"
           Grid.Row="2"
           Foreground="Red" />
…
<!-- Result -->
<Rectangle Grid.Column="3"
           Grid.Row="0"
           Grid.RowSpan="3">
    <Rectangle.Fill>
        <SolidColorBrush Color="{Binding Color}" />
    </Rectangle.Fill>
</Rectangle>
```

It's possible to mix the two approaches. For example, you can instantiate the View Model in the *Resource* collection of the XAML file:

```
<Page.Resources>
    …
    <local:RgbViewModel x:Key="rgbViewModel" />
    …
```

```
</Page.Resources>
```

Then at the earliest convenient place in the visual tree, you can set a *DataContext* property:

```
<Grid … DataContext="{StaticResource rgbViewModel}" … >
```

Or:

```
<Grid … DataContext="{Binding Source={StaticResource rgbViewModel}}" … >
```

The second form is particularly useful if you want to set the *DataContext* to a property of the View Model. You'll see examples when I begin discussing collections.

Bindings and *TextBox*

One of the big advantages to isolating underlying business logic is the ability to completely revamp the user interface without touching the View Model. For example, suppose you want a color-selection program that is similar to ColorScroll but where each color component is entered in a *TextBox*. Such a program might be a little clumsy to use, but it should be possible.

The ColorTextBoxes project has the same *RgbViewModel* class as the ColorScrollWithDataContext program. The code-behind file has the same constructor as that project as well:

Project: ColorTextBoxes | File: BlankPage.xaml.cs (excerpt)

```
public BlankPage()
{
    this.InitializeComponent();
    this.DataContext = new RgbViewModel();

    // Initialize to highlight color
    (this.DataContext as RgbViewModel).Color =
        (this.Resources["ControlHighlightBrush"] as SolidColorBrush).Color;
}
```

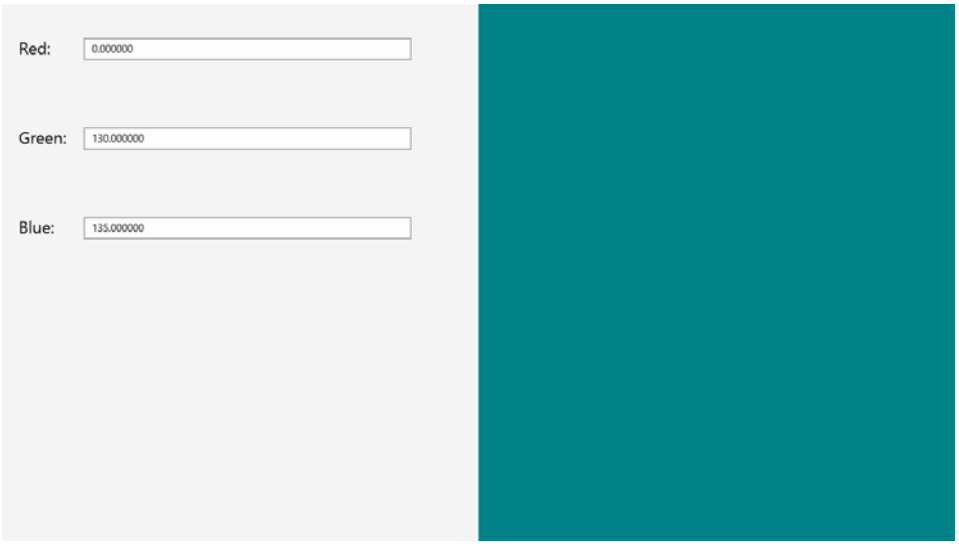
The XAML file instantiates three *TextBox* controls and defines data bindings between the *Red*, *Green*, and *Blue* properties of *RgbViewModel*:

Project: ColorTextBoxes | File: BlankPage.xaml (excerpt)

```
<Page … >

    <Page.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="FontSize" Value="24" />
            <Setter Property="Margin" Value="24 0 0 0" />
            <Setter Property="VerticalAlignment" Value="Center" />
        </Style>

        <Style TargetType="TextBox">
            <Setter Property="Margin" Value="24 48 96 48" />
            <Setter Property="VerticalAlignment" Value="Center" />
```

```
        </Style>
    </Page.Resources>

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Grid Grid.Column="0">
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>

            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>

            <TextBlock Text="Red: "
                       Grid.Row="0"
                       Grid.Column="0" />

            <TextBox Text="{Binding Red, Mode=TwoWay}"
                     Grid.Row="0"
                     Grid.Column="1" />

            <TextBlock Text="Green: "
                       Grid.Row="1"
                       Grid.Column="0" />

            <TextBox Text="{Binding Green, Mode=TwoWay}"
                     Grid.Row="1"
                     Grid.Column="1" />

            <TextBlock Text="Blue: "
                       Grid.Row="2"
                       Grid.Column="0" />

            <TextBox Text="{Binding Blue, Mode=TwoWay}"
                     Grid.Row="2"
                     Grid.Column="1" />
        </Grid>

        <!-- Result -->
        <Rectangle Grid.Column="1">
            <Rectangle.Fill>
                <SolidColorBrush Color="{Binding Color}" />
            </Rectangle.Fill>
        </Rectangle>
    </Grid>
</Page>
```

When the program runs, the individual *TextBox* controls are initialized with color values that have an excessive number of decimal places (which we probably expect by now) but which are correct:



Now tap one of the *TextBox* controls, and try entering another number. Nothing happens. Now tap another *TextBox*, or press the Tab key to shift the input focus to the next *TextBox*. Aha! Now the number you entered in the first *TextBox* has finally been acknowledged and used to update the color.

As you experiment with this program, you'll find that the Windows Runtime is extremely lenient about accepting letters and symbols in these text strings without raising exceptions but that any new value you type registers only when the *TextBox* loses input focus.

This behavior is by design. Suppose a View Model bound to a *TextBox* is using a Model to update a database through a network connection. As the user types text into a *TextBox*—perhaps making mistakes and backspacing—do you really want each and every change going over the network? For that reason, user entry in the *TextBox* is considered to be completed and ready for processing only when the *TextBox* loses input focus.

Unfortunately, there's currently no option to change this behavior. Nor is there any way to include validation in these data bindings. If the *TextBox* binding behavior is unacceptable, the only real choice you have is abandoning bindings for this case and using the *TextChanged* event handler instead.

The ColorTextBoxesWithEvents project shows one possible approach. The project still uses the same *RgbViewModel* class. The XAML file is similar to the previous project except that the *TextBox* controls now have names and *TextChanged* handlers assigned:

Project: ColorTextBoxesWithEvents | File: BlankPage.xaml (excerpt)

```
<TextBlock Text="Red: "
           Grid.Row="0"
           Grid.Column="0" />
```

```
<TextBox Name="redTextBox"
         Grid.Row="0"
         Grid.Column="1"
         Text="0"
         TextChanged="OnTextBoxTextChanged" />

<TextBlock Text="Green: "
           Grid.Row="1"
           Grid.Column="0" />

<TextBox Name="greenTextBox"
         Grid.Row="1"
         Grid.Column="1"
         Text="0"
         TextChanged="OnTextBoxTextChanged" />

<TextBlock Text="Blue: "
           Grid.Row="2"
           Grid.Column="0" />

<TextBox Name="blueTextBox"
         Grid.Row="2"
         Grid.Column="1"
         Text="0"
         TextChanged="OnTextBoxTextChanged" />
```

The *Rectangle*, however, still has the same data binding as in the earlier programs.

Because we're replacing two-way bindings, not only do we need event handlers on the *TextBox* controls, but we need to install a handler for the *PropertyChanged* event of *RgbViewModel*. Updating a *TextBox* when a View Model property changes is fairly easy—and it prevents the decimal point and row of zeroes—but I also decided I wanted to actually validate the text entered by the user:

Project: ColorTextBoxesWithEvents | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    RgbViewModel rgbViewModel;
    Brush textBoxTextBrush;
    Brush textBoxErrorBrush = new SolidColorBrush(Colors.Red);

    public BlankPage()
    {
        this.InitializeComponent();

        // Get TextBox brush
        textBoxTextBrush = this.Resources["TextBoxTextBrush"] as SolidColorBrush;

        // Create RgbViewModel and save as field
        rgbViewModel = new RgbViewModel();
        rgbViewModel.PropertyChanged += OnRgbViewModelPropertyChanged;
        this.DataContext = rgbViewModel;
```

```
        // Initialize to highlight color
        rgbViewModel.Color = (this.Resources["ControlHighlightBrush"] as SolidColorBrush).Color;
    }

    void OnRgbViewModelPropertyChanged(object sender, PropertyChangedEventArgs args)
    {
        switch (args.PropertyName)
        {
            case "Red":
                redTextBox.Text = rgbViewModel.Red.ToString("F0");
                break;

            case "Green":
                greenTextBox.Text = rgbViewModel.Green.ToString("F0");
                break;

            case "Blue":
                blueTextBox.Text = rgbViewModel.Blue.ToString("F0");
                break;
        }
    }

    void OnTextBoxTextChanged(object sender, TextChangedEventArgs args)
    {
        byte value;

        if (sender == redTextBox && Validate(redTextBox, out value))
            rgbViewModel.Red = value;

        if (sender == greenTextBox && Validate(greenTextBox, out value))
            rgbViewModel.Green = value;

        if (sender == blueTextBox && Validate(blueTextBox, out value))
            rgbViewModel.Blue = value;
    }

    bool Validate(TextBox txtbox, out byte value)
    {
        bool valid = byte.TryParse(txtbox.Text, out value);
        txtbox.Foreground = valid ? textBoxTextBrush : textBoxErrorBrush;
        return valid;
    }
}
```

The *Validate* method uses the standard *TryParse* method to convert the text into a *byte* value. If successful, the View Model is updated with the value. If not, the text is displayed in red, indicating a problem.

This works well except when the numbers being entered are preceded with leading blanks or zeros. For example, suppose you type **0** in the first *TextBox*. That's a valid *byte*, so the *Red* property in *RgbViewModel* is updated with this value, which triggers a *PropertyChanged* method, and the *TextBox* is assigned a *Text* value of "0". No problem. Now type a **5**. The *TextBox* contains "05". The *TryParse* method considers this to be a valid *byte* string, and the *Red* property is updated with the value 5. Now

the *PropertyChanged* handler sets the *Text* property of the *TextBox* to the string "5", replacing "05". But the cursor location is not changed, so it's between the 0 and the 5 instead of being after the 5.

Perhaps the best way to prevent this problem is to ignore *PropertyChanged* events from the View Model while setting a property in the View Model from the *TextChanged* handler. You can do this with a simple flag:

```
bool blockViewModelUpdates;

…

void OnRgbViewModelPropertyChanged(object sender, PropertyChangedEventArgs args)
{
    if (blockViewModelUpdates)
        return;
    …
}

void OnTextBoxTextChanged(object sender, TextChangedEventArgs args)
{
    blockViewModelUpdates = true;
    …
    blockViewModelUpdates = false;
}
```

You'll probably also want to clean up the displayed values when each TextBox loses input focus.

Buttons and MVVM

At first, the idea that you can use MVVM to eliminate most of the code-behind file seems valid only for controls that generate values. The concept starts to crumble when you consider buttons. A *Button* fires a *Click* event. That *Click* event must be handled in the code-behind file. If a View Model is actually implementing the logic for that button (which is likely), the *Click* handler must call a method in the View Model. That might be architecturally legal, but it's still rather cumbersome.

Fortunately, there's an alternative to the *Click* event that is ideal for MVVM. This is sometimes informally referred to as the "command interface." *ButtonBase* defines properties named *Command* (of type *ICommand*) and *CommandParameter* (of type object) that allow a *Button* to effectively make a call into a View Model. *Command* and *CommandParameter* are both backed by dependency properties, which means they can be binding targets. *Command* is almost always the target of a data binding. *CommandParameter* is optional. It's useful for differentiating between buttons bound to the same *Command* object, and it's usually treated like a *Tag* property.

Perhaps you've written a calculator application where you've implemented the engine as a View Model that's set as the *DataContext*. The calculator button for the + (plus) command might be instantiated in XAML like so:

```
<Button Content="+"
```

```
        Command="{Binding CalculateCommand}"
        CommandParameter="add" />
```

What this means is that the View Model has a property named *CalculateCommand* of type *ICommand*, perhaps defined like this:

```
public ICommand CalculateCommand { protected set; get; }
```

The View Model must initialize the *CalculateCommand* property by setting it to an instance of a class that implements the *ICommand* interface, which is defined like so:

```
public interface ICommand
{
    void Execute(object param);
    bool CanExecute(object param)
    event EventHandler<object> CanExecuteChanged;
}
```

When this particular *Button* is clicked, the *Execute* method is called in the class referenced by *CalculateCommand* with an argument of *"add"*. This is how a *Button* basically makes a call right into the View Model (or rather, the class containing that *Execute* method).

The other two-thirds of the *ICommand* interface contain the phrase "can execute" and involve the validity of the particular command at a particular time. If this command is not currently valid—perhaps the calculator can't add right now because no number has been entered—the *Button* should be disabled.

Here's how it works: As the XAML is being parsed and loaded at run time, the *Command* property of the *Button* is assigned a binding to (in this example) the *CalculateCommand* object. The *Button* installs a handler for the *CanExecuteChanged* event and calls the *CanExecute* method in this object with an argument (in this example) of *"add"*. If *CanExecute* returns *false*, the *Button* disables itself. Thereafter, the *Button* calls *CanExecute* again whenever the *CanExecuteChanged* event is fired.

To include a command in your View Model, you must provide a class that implements the *ICommand* interface. However, it's very likely that this class needs to access properties in the View Model class, and vice versa.

So you might wonder: can these two classes be one and the same?

In theory, yes they can, but only if you use the same *Execute* and *CanExecute* methods for all the buttons on the page, which means that each button must have a unique *CommandParameter* so that the methods can distinguish between them.

I have not been able to get it to work, however, so let me show you the standard way of implementing commands in a View Model.

The *DelegateCommand* Class

Let's rewrite the SimpleKeypad application from Chapter 5 so that it uses a View Model to accumulate the keystrokes and generate a formatted text string. Besides implementing the *INotifyPropertyChanged* interface (via the *BindableBase* class), the View Model will also process commands from all the buttons in the keypad. There will be no more *Click* handlers.

Here's the problem: For the View Model to process button commands, it must have one or more properties of type *ICommand*, which means that we need one or more classes that implement the *ICommand* interface. To implement *ICommand*, these classes must contain *Execute* and *CanExecute* methods and the *CanExecuteChanged* event. Yet, the bodies of these methods undoubtedly need to interact with the other parts of the View Model.

The solution is to define all the *Execute* and *CanExecute* methods in the View Model class but with different and unique names. Then, a special class can be defined that implements *ICommand* but that actually calls the methods in the View Model.

This special class is often named *DelegateCommand*, and if you search around, you'll find several somewhat different implementations of this class, including one in Microsoft's Prism framework, which helps developers implement MVVM in Windows Presentation Foundation (WPF) and Silverlight. The version here is my variation.

DelegateCommand implements the *ICommand* interface, which means it has *Execute* and *CanExecute* methods and the *CanExecuteChanged* event, but it turns out that *DelegateCommand* also needs another method to fire the *CanExecuteChanged* event. Let's call this additional method *RaiseCanExecuteChanged*. The first job is to define an interface that implements *ICommand* but that includes this additional method:

Project: KeypadWithViewModel | File: IDelegateCommand.cs

```
using System.Windows.Input;

namespace KeypadWithViewModel
{
    public interface IDelegateCommand : ICommand
    {
        void RaiseCanExecuteChanged();
    }
}
```

Simple enough. The *DelegateCommand* class implements the *IDelegateCommand* interface and makes use of a couple simple (but useful) delegates defined in the *System* namespace. The *Action<object>* delegate represents a method with a single object argument and a void return value; not coincidentally, this is the signature of the *Execute* method. The *Func<object, bool>* delegate represents a method with an *object* argument that returns a *bool*; this is the signature of the *CanExecute* method. *DelegateCommand* defines two fields of these types for storing methods with these signatures:

Project: KeypadWithViewModel | File: DelegateCommand.cs

```
using System;

namespace KeypadWithViewModel
{
    public class DelegateCommand : IDelegateCommand
    {
        Action<object> execute;
        Func<object, bool> canExecute;

        // Event required by ICommand
        public event EventHandler CanExecuteChanged;

        // Two constructors
        public DelegateCommand(Action<object> execute, Func<object, bool> canExecute)
        {
            this.execute = execute;
            this.canExecute = canExecute;
        }
        public DelegateCommand(Action<object> execute)
        {
            this.execute = execute;
            this.canExecute = this.AlwaysCanExecute;
        }

        // Methods required by ICommand
        public void Execute(object param)
        {
            execute(param);
        }
        public bool CanExecute(object param)
        {
            return canExecute(param);
        }

        // Method required by IDelegateCommand
        public void RaiseCanExecuteChanged()
        {
            if (CanExecuteChanged != null)
                CanExecuteChanged(this, EventArgs.Empty);
        }

        // Default CanExecute method
        bool AlwaysCanExecute(object param)
        {
            return true;
        }
    }
}
```

This class implements *Execute* and *CanExecute* methods, but these methods merely call the methods saved as fields. These fields are set by the constructor of the class from constructor arguments.

For example, if the calculator View Model has a command to calculate, it can define the

CalculateCommand property like so:

```
public IDelegateCommand CalculateCommand { protected set; get; }
```

The View Model also defines two methods named *ExecuteCalculate* and *CanExecuteCalculate*:

```
void ExecuteCalculate(object param)
{
    …
}
bool CanExecuteCalculate(object param)
{
    …
}
```

The constructor of the View Model class creates the *CalculateCommand* property by instantiating *DelegateCommand* with these two methods:

```
this.CalculateCommand = new DelegateCommand(ExecuteCalculate, CanExecuteCalculate);
```

Now that you see the general idea, let's look at the View Model for the keypad. The class derives from *BindableBase* for the *INotifyPropertyChanged* implementation. For the text entered into and displayed by the keypad, this View Model defines two properties named *InputString* and the formatted version, *DisplayText*.

The View Model also defines two properties of type *IDelegateCommand* named *AddCharacterCommand* (for all the numeric and symbol keys) and *DeleteCharacterCommand*. These properties are created by instantiating *DelegateCommand* with the methods *ExecuteAddCharacter*, *ExecuteDeleteCharacter*, and *CanExecuteDeleteCharacter*. There's no *CanExecuteAddCharacter* because the keys are always valid.

Project: KeypadWithViewModel | File: KeypadViewModel.cs

```
using System;
using KeypadWithViewModel.Common;

namespace KeypadWithViewModel
{
    public class KeypadViewModel : BindableBase
    {
        string inputString = "";
        string displayText = "";
        char[] specialChars = { '*', '#' };

        // Constructor
        public KeypadViewModel()
        {
            this.AddCharacterCommand = new DelegateCommand(ExecuteAddCharacter);
            this.DeleteCharacterCommand =
                new DelegateCommand(ExecuteDeleteCharacter, CanExecuteDeleteCharacter);
        }

        // Public properties
        public string InputString
```

```
{
    protected set
    {
        bool previousCanExecuteDeleteChar = this.CanExecuteDeleteCharacter(null);

        if (this.SetProperty<string>(ref inputString, value))
        {
            this.DisplayText = FormatText(inputString);

            if (previousCanExecuteDeleteChar != this.CanExecuteDeleteCharacter(null))
                this.DeleteCharacterCommand.RaiseCanExecuteChanged();
        }
    }

    get { return inputString; }
}

public string DisplayText
{
    protected set { this.SetProperty<string>(ref displayText, value); }
    get { return displayText; }
}

// ICommand implementations
public IDelegateCommand AddCharacterCommand { protected set; get; }

public IDelegateCommand DeleteCharacterCommand { protected set; get; }

// Execute and CanExecute methods
void ExecuteAddCharacter(object param)
{
    this.InputString += param as string;
}

void ExecuteDeleteCharacter(object param)
{
    this.InputString = this.InputString.Substring(0, this.InputString.Length - 1);
}

bool CanExecuteDeleteCharacter(object param)
{
    return this.InputString.Length > 0;
}

// Private method called from InputString
string FormatText(string str)
{
    bool hasNonNumbers = str.IndexOfAny(specialChars) != -1;
    string formatted = str;

    if (hasNonNumbers || str.Length < 4 || str.Length > 10)
    {
    }
    else if (str.Length < 8)
```

```
            {
                formatted = String.Format("{0}-{1}", str.Substring(0, 3),
                                                     str.Substring(3));
            }
            else
            {
                formatted = String.Format("({0}) {1}-{2}", str.Substring(0, 3),
                                                           str.Substring(3, 3),
                                                           str.Substring(6));
            }
            return formatted;
        }
    }
}
```

The *ExecuteAddCharacter* method expects that the parameter is the character entered by the user. This is how the single command is shared among multiple buttons.

The *CanExecuteDeleteCharacter* returns *true* only if there are characters to delete. The delete button should be disabled otherwise. But this method is called only initially when the binding is first established and thereafter only if the *CanExecuteChanged* event is fired. The logic to fire this event is in the *set* access of *InputString*, which compares the *CanExecuteDeleteCharacter* return values before and after the input string is modified.

The XAML file instantiates the View Model as a resource and then defines a *DataContext* in the *Grid*. Notice the simplicity of the *Command* bindings on the thirteen *Button* controls and the use of *CommandParameter* on the numeric and symbol keys:

Project: KeypadWithViewModel | File: BlankPage.xaml (excerpt)

```
<Page … >

    <Page.Resources>
        <local:KeypadViewModel x:Key="viewModel" />
    </Page.Resources>

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}"
          DataContext="{StaticResource viewModel}">

        <Grid HorizontalAlignment="Center"
              VerticalAlignment="Center"
              Width="288">

            <Grid.Resources>
                <Style TargetType="Button">
                    <Setter Property="ClickMode" Value="Press" />
                    <Setter Property="HorizontalAlignment" Value="Stretch" />
                    <Setter Property="Height" Value="72" />
                    <Setter Property="FontSize" Value="36" />
                </Style>
            </Grid.Resources>

            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
```

```
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Grid Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="3">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="Auto" />
            </Grid.ColumnDefinitions>

            <Border Grid.Column="0"
                    HorizontalAlignment="Left">

                <TextBlock Text="{Binding DisplayText}"
                           HorizontalAlignment="Right"
                           VerticalAlignment="Center"
                           FontSize="24" />
            </Border>

            <Button Content="&#x21E6;"
                    Command="{Binding DeleteCharacterCommand}"
                    Grid.Column="1"
                    FontFamily="Segoe Symbol"
                    HorizontalAlignment="Left"
                    Padding="0"
                    BorderThickness="0" />
        </Grid>

        <Button Content="1"
                Command="{Binding AddCharacterCommand}"
                CommandParameter="1"
                Grid.Row="1" Grid.Column="0" />

        …

        <Button Content="#"
                Command="{Binding AddCharacterCommand}"
                CommandParameter="#"
                Grid.Row="4" Grid.Column="2" />
    </Grid>
  </Grid>
</Page>
```

The really boring part of this project is the code-behind file, which now contains nothing but a call to *InitializeComponent*.

Mission accomplished.

Chapter 7

Building an Application

Even after becoming familiar with various features of the Windows Runtime, putting it all together to create an application can still be a challenge. For that reason, this chapter is mostly devoted to building a rather larger application than anything else in this book.

XamlCruncher features a multiline *TextBox* for editing a XAML file and a display area that shows the visual tree created by running that XAML through the *XamlReader.Load* method. I demonstrated *XamlReader.Load* briefly in Chapter 2, "XAML Syntax," when I used it to convert some XAML path markup syntax to a *PathGeometry* object. The method can handle more complex visual trees, and a tool such as XamlCruncher is very useful for interactively experimenting with XAML and learning about it.

XamlCruncher also features some custom controls, and it demonstrates common application needs:

- An application bar
- A pop-up dialog (or "popup") for customizing program settings
- Saving and retrieving user settings in isolated application storage
- Saving and retrieving files in the Documents area

I'll add additional features to XamlCruncher in future chapters.

Two aspects of this job—file input/output and asynchronous operations—are the subjects of future chapters as well, but it will be necessary to at least become acquainted with these topics in this chapter. To allow me to focus more sharply on these two topics, I'll discuss them in connection with a simpler program with fewer features called MetroPad, which is similar to the traditional Windows Notepad program.

Commands, Options, and Settings

The Windows Runtime supports several methods for applications to implement commands and program options. The most important is the application bar, which is intended to implement basic program commands in a manner similar to a traditional menu or toolbar. The application bar is a class named *AppBar*, and it's invoked when the user sweeps a finger on the top or bottom of the screen. The application bar then often disappears when a command has been selected.

An application bar can appear at the top of the page, or the bottom, or both. The *Page* class defines two properties named *TopAppBar* and *BottomAppBar* that you generally set to *AppBar* tags in XAML.

AppBar derives from *ContentControl*, and you'll usually set the *Content* property to a panel that contains the controls that appear on the application bar.

Perhaps the best way to become familiar with the use of application bars in real programs is to explore some of the standard Metro style applications that are part of Windows 8.

The application bars in the Metro style version of Internet Explorer demonstrates that an application bar can contain a variety of controls. However, very often the *BottomAppBar* contains only a row of circular *Button* controls. In the StandardStyles.xaml file that Visual Studio creates in the Common folder of a standard application, you'll find a *Style* definition with the name *AppBarButtonStyle* that defines this circular button. In addition, StandardStyles.xaml also defines 29 additional styles based on *AppBarButtonStyle* for common commands such as Play, Edit, Save, and Delete. These styles include a template that references the *AutomationProperties.Name* attached property for the text that appears under the button. The button content in these 29 styles is set to character codes ranging from 0xE100 to 0xE11C. This is a private use area in the Unicode standard and makes sense only for the Segoe UI Symbol font. This font has additional symbol characters beyond 0xE11C that you can also use for application bar buttons.

In addition, the Segoe UI Symbol font supports character codes from 0x1F300 through 0x1F5FF that map to emoji characters. These are icon characters that originated in Japan but that have also found their way into the Microsoft Windows Phone and the Apple iPhone. Some of these characters might also be suitable for application bar buttons. (An application to display these symbols is coming up.)

Unfortunately, the *AppBarButtonStyle* has a *TargetType* of *Button*, and even if you change that to *ButtonBase*, you cannot use the style for *ToggleButton* or *RadioButton*. This is unfortunate, because some standard Metro style applications use application bar buttons in this way. For example, in the Calendar application, the Day, Week, and Month buttons work like a trio of *RadioButton* controls, and the Show Traffic button in the Map application works like a *ToggleButton*. I suspect in a future version of StandardStyles.xaml we'll see *AppBarToggleButtonStyle* and *AppBarRadioButtonStyle*, or perhaps controls designed specifically for application bars.

Another approach to implement functionality similar to a *ToggleButton* in an application bar is illustrated in the Weather application. When tapped, the *Button* labeled "Change to Celsius" changes to "Change to Fahrenheit."

A button on an application bar can also invoke a pop-up dialog. For example, press the button in the Metro style Internet Explorer with the wrench icon and the mouse-over tooltip "Page tools." A little popup appears with two additional commands: "Find on page" and "View on the desktop." Or try the Map Style button in Maps to see two mutually exclusive options "Road View" and "Aerial View" with a checkmark indicating the current selection. Or press the "Camera options" command in the Camera application. You get a popup with combo boxes, a toggle switch, and a link for "More," which displays a larger pop-up dialog.

All these dialogs are probably instances of *Popup*, defined in the *Windows.UI.Xaml.Controls.Primitives* namespace. *Popup* has a property named *Child* that you normally

set to a *Panel* derivative to display a bunch of controls. I'll show you how to use *Popup* shortly.

There's also a class named *PopupMenu* from the *Windows.UI.Popups* namespace. As the name suggests, *PopupMenu* is mostly for context menus, such as the Cut/Copy/Paste menu that appears when you press and hold some selected text in the *TextBox* control. You can create a *PopupMenu* on your own, but it is restricted to text commands and you have no control over the formatting.

Also in the *Windows.UI.Popups* namespace is *MessageDialog*, which is the Metro style version of the message box. I'll have some examples of *MessageDialog* later in this chapter.

If you sweep your finger on the right side of the screen while an application is running, you'll bring up the standard list of charms: Search, Share, Devices, and Settings. I'll demonstrate in a later chapter how your application can hook into these charms. In particular, the Settings button often invokes a list of options that can include About and Help as well as Settings. However, some applications include an Options item on the application bar, and the application bar can also contain a Settings item. Indeed, StandardStyles.xaml includes a *SettingsAppBarButtonStyle* that displays a gear icon and the word "Settings." How you divide program functionality among these items is up to you, but generally you'll use an application bar Options button for items accessed more frequently than the Settings and you'll use the Settings button on the application bar for items accessed more frequently than those on the Settings charm.

The Segoe UI Symbol Font

To help you (and me) select symbols for an application bar, I've written a program named SegoeSymbols that displays all the characters from 0x0000 through 0x1FFFF in the Segoe UI Symbol font, which is the font specified by *AppBarButtonStyle*.

As you might know, Unicode started out as a 16-bit character encoding with codes ranging from 0x0000 through 0xFFFF. When it became evident that 65,536 code points were not sufficient, Unicode began incorporating character codes in the range 0x10000 through 0x10FFFF, increasing the number of characters to over 1.1 million. This expansion of Unicode also included a system to represent these additional characters using a pair of 16-bit values.

The use of a 32-bit value to represent a character code is known as UTF-32, or 32-bit Unicode Transformation Format. But that's a bit of misnomer because with UTF-32 there is no transformation: a one-to-one mapping exists from the 32-bit numeric codes to Unicode characters.

Most modern programming languages and operating systems instead support UTF-16. For example, the *Char* structure supported by the Windows Runtime is basically a 16-bit integer, and that's the basis for the *char* data type in C#. To represent the additional characters in the range 0x10000 through 0x10FFFF, UTF-16 uses two 16-bit characters in sequence. These are known as *surrogates*, and a special range of 16-bit codes in Unicode has been set aside for their use. The leading surrogate is in the range 0xD800 through 0xDBFF, and the trailing surrogate is in the range 0xDC00 through 0xDFFF. That's

1,024 possible leading surrogates, and 1,024 possible trailing surrogates, which is sufficient for the 1,048,576 codes in the range 0x10000 through 0x10FFFF. (You'll see the actual algorithm shortly.)

Text in languages that use the Latin alphabet is mostly restricted to ASCII character codes in the range 0x0020 and 0x007E, so most web pages and other files save lots of space by using a system called UTF-8 for text. UTF-8 encodes these 7-bit characters directly but uses one to three additional bytes for other Unicode characters.

Because I wrote SegoeSymbols mostly to let me examine the symbols that might be useful in application bars, the program goes up to character codes of 0x1FFFF only. The XAML file has a simple title, a *Grid* awaiting rows and columns to display a block of 256 characters, and a *Slider*:

Project: SegoeSymbols | File: BlankPage.xaml (excerpt)

```
<Page … >

    <Page.Resources>
        <local:DoubleToStringHexByteConverter x:Key="hexByteConverter" />
    </Page.Resources>

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <TextBlock Name="titleText"
                   Grid.Row="0"
                   Text="Segoe UI Symbol"
                   HorizontalAlignment="Center"
                   Style="{StaticResource HeaderTextStyle}" />

        <Grid Name="characterGrid"
              Grid.Row="1"
              HorizontalAlignment="Center"
              VerticalAlignment="Center" />

        <Slider Grid.Row="2"
                Orientation="Horizontal"
                Margin="24 0"
                Minimum="0"
                Maximum="511"
                SmallChange="1"
                LargeChange="16"
                ThumbToolTipValueConverter="{StaticResource hexByteConverter}"
                ValueChanged="OnSliderValueChanged" />
    </Grid>
</Page>
```

Notice that the *Slider* has a Maximum value of 511, which is the maximum character code I want to display (0x1FFFF) divided by 256. The *DoubleToStringHexByteConverter* class referenced in the *Resources* section is similar to one you've seen before, but it displays a couple underlines as well to be

consistent with the screen visuals:

Project: SegoeSymbols | File: DoubleToStringHexByteConverter.cs (excerpt)

```
public class DoubleToStringHexByteConverter : IValueConverter
{
    public object Convert(object value, Type targetType, object parameter, string language)
    {
        return ((int)(double)value).ToString("X2") + "__";
    }
    public object ConvertBack(object value, Type targetType, object parameter, string language)
    {
        return value;
    }
}
```

Each *Slider* value corresponds to a display of 256 characters in a 16 × 16 array. The code to build the *Grid* that displays these 256 characters is rather messy because I decided that there should be lines between all the rows and columns of characters and that these lines should have their own rows and columns in the *Grid*.

Project: SegoeSymbols | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    const int CellSize = 36;
    const int LineLength = (CellSize + 1) * 16 + 18;
    FontFamily symbolFont = new FontFamily("Segoe UI Symbol");

    TextBlock[] txtblkColumnHeads = new TextBlock[16];
    TextBlock[,] txtblkCharacters = new TextBlock[16, 16];

    public BlankPage()
    {
        this.InitializeComponent();

        for (int row = 0; row < 34; row++)
        {
            RowDefinition rowdef = new RowDefinition();

            if (row == 0 || row % 2 == 1)
                rowdef.Height = GridLength.Auto;
            else
                rowdef.Height = new GridLength(CellSize, GridUnitType.Pixel);

            characterGrid.RowDefinitions.Add(rowdef);

            if (row != 0 && row % 2 == 0)
            {
                TextBlock txtblk = new TextBlock
                {
                    Text = (row / 2 - 1).ToString("X1"),
                    VerticalAlignment = VerticalAlignment.Center
                };
                Grid.SetRow(txtblk, row);
```

```
            Grid.SetColumn(txtblk, 0);
            characterGrid.Children.Add(txtblk);
        }

        if (row % 2 == 1)
        {
            Rectangle rectangle = new Rectangle
            {
                Stroke = this.Foreground,
                StrokeThickness = row == 1 || row == 33 ? 1.5 : 0.5,
                Height = 1
            };
            Grid.SetRow(rectangle, row);
            Grid.SetColumn(rectangle, 0);
            Grid.SetColumnSpan(rectangle, 34);
            characterGrid.Children.Add(rectangle);
        }
    }

    for (int col = 0; col < 34; col++)
    {
        ColumnDefinition coldef = new ColumnDefinition();

        if (col == 0 || col % 2 == 1)
            coldef.Width = GridLength.Auto;
        else
            coldef.Width = new GridLength(CellSize);

        characterGrid.ColumnDefinitions.Add(coldef);

        if (col != 0 && col % 2 == 0)
        {
            TextBlock txtblk = new TextBlock
            {
                Text = "00" + (col / 2 - 1).ToString("X1") + "_",
                HorizontalAlignment = HorizontalAlignment.Center
            };
            Grid.SetRow(txtblk, 0);
            Grid.SetColumn(txtblk, col);
            characterGrid.Children.Add(txtblk);
            txtblkColumnHeads[col / 2 - 1] = txtblk;
        }

        if (col % 2 == 1)
        {
            Rectangle rectangle = new Rectangle
            {
                Stroke = this.Foreground,
                StrokeThickness = col == 1 || col == 33 ? 1.5 : 0.5,
                Width = 1
            };
            Grid.SetRow(rectangle, 0);
            Grid.SetColumn(rectangle, col);
            Grid.SetRowSpan(rectangle, 34);
```

```
                characterGrid.Children.Add(rectangle);
            }
        }

        for (int col = 0; col < 16; col++)
            for (int row = 0; row < 16; row++)
            {
                TextBlock txtblk = new TextBlock
                {
                    Text = ((char)(16 * col + row)).ToString(),
                    FontFamily = symbolFont,
                    FontSize = 24,
                    HorizontalAlignment = HorizontalAlignment.Center,
                    VerticalAlignment = VerticalAlignment.Center
                };
                Grid.SetRow(txtblk, 2 * row + 2);
                Grid.SetColumn(txtblk, 2 * col + 2);
                characterGrid.Children.Add(txtblk);
                txtblkCharacters[col, row] = txtblk;
            }
    }
    …
}
```

The *ValueChanged* handler for the *Slider* has the relatively easier job of inserting the correct text into the existing *TextBlock* elements, but there is that irksome matter of dealing with character codes above 0xFFFF:

Project: SegoeSymbols | File: BlankPage.xaml.cs (excerpt)

```
void OnSliderValueChanged(object sender, RangeBaseValueChangedEventArgs args)
{
    int baseCode = 256 * (int)args.NewValue;

    for (int col = 0; col < 16; col++)
    {
        txtblkColumnHeads[col].Text = (baseCode / 16 + col).ToString("X3") + "_";

        for (int row = 0; row < 16; row++)
        {
            int code = baseCode + 16 * col + row;
            string strChar = null;

            if (code <= 0x0FFFF)
            {
                strChar = ((char)code).ToString();
            }
            else
            {
                code -= 0x10000;
                int lead = 0xD800 + code / 1024;
                int trail = 0xDC00 + code % 1024;
                strChar = ((char)lead).ToString() + (char)trail;
            }
            txtblkCharacters[col, row].Text = strChar;
```

```
        }
    }
}
```

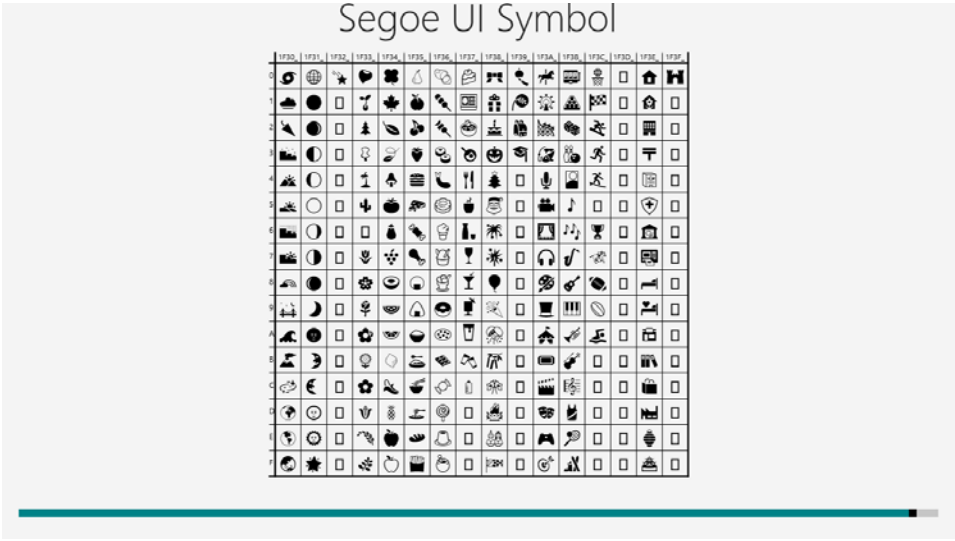
Four statements towards the end of the handler demonstrate the mathematics that separate a Unicode character code between 0x10000 and 0x10FFFF into two 10-bit values to construct leading and trailing surrogates, which in sequence in a string define a single character.

If you're the type of person who prefers not witnessing how sausage is made, you can replace those four lines with:

```
strChar = Char.ConvertFromUtf32(code);
```

For a 16-bit code, *Char.ConvertFromUtf32* returns a string consisting of one character; for codes above 0xFFFF, the string has two characters. Passing the method a surrogate code (0xD800 through 0xDFFF) raises an exception.

The areas that are of most interest in constructing application bar buttons begin at 0xE100 (the private use area used by the Seqoe UI Symbol font) and 0x1F300 (emoji). Here's the first screen of the emoji characters:



You can specify a character beyond 0xFFFF in XAML like so:

```
<TextBlock FontFamily="Segoe UI Symbol"
           FontSize="24"
           Text="&#x1F3B7;" />
```

That's the saxophone symbol. The Visual Studio designer will complain upon encountering a five-digit character code, but it will compile the application regardless and Windows 8 will run it.

The Application Bar

The two MetroPad programs coming up might represent a first step in creating a Metro style Notepad application. Rather than a menu, these programs expose their commands using an application bar. To keep the programs reasonably short, I've eliminated some features that might be expected in a real Notepad-like application. MetroPad1 and MetroPad2 are functionally equivalent but use different methods for asynchronous file I/O. Here's the BlankPage.xaml file for MetroPad1:

Project: MetroPad1 | File: BlankPage.xaml (excerpt)

```
<Page … >

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <TextBox Name="txtbox"
                 FontSize="24"
                 AcceptsReturn="True" />
    </Grid>

    <Page.BottomAppBar>
        <AppBar Padding="10 0">
            <Grid>
                <StackPanel Orientation="Horizontal"
                            HorizontalAlignment="Left">

                    <Button Style="{StaticResource AppBarButtonStyle}"
                            Content="&#xE15E;"
                            AutomationProperties.Name="Wrap options"
                            Click="OnWrapOptionsAppBarButtonClick" />
                </StackPanel>

                <StackPanel Orientation="Horizontal"
                            HorizontalAlignment="Right">

                    <Button Style="{StaticResource AppBarButtonStyle}"
                            Content="&#xE184;"
                            AutomationProperties.Name="Open"
                            Click="OnOpenAppBarButtonClick" />

                    <Button Style="{StaticResource SaveAppBarButtonStyle}"
                            AutomationProperties.Name="Save As"
                            Click="OnSaveAsAppBarButtonClick" />
                </StackPanel>
            </Grid>
        </AppBar>
    </Page.BottomAppBar>
</Page>
```

I've given the *TextBox* a little larger font so that it's easier to experiment with scrolling and word wrapping. In the constructor of the code-behind file, the handler for the *Loaded* event gives *TextBox* input focus so the program is ready for your typing:

Project: MetroPad1 | File: BlankPage.xaml.cs (excerpt)

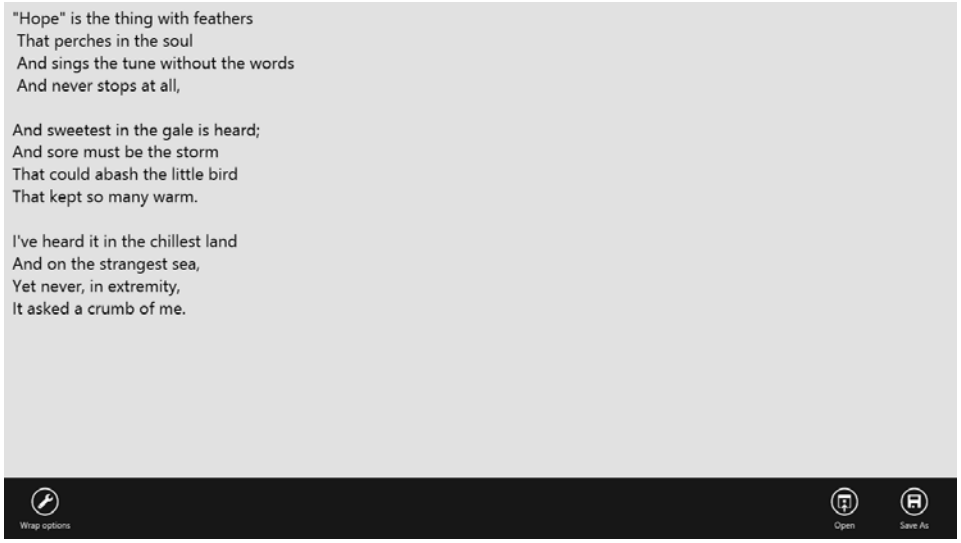
```
public BlankPage()
{
    this.InitializeComponent();

    Loaded += (sender, args) =>
        {
            txtbox.Focus(FocusState.Keyboard);
        };
}
```

As you can see in the XAML file, an application bar is added to the page by splitting out the *BottomAppBar* property of *Page* as a property element and setting it to an *AppBar* element. The *Padding* value of "10 0" is standard and puts 10 pixels of padding at the left and right to prevent the contents from getting too close to the edge.

Generally an application bar has some buttons on the left and some on the right. When holding a tablet, these are more convenient than buttons in the middle. You can use XAML in a couple ways to divide the buttons between left and right. Perhaps the easiest approach is to put two horizontal *StackPanel* elements in a single-cell *Grid* and align them on the right and left.

It's recommended that a New (or Add) button be on the far right, and although this program does not have a New button, the other file-related buttons should also appear on the right side because they are related to New. I was able to use the predefined *SaveAppBarButtonStyle*, but I had to specify my own symbols and text for the other two items.

When you run MetroPad1, it might not be obvious that anything is happening because the program consists entirely of a *TextBox* with an off-white background. But you can type some poetry (or other text) into the *TextBox* and when you sweep your finger on the top or bottom of the screen, here's what you'll see:



Do not specify a *RequestedTheme* of *Light* when using an application bar. The *AppBar* has a black background regardless, and the dark outline and text of the buttons will be nearly invisible.

AppBar defines an *IsOpen* property that you can initialize to *true* if you want the application bar to be visible when the user first runs the program. This might make sense if the program is not usable unless a user executes one of the commands.

Clicking one of the buttons does not automatically dismiss the application bar. That must be done in code by setting *IsOpen* to *false*. However, the user can manually dismiss the application bar in one of two ways: by sweeping a finger again on the top or bottom, or by touching anywhere outside the application bar. The first type of dismissal always works. The second is called *light dismiss* and you can override the default behavior by setting the *IsSticky* property to *true*.

AppBar also defines *Opened* and *Closed* events if you need to initialize an application bar when it's opening or save settings when it closes.

Popups and Dialogs

When the *Button* in the MetroPad1 application bar labeled "Wrap options" is clicked, the program displays a little dialog with "Wrap" and "No wrap" items. Such a dialog is normally defined as a *UserControl*, and I've called mine *WrapOptionsDialog*. The XAML file represents the two options with *RadioButton* controls:

Project: MetroPad1 | File: WrapOptionsDialog.xaml (excerpt)

```
<UserControl … >

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <StackPanel Name="stackPanel"
                    Margin="24">
            <RadioButton Content="Wrap"
                         Checked="OnRadioButtonChecked">
                <RadioButton.Tag>
                    <TextWrapping>Wrap</TextWrapping>
                </RadioButton.Tag>
            </RadioButton>

            <RadioButton Content="No wrap"
                         Checked="OnRadioButtonChecked">
                <RadioButton.Tag>
                    <TextWrapping>NoWrap</TextWrapping>
                </RadioButton.Tag>
            </RadioButton>
        </StackPanel>
    </Grid>
</UserControl>
```

A few words on color. You'll notice that this *Grid* has the standard background brush. It needs to have some kind of brush or the background will be transparent. I mentioned earlier that you can't set

RequestedTheme to *Light* when you implement an application bar or the buttons fade into the background. Because a dark theme is in effect here, this dialog will have a black background with a white foreground.

All of the dialogs that I've seen in Metro style applications have a white background and black foreground. However, I've had frustrating experiences trying to flip the colors in the dialog box. You can set the *Grid* background to *ApplicationTextBrush* or explicitly to white, but setting the *Foreground* on the root element does not property propagate to the *RadioButton* controls, and even explicitly setting *Foreground* on the individual controls (or using a style) does not color them properly.

This means that the dialogs in this book will have a black background and white foreground until the controls are fixed or more guidance comes from above.

The code-behind file for the dialog defines a dependency property named *TextWrapping* of type *TextWrapping*. The property-changed handler checks a *RadioButton* when this property is set, and the property is set when a user checks a *RadioButton*:

Project: MetroPad1 | File: WrapOptionsDialog.xaml.cs (excerpt)

```
public sealed partial class WrapOptionsDialog : UserControl
{
    static WrapOptionsDialog()
    {
        TextWrappingProperty = DependencyProperty.Register("TextWrapping",
            typeof(TextWrapping),
            typeof(WrapOptionsDialog),
            new PropertyMetadata(TextWrapping.NoWrap, OnTextWrappingChanged));
    }

    public static DependencyProperty TextWrappingProperty { private set; get; }

    public WrapOptionsDialog()
    {
        this.InitializeComponent();
    }

    public TextWrapping TextWrapping
    {
        set { SetValue(TextWrappingProperty, value); }
        get { return (TextWrapping)GetValue(TextWrappingProperty); }
    }

    static void OnTextWrappingChanged(DependencyObject obj,
                                      DependencyPropertyChangedEventArgs args)
    {
        (obj as WrapOptionsDialog).OnTextWrappingChanged(args);
    }

    void OnTextWrappingChanged(DependencyPropertyChangedEventArgs args)
    {
        foreach (UIElement child in stackPanel.Children)
        {
```

```
            RadioButton radioButton = child as RadioButton;
            radioButton.IsChecked = (TextWrapping)radioButton.Tag == (TextWrapping)args.NewValue;
        }
    }

    void OnRadioButtonChecked(object sender, RoutedEventArgs args)
    {
        this.TextWrapping = (TextWrapping)(sender as RadioButton).Tag;
    }
}
```

The event handler for the "Wrap options" application bar button is in the *BlankPage* code-behind file. The event handler instantiates a *WrapOptionsDialog* object and initializes its *TextWrapping* property from the *TextWrapping* property of the *TextBox*. It then defines a binding in code between the two *TextWrapping* properties. This allows the user to see the result of changing this property directly in the *TextBox*. The *WrapOptionsDialog* object is then made a child of a new *Popup* object:

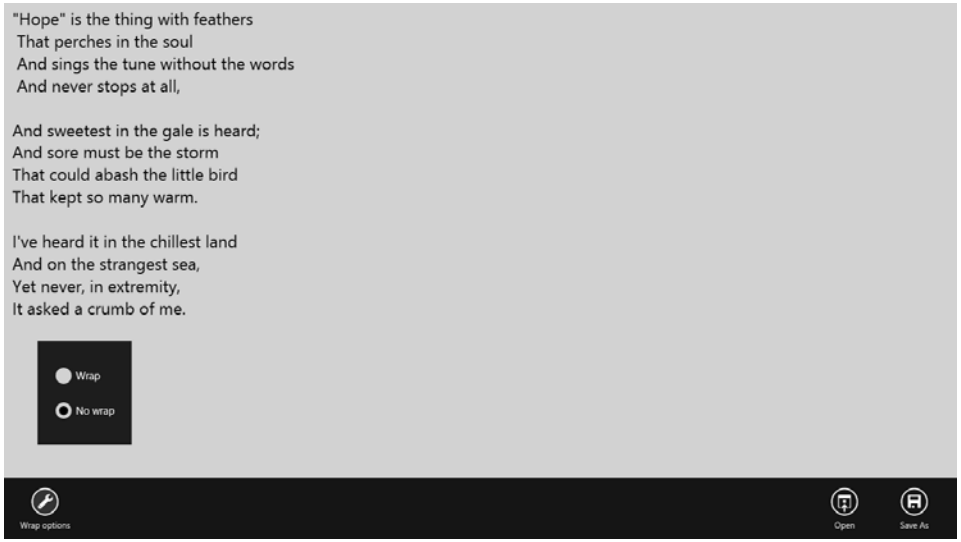
Project: MetroPad1 | File: BlankPage.xaml.cs (excerpt)

```
void OnWrapOptionsAppBarButtonClick(object sender, RoutedEventArgs args)
{
    // Create dialog
    WrapOptionsDialog wrapOptionsDialog = new WrapOptionsDialog
    {
        TextWrapping = txtbox.TextWrapping
    };

    // Bind dialog to TextBox
    Binding binding = new Binding
    {
        Source = wrapOptionsDialog,
        Path = new PropertyPath("TextWrapping"),
        Mode = BindingMode.TwoWay
    };
    txtbox.SetBinding(TextBox.TextWrappingProperty, binding);

    // Create popup
    Popup popup = new Popup
    {
        Child = wrapOptionsDialog,
        IsLightDismissEnabled = true
    };

    // Adjust location based on content size
    wrapOptionsDialog.SizeChanged += (dialogSender, dialogArgs) =>
    {
        popup.VerticalOffset = this.ActualHeight - wrapOptionsDialog.ActualHeight
                                                  - this.BottomAppBar.ActualHeight - 48;
        popup.HorizontalOffset = 48;
    };

    // Open the popup
    popup.IsOpen = true;
}
```

As you can see, *Popup* also has a "light dismiss" mode that lets you dismiss the *Popup* by tapping anywhere outside it. By default this property is not enabled, but in most cases it should be.

The hard part is positioning the *Popup*. It has *VerticalOffset* and *HorizontalOffset* properties for that purpose, but generally popups such as this are positioned just above the application bar, which means that you need to know the height of the popup, the height of the page, and the height of the application bar to get it right. I've found that setting a *SizeChanged* event on the dialog control is a good way to obtain this information and perform the calculation.

The *Click* handler concludes by setting the *IsOpen* property of the *Popup* to *true*, and here it is:



The *Popup* is automatically dismissed when the user taps anywhere outside the *Popup*, and then the user needs to tap once more to dismiss the application bar. Like *AppBar*, *Popup* has *Opened* and *Closed* events if you need to perform some initialization or cleanup. For example, it's possible to install a handler for the *Closed* event of *Popup* and use that to set the *IsOpen* property of the *AppBar* to *false*.

Windows Runtime File I/O

MetroPad1 has Open and Save As buttons. If it were a real application, it would also have New and Save buttons and it would prompt you to save a file if you pressed New or Open without saving your previous work. That logic is coming up in XamlCruncher. The more modest goal here is to introduce you to the Windows Runtime *FileOpenPicker* and *FileSavePicker* classes, as well as some rudimentary Windows Runtime file I/O.

If you're familiar with the .NET *System.IO* namespace, you can leverage some of what you already know, but the Windows 8 version of *System.IO* might look a bit emaciated in comparison. Be prepared for plenty of new file I/O classes and concepts. The whole file and stream interface has been revamped,

and any method that accesses a disk is asynchronous. Fortunately, C# 5.0 has introduced two new keywords *await* and *async*, which make working with asynchronous methods very easy. But first I want to show you how to use these asynchronous methods without *await* and *async*.

The *FileOpenPicker* and *FileSavePicker* classes are defined in the *Windows.Storage.Pickers* namespace. These pickers take over the screen from your application and don't return control to the application until they have completed. If this is unacceptable to you, you'll probably want to explore the *FolderInformation* class in the *Windows.Storage.BulkAccess* namespace for obtaining files and subdirectories on your own.

The *FileOpenPicker* and *FileSavePicker* classes deliver an object of type *StorageFile* back to your application. *StorageFile* is defined in the *Windows.Storage* namespace and represents an unopened file. Calling one of the *Open* methods on this *StorageFile* object gives you a stream object represented as an interface such as *IInputStream* or *IRandomAccessStream*. You can then attach a *DataReader* or *DataWriter* object to this stream for reading or writing. The stream classes and interfaces are found in *Windows.Storage.Streams*. Through extension methods defined in *System.IO*, it's also possible to create a .NET Stream object from the Windows Runtime object, and then use some familiar .NET objects, such as *StreamReader* or *StreamWriter*, for dealing with these files. You might be able to salvage some existing code that uses .NET streams, and you'll also need these .NET stream objects for reading and writing XML files.

The only prerequisite for invoking *FileOpenPicker* is adding at least one string to the *FileTypeFilter* collection (for example, ".txt"). You then call the *PickSingleFileAsync* method.

Notice the last five letters of that method name: *Async*, short for "asynchronous." That's a very important sequence of five letters in the Windows Runtime.

As you know, a Windows Metro program is similar to a Windows Desktop program in being event-driven and structured much like a state machine. Following initialization, a program sits dormant in memory waiting for events. Very often these events signal activity in the user interface. Sometimes they signal systemwide changes, such as a switch in the orientation of the display. Sometimes they signal that a file download has progressed or completed or failed.

It's important that applications process events as quickly as possible and then return control back to the operating system to wait for more events. If an application doesn't process events quickly, it could become unresponsive. For this reason, applications should relegate very lengthy jobs to secondary threads of execution. The thread devoted to the user interface should remain free and unencumbered of heavy processing.

But what if a particular method call in the Windows Runtime itself takes a long time? Is an application programmer expected to anticipate that problem and put that call in a secondary thread?

No, that seems unreasonable. For that reason, when the Microsoft developers were designing the Windows Runtime, they attempted to identify any method call that could require more than 50 milliseconds to return control to the application. Approximately 10–15% of the Windows Runtime

qualified. These methods were made asynchronous, meaning that the methods themselves spin off secondary threads to do the lengthy processing. They return control back to the application very quickly and later notify the application when they've completed.

These asynchronous methods are all identified with the *Async* suffix, and they all have similar definition patterns.

The method call in *FileOpenPicker* class that displays the picker and returns a file selected by the user definitely will not return control to the program in under 50 milliseconds. Consequently, instead of a method call named *PickSingleFile*, it has a method call named *PickSingleFileAsync*.

Here's the *Click* handler for the application bar Open button in MetroPad1 showing the creation and initialization of *FileOpenPicker*, and the *PickSingleFileAsync* call:

Project: MetroPad1 | File: BlankPage.xaml.cs (excerpt)

```
void OnOpenAppBarButtonClick(object sender, RoutedEventArgs args)
{
    FileOpenPicker picker = new FileOpenPicker();
    picker.FileTypeFilter.Add(".txt");
    IAsyncOperation<StorageFile> asyncOp = picker.PickSingleFileAsync();
    asyncOp.Completed = OnPickSingleFileCompleted;
}

void OnPickSingleFileCompleted(IAsyncOperation<StorageFile> asyncInfo, AsyncStatus asyncStatus)
{
    …
}
```

PickSingleFileAsync returns quickly, but instead of returning with a *StorageFile* selected by the user, it returns an object of the generic type *IAsyncOperation<StorageFile>*. That *StorageFile* type is important, because that's what the *PickSingleFileAsync* will *eventually* deliver to your program but just not right away. For that reason, sometimes an object like *IAsyncOperation* is called a "future" or a "promise."

IAsyncOperation<T> derives from the *IAsyncInfo* interface, which defines methods named *Cancel* and *Close* and properties named *Id*, *Status*, and *ErrorCode*. The *IAsyncOperation<T>* interface additionally defines a property named *Completed*, which you'll notice is set in this code.

This *Completed* property is a delegate of type *AsyncOperationCompletedHandler<T>*. Although it's defined as a property, it functions like an event. (The difference is that an event can have multiple handlers but a property can have only one.)

To actually initiate the display of the *FileOpenPicker*, your program simply sets the *Completed* property of the *IAsyncOperation<StorageFile>* object to a method of the required type, named here *OnPickSingleFileCompleted*.

There is no separate *Start* method. Setting the *Completed* property starts it going but perhaps not right away. If your method contains any code after the *Completed* property is set, that code will be executed first. Only after *OnOpenAppBarButtonClick* returns control back to the operating system is the

file picker displayed. The user then interacts with it.

When the user selects a file from the picker and presses OK (or presses Cancel), Windows is ready to deliver a file back to your program. The *Completed* callback method in your program (here called *OnPickSingleFileCompleted*) is called with a first argument that is the same object that *PickSingleFileAsync* returned, but I've given it a somewhat different name (*asyncInfo*) because now it actually has some information for us.

If an error occurred—which is unlikely in this case—the *ErrorCode* property of this *asyncInfo* argument is non-*null* and equals an *Exception* object that describes the problem. (For the most part I will be ignoring errors in this little exercise.) Otherwise, the *Completed* handler calls *GetResults* on the *IAsyncOperation* object. This returns on object of type *StorageFile* indicating the file selected by the user. However, if *GetResults* returns *null*, the user dismissed the file picker by pressing Cancel, and there's nothing further to do. Here's the code so far:

Project: MetroPad1 | File: BlankPage.xaml.cs (excerpt)

```
void OnPickSingleFileCompleted(IAsyncOperation<StorageFile> asyncInfo, AsyncStatus asyncStatus)
{
    if (asyncInfo.ErrorCode != null)
        return;

    StorageFile storageFile = asyncInfo.GetResults();

    if (storageFile == null)
        return;
    …
}
```

To open that *StorageFile* object for reading, you can call *OpenReadAsync* on it. Oh no! That's another asynchronous operation! Of course, it makes sense that opening a file is asynchronous because the call must access the disk, and that could take longer than 50 milliseconds. So, similar to the first case, *OpenReadAsync* returns an object of type (hold your breath) *IAsyncOperation<IRandomAccessStreamWithContentType>*. Once again, set the *Completed* handler to start the operation going.

Here's the complete *OnPickSingleFileCompleted* handler with the handler for the *Completed* event of *OpenReadAsync*:

Project: MetroPad1 | File: BlankPage.xaml.cs (excerpt)

```
void OnPickSingleFileCompleted(IAsyncOperation<StorageFile> asyncInfo, AsyncStatus asyncStatus)
{
    if (asyncInfo.ErrorCode != null)
        return;

    StorageFile storageFile = asyncInfo.GetResults();

    if (storageFile == null)
        return;
```

```
    IAsyncOperation<IRandomAccessStreamWithContentType> asyncOp = storageFile.OpenReadAsync();
    asyncOp.Completed = OnFileOpenReadCompleted;
}

void OnFileOpenReadCompleted(IAsyncOperation<IRandomAccessStreamWithContentType> asyncInfo,
                             AsyncStatus asyncStatus)
{
    …
}
```

When *OnFileOpenReadCompleted* is called, the file has been opened and is ready for reading and the *GetResults* method of the *asyncInfo* argument returns an object of type *IRandomAccessStreamWithContentType*. You create a *DataReader* object based on this stream, and the next step is to call *LoadAsync* to actually read the contents of the file into an internal buffer. Another asynchronous operation requires another *Completed* handler:

Project: MetroPad1 | File: BlankPage.xaml.cs (excerpt)

```
DataReader dataReader;
…
void OnFileOpenReadCompleted(IAsyncOperation<IRandomAccessStreamWithContentType> asyncInfo,
                             AsyncStatus asyncStatus)
{
    if (asyncInfo.ErrorCode != null)
        return;

    using (IRandomAccessStreamWithContentType stream = asyncInfo.GetResults())
    {
        using (dataReader = new DataReader(stream))
        {
            uint length = (uint)stream.Size;
            DataReaderLoadOperation asyncOp = dataReader.LoadAsync(length);
            asyncOp.Completed = OnDataReaderLoadCompleted;
        }
    }
}

void OnDataReaderLoadCompleted(IAsyncOperation<uint> asyncInfo, AsyncStatus asyncStatus)
{
    …
}
```

Both *IRandomAccessStreamWithContentType* and *DataReader* implement *IClosable* (which is the same as the .NET *IDisposable*), so they appear in *using* statements to automatically close and dispose of the object when it's no longer needed. Also notice that the *DataReader* is saved as a field.

A call to the *OnDataReaderLoadCompleted* handler indicates that the file is now present in memory, so the contents can be transferred to the *TextBox*.

Not so fast!

When you set the *Completed* property of a method like *LoadAsync*, the *DataReader* class creates a secondary thread of execution that performs the job of accessing the file and reading it into memory.

The *Completed* handler in your code is then called, and it runs in that secondary thread. You cannot access user interface objects from that thread.

For any particular window, there can be only one application thread that handles user input and displays graphics that interact with this input. This "UI thread" (as it's called) is consequently very important and very special to Windows applications because all interaction with the user must occur through this thread.

This prohibition can be generalized: *DependencyObject* is not thread safe. Any object based on a class that derives from *DependencyObject* can only be accessed by the thread that creates that object.

In the particular problem we've encountered, the code that transfers text into a *TextBox* must run in the UI thread. Fortunately, there's a way to do it. To compensate for the fact that it's not thread safe, *DependencyObject* has a property named *Dispatcher* that returns an object of type *CoreDispatcher*. The *HasThreadAccess* property of *CoreDispatcher* lets you know if you can access this particular *DependencyObject* from the thread in which the code is running. If you can't (and even if you can), you can put a chunk of code on a queue for execution by the thread that created the object. You do this by calling the *Invoke* method referencing a method in your code that will run in the proper thread.

Here's the *OnDataReaderLoadCompleted* method calling *Invoke* on the *Dispatcher* property of the page. It doesn't matter whose *CoreDispatcher* object you use; because all the user interface objects were created in the same UI thread, they all work identically. The last argument passed to *Invoke* is passed to the handler as the *Context* property of the event arguments:

Project: MetroPad1 | File: BlankPage.xaml.cs (excerpt)

```
void OnDataReaderLoadCompleted(IAsyncOperation<uint> asyncInfo, AsyncStatus asyncStatus)
{
    if (asyncInfo.ErrorCode != null)
        return;

    uint length = asyncInfo.GetResults();
    string text = dataReader.ReadString(length);

    this.Dispatcher.Invoke(CoreDispatcherPriority.Normal,
                           SetTextBoxText, this, text);
}

void SetTextBoxText(object sender, InvokedHandlerArgs args)
{
     string text = args.Context as string;
     txtbox.Text = text;
}
```

The *SetTextBoxText* method runs in the UI thread so that it can safely set the text from the file into the *TextBox*.

Very often, the method to be executed in the UI thread is passed as an anonymous method to *Invoke*, like this:

```
this.Dispatcher.Invoke(CoreDispatcherPriority.Normal, (sender, args) =>
    {
        txtbox.Text = text;
    },
    this, null);
```

Indeed, all the *Completed* methods can be defined as anonymous methods, and that's what I've done for the logic to save a file in MetroPad1.

Even so, saving to a file is potentially more involved that opening a file because four asynchronous operations are involved: *PickSaveFileAsync* on the *FileSavePicker*, *OpenAsync* on the *StorageFile* to get a stream from which to create a *DataWriter*, and then, after calling *WriteString* on this *DataWriter*, calling *StoreAsync* and *FlushAsync*. However, there are some shortcuts. The *FileIO* class in *Windows.Storage* contains static methods that can read and write entire *StorageFile* objects in one big gulp.

In summary, in implementing the Save As button, I've used the shortcut methods for the file I/O and anonymous methods for the *Completed* handler and *Invoke* method. Everything goes in the *Click* handler for the button:

Project: MetroPad1 — File: BlankPage.xaml.cs (excerpt)

```
void OnSaveAsAppBarButtonClick(object sender, RoutedEventArgs args)
{
    FileSavePicker picker = new FileSavePicker();
    picker.DefaultFileExtension = ".txt";
    picker.FileTypeChoices.Add("Text", new List<string> { ".txt" });

    picker.PickSaveFileAsync().Completed = (asyncInfo, asyncStatus) =>
        {
            if (asyncInfo.ErrorCode != null)
                return;

            StorageFile storageFile = asyncInfo.GetResults();

            if (storageFile == null)
                return;

            string text = null;

            this.Dispatcher.Invoke(CoreDispatcherPriority.Normal,
                                   (dispatcherSender, dispatcherArgs) =>
                {
                    text = txtbox.Text;
                },
                this, null);

            FileIO.WriteTextAsync(storageFile, text).Completed = (asyncInfo2, asyncStatus2) =>
                {
                };
        };
}
```

This actually isn't too bad, but as the number of nested anonymous handlers builds up, the structure

can become quite awkward and particularly messy to trace program flow or implement a simple *return* statement.

Another solution is desperately needed. Fortunately, it exists.

Await and *Async*

The C# 5.0 keyword *await* allows us to work with asynchronous operations as if they were relatively normal method calls. The MetroPad2 program is the same as MetroPad1 except for the processing of the Open and Save As buttons on the application bar. Here's the *Click* handler for the Save As button:

Project: MetroPad2 — File: BlankPage.xaml.cs (excerpt)

```
async void OnSaveAsAppBarButtonClick(object sender, RoutedEventArgs args)
{
    FileSavePicker picker = new FileSavePicker();
    picker.DefaultFileExtension = ".txt";
    picker.FileTypeChoices.Add("Text", new List<string> { ".txt" });

    // Asynchronous call!
    StorageFile storageFile = await picker.PickSaveFileAsync();

    if (storageFile == null)
        return;

    // Asynchronous call!
    await FileIO.WriteTextAsync(storageFile, txtbox.Text);
}
```

Notice the two occurrences of *await* preceded with comments. *PickSaveFileAsync* actually returns an *IAsyncOperation* on which you must normally set a *Completed* handler and then call *GetResults* in the *Completed* callback to get a *StorageFile* object. The *await* operator seems to bypass all the messy stuff and simply return the *StorageFile* directly. And that's exactly what it does, except not quite right away.

It looks like magic, but much of the messy implementation details are now hidden. The C# compiler generates the callback and the *GetResults* call. But what the *await* operator also does is turn the method in which it's used into a state machine. The *OnSaveAsAppBarButtonClick* method begins executing normally, until *PickSaveFileAsync* is called and the first *await* appears.

Despite its name, that *await* does *not* wait until the operation completes. Instead, the *Click* handler is exited at that point. Control returns back to Windows. Other code on the program's user interface thread can then run, as can the file-picker itself. When the file-picker is dismissed, and a result is ready, and the UI thread is ready to run some code, execution of the *Click* handler continues with the assignment to the *storageFile* variable and then continues until the next *await* operator. And so forth with as many *await* operators as you like until the method completes.

The last line in this *Click* handler calls the static *FileIO.WriteTextAsync* method. Strictly speaking, the *await* operator is not needed here because conclusion of the *Click* handler doesn't need to wait for this

method to conclude. The *FileIO.WriteTextAsync* method doesn't return anything, and nothing else in the *Click* handler is dependent on its conclusion. That *await* operator can be removed in this case and the program will work the same:

```
FileIO.WriteTextAsync(storageFile, txtbox.Text);
```

You'll get a warning message from the compiler, but it's OK. The *await* operator is crucial only when you need a return value from the asynchronous method or when the method must complete before program logic continues.

Here's the *Click* handler for the Open button now using the static *FileIO.ReadTextAsync* shortcut method:

Project: MetroPad2 | File: BlankPage.xaml.cs (excerpt)

```
async void OnOpenAppBarButtonClick(object sender, RoutedEventArgs args)
{
    FileOpenPicker picker = new FileOpenPicker();
    picker.FileTypeFilter.Add(".txt");

    // Asynchronous call!
    StorageFile storageFile = await picker.PickSingleFileAsync();

    if (storageFile == null)
        return;

    // Asynchronous call!
    txtbox.Text = await FileIO.ReadTextAsync(storageFile);
}
```

Prior to *await*, calling asynchronous operations in C# always seemed to me to violate the imperative structure of the language. The *await* operator brings back that imperative structure and turns asynchronous calls into what appears to be a series of sequential normal method calls. Moreover, everything in those *Click* handlers now runs in the UI thread, so you don't have to worry about accessing user interface objects. But despite the ease of *await*, you'll probably want to keep in mind that a method in which *await* appears is actually chopped up into pieces behind the scenes.

There are some restrictions on the *await* operator. It cannot appear in the *catch* or *finally* clause of an exception handler. However, it can appear in the *try* clause, and this is how you'll trap errors that occur in the asynchronous method. There are also ways to cancel operations, and some asynchronous methods report progress as well. More details in the later chapter devoted to asynchronous operations.

The method in which the *await* operator appears must be flagged as *async*, but the *async* keyword doesn't do much of anything. In earlier versions of C#, *await* was not a keyword, so programmers could use the word for variable names or property names or whatever. Adding a new *await* keyword to C# 5.0 would break this code, but restricting *await* to methods flagged with *async* avoids that problem. The *async* modifier does not change the signature of the method—the method above is still a valid *Click* handler. But you can't use *async* (and hence *await*) with methods that serve as entry points, such as *Main* or class constructors.

If you need to call asynchronous methods while initializing a *FrameworkElement* derivative, do them in the handler for the *Loaded* event and flag it as *async*:

```
public BlankPage()
{
    this.InitializeComponent();
    …
    Loaded += OnLoaded;
}
async void OnLoaded(object sender, RoutedEventArgs arg)
{
    …
}
```

Or, if you prefer defining the Loaded handler as an anonymous method:

```
public BlankPage()
{
    this.InitializeComponent();
    …
    Loaded += async (sender, args) =>
        {
            …
        };
}
```

See the *async* before the argument list?

Calling Your Own *Async* Methods

Suppose you want to isolate the file-save logic in a method like this:

```
async void SaveFile(string text)
{
    FileSavePicker picker = new FileSavePicker();
    picker.DefaultFileExtension = ".txt";
    picker.FileTypeChoices.Add("Text", new List<string> { ".txt" });
    StorageFile storageFile = await picker.PickSaveFileAsync();

    if (storageFile == null)
        return;

    await FileIO.WriteTextAsync(storageFile, txtbox.Text);
}
```

The method must be flagged as *async* because it contains *await* keywords. You can then call this method from *OnSaveAsAppBarButtonClick* like so:

```
void OnSaveAsAppBarButtonClick(object sender, RoutedEventArgs args)
{
    SaveFile(txtbox.Text);
}
```

What happens here is that *OnSaveAsAppBarButtonClick* calls *SaveFile* and *SaveFile* begins executing until the first *await* on the *PickSaveFileAsync* call. At that point, *SaveFile* returns control back to *OnSaveAsAppBarButtonClick* and that method terminates. When *PickSaveFileAsync* has a result ready, the rest of the *SaveFile* method proceeds.

In this particular case, this might be OK. However, if you want the *OnSaveAsAppBarButtonClick* method to await the execution of *SaveFile*, it must include an *await* keyword and be flagged as *async*:

```
async void OnSaveAsAppBarButtonClick(object sender, RoutedEventArgs args)
{
    await SaveFile(txtbox.Text);
}
```

But when you do that, *SaveFile* must be changed as well. It can no longer return void. You can modify *SaveFile* in several ways, but perhaps the easiest is simply changing the return type to *Task*:

```
async Task SaveFile(string text)
{
    …
}
```

At this point, you probably also want to change the name of the method to *SaveFileAsync* to indicate that it's an asynchronous method that can be awaited. Although the code that you write in this *SaveFileAsync* method does not run in a secondary thread, the other asynchronous methods that *SaveFileAsync* calls do so.

A similar separation of *OnOpenAppBarButtonClick* and a *ReadFileAsync* method is a little different. You probably want the *ReadFileAsync* method to return the text contents of the file, so the return type isn't *Task* but *Task<string>*:

```
async Task<string> ReadFileAsync()
{
    FileOpenPicker picker = new FileOpenPicker();
    picker.FileTypeFilter.Add(".txt");
    StorageFile storageFile = await picker.PickSingleFileAsync();

    if (storageFile == null)
        return null;

    return await FileIO.ReadTextAsync(storageFile);
}
```

You can then call *ReadFileAsync* like so:

```
async void OnOpenAppBarButtonClick(object sender, RoutedEventArgs args)
{
    string text = await ReadFileAsync();

    if (text != null)
        txtbox.Text = text;
}
```

It's important to realize that all of this code runs in the user interface thread. You don't have to worry about accessing user interface objects. The two methods in the *FileIO* class certainly spin off secondary threads to do work, but the *ReadFileAsync* method I've shown is considered to be asynchronous only because it calls other asynchronous methods. The other code in the method runs in the user interface thread.

If you're writing some of your own code that requires a lot of processing time, you don't want to do that job in the user interface thread. But instead of using traditional techniques to create and execute threads, consider using a task-based approach like the Windows Runtime. You can execute program code asynchronously by passing it as a method to the static *Task.Run* method. Generally this is done as an anonymous method:

```
Task<double> BigJobAsync(int arg1, int arg2)
{
    return Task.Run<double>(() =>
    {
        double val = 0;
        // ... lengthy code
        return val;
    });
}
```

Everything in that anonymous method runs in a secondary thread, and hence it cannot access user interface objects. You can then call this method like so:

```
double value = await BigJobAsync(22, 33);
```

If, perchance, the anonymous method in *BigJobAsync* includes its own *await* operators, you would need to flag the anonymous method as *async*:

```
Task<double> BigJobAsync(int arg1, int arg2)
{
    return Task.Run<double>(async () =>
    {
        double val = 0;
        // ... lengthy code
        return val;
    });
}
```

I'll have much more to say about asynchronous processing in the chapter devoted to the subject.

Controls for XamlCruncher

Now that you've seen some rudimentary file I/O and asynchronous processing, it's time to start looking at XamlCruncher. I won't pretend that this program is commercial grade or even that it doesn't have some serious flaws. But it's a real program with real Metro style features, and I'll surely be enhancing it to fix any problems or deficiencies the first version might have.

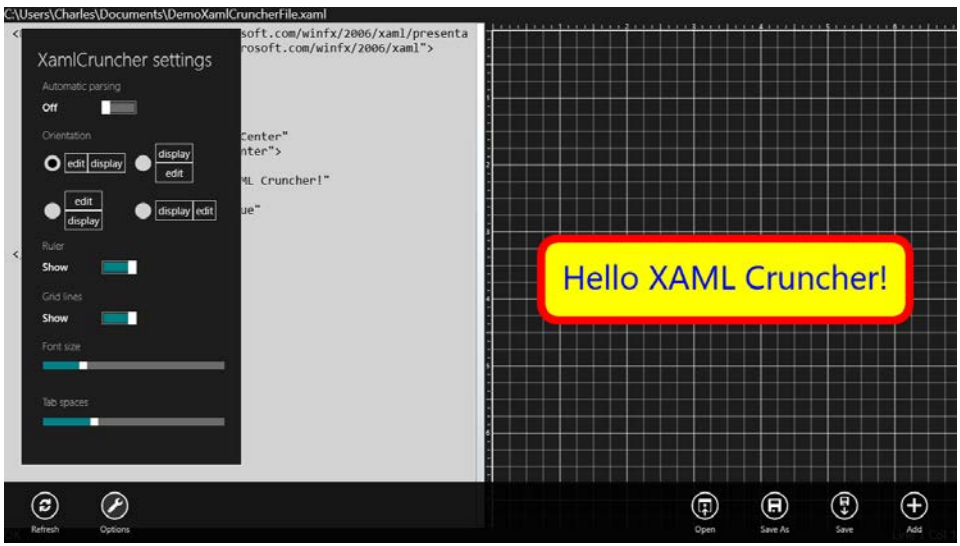
XamlCruncher lets you type in XAML and see the resultant objects and visual tree. The magic method that XamlCruncher uses is *XamlReader.Load*, which you had a brief glimpse of in the PathMarkupSyntaxCode project in Chapter 2. The XAML processed by *XamlReader.Load* cannot reference event handlers or external assemblies.

Here's a view of the program with some XAML in the editor on the left and the resultant objects in a display area on the right:

The editor doesn't include any amenities. It won't even automatically generate a closing tag when you type a start tag; it doesn't use different colors for elements, attributes, and strings; and it doesn't have anything close to IntelliSense. However, the configuration of the page is changeable: you can put the edit window on the top, right, or bottom.

The application bar has Add, Open, Save, and Save As buttons as well as a Refresh button and a button for application options:

You can select whether XamlCruncher reparses the XAML with each keystroke or only with a press of the Refresh button. That option and others are available from the dialog invoked when you press the Options button:



I've turned on the Ruler and Grid Lines options to show you the result in the display area on the right. All these options are saved for the next time the program is run.

Most of the page is a custom *UserControl* derivative called *SplitContainer*. In the center is a *Thumb* control that lets you select the proportion of space in the left and right panels (or top and bottom panels). In the screen shots, this *Thumb* is a lighter gray vertical bar in the center of the screen. The XAML file for *SplitContainer* consists of a *Grid* defined for both horizontal and vertical configurations:

Project: XamlCruncher | File: SplitContainer.xaml

```
<UserControl
    x:Class="XamlCruncher.SplitContainer"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:XamlCruncher">

    <Grid>
        <!-- Default Orientation is Horizontal -->
        <Grid.ColumnDefinitions>
            <ColumnDefinition x:Name="coldef1" Width="*" MinWidth="100" />
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition x:Name="coldef2" Width="*" MinWidth="100" />
        </Grid.ColumnDefinitions>

        <!-- Alternative Orientation is Vertical -->
        <Grid.RowDefinitions>
            <RowDefinition x:Name="rowdef1" Height="*" />
            <RowDefinition Height="Auto" />
            <RowDefinition x:Name="rowdef2" Height="0" />
        </Grid.RowDefinitions>

        <Grid Name="grid1"
              Grid.Row="0"
              Grid.Column="0" />

        <Thumb Name="thumb"
               Grid.Row="0"
               Grid.Column="1"
               Width="12"
               DragStarted="OnThumbDragStarted"
               DragDelta="OnThumbDragDelta" />

        <Grid Name="grid2"
              Grid.Row="0"
              Grid.Column="2" />
    </Grid>
</UserControl>
```

You've seen similar markup in the OrientableColorScroll program, which altered a *Grid* when the aspect ratio of the page changed between landscape and portrait.

The code-behind file defines five properties backed by dependency properties. Normally you'll set the *Child1* and *Child2* properties to the elements to appear in the left and right of the control, but where they actually appear is governed by the *Orientation* and *SwapChildren* properties:

Project: XamlCruncher | File: SplitContainer.xaml.cs (excerpt)

```
public sealed partial class SplitContainer : UserControl
{
    // Static constructor and properties
    static SplitContainer()
    {
        Child1Property =
```

```
            DependencyProperty.Register("Child1",
                typeof(UIElement), typeof(SplitContainer),
                new PropertyMetadata(null, OnChildChanged));

        Child2Property =
            DependencyProperty.Register("Child2",
                typeof(UIElement), typeof(SplitContainer),
                new PropertyMetadata(null, OnChildChanged));

        OrientationProperty =
            DependencyProperty.Register("Orientation",
                typeof(Orientation), typeof(SplitContainer),
                new PropertyMetadata(Orientation.Horizontal, OnOrientationChanged));

        SwapChildrenProperty =
            DependencyProperty.Register("SwapChildren",
                typeof(bool), typeof(SplitContainer),
                new PropertyMetadata(false, OnSwapChildrenChanged));

        MinimumSizeProperty =
            DependencyProperty.Register("MinimumSize",
                typeof(double), typeof(SplitContainer),
                new PropertyMetadata(100.0, OnMinSizeChanged));
    }

    public static DependencyProperty Child1Property { private set; get; }
    public static DependencyProperty Child2Property { private set; get; }
    public static DependencyProperty OrientationProperty { private set; get; }
    public static DependencyProperty SwapChildrenProperty { private set; get; }
    public static DependencyProperty MinimumSizeProperty { private set; get; }

    // Instance constructor and properties
    public SplitContainer()
    {
        this.InitializeComponent();
    }

    public UIElement Child1
    {
        set { SetValue(Child1Property, value); }
        get { return (UIElement)GetValue(Child1Property); }
    }

    public UIElement Child2
    {
        set { SetValue(Child2Property, value); }
        get { return (UIElement)GetValue(Child2Property); }
    }

    public Orientation Orientation
    {
        set { SetValue(OrientationProperty, value); }
        get { return (Orientation)GetValue(OrientationProperty); }
    }
```

```
    public bool SwapChildren
    {
        set { SetValue(SwapChildrenProperty, value); }
        get { return (bool)GetValue(SwapChildrenProperty); }
    }

    public double MinimumSize
    {
        set { SetValue(MinimumSizeProperty, value); }
        get { return (double)GetValue(MinimumSizeProperty); }
    }
    …
}
```

The *Orientation* property is of type *Orientation*, the same enumeration used for *StackPanel* and *VariableSizedWrapGrid*. It's always nice to use existing types for dependency properties rather than inventing your own. Notice that the *MinimumSize* is of type *double* and hence is initialized as 100.0 rather than 100 to prevent a type mismatch at run time.

The property-changed handlers show two different approaches that programmers use in calling the instance property-changed handler from the static handler. I've already shown you the approach where the static handler simply calls the instance handler with the same *DependencyPropertyChangedEventArgs* object. Sometimes—as with the handlers for the *Orientation*, *SwapChildren*, and *MinimumSize* properties—it's more convenient for the static handler to call the instance handler with the old value and new value cast to the proper type:

Project: XamlCruncher | File: SplitContainer.xaml.cs (excerpt)

```
public sealed partial class SplitContainer : UserControl
{
    …
    // Property changed handlers
    static void OnChildChanged(DependencyObject obj,
                               DependencyPropertyChangedEventArgs args)
    {
        (obj as SplitContainer).OnChildChanged(args);
    }

    void OnChildChanged(DependencyPropertyChangedEventArgs args)
    {
        Grid targetGrid = (args.Property == Child1Property ^ this.SwapChildren) ? grid1 : grid2;
        targetGrid.Children.Clear();

        if (args.NewValue != null)
            targetGrid.Children.Add(args.NewValue as UIElement);
    }

    static void OnOrientationChanged(DependencyObject obj,
                                     DependencyPropertyChangedEventArgs args)
    {
        (obj as SplitContainer).OnOrientationChanged((Orientation)args.OldValue,
                                                     (Orientation)args.NewValue);
```

```
}

void OnOrientationChanged(Orientation oldOrientation, Orientation newOrientation)
{
    // Shouldn't be necessary, but...
    if (newOrientation == oldOrientation)
        return;

    if (newOrientation == Orientation.Horizontal)
    {
        coldef1.Width = rowdef1.Height;
        coldef2.Width = rowdef2.Height;

        coldef1.MinWidth = this.MinimumSize;
        coldef2.MinWidth = this.MinimumSize;

        rowdef1.Height = new GridLength(1, GridUnitType.Star);
        rowdef2.Height = new GridLength(0);

        rowdef1.MinHeight = 0;
        rowdef2.MinHeight = 0;

        thumb.Width = 12;
        thumb.Height = Double.NaN;

        Grid.SetRow(thumb, 0);
        Grid.SetColumn(thumb, 1);

        Grid.SetRow(grid2, 0);
        Grid.SetColumn(grid2, 2);
    }
    else
    {
        rowdef1.Height = coldef1.Width;
        rowdef2.Height = coldef2.Width;

        rowdef1.MinHeight = this.MinimumSize;
        rowdef2.MinHeight = this.MinimumSize;

        coldef1.Width = new GridLength(1, GridUnitType.Star);
        coldef2.Width = new GridLength(0);

        coldef1.MinWidth = 0;
        coldef2.MinWidth = 0;

        thumb.Height = 12;
        thumb.Width = Double.NaN;

        Grid.SetRow(thumb, 1);
        Grid.SetColumn(thumb, 0);

        Grid.SetRow(grid2, 2);
        Grid.SetColumn(grid2, 0);
    }
```

```
    }

    static void OnSwapChildrenChanged(DependencyObject obj,
                                      DependencyPropertyChangedEventArgs args)
    {
        (obj as SplitContainer).OnSwapChildrenChanged((bool)args.OldValue,
                                                      (bool)args.NewValue);
    }

    void OnSwapChildrenChanged(bool oldOrientation, bool newOrientation)
    {
        grid1.Children.Clear();
        grid2.Children.Clear();

        grid1.Children.Add(newOrientation ? this.Child2 : this.Child1);
        grid2.Children.Add(newOrientation ? this.Child1 : this.Child2);
    }

    static void OnMinSizeChanged(DependencyObject obj,
                                 DependencyPropertyChangedEventArgs args)
    {
        (obj as SplitContainer).OnMinSizeChanged((double)args.OldValue,
                                                 (double)args.NewValue);
    }

    void OnMinSizeChanged(double oldValue, double newValue)
    {
        if (this.Orientation == Orientation.Horizontal)
        {
            coldef1.MinWidth = newValue;
            coldef2.MinWidth = newValue;
        }
        else
        {
            rowdef1.MinHeight = newValue;
            rowdef2.MinHeight = newValue;
        }
    }
    …
}
```

My original version of the property-changed handler for *Orientation* assumed that the *Orientation* property was actually changing, as should be the case whenever a property-changed handler is called. However, I discovered that sometimes the property-changed handler was called when the property was set to its existing value.

All that's left is looking at the event handlers for the *Thumb*. The idea here is that the two columns (or rows) of the *Grid* are allocated size based on the star specification so that the relative size of the columns (or rows) remains the same when the size or aspect ratio of the *Grid* changes. However, to keep the *Thumb* dragging logic reasonably simple, it helps if the numeric proportions associated with the star specifications are actual pixel dimensions. These are initialized in the *OnThumbDragStarted* method and changed in *OnDragThumbDelta*:

Project: XamlCruncher | File: SplitContainer.xaml.cs (excerpt)

```
public sealed partial class SplitContainer : UserControl
{
    …
    // Thumb event handlers
    void OnThumbDragStarted(object sender, DragStartedEventArgs args)
    {
        if (this.Orientation == Orientation.Horizontal)
        {
            coldef1.Width = new GridLength(coldef1.ActualWidth, GridUnitType.Star);
            coldef2.Width = new GridLength(coldef2.ActualWidth, GridUnitType.Star);
        }
        else
        {
            rowdef1.Height = new GridLength(rowdef1.ActualHeight, GridUnitType.Star);
            rowdef2.Height = new GridLength(rowdef2.ActualHeight, GridUnitType.Star);
        }
    }

    void OnThumbDragDelta(object sender, DragDeltaEventArgs args)
    {
        if (this.Orientation == Orientation.Horizontal)
        {
            double newWidth1 = Math.Max(0, coldef1.Width.Value + args.HorizontalChange);
            double newWidth2 = Math.Max(0, coldef2.Width.Value - args.HorizontalChange);

            coldef1.Width = new GridLength(newWidth1, GridUnitType.Star);
            coldef2.Width = new GridLength(newWidth2, GridUnitType.Star);
        }
        else
        {
            double newHeight1 = Math.Max(0, rowdef1.Height.Value + args.VerticalChange);
            double newHeight2 = Math.Max(0, rowdef2.Height.Value - args.VerticalChange);

            rowdef1.Height = new GridLength(newHeight1, GridUnitType.Star);
            rowdef2.Height = new GridLength(newHeight2, GridUnitType.Star);
        }
    }
}
```

The last of the earlier screen shots of XamlCruncher showed a ruler and grid lines in the display area. The ruler is in units of inches, based on 96 pixels to the inch, so the grid lines are 24 pixels apart. The ruler and grid lines are useful if you're interactively designing some vector graphics or other precise layout.

The ruler and grid lines are independently optional. The *UserControl* derivative that displays them is called *RulerContainer*. As you'll see when the XamlCruncher page is constructed, an instance of *RulerContainer* is set to the *Child2* property of the *SplitContainer* object. Here's the XAML file for *RulerContainer*:

Project: XamlCruncher | File: RulerContainer.xaml (excerpt)

```
<UserControl … >
```

```
    <Grid SizeChanged="OnGridSizeChanged">
        <Canvas Name="rulerCanvas" />
        <Grid Name="innerGrid">
            <Grid Name="gridLinesGrid" />
            <Border Name="border" />
        </Grid>
    </Grid>
</UserControl>
```

This *RulerContainer* control has a *Child* property, and the child of this control is set to the *Child* property of the *Border*. Visually behind this *Border* is the grid of horizontal and vertical lines, which are children of the *Grid* labeled "gridLinesGrid." If the ruler is also present, the *Grid* labeled "innerGrid" is given a nonzero *Margin* on the left and top to accommodate this ruler. The tick marks and numbers that comprise the ruler are children of the *Canvas* named "rulerCanvas."

Here's all the overhead for the dependency property definitions in the code-behind file:

Project: XamlCruncher | File: RulerContainer.xaml.cs (excerpt)

```
public sealed partial class RulerContainer : UserControl
{
    …
    static RulerContainer()
    {
        ChildProperty =
            DependencyProperty.Register("Child",
                typeof(UIElement), typeof(RulerContainer),
                new PropertyMetadata(null, OnChildChanged));

        ShowRulerProperty =
            DependencyProperty.Register("ShowRuler",
                typeof(bool), typeof(RulerContainer),
                new PropertyMetadata(false, OnShowRulerChanged));

        ShowGridLinesProperty =
            DependencyProperty.Register("ShowGridLines",
                typeof(bool), typeof(RulerContainer),
                new PropertyMetadata(false, OnShowGridLinesChanged));
    }

    public static DependencyProperty ChildProperty { private set; get; }
    public static DependencyProperty ShowRulerProperty { private set; get; }
    public static DependencyProperty ShowGridLinesProperty { private set; get; }

    public RulerContainer()
    {
        this.InitializeComponent();
    }

    public UIElement Child
    {
        set { SetValue(ChildProperty, value); }
        get { return (UIElement)GetValue(ChildProperty); }
    }
```

```
    public bool ShowRuler
    {
        set { SetValue(ShowRulerProperty, value); }
        get { return (bool)GetValue(ShowRulerProperty); }
    }

    public bool ShowGridLines
    {
        set { SetValue(ShowGridLinesProperty, value); }
        get { return (bool)GetValue(ShowGridLinesProperty); }
    }

    // Property changed handlers
    static void OnChildChanged(DependencyObject obj,
                               DependencyPropertyChangedEventArgs args)
    {
        (obj as RulerContainer).border.Child = (UIElement)args.NewValue;
    }

    static void OnShowRulerChanged(DependencyObject obj,
                                   DependencyPropertyChangedEventArgs args)
    {
        (obj as RulerContainer).RedrawRuler();
    }

    static void OnShowGridLinesChanged(DependencyObject obj,
                                       DependencyPropertyChangedEventArgs args)
    {
        (obj as RulerContainer).RedrawGridLines();
    }

    void OnGridSizeChanged(object sender, SizeChangedEventArgs args)
    {
        RedrawRuler();
        RedrawGridLines();
    }

    …

}
```

Also shown here are the property-changed handlers (which are simple enough to use in the static versions) as well as the *SizeChanged* handler for the *Grid*. Two redraw methods handle all the drawing, which involves creating *Line* elements and *TextBlock* elements and organizing them in the two panels:

```
public sealed partial class RulerContainer : UserControl
{
    const double RULER_WIDTH = 12;
    …
    void RedrawGridLines()
    {
        gridLinesGrid.Children.Clear();
```

```
    if (!this.ShowGridLines)
        return;

    // Vertical grid lines every 1/4"
    for (double x = 24; x < gridLinesGrid.ActualWidth; x += 24)
    {
        Line line = new Line
        {
            X1 = x,
            Y1 = 0,
            X2 = x,
            Y2 = gridLinesGrid.ActualHeight,
            Stroke = this.Foreground,
            StrokeThickness = x % 96 == 0 ? 1 : 0.5
        };
        gridLinesGrid.Children.Add(line);
    }

    // Horizontal grid lines every 1/4"
    for (double y = 24; y < gridLinesGrid.ActualHeight; y += 24)
    {
        Line line = new Line
        {
            X1 = 0,
            Y1 = y,
            X2 = gridLinesGrid.ActualWidth,
            Y2 = y,
            Stroke = this.Foreground,
            StrokeThickness = y % 96 == 0 ? 1 : 0.5
        };
        gridLinesGrid.Children.Add(line);
    }
}

void RedrawRuler()
{
    rulerCanvas.Children.Clear();

    if (!this.ShowRuler)
    {
        innerGrid.Margin = new Thickness();
        return;
    }

    innerGrid.Margin = new Thickness(RULER_WIDTH, RULER_WIDTH, 0, 0);

    // Ruler across the top
    for (double x = 0; x < gridLinesGrid.ActualWidth - RULER_WIDTH; x += 12)
    {
        // Numbers every inch
        if (x > 0 && x % 96 == 0)
        {
            TextBlock txtblk = new TextBlock
            {
```

```
            Text = (x / 96).ToString("F0"),
            FontSize = RULER_WIDTH - 2
        };

        txtblk.Measure(new Size());
        Canvas.SetLeft(txtblk, RULER_WIDTH + x - txtblk.ActualWidth / 2);
        Canvas.SetTop(txtblk, 0);
        rulerCanvas.Children.Add(txtblk);
    }
    // Tick marks every 1/8"
    else
    {
        Line line = new Line
        {
            X1 = RULER_WIDTH + x,
            Y1 = x % 48 == 0 ? 2 : 4,
            X2 = RULER_WIDTH + x,
            Y2 = x % 48 == 0 ? RULER_WIDTH - 2 : RULER_WIDTH - 4,
            Stroke = this.Foreground,
            StrokeThickness = 1
        };
        rulerCanvas.Children.Add(line);
    }
}

// Heavy line underneath the tick marks
Line topLine = new Line
{
    X1 = RULER_WIDTH - 1,
    Y1 = RULER_WIDTH - 1,
    X2 = rulerCanvas.ActualWidth,
    Y2 = RULER_WIDTH - 1,
    Stroke = this.Foreground,
    StrokeThickness = 2
};
rulerCanvas.Children.Add(topLine);

// Ruler down the left side
for (double y = 0; y < gridLinesGrid.ActualHeight - RULER_WIDTH; y += 12)
{
    // Numbers every inch
    if (y > 0 && y % 96 == 0)
    {
        TextBlock txtblk = new TextBlock
        {
            Text = (y / 96).ToString("F0"),
            FontSize = RULER_WIDTH - 2,
        };

        txtblk.Measure(new Size());
        Canvas.SetLeft(txtblk, 2);
        Canvas.SetTop(txtblk, RULER_WIDTH + y - txtblk.ActualHeight / 2);
        rulerCanvas.Children.Add(txtblk);
    }
```

```
            // Tick marks every 1/8"
            else
            {
                Line line = new Line
                {
                    X1 = y % 48 == 0 ? 2 : 4,
                    Y1 = RULER_WIDTH + y,
                    X2 = y % 48 == 0 ? RULER_WIDTH - 2 : RULER_WIDTH - 4,
                    Y2 = RULER_WIDTH + y,
                    Stroke = this.Foreground,
                    StrokeThickness = 1
                };
                rulerCanvas.Children.Add(line);
            }
        }

        Line leftLine = new Line
        {
            X1 = RULER_WIDTH - 1,
            Y1 = RULER_WIDTH - 1,
            X2 = RULER_WIDTH - 1,
            Y2 = rulerCanvas.ActualHeight,
            Stroke = this.Foreground,
            StrokeThickness = 2
        };
        rulerCanvas.Children.Add(leftLine);
    }
}
```

These two methods make extensive use of the *Line* element, which renders a single straight line between the points (*X1*, *Y1*) and (*X2*, *Y2*). This *RedrawRuler* code also illustrates a technique for obtaining the rendered size of a *TextBlock*.

When you create a new *TextBlock*, the *ActualWidth* and *ActualHeight* properties are both zero. These properties are normally not calculated until the *TextBlock* becomes part of a visual tree and is subjected to layout. However, you can force the *TextBlock* to calculate a size for itself by calling its *Measure* method. This method is defined by *UIElement* and is an important component of the layout system.

The argument to the *Measure* method is a *Size* value indicating the size available for the element, but you can set the size to zero for this purpose:

```
txtblk.Measure(new Size());
```

If you need to find the size of a *TextBlock* that wraps text, you must supply a nonzero first argument to the *Size* constructor so that *TextBlock* knows the width in which to wrap the text.

Following the *Measure* call, the *ActualWidth* and *ActualHeight* properties of *TextBlock* are valid and usable for positioning the *TextBlock* in a *Canvas*. Calling the *Canvas.SetLeft* and *Canvas.SetTop* properties is necessary only when positioning the *TextBlock* elements in the *Canvas*. In either a single-cell *Grid* or *Canvas*, the *Line* elements are positioned based on their coordinates.

As you'll see, an instance of *RulerContainer* is set to the *Child2* property of the *SplitContainer* that dominates the XamlCruncher page. The *Child1* property appears to be a *TextBox*, but it's actually an instance of another custom control named *TabbableTextBox*, which derives from *TextBox*.

The standard *TextBox* does not respond to the Tab key, and when you're typing XAML into an editor, you really want tabs. That's the primary feature of *TabbableTextBox*, shown here in its entirety:

Project: XamlCruncher | File: TabbableTextBox.cs

```
using Windows.System;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Input;

namespace XamlCruncher
{
    public class TabbableTextBox : TextBox
    {
        static TabbableTextBox()
        {
            TabSpacesProperty =
                DependencyProperty.Register("TabSpaces",
                    typeof(int), typeof(TabbableTextBox),
                    new PropertyMetadata(4));
        }

        public static DependencyProperty TabSpacesProperty { private set; get; }

        public int TabSpaces
        {
            set { SetValue(TabSpacesProperty, value); }
            get { return (int)GetValue(TabSpacesProperty); }
        }

        public bool IsModified { set; get; }

        protected override void OnKeyDown(KeyEventArgs args)
        {
            this.IsModified = true;

            if (args.Key == VirtualKey.Tab)
            {
                int line, col;
                GetPositionFromIndex(this.SelectionStart, out line, out col);
                int insertCount = this.TabSpaces - col % this.TabSpaces;
                this.SelectedText = new string(' ', insertCount);
                this.SelectionStart += insertCount;
                this.SelectionLength = 0;
                args.Handled = true;
                return;
            }
            base.OnKeyDown(args);
        }
```

```
        public void GetPositionFromIndex(int index, out int line, out int col)
        {
            if (index > Text.Length)
            {
                line = col = -1;
                return;
            }

            line = col = 0;
            bool justFoundEol = false;

            for (int i = 0; i < index; i++)
            {
                if (Text[i] == '\r' || Text[i] == '\n')
                {
                    if (!justFoundEol)
                        line++;

                    col = 0;
                    justFoundEol = true;
                }
                else
                {
                    col++;
                    justFoundEol = false;
                }
            }
            return;
        }
    }
}
```

The class intercepts the *OnKeyDown* method to determine if the Tab key is being pressed. If that's the case, it inserts blanks into the *Text* object so that the cursor moves to a text column that is an integral multiple of the *TabSpaces* property. This calculation requires knowing the character position of the cursor on the current line. To obtain this information, it uses the *GetPositionFromIndex* method also defined in this class. This method is public and is also used by XamlCruncher to display the current position of the cursor and the current selection (if any).

Another property—not backed by a dependency property—is also defined by *TabbableTextBox*. This is *IsModified*, which is set to *true* whenever a *KeyDown* event occurs.

Like many programs that deals with documents, XamlCruncher keeps track if the text file has changed since the last save. If the user initiates an operation to create a new file or open an existing file, and the current document is in a modified state, the program asks if the user wants to save that document.

Often this logic occurs entirely external to the *TextBox* control. The program sets an *IsModified* flag to *true* when a new file is loaded or the file is saved and to *false* on receipt of a *TextChanged* event. However, the *TextChanged* event is fired when the *Text* property of the *TextBox* is set programmatically, so even if the *TextBox* is being set to a newly loaded file, the *TextChanged* event is

fired and the *IsModified* flag would be set by the *TextChanged* handler. You might think that setting the *IsModified* flag in that case might be avoided by setting a flag when the *Text* property is set programmatically. However, the *TextChanged* handler is not called until the method setting the *Text* property has returned control back to the operating system, which makes the logic rather messy. Implementing the *IsModified* flag in the *TextBox* derivative helps.

Application Settings and Isolated Storage

Many applications maintain user settings and preferences between invocations of the program. The Windows Runtime provides an area of application data storage (sometimes known as "isolated storage") specifically for the use of the application in storing information of this sort. A program obtains access to this storage through the *ApplicationData* class in the *Windows.Storage* namespace.

An instance of *ApplicationData* applicable for the current application is available from the static *ApplicationData.Current* method. From that object, a *TemporaryFolder* property provides a disk area suitable for temporary data. Other properties—*LocalFolder*, *LocalSettings*, *RoamingFolder*, and *RoamingSettings*—are also available for storing more permanent data.

The *LocalSettings* property gives you access to a dictionary in which you can store program settings with names and values. But I don't like to use this. I prefer to store program settings in an XML file that is serialized from a class in the program that I generally called *AppSettings*. This class implements *INotifyPropertyChanged* so that it can be used for data binding. It's basically a View Model, or perhaps (in larger applications) part of a View Model. An XML file serialized from *AppSettings* can be stored in the *AppicationData.Current.LocalFolder* directory, which, you'll discover, maps to this location on the machine's main drive:

/Users/[username]/AppData/Local/Packages/[package family name]/LocalState

The *[username]* is the user's name on the computer, and *[package family name]* is mostly a GUID that uniquely identifies the application. For any Visual Studio application project you can find this name by opening the Package.appmanifest file and clicking the Packaging tab.

One program option that should be saved is the orientation of the edit and display areas. As you'll recall, the *SplitContainer* has two properties named *Orientation* and *SwapChildren*. For storing user settings, I wanted something more specific to this application. The *TextBox* (or rather, the *TabbableTextBox*) can be on the left, top, right, or bottom, and this enumeration encapsulates those options:

Project: XamlCruncher | File: EditOrientation.cs

```
namespace XamlCruncher
{
    public enum EditOrientation
    {
        Left, Top, Right, Bottom
    }
```

```
}
```

Here's the first half of *AppSettings* showing all the properties that comprise program settings. The class derives from *BindableBase* to implement *INotifyPropertyChanged*. All the property values are backed by fields initialized with the program's default settings. Notice that the *EditOrientation* property is based on the *EditOrientation* enumeration:

Project: XamlCruncher | File: AppSettings.cs

```
public class AppSettings : XamlCruncher.Common.BindableBase
{
    …
    // Application settings initial values
    EditOrientation editOrientation = EditOrientation.Left;
    Orientation orientation = Orientation.Horizontal;
    bool swapEditAndDisplay = false;
    bool autoParsing = false;
    bool showRuler = false;
    bool showGridLines = false;
    double fontSize = 18;
    int tabSpaces = 8;

    public EditOrientation EditOrientation
    {
        set
        {
            if (SetProperty<EditOrientation>(ref editOrientation, value))
            {
                switch (editOrientation)
                {
                    case EditOrientation.Left:
                        this.Orientation = Orientation.Horizontal;
                        this.SwapEditAndDisplay = false;
                        break;

                    case EditOrientation.Top:
                        this.Orientation = Orientation.Vertical;
                        this.SwapEditAndDisplay = false;
                        break;

                    case EditOrientation.Right:
                        this.Orientation = Orientation.Horizontal;
                        this.SwapEditAndDisplay = true;
                        break;

                    case EditOrientation.Bottom:
                        this.Orientation = Orientation.Vertical;
                        this.SwapEditAndDisplay = true;
                        break;
                }
            }
        }
        get { return editOrientation; }
    }
```

```
    [XmlIgnore]
    public Orientation Orientation
    {
        protected set { SetProperty<Orientation>(ref orientation, value); }
        get { return orientation; }
    }

    [XmlIgnore]
    public bool SwapEditAndDisplay
    {
        protected set { SetProperty<bool>(ref swapEditAndDisplay, value); }
        get { return swapEditAndDisplay; }
    }

    public bool AutoParsing
    {
        set { SetProperty<bool>(ref autoParsing, value); }
        get { return autoParsing; }
    }

    public bool ShowRuler
    {
        set { SetProperty<bool>(ref showRuler, value); }
        get { return showRuler; }
    }

    public bool ShowGridLines
    {
        set { SetProperty<bool>(ref showGridLines, value); }
        get { return showGridLines; }
    }

    public double FontSize
    {
        set { SetProperty<double>(ref fontSize, value); }
        get { return fontSize; }
    }

    public int TabSpaces
    {
        set { SetProperty<int>(ref tabSpaces, value); }
        get { return tabSpaces; }
    }
    …
}
```

Besides *EditOrientation*, *AppSettings* defines two additional properties that more directly correspond to properties of the *SplitContainer*. These are *Orientation* and *SwapEditAndDisplay*. The *set* accessors are protected, and the properties are set only from the *set* accessor of *EditOrientation*. These two properties are also flagged with the attribute *XmlIgnore*, indicating that these properties should be ignored when the *AppSettings* object is serialized into XML. They are not actually part of application settings, but they are easily derived from application settings and make the bindings easier.

AppSettings also has methods to serialize an instance of itself to XML and save it as a file and to deserialize that file back into an *AppSettings* instance. The *LoadAsync* and *SaveAsync* methods to load and save these files are, as the names suggest, asynchronous. They use a combination of Windows Runtime classes (*StorageFolder*, *StorageFile*, and *FileIO*) and .NET classes (*XmlSerializer*, *StringReader*, and *StringWriter*).

The *LoadAsync* method must be static because it is the only way to create an instance of *AppSettings*. Some programmers like to use a static property called *Current* for this purpose to ensure that *AppSettings* is a singleton—in other words, to ensure that only one instance of *AppSettings* exists anywhere in the program.

Project: XamlCruncher | File: AppSettings.cs (excerpt)

```
public class AppSettings : XamlCruncher.Common.BindableBase
{
    const string FILENAME = "applicationsettings.xml";

    …

    public async static Task<AppSettings> LoadAsync()
    {
        StorageFolder storageFolder = ApplicationData.Current.LocalFolder;
        StorageFile appSettingsFile = null;
        AppSettings appSettings = null;

        try
        {
            appSettingsFile = await storageFolder.GetFileAsync(FILENAME);
        }
        catch (Exception)
        {
            // This happens the first time the program is run
        }

        if (appSettingsFile == null)
        {
            appSettings = new AppSettings();
        }
        else
        {
            string str = await FileIO.ReadTextAsync(appSettingsFile);
            XmlSerializer xmlSerializer = new XmlSerializer(typeof(AppSettings));

            using (StringReader reader = new StringReader(str))
            {
                appSettings = xmlSerializer.Deserialize(reader) as AppSettings;
            }
        }
        return appSettings;
    }

    public async Task SaveAsync()
    {
```

```
        string settingsXml = null;

        using (StringWriter stringWriter = new StringWriter())
        {
            XmlSerializer xmlSerializer = new XmlSerializer(typeof(AppSettings));
            xmlSerializer.Serialize(stringWriter, this);
            settingsXml = stringWriter.ToString();
        }
        StorageFolder storageFolder = ApplicationData.Current.LocalFolder;
        StorageFile storageFile = null;

        try
        {
            storageFile = await storageFolder.CreateFileAsync(FILENAME,
                                                CreationCollisionOption.ReplaceExisting);
        }
        catch (Exception)
        {
            // TODO: This shouldn't happen, but it might
        }
        await FileIO.WriteTextAsync(storageFile, settingsXml);
    }
}
```

When the program is first run, *AppSettings.LoadAsync* attempts to access the settings file, but it won't exist. An exception is thrown, and instead the method simply instantiates *AppSettings*. That instance will have all default values.

The XamlCruncher Page

Sufficient pieces have now been created to let us begin assembling this application. Here's BlankPage.xaml:

Project: XamlCruncher | File: BlankPage.xaml (excerpt)

```
<Page … >

    <Grid Background="{StaticResource ApplicationPageBackgroundBrush}">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>

        <TextBlock Name="filenameText"
                   Grid.Row="0"
                   Grid.Column="0"
```

```
                Grid.ColumnSpan="2"
                FontSize="18"
                TextTrimming="WordEllipsis" />

    <local:SplitContainer x:Name="splitContainer"
                                Orientation="{Binding Orientation}"
                                SwapChildren="{Binding SwapEditAndDisplay}"
                                MinimumSize="200"
                                Grid.Row="1"
                                Grid.Column="0"
                                Grid.ColumnSpan="2">
        <local:SplitContainer.Child1>
            <local:TabbableTextBox x:Name="editBox"
                                    AcceptsReturn="True"
                                    FontSize="{Binding FontSize}"
                                    TabSpaces="{Binding TabSpaces}"
                                    TextChanged="OnEditBoxTextChanged"
                                    SelectionChanged="OnEditBoxSelectionChanged"/>
        </local:SplitContainer.Child1>

        <local:SplitContainer.Child2>
            <local:RulerContainer x:Name="resultContainer"
                                ShowRuler="{Binding ShowRuler}"
                                ShowGridLines="{Binding ShowGridLines}" />
        </local:SplitContainer.Child2>
    </local:SplitContainer>

    <TextBlock Name="statusText"
               Text="OK"
               Grid.Row="2"
               Grid.Column="0"
               FontSize="18"
               TextWrapping="Wrap" />

    <TextBlock Name="lineColText"
               Grid.Row="2"
               Grid.Column="1"
               FontSize="18" />
</Grid>

<Page.BottomAppBar>
    <AppBar Padding="10 0">
        <Grid>
            <StackPanel Orientation="Horizontal" HorizontalAlignment="Left">
                <Button Style="{StaticResource RefreshAppBarButtonStyle}"
                        Click="OnRefreshAppBarButtonClick" />

                <Button Style="{StaticResource AppBarButtonStyle}"
                        Content="&#xE15E;"
                        AutomationProperties.Name="Options"
                        Click="OnOptionsAppBarButtonClick" />
            </StackPanel>

            <StackPanel Orientation="Horizontal" HorizontalAlignment="Right">
```

```
                    <Button Style="{StaticResource AppBarButtonStyle}"
                            Content="&#xE184;"
                            AutomationProperties.Name="Open"
                            Click="OnOpenAppBarButtonClick"
                            />

                    <Button Style="{StaticResource SaveAppBarButtonStyle}"
                            AutomationProperties.Name="Save As"
                            Click="OnSaveAsAppBarButtonClick" />

                    <Button Style="{StaticResource AppBarButtonStyle}"
                            Content="&#xE159;"
                            AutomationProperties.Name="Save"
                            Click="OnSaveAppBarButtonClick" />

                    <Button Style="{StaticResource AddAppBarButtonStyle}"
                            Click="OnAddAppBarButtonClick" />
                </StackPanel>
            </Grid>
        </AppBar>
    </Page.BottomAppBar>
</Page>
```

The main *Grid* has three rows:

- for the name of the loaded file (the *TextBlock* named "filenameText"),
- the *SplitContainer*,
- and the status bar at the bottom.

The status bar consists of two *TextBlock* elements named "statusText" (to indicate possible XAML parsing errors) and "lineColText" (for the line and column of the *TabbableTextBox*). The *Grid* is further divided into two columns for the two components of that status bar.

Most of the page is occupied by the *SplitContainer*, and you'll see that it contains bindings to the *Orientation* and *SwapEditAndDisplay* properties of *AppSettings*. The *SplitContainer* contains a *TabbableTextBox* (with bindings to the *FontSize* and *TabSpaces* properties of *AppSettings*) and a *RulerContainer* (with bindings to *ShowRuler* and *ShowGridLines*). All these bindings strongly suggest that the *DataContext* of *BlankPage* is set to an instance of *AppSettings*.

The bottom of the XAML file has the *Button* definitions for the application bar.

As you might expect, the code-behind file is the longest file in the project, but I'm going to discuss it in various modular sections so that the discussion won't be too overwhelming. Here's the constructor, *Loaded* handler and a few simple methods:

Project: XamlCruncher | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    …
    AppSettings appSettings;
```

```
StorageFile loadedStorageFile;

public BlankPage()
{
    this.InitializeComponent();

    …

    // Why aren't these set in the generated C# files?
    editBox = splitContainer.Child1 as TabbableTextBox;
    resultContainer = splitContainer.Child2 as RulerContainer;

    // Set a fixed-pitch font for the TextBox
    Language language = new Language();
    LanguageFontGroup languageFontGroup = new LanguageFontGroup(language.LanguageTag);
    LanguageFont languageFont = languageFontGroup.FixedWidthTextFont;
    editBox.FontFamily = new FontFamily(languageFont.FontFamily);

    Loaded += OnLoaded;
}

async void OnLoaded(object sender, RoutedEventArgs args)
{
    // Load AppSettings and set to DataContext
    appSettings = await AppSettings.LoadAsync();
    this.DataContext = appSettings;

    // Other initialization
    await SetDefaultXamlFile();
    ParseText();
    editBox.Focus(FocusState.Keyboard);
    DisplayLineAndColumn();
    …
}

async Task SetDefaultXamlFile()
{
    editBox.Text =
        "<Page xmlns=\"http://schemas.microsoft.com/winfx/2006/xaml/presentation\"\r\n" +
        "      xmlns:x=\"http://schemas.microsoft.com/winfx/2006/xaml\">\r\n\r\n" +
        "    <TextBlock Text=\"Hello, Windows 8!\"\r\n" +
        "               FontSize=\"48\" />\r\n\r\n" +
        "</Page>";

    editBox.IsModified = false;
    loadedStorageFile = null;
    filenameText.Text = "";
}
…
void OnEditBoxSelectionChanged(object sender, RoutedEventArgs args)
{
    DisplayLineAndColumn();
}
```

```
    void DisplayLineAndColumn()
    {
        int line, col;
        editBox.GetPositionFromIndex(editBox.SelectionStart, out line, out col);
        lineColText.Text = String.Format("Line {0} Col {1}", line + 1, col + 1);

        if (editBox.SelectionLength > 0)
        {
            editBox.GetPositionFromIndex(editBox.SelectionStart + editBox.SelectionLength - 1,
                                         out line, out col);
            lineColText.Text += String.Format(" - Line {0} Col {1}", line + 1, col + 1);
        }
    }
    …
}
```

The constructor begins by fixing a little bug involving the *editBox* and *resultContainer* fields. The XAML parser definitely creates these fields during compilation, but they not set by the *InitializeComponent* call at run time.

The remainder of the constructor sets a fixed-pitch font in the *TabbableTextBox* based on the predefined fonts available from the *LanguageFontGroup* class. This is apparently the only way to get actual font family names from the Windows Runtime.

The remaining initialization occurs in the *Loaded* event handler because it needs to call the *AppSettings.LoadAsync* asynchronous method and asynchronous methods can't be called in constructors. The *DataContext* of the page is set to the *AppSettings* instance, as you probably anticipated from the data bindings in the BlankPage.xaml file.

The *OnLoaded* method begins by setting a default piece of XAML in the *TabbableTextBox* and calling *ParseText* to parse it. (You'll see how this works soon.) The *TabbableTextBox* is assigned keyboard input focus, and *OnLoaded* concludes by displaying the initial line and column, which is then updated whenever the *TextBox* selection changes.

You might wonder why *SetDefaultXamlFile* is defined as *async* and returns *Task* when it does not actually contain any asynchronous code. You'll see later that this method is used as an argument to another method in the file I/O logic, and that's the sole reason I had to define it oddly. The compiler generates a warning message because it doesn't contain any *await* logic.

Parsing the XAML

The major job of XamlCruncher is to pass a piece of XAML to *XamlReader.Load* and get out an object. A property of the *AppSettings* class named *AutoParsing* allows this to happen with every keystroke, or it waits until you press the Refresh button on the application bar.

If *XamlReader.Load* encounters an error, it raises an exception, and the program then displays that error in red in the status bar at the bottom of the page and also colors the text in the *TabbableTextBox*

red.

Project: XamlCruncher | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    Brush textBlockBrush, textBoxBrush, errorBrush;
    …
    public BlankPage()
    {
        …
        // Set brushes
        textBlockBrush = Resources["ApplicationTextBrush"] as SolidColorBrush;
        textBoxBrush = Resources["TextBoxTextBrush"] as SolidColorBrush;
        errorBrush = new SolidColorBrush(Colors.Red);
        …
    }

    …

    void OnRefreshAppBarButtonClick(object sender, RoutedEventArgs args)
    {
        ParseText();
        this.BottomAppBar.IsOpen = false;
    }
    …
    void OnEditBoxTextChanged(object sender, RoutedEventArgs e)
    {
        if (appSettings.AutoParsing)
            ParseText();
    }

    void ParseText()
    {
        object result = null;

        try
        {
            result = XamlReader.Load(editBox.Text);
        }
        catch (Exception exc)
        {
            SetErrorText(exc.Message);
            return;
        }

        if (result == null)
        {
            SetErrorText("Null result");
        }
        else if (!(result is UIElement))
        {
            SetErrorText("Result is " + result.GetType().Name);
        }
        else
```

```
            {
                resultContainer.Child = result as UIElement;
                SetOkText();
                return;
            }
        }

        void SetErrorText(string text)
        {
            SetStatusText(text, errorBrush, errorBrush);
        }

        void SetOkText()
        {
            SetStatusText("OK", textBlockBrush, textBoxBrush);
        }

        void SetStatusText(string text, Brush statusBrush, Brush editBrush)
        {
            statusText.Text = text;
            statusText.Foreground = statusBrush;
            editBox.Foreground = editBrush;
        }
}
```

It could be that a chunk of XAML successfully passes *XamlReader.Load* with no errors but then raises an exception later on. This can happen particularly when XAML animations are involved because the animation doesn't start up until the visual tree is loaded.

The only real solution is to install a handler for the *UnhandledException* event defined by the *Application* object, and that's done in the conclusion of the *Loaded* handler:

Project: XamlCruncher | File: BlankPage.xaml.cs (excerpt)

```
async void OnLoaded(object sender, RoutedEventArgs args)
{
    …
    Application.Current.UnhandledException += (excSender, excArgs) =>
        {
            SetErrorText(excArgs.Message);
            excArgs.Handled = true;
        };
}
```

The problem with something like this is that you want to make sure that the program isn't going to have some other kind of unhandled exception that isn't a result of some errant XAML.

Also, when Visual Studio is running a program in its debugger, it wants to snag the unhandled exceptions so that it can report them to you. Use the Exceptions dialog from the Debug menu to indicate which exceptions you want Visual Studio to intercept and which should be left to the program.

XAML Files In and Out

Whenever I approach the code involved in loading and saving documents, I always think it's going to be easier than it turns out to be. Here's the basic problem. Whenever a New or Open command occurs, you need to check if the current document has been modified without being saved. If that's the case, a message box should be displayed asking whether the user wants to save the file. The options are Save, Don't Save, and Cancel.

The easy answer is Cancel. The program doesn't need to do anything further. If the user selects the Don't Save option, the current document can be abandoned and the New or Open command can proceed.

If the user answers Save, the existing document needs to be saved under its filename. But that filename might not exist if the document wasn't loaded from a disk file or previously saved. At that point, the Save As dialog box needs to be displayed. But the user can select Cancel from that dialog box as well, and the New or Open operation ends. Otherwise, the existing file is first saved.

Let's first look at the methods involved in saving documents. The application button has Save and Save As buttons, but the Save button needs to invoke the Save As dialog box if it doesn't have a filename for the document:

Project: XamlCruncher | File: BlankPage.xaml.cs (excerpt)

```
async void OnSaveAsAppBarButtonClick(object sender, RoutedEventArgs args)
{
     StorageFile storageFile = await GetFileFromSavePicker();

     if (storageFile == null)
         return;

     await SaveXamlToFile(storageFile);
}

async void OnSaveAppBarButtonClick(object sender, RoutedEventArgs args)
{
    Button button = sender as Button;
    button.IsEnabled = false;

    if (loadedStorageFile != null)
    {
        await SaveXamlToFile(loadedStorageFile);
    }
    else
    {
        StorageFile storageFile = await GetFileFromSavePicker();

        if (storageFile != null)
        {
            await SaveXamlToFile(storageFile);
        }
```

```
    }
    button.IsEnabled = true;
}

async Task<StorageFile> GetFileFromSavePicker()
{
    FileSavePicker picker = new FileSavePicker();
    picker.DefaultFileExtension = ".xaml";
    picker.FileTypeChoices.Add("XAML", new List<string> { ".xaml" });
    picker.SuggestedSaveFile = loadedStorageFile;
    return await picker.PickSaveFileAsync();
}

async Task SaveXamlToFile(StorageFile storageFile)
{
    loadedStorageFile = storageFile;
    string exception = null;

    try
    {
        await FileIO.WriteTextAsync(storageFile, editBox.Text);
    }
    catch (Exception exc)
    {
        exception = exc.Message;
    }

    if (exception != null)
    {
        string message = String.Format("Could not save file {0}: {1}",
                                       storageFile.Name, exception);
        MessageDialog msgdlg = new MessageDialog(message, "XAML Cruncher");
        await msgdlg.ShowAsync();
    }
    else
    {
        editBox.IsModified = false;
        filenameText.Text = storageFile.Path;
    }
}
```

For the Save button, the handler disables the button and then enables it when it's completed. I'm worried that the button might be re-pressed during the time the file is being saved and there might even be a reentrancy problem if the handler tries to save it again when the first save hasn't completed. More research into how this problem can occur is surely warranted.

In the final method, the *FileIO.WriteTextAsync* call is in a *try* block. If an exception occurs while saving the file, the program wants to use *MessageDialog* to inform the user. But asynchronous methods such as *ShowAsync* can't be called in a *catch* block, so the exception is simply saved for checking afterward.

For both Add and Open, XamlCruncher needs to check if the file has been modified. If so, a

message box must be displayed to inform the user and request further direction. This occurs in a method I've called *CheckIfOkToTrashFile*. Because this method is applicable for both the Add and Open buttons, I gave this method an argument named *commandAction* of type *Func<Task>*, a delegate meaning a method with no arguments that returns a *Task*. The *Click* handler for the Open event passes the *LoadFileFromOpenPicker* method as this argument, and the handler for the Add button uses the aforementioned *SetDefaultXamlFile*.

Project: XamlCruncher | File: BlankPage.xaml.cs (excerpt)

```
async void OnAddAppBarButtonClick(object sender, RoutedEventArgs args)
{
    Button button = sender as Button;
    button.IsEnabled = false;
    await CheckIfOkToTrashFile(SetDefaultXamlFile);
    button.IsEnabled = true;
    this.BottomAppBar.IsOpen = false;
}

async void OnOpenAppBarButtonClick(object sender, RoutedEventArgs args)
{
    Button button = sender as Button;
    button.IsEnabled = false;
    await CheckIfOkToTrashFile(LoadFileFromOpenPicker);
    button.IsEnabled = true;
    this.BottomAppBar.IsOpen = false;
}

async Task CheckIfOkToTrashFile(Func<Task> commandAction)
{
    if (!editBox.IsModified)
    {
        await commandAction();
        return;
    }

    string message =
        String.Format("Do you want to save changes to {0}?",
            loadedStorageFile == null ? "(untitled)" : loadedStorageFile.Name);

    MessageDialog msgdlg = new MessageDialog(message, "XAML Cruncher");
    msgdlg.Commands.Add(new UICommand("Save", null, "save"));
    msgdlg.Commands.Add(new UICommand("Don't Save", null, "dont"));
    msgdlg.Commands.Add(new UICommand("Cancel", null, "cancel"));
    msgdlg.DefaultCommandIndex = 0;
    msgdlg.CancelCommandIndex = 2;
    IUICommand command = await msgdlg.ShowAsync();

    if ((string)command.Id == "cancel")
        return;

    if ((string)command.Id == "dont")
    {
        await commandAction();
        return;
```

```
    }

    if (loadedStorageFile == null)
    {
        StorageFile storageFile = await GetFileFromSavePicker();

        if (storageFile == null)
            return;

        loadedStorageFile = storageFile;
    }

    await SaveXamlToFile(loadedStorageFile);
    await commandAction();
}

async Task LoadFileFromOpenPicker()
{
    FileOpenPicker picker = new FileOpenPicker();
    picker.FileTypeFilter.Add(".xaml");
    StorageFile storageFile = await picker.PickSingleFileAsync();

    if (storageFile != null)
    {
        string exception = null;

        try
        {
            editBox.Text = await FileIO.ReadTextAsync(storageFile);
        }
        catch (Exception exc)
        {
            exception = exc.Message;
        }

        if (exception != null)
        {
            string message = String.Format("Could not load file {0}: {1}",
                                           storageFile.Name, exception);
            MessageDialog msgdlg = new MessageDialog(message, "XAML Cruncher");
            await msgdlg.ShowAsync();
        }
        else
        {
            editBox.IsModified = false;
            loadedStorageFile = storageFile;
            filenameText.Text = loadedStorageFile.Path;
        }
    }
}
```

The *CheckIfOkToTrashFile* method also demonstrates how additional commands are added to the *MessageDialog*. By default, the only button is labeled Close.

The Settings Dialog

When the user clicks the Options button, the handler instantiates a *UserControl* derivative named *SettingsDialog* and makes it the child of a *Popup*. Among these options is the orientation of the display. You'll recall I defined an *EditOrientation* enumeration for the four possibilities. Accordingly, the project also contains an *EditOrientationRadioButton* for storing one of the four values as a custom tag:

Project: XamlCruncher | File: EditOrientationRadioButton.cs

```
using Windows.UI.Xaml.Controls;

namespace XamlCruncher
{
    public class EditOrientationRadioButton : RadioButton
    {
        public EditOrientation EditOrientationTag { set; get; }
    }
}
```

The SettingsDialog.xaml file arranges all the controls in a *StackPanel*:

Project: XamlCruncher | File: SettingsDialog.xaml (excerpt)

```
<UserControl … >

    <UserControl.Resources>
        <Style x:Key="DialogCaptionTextStyle"
               TargetType="TextBlock"
               BasedOn="{StaticResource CaptionTextStyle}">
            <Setter Property="FontSize" Value="14.67" />
            <Setter Property="FontWeight" Value="SemiLight" />
            <Setter Property="Margin" Value="7 0 0 0" />
        </Style>
    </UserControl.Resources>

    <Border Background="{StaticResource ApplicationPageBackgroundBrush}"
            BorderBrush="{StaticResource ApplicationTextBrush}"
            BorderThickness="1">
        <StackPanel Margin="24">
            <TextBlock Text="XamlCruncher settings"
                       Style="{StaticResource SubheaderTextStyle}"
                       Margin="0 0 0 12" />

            <!-- Auto parsing -->
            <ToggleSwitch Header="Automatic parsing"
                          IsOn="{Binding AutoParsing, Mode=TwoWay}" />

            <!-- Orientation -->
            <TextBlock Text="Orientation"
                       Style="{StaticResource DialogCaptionTextStyle}" />

            <Grid Name="orientationRadioButtonGrid"
                  Margin="7 0 0 0">
```

```
<Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
</Grid.RowDefinitions>

<Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="Auto" />
</Grid.ColumnDefinitions>

<Grid.Resources>
    <Style TargetType="Border">
        <Setter Property="BorderBrush"
                Value="{StaticResource ApplicationTextBrush}" />
        <Setter Property="BorderThickness" Value="1" />
        <Setter Property="Padding" Value="3" />
    </Style>

    <Style TargetType="TextBlock">
        <Setter Property="TextAlignment" Value="Center" />
    </Style>

    <Style TargetType="local:EditOrientationRadioButton">
        <Setter Property="Margin" Value="0 6 12 6" />
    </Style>
</Grid.Resources>

<local:EditOrientationRadioButton Grid.Row="0" Grid.Column="0"
                                  EditOrientationTag="Left"
             Checked="OnOrientationRadioButtonChecked">
    <StackPanel Orientation="Horizontal">
        <Border>
            <TextBlock Text="edit" />
        </Border>
        <Border>
            <TextBlock Text="display" />
        </Border>
    </StackPanel>
</local:EditOrientationRadioButton>

<local:EditOrientationRadioButton Grid.Row="0" Grid.Column="1"
                                  EditOrientationTag="Bottom"
             Checked="OnOrientationRadioButtonChecked">
    <StackPanel>
        <Border>
            <TextBlock Text="display" />
        </Border>
        <Border>
            <TextBlock Text="edit" />
        </Border>
    </StackPanel>
</local:EditOrientationRadioButton>

<local:EditOrientationRadioButton Grid.Row="1" Grid.Column="0"
```

```
                                    EditOrientationTag="Top"
                 Checked="OnOrientationRadioButtonChecked">
        <StackPanel>
            <Border>
                <TextBlock Text="edit" />
            </Border>
            <Border>
                <TextBlock Text="display" />
            </Border>
        </StackPanel>
    </local:EditOrientationRadioButton>

    <local:EditOrientationRadioButton Grid.Row="1" Grid.Column="1"
                                    EditOrientationTag="Right"
                 Checked="OnOrientationRadioButtonChecked">
        <StackPanel Orientation="Horizontal">
            <Border>
                <TextBlock Text="display" />
            </Border>
            <Border>
                <TextBlock Text="edit" />
            </Border>
        </StackPanel>
    </local:EditOrientationRadioButton>
</Grid>

<!-- Ruler -->
<ToggleSwitch Header="Ruler"
              OnContent="Show"
              OffContent="Hide"
              IsOn="{Binding ShowRuler, Mode=TwoWay}" />

<!-- Grid lines -->
<ToggleSwitch Header="Grid lines"
              OnContent="Show"
              OffContent="Hide"
              IsOn="{Binding ShowGridLines, Mode=TwoWay}" />

<!-- Font size -->
<TextBlock Text="Font size"
           Style="{StaticResource DialogCaptionTextStyle}" />

<Slider Value="{Binding FontSize, Mode=TwoWay}"
        Minimum="10"
        Maximum="48"
        Margin="7 0 0 0" />

<!-- Tab spaces -->
<TextBlock Text="Tab spaces"
           Style="{StaticResource DialogCaptionTextStyle}" />

<Slider Value="{Binding TabSpaces, Mode=TwoWay}"
        Minimum="1"
        Maximum="12"
```

```
                    Margin="7 0 0 0" />
        </StackPanel>
    </Border>
</UserControl>
```

All the two-way bindings strongly suggest that the *DataContext* is set to an instance of *AppSettings*, just like *BlankPage*. It's actually the *same* instance of *AppSettings*, which means that any changes in this dialog are automatically applied to the program.

This means that you can't make a bunch of changes in the dialog and hit Cancel. There is no Cancel button. To compensate, it might make sense for a dialog to have a Defaults button that restores everything to its factory-new condition.

A significant chunk of the XAML file is devoted to the four *EditOrientationRadioButton* controls. The content of each of these is a *StackPanel* with two bordered *TextBlock* elements, to create a little graphic that resembles the four layout options you saw in the earlier screen shot (that is, the third screen shot in the "Controls for XamlCruncher" section).

The dialog contains three instances of *ToggleSwitch*. By default, the *OnContent* and *OffContent* properties are set to the text string "On" and "Off," but I thought "Show" and "Hide" were better for the ruler and grid displays.

ToggleSwitch also has a *Header* property that displays text above the switch. In the screen shot I just referred to, the labels "Automatic parsing," "Ruler," and "Grid lines" are all displayed by the *ToggleSwitch*. I thought the labels looked good, so I made an effort to duplicate the font and placement with the *Style* labeled as "DialogCaptionTextStyle."

A *Slider* is used to set the font size, which might seem reasonable, but I also use a *Slider* to set the number of tab spaces, which I'll admit doesn't seem reasonable at all. Even though the *AppSettings* class defines the *TabSpaces* property as an integer, the binding with the *Value* property of the *Slider* works regardless, and the *Slider* proves to be a convenient way to change the property.

The only chore left for the code-behind file is to manage the *RadioButton* controls:

Project: XamlCruncher | File: SettingsDialog.xaml.cs

```
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;

namespace XamlCruncher
{
    public sealed partial class SettingsDialog : UserControl
    {
        public SettingsDialog()
        {
            this.InitializeComponent();
            Loaded += OnLoaded;
        }

        // Initialize RadioButton for edit orientation
        void OnLoaded(object sender, RoutedEventArgs args)
```

```
        {
            AppSettings appSettings = DataContext as AppSettings;

            if (appSettings != null)
            {
                foreach (UIElement child in orientationRadioButtonGrid.Children)
                {
                    EditOrientationRadioButton radioButton = child as EditOrientationRadioButton;
                    radioButton.IsChecked =
                        appSettings.EditOrientation == radioButton.EditOrientationTag;
                }
            }
        }

        // Set EditOrientation based on checked RadioButton
        void OnOrientationRadioButtonChecked(object sender, RoutedEventArgs args)
        {
            AppSettings appSettings = DataContext as AppSettings;
            EditOrientationRadioButton radioButton = sender as EditOrientationRadioButton;

            if (appSettings != null)
                appSettings.EditOrientation = radioButton.EditOrientationTag;
        }
    }
}
```

The display of the dialog is very similar to the MetroPad programs:

Project: XamlCruncher | File: BlankPage.xaml.cs (excerpt)

```
public sealed partial class BlankPage : Page
{
    …
    void OnOptionsAppBarButtonClick(object sender, RoutedEventArgs args)
    {
        SettingsDialog settingsDialog = new SettingsDialog();
        settingsDialog.DataContext = appSettings;

        Popup popup = new Popup
        {
            Child = settingsDialog,
            IsLightDismissEnabled = true
        };

        settingsDialog.SizeChanged += (dialogSender, dialogArgs) =>
            {
                popup.VerticalOffset = this.ActualHeight - settingsDialog.ActualHeight
                                                          - this.BottomAppBar.ActualHeight - 24;
                popup.HorizontalOffset = 24;
            };

        popup.Closed += OnPopupClose;
        popup.IsOpen = true;
    }
```

```
    async void OnPopupClose(object sender, object args)
    {
        try
        {
            await appSettings.SaveAsync();
        }
        catch (Exception exc)
        {
        }
        this.BottomAppBar.IsOpen = false;
    }
    …
}
```

The *Closed* event handler for the *Popup* saves the updated settings.

What happens if XamlCruncher terminates (either normally or unexpectedly) when the *SettingsDialog* is still displayed? Well, any changes that the user made to the settings won't be saved. The same goes for a document that was modified, which is potentially a much greater loss.

One of the big "to do" items is to handle the *Suspending* event of the *App* object. This event indicates when Windows 8 is suspending an application but also when the application is about to terminate. My thinking now is that the program should save any edited document in the *LocalFolder* area and then check for the existence of the document the next time the program starts up. One philosophy holds that applications should seem to be continuous experiences even when they are terminated and restarted.

Beyond the Windows Runtime

Earlier I mentioned some limitations to the XAML that you can enter in XamlCruncher. Elements cannot have their events set, because events require event handlers and event handlers must be implemented in code. Nor can the XAML contain references to external classes or assemblies.

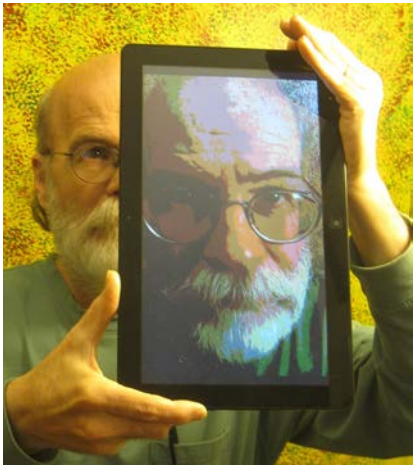
However, the parsed XAML runs in the XamlCruncher process, which means that it does have access to any classes that XamlCruncher has access to, including the custom classes I created for the program. Here's a piece of XAML that includes a namespace declaration for *local*. This enables it to use the *SplitContainer* and nests two instances of it:

This piece of XAML is among the downloadable code for this chapter, as is the XAML used for the earlier screen shots.

This is interesting, because it means that XamlCruncher really can go beyond the Windows Runtime and let you experiment with custom classes.

More to come.

Author Bio



Charles Petzold began programming for Windows 27 years ago with beta versions of Windows 1. He wrote the first articles about Windows programming to appear in a magazine and wrote one of the first books on the subject, *Programming Windows*, first published in 1988. Over the past decade, he has written seven books on .NET programming, including the recent *Programming Windows Phone 7* (Microsoft Press, 2010), and he currently writes a column on touch-oriented user interfaces for *MSDN Magazine*. Petzold's books also include *Code: The Hidden Language of Computer Hardware and Software* (Microsoft Press, 1999), a unique exploration of digital technologies, and *The Annotated Turing: A Guided Tour through Alan Turing's Historic Paper on Computability and the Turing Machine* (Wiley, 2008). His website is www.charlespetzold.com.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft® Press