

Programming Windows®

SIXTH EDITION

Release 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

Copyright © 2012 Charles Petzold

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

ISBN: 978-0-7356-7176-8

This document supports a preliminary release of a software product that may be changed substantially prior to final commercial release. This document is provided for informational purposes only and Microsoft makes no warranties, either express or implied, in this document. Information in this document, including URL and other Internet website references, is subject to change without notice. The entire risk of the use or the results from the use of this document remains with the user.

Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft and the trademarks listed at <http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions, Developmental, and Project Editor: Devon Musgrave

Technical Reviewer: Marc Young

Cover: Twist Creative • Seattle

Introduction 9

The Versions of Windows 8 9

The Focus of This Book 10

The Approach 12

My Setup 13

The *Programming Windows* Heritage 13

Behind the Scenes 16

Errata & Book Support 17

We Want to Hear from You 17

Stay in Touch 17

Chapter 1: Markup and Code 18

The First Project 18

Graphical Greetings 24

Variations in Text 28

Media As Well 36

The Code Alternatives 37

Images in Code 41

Not Even a Page 43

Chapter 2: XAML Syntax 45

The Gradient Brush in Code 45

Property Element Syntax 48

Content Properties 51

The *TextBlock* Content Property 55

Sharing Brushes (and Other Resources) 57

Resources Are Shared 61

A Bit of Vector Graphics 62

Stretching with *Viewbox* 71

Styles 74

A Taste of Data Binding 80

Chapter 3: Basic Event Handling	83
The <i>Tapped</i> Event.....	83
Routed Event Handling.....	86
Overriding the <i>Handled</i> Setting	92
Input, Alignment, and Backgrounds	93
Size and Orientation Changes	97
Bindings to <i>Run</i> ?	101
Timers and Animation	103
Chapter 4: The <i>Border</i> Element.....	111
The <i>Border</i> Element	111
Rectangle and Ellipse	115
The <i>StackPanel</i>	117
Horizontal Stacks.....	121
WhatSize with Bindings (and a Converter)	124
The <i>ScrollView</i> Solution	128
Layout Weirdness or Normalcy?	134
Making an E-Book	135
Fancier <i>StackPanel</i> Items	138
Deriving from <i>UserControl</i>	141
Creating Windows Runtime Libraries	143
The Wrap Alternative.....	146
The <i>Canvas</i> and Attached Properties	148
The Z-Index	153
Canvas Weirdness	153
Chapter 5: Control Interaction.....	155
The <i>Control</i> Difference	155
The <i>Slider</i> for Ranges	157
The <i>Grid</i>	161
Orientation and Aspect Ratios.....	168
<i>Slider</i> and the Formatted String Converter	171

Tooltips and Conversions	171
Sketching with Sliders.....	174
The Varieties of Button Experience	176
Defining Dependency Properties.....	184
<i>RadioButton</i> Tags	194
Keyboard Input and <i>TextBox</i>	200
Touch and <i>Thumb</i>	204
Chapter 6: WinRT and MVVM	211
MVVM (Brief and Simplified).....	211
Data Binding Notifications.....	212
A View Model for ColorScroll.....	214
Syntactic Shortcuts	219
The <i>DataContext</i> Property	222
Bindings and <i>TextBox</i>	224
Buttons and MVVM.....	230
The <i>DelegateCommand</i> Class.....	231
Chapter 7: Building an Application	238
Commands, Options, and Settings.....	238
The Segoe UI Symbol Font.....	240
The Application Bar	246
Popups and Dialogs	248
Windows Runtime File I/O	251
<i>Await</i> and <i>Async</i>	258
Calling Your Own <i>Async</i> Methods.....	260
Controls for XamlCruncher	263
Application Settings and Isolated Storage.....	278
The XamlCruncher Page	282
Parsing the XAML.....	286
XAML Files In and Out.....	288
The Settings Dialog	292

Beyond the Windows Runtime	298
Chapter 8: Animation	299
The <i>Windows.UI.Xaml.Media.Animation</i> Namespace.....	299
Animation Basics	300
Animation Variation Appreciation.....	303
Other Double Animations	310
Animating Attached Properties.....	317
The Easing Functions	320
All-XAML Animations.....	329
Animating Custom Classes.....	333
Key Frame Animations.....	337
The <i>Object</i> Animation.....	341
Predefined Animations and Transitions.....	343
Chapter 9: Transforms.....	347
A Brief Overview.....	347
Rotation (Manual and Animated).....	350
Visual Feedback.....	356
Translation.....	358
Transform Groups	361
The Scale Transform.....	366
Building an Analog Clock.....	370
Skew.....	375
Making an Entrance	378
Transform Mathematics	379
The CompositeTransform.....	386
Geometry Transforms.....	389
Brush Transforms.....	391
Dude, Where's My Element?	395
Projection Transforms.....	398
Deriving a Matrix3D.....	405

Chapter 13: Touch, Etc.....	416
A <i>Pointer</i> Roadmap	417
A First Dab at Finger Painting	420
Capturing the Pointer	423
Editing with a Popup Menu.....	431
Pressure Sensitivity	435
How Do I Save My Drawings?.....	438
A Touch Piano	439
Manipulation, Fingers, and Elements	444
Working with Inertia	452
An <i>XYSlider</i> Control	456
Centered Scaling and Rotation.....	462
Single-Finger Rotation.....	466
Chapter 14: Bitmaps	473
Pixel Bits	474
Transparency and Premultiplied Alphas.....	480
A Radial Gradient Brush.....	485
Loading and Saving Image Files.....	493
In Progress.....	503
Completed.....	503
Chapter 15: Printing	504
Basic Printing	504
Printable and Unprintable Margins.....	511
The Pagination Process	515
Custom Printing Properties.....	522
Printing a Monthly Planner.....	527
Printing a Range of Pages	537
Where to Do the Big Jobs?	548

Chapter 16: Going Native	549
An Introduction to P/Invoke.....	549
Some Help.....	555
Time Zone Information	555
A Windows Runtime Component Wrapper for DirectX	577
About the Author	578

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 (August 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 ebook is based on the Release Preview of Windows 8 (build 8400), which was released on May 31, 2012.

Microsoft has announced that Windows 8 will be released for general availability on October 26, 2012. Microsoft Press and I are targeting the release of the final version of this book for mid-November.

To use this book, you'll need to download and install the Windows 8 Release Preview, as well as Microsoft Visual Studio Express 2012 RC 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
- New Windows 8 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. To run these desktop applications, Windows 8 includes a familiar Windows desktop screen.

The new Windows 8 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. These applications will be purchasable and installable only from an application store run by Microsoft.

A design paradigm is sometimes associated with these new Windows 8 applications. Somewhat inspired by signage in urban environments, this design paradigm emphasizes content over program “chrome” and is characterized by the use of unadorned fonts, clean open styling, a tile-based interface, and transitional animations.

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 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 new Windows 8 applications only.

Many developers were first introduced to the Windows 8 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 Windows 8 applications treats touch, mouse, and pen input 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 new Windows 8 applications. Plenty of other books already exist for writing Win32 desktop applications, including the 5th edition of *Programming Windows*. I’ll occasionally make reference to the Win32 API and desktop applications, but this book is really all about writing new Windows 8 applications.

For writing these 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. Externally, it is very object-oriented.

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 Windows 8 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 Windows 8 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 (called Windows Runtime Components) 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 direct 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. Windows 8 applications written in these languages can't access Win32, COM, or DirectX APIs with as much ease as the C++ programmer, but it is possible and some sample programs in this book show how. A stripped-down version of .NET is also 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 Windows 8 apps written in JavaScript.

After much consideration (and some anguish), I decided that this book would focus almost exclusively on the C# and XAML option. 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 Windows 8 applications using the other options.

With that said, I have greatly enjoyed the renewed debate about the advantages of C++ and native code in crafting high-performance applications. No single tool is best for every problem, and I hope to have the opportunity to explore C++ and DirectX development for Windows 8 more in the future. As a modest start, the companion content for this book includes all the program samples converted to C++.

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 in early 2013 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 Express 2012 RC 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 9

chapters out of the 12 that will eventually make up Part I are included in this 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. This edition includes 4 chapters that will eventually be in these later parts.

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 and the Release Preview (build 8400) in June 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

This is the 6th edition of *Programming Windows*, a book that was first conceived by Microsoft Press in the fall of 1986. The project came to involve me because I was writing articles about Windows programming for *Microsoft Systems Journal* at the time.

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 at 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 programming from National Public Radio. 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 advance 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, and Brent Rector for an email with a crucial solution for an issue involving touch. 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 and Roscoe, New York

August 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 the new world of Windows 8.

I’ll assume you have the Windows 8 Release Preview installed with the development tools and software development kit, specifically Microsoft Visual Studio Express 2012 RC 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.

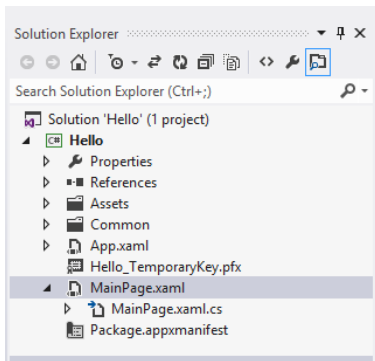
The First Project

On the opening screen in Visual Studio, the Get Started tab should already be selected. Over at the left 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 the option for creating a new Windows 8 project. From the list of available templates in the central area, select Blank App (or the equivalent). 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 MainPage.xaml, and if you click the little arrowhead next to that file, you’ll see a file named MainPage.xaml.cs indented underneath MainPage.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 MainPage.xaml and MainPage.xaml.cs files are linked in the Solution Explorer because they both contribute to the definition of a class named *MainPage*. For a simple program like Hello, this *MainPage* class defines all the visuals and user interface for the application.

Despite its funny file name, MainPage.xaml.cs definitely has a .cs extension, which stands for "C Sharp." Stripped of all its comments, the skeleton MainPage.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 MainPage : Page
    {
        public MainPage()
        {
            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 `MainPage.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 new Windows 8 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 `MainPage.xaml.cs` file defines a namespace named *Hello* (the same as the project name) and a class named *MainPage* 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 `MainPage.xaml.cs` source code, and you'll discover that *Page* is in the *Windows.UI.Xaml.Controls* namespace.

The constructor of the *MainPage* class calls an *InitializeComponent* method (which I'll discuss shortly), and the class also contains an override of a method named *OnNavigatedTo*. Windows 8 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 *MainPage* 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 *MainPage* class is not another C# code file but the `MainPage.xaml` file:

```
<Page
  x:Class="Hello.MainPage"
  IsTabStop="false"
  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 ApplicationPageBackgroundThemeBrush}">
```



```
</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.MainPage"
```

The *x:Class* attribute can appear only on the root element in a XAML file. This particular *x:Class* attribute translates as "a class *MainPage* 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 setting of the *IsTabStop* property of the *Page* class, which regulates behavior when the user navigates among controls by using the Tab key on the keyboard. This setting isn't needed because *IsTabStop* is *false* by default for the *Page* class.

Following that are 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 *Windows.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 Windows 8 applications. In the *MainPage.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: *MainPage.xaml* (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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. If you've created the project yourself, the tile is not very pretty, but the program's tiles are all stored in the Assets directory of the project so you can spruce them up if you want. (The projects in the downloadable companion content for this book have been given custom tiles.) You can run the program again outside of the Visual Studio debugger right from the Windows 8 start screen.

Another option is to run your programs in a simulator that lets you control resolution, orientation, and other characteristics. In the Visual Studio toolbar, you'll see a drop-down list with the current setting Local Machine. Simply change that to Simulator.

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: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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.

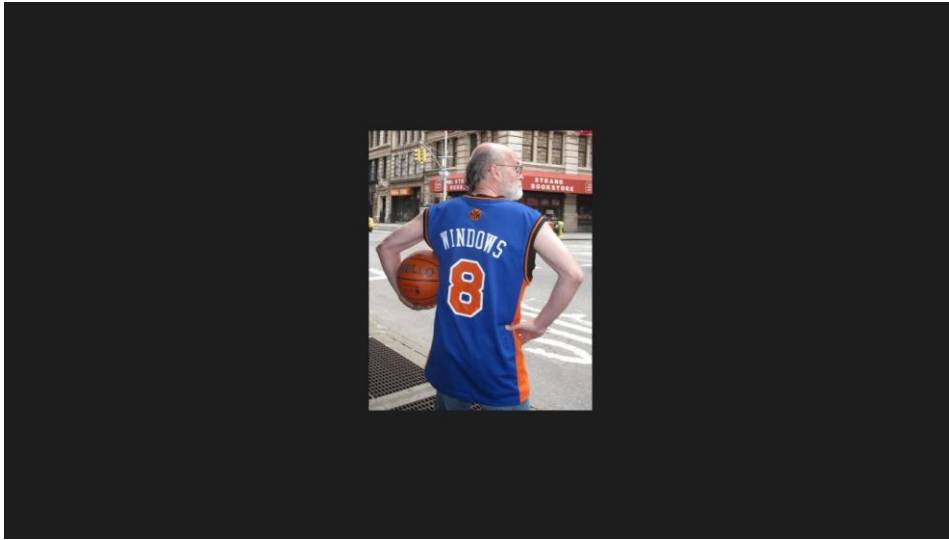
You can override the default display of this bitmap by using the *Stretch* property defined by *Image*. The default value is the enumeration member *Stretch.Uniform*. Try setting it to *Fill*:

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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):

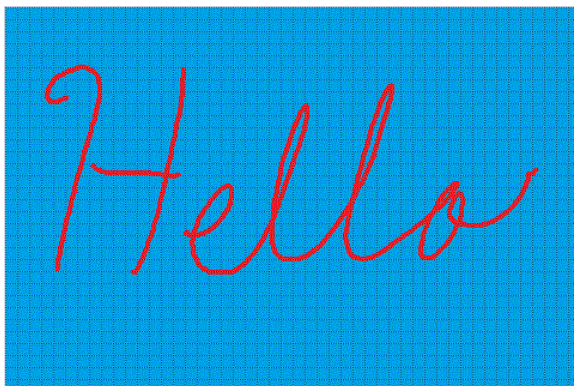


You can control where it appears on the page by using the same *HorizontalAlignment* and *VerticalAlignment* properties you use with *TextBlock*.

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 pen, 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.

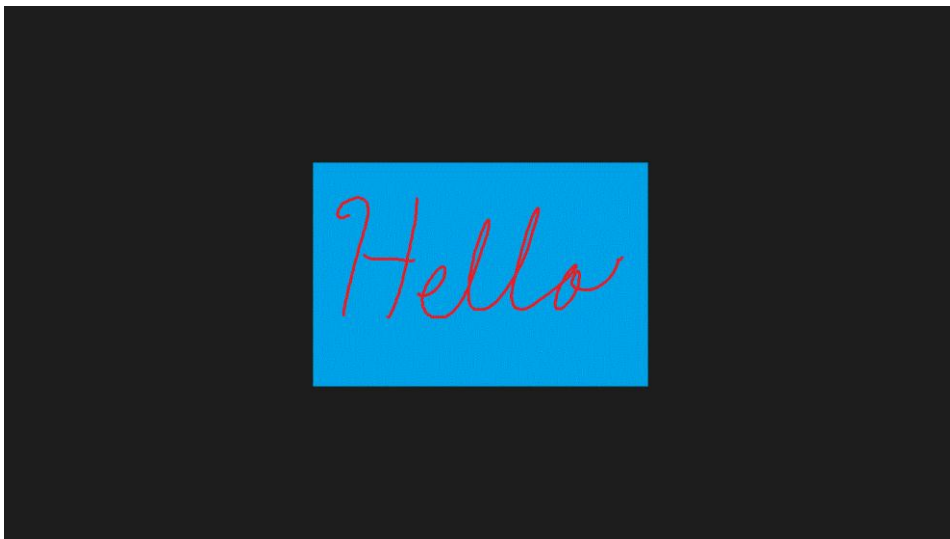
Now right-click the Images folder and choose Add and Existing Item. Navigate to the Greeting.png file you saved and click the Add button. Once the file is added to the project, you'll want to right-click the Greeting.png file name and select Properties. In the Properties panel, make sure the Build Action 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: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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 logo bitmaps. 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, but the distinction is useful nonetheless. Visually, controls are built from elements, and the visual appearance of the control can be customizable through a template. 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 *MainPage.xaml* file now looks like this:

```
<Page
  x:Class="Hello.MainPage"
  IsTabStop="false"
  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 ApplicationPageBackgroundThemeBrush}">
```



```

        <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.

Many of the classes that you'll use in constructing the user interface of a Windows 8 application have conventional properties as well as corresponding properties called “dependency properties” of type *DependencyProperty*. Interestingly enough, the class hierarchy I just showed you has a class named *DependencyObject*. 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.

Dependency properties are intended to solve some fundamental problems that come about in sophisticated user interfaces. In a Windows 8 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 Windows 8 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 *ApplicationPageBackgroundThemeBrush* 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 when you recompile and run the program, it has a light background, which means the color referenced by the *ApplicationPageBackgroundThemeBrush* 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 video 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"
```

When an alpha value is included at the beginning, these are sometimes referred to as ARGB colors. The *UIElement* class also defines an *Opacity* property that can be set to values between 0 (transparent) and 1 (opaque). In HellowImage, 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: *MainPage.xaml* (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">  
    <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*, *Center*, or *Justify*, which causes extra space to be inserted between words so that the text is even on both the left and right.

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

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

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 ('‑'). Any soft hyphens ('­') 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: *OverlappedStackedText* | File: *MainPage.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: MainPage.xaml (excerpt)

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

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

    <!-- Urdu -->
    <TextBlock Text="دنیا، بیلو"
      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="العالم مرحبا،"
      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: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <MediaElement Source="http://www.charlespetzold.com/pw6/AudioGreeting.wma" />
</Grid>
```

The *MediaElement* class derives from *FrameworkElement* and has no user interface, although it provides enough information for you to build your own.

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

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

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <MediaElement 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 MainPage.xaml file only long enough to give the *Grid* a name:

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

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

</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 ApplicationPageBackgroundThemeBrush}">

</Grid>
```

For most cases, 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.

However, sometimes, when naming a custom control that is defined in the application assembly, *Name* doesn't work and *x:Name* is required.

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 MainPage.xaml.cs file you'll want two additional *using* directives:

Project: HelloCode | File: MainPage.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 MainPage.xaml.cs file.)

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

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

```
public MainPage()
{
    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* value 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
};

```

I've tended to use this style a lot in this book, but not implicit typing (the *var* keyword) also introduced in C# 3.0 because it obscures rather than illuminates the code. 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 *MainPage* 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 *MainPage.g.cs* and *MainPage.g.i.cs*. The "g" stands for "generated." Both these files define *MainPage* classes derived from *Page* with the *partial* keyword. The composite *MainPage* class thus consists of the *MainPage.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 *MainPage.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 *MainPage.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 *MainPage* 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 something 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, and that source code is likely written in C++ rather than C#, 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 in XAML on the *Page* tag rather than the *TextBlock* and how these properties are inherited by the *TextBlock*. Of course you can do the same thing in code:

```

public MainPage()
{
    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 *MainPage* constructor:

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

```
public MainPage()
{
    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 *MainPage*:

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

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

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    public MainPage()
    {
        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 *MainPage* (which is how *MainPage* gets instantiated), and then sets this *Frame* object to a precreated *Window* object accessible through the *Window.Current* static property. Here’s the simplified code:

```

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

```

A Windows 8 application doesn’t require a *Page* derivative, 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, MainPage.xaml, and MainPage.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 StrippedDownHello

```

```

{
    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 = HorizontalAlignment.Center,
                VerticalAlignment = 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 Windows 8 applications in this book, but obviously it works.

Chapter 2

XAML Syntax

A Windows 8 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 off topic: 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: MainPage.xaml (excerpt)

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

    <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: MainPage.xaml.cs (excerpt)

```
public MainPage()
{
    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?

Property Element Syntax

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 number pair 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>
```

```

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

```

And there we have it: a rather complex property setting 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 such a 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 an attribute named *ContentProperty*. If these classes were defined in C#, it would look like this:

```
[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: MainPage.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" />
  </Grid.Children>
```

```

        <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: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <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 derivatives:

Here is text in a Times New Roman font, as well as text in a 36-pixel height.

Here is some **bold** and here is some *italic* and here is some underline and here is some ***bold italic underline*** and **bigger and Red as well.**

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 a run-time error explaining that you can't add an instance of *InlineUIContainer* to an *InlineCollection*.

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: MainPage.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. Also available are *x:Boolean* and *x:Int32* types.

Notice as well 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: MainPage.xaml (excerpt)

```

<Page ... >
...
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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" />

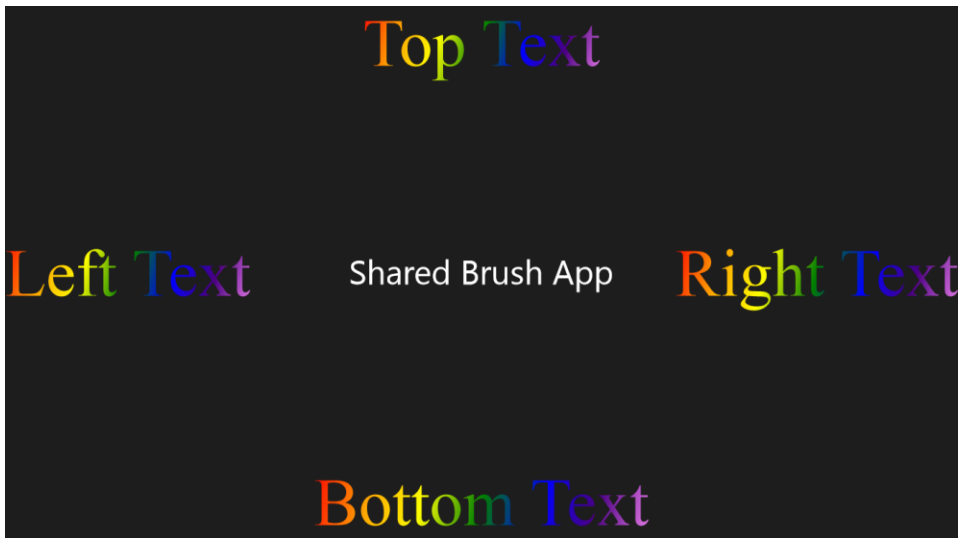
<TextBlock Text="Right Text"
    Foreground="{StaticResource rainbowBrush}"
    FontFamily="{StaticResource fontFamily}"
    FontSize="{StaticResource fontSize}"
    HorizontalAlignment="Right"
    VerticalAlignment="Center" />

<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 Windows 8 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 *MainPage.xaml* file, but add that item to the dictionary in the *MainPage* 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 *ApplicationPageBackgroundThemeBrush* 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 *MainPage.xaml* file and viewing the available *Background* brush identifiers in the Properties view in the lower-right corner of Visual Studio.

After *ApplicationPageBackgroundThemeBrush*, the next most important predefined resource identifier is *ApplicationForegroundThemeBrush*, 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 *ApplicationPressedForegroundThemeBrush* is also convenient for a splash of color that contrasts with both the background and foreground.

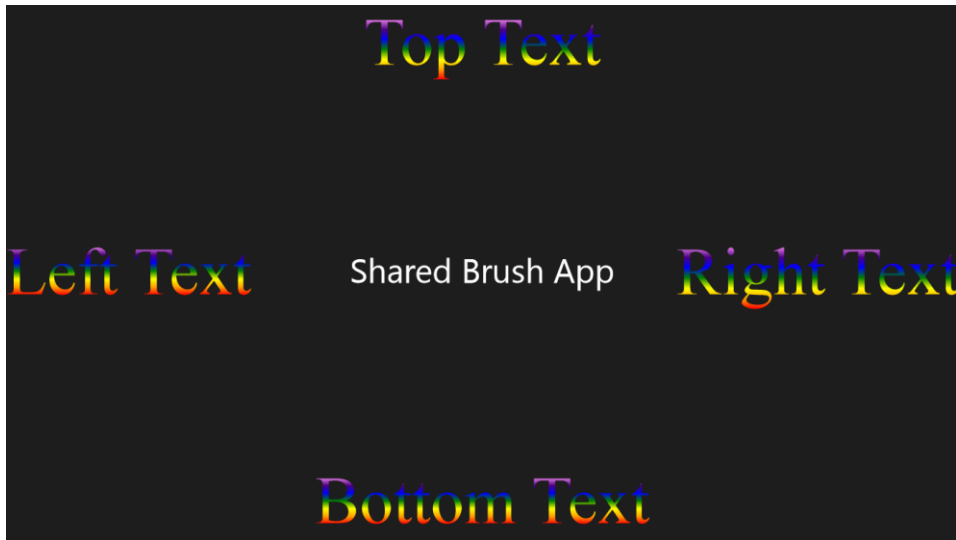
Resources Are Shared

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 Windows 8 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 in the Windows Runtime! Methods with names like those exist in DirectX, which you can use in a Windows 8 application, but when using the Windows Runtime you instead 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: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Polyline Name="polyline"
        Stroke="{StaticResource ApplicationForegroundThemeBrush}"
        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 becomes 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: MainPage.xaml.cs (excerpt)

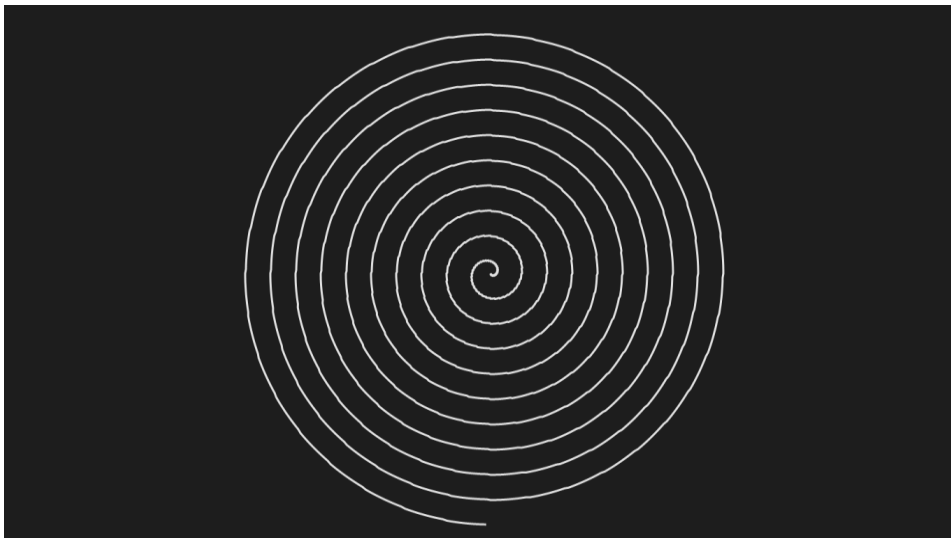
```
public MainPage()
{
    this.InitializeComponent();

    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 $\frac{3}{4}$ 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: MainPage.xaml (excerpt)

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

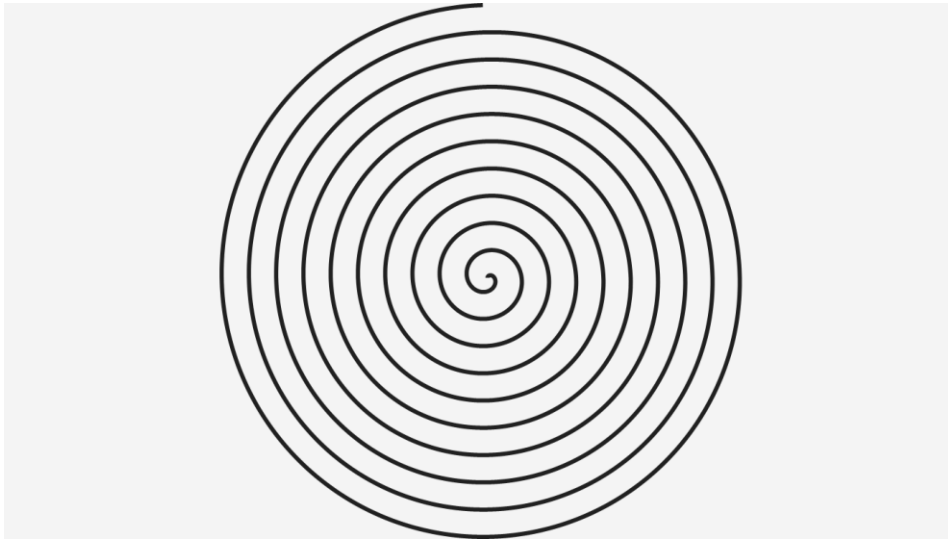
The code-behind file calculates the coordinates of the spiral using arbitrary coordinates, which in this case I've chosen based on a radius of 1000:

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

```
public MainPage()
{
    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 *ApplicationForegroundThemeBrush* 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: *MainPage.xaml* (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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 using *Path* and *PathGeometry*:

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

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

```

<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's why it's set to 1/10th pixel short. An alternative is to split the *ArcSegment* into two, each drawing half the circle.)

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: *MainPage.xaml* (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <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: MainPage.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 MainPage : Page
    {
        public MainPage()
        {
            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 *MainPage.xaml.cs* file that Visual Studio generates does not include a *using* directive for *Windows.UI.Xaml.Shapes* 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*.

Stretching with *Viewbox*

Both the *Image* class and the *Shape* class define a *Stretch* property that can stretch the bitmap or vector graphics to the size of its container. This property is not universal among the *FrameworkElement* derivatives. After all, why would you ever want to stretch a *TextBlock* in such a way?

Well, sometimes you need to do precisely that. Suppose you were displaying a bunch of objects with text titles. You want these items to look similar with each title restricted to a particular rectangular area. But the length of the text might be variable. Perhaps the user types in this text. If the text is very long, you might prefer that it be shrunk down a bit to fit the rectangle. While you could always perform a *FontSize* calculation in the code-behind file, it would be nice to have the *TextBlock* sized automatically to fit a particular space.

This is a job for *Viewbox*, which has a *Child* property of type *UIElement* and which stretches that child to its own size. Like *Image* and *Shape*, *Viewbox* defines a *Stretch* property. The default setting is *Uniform* (the same default as *Image*), but the following program sets *Stretch* to *Fill* to ignore the aspect ratio of a *TextBlock* and make it fill the screen:

Project: TextStretch | File: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Viewbox Stretch="Fill">
        <TextBlock Text="Stretch Windows 8!" />
    </Viewbox>
</Grid>
```

TextBlock always calculates its height to encompass diacritics and descenders even if they don't exist in the text, which is why the text doesn't quite extend to the full height of the window:



Still, it definitely no longer has the correct aspect ratio.

Unlike *Image* and *Shape*, *Viewbox* defines a *StretchDirection* property that can take on values of *UpOnly*, *DownOnly*, or *Both* (the default). This instructs *Viewbox* to only increase the size of its child or only decrease the size if that's what you want.

Suppose you wanted to modify the *HelloVectorGraphics* program so that each letter is a different color. Instead of using one *Path* element you'd need to split it up into five *Path* elements. But if you

then try to use the *Stretch* property of *Path* to stretch each letter to the size of the window, it wouldn't work because each letter has a different size.

Instead, put all five *Path* elements in a *Grid*, and put the *Grid* inside a *Viewbox*:

Project: VectorGraphicsStretch | File: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <Viewbox Stretch="Fill">
    <Grid Margin="6 6 0 0">
      <!-- H -->
      <Path Stroke="Red"
            StrokeThickness="12"
            StrokeLineJoin="Round"
            Data="M 0 0 L 0 100 M 0 50 L 50 50 M 50 0 L 50 100" />

      <!-- E -->
      <Path Stroke="#C00040"
            StrokeThickness="12"
            StrokeLineJoin="Round"
            Data="M 125 0 C 60 -10, 60 60, 125 50, 60 40, 60 110, 125 100" />

      <!-- L -->
      <Path Stroke="#800080"
            StrokeThickness="12"
            StrokeLineJoin="Round"
            Data="M 150 0 L 150 100, 200 100" />

      <!-- L -->
      <Path Stroke="#4000C0"
            StrokeThickness="12"
            StrokeLineJoin="Round"
            Data="M 225 0 L 225 100, 275 100" />

      <!-- O -->
      <Path Stroke="Blue"
            StrokeThickness="12"
            StrokeLineJoin="Round"
            Data="M 300 50 A 25 50 0 1 0 300 49.9" />
    </Grid>
  </Viewbox>
</Grid>
```

Now the whole ensemble of vector graphics is sized uniformly:



Notice also that the *Viewbox* increases the stroke width along with the size of the graphics, whereas setting *Stretch* on the *Path* element does not.

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 Chapter 10, "The Two Templates."

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: *MainPage.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>
</Page.Resources>
```

```

</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: MainPage.xaml (excerpt)

```

<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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 property of type *DependencyProperty* 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 style 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: *MainPage.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: *MainPage.xaml* (excerpt)

```

<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

```

```

<TextBlock Text="{StaticResource appName}"
           FontFamily="Portable User Interface"
           FontSize="48"
           Foreground="{StaticResource ApplicationForegroundThemeBrush}"
           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 effectively 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*, *CaptionTextStyle*, *PageHeaderTextStyle*, *PageSubheaderTextStyle*, and *SnappedPageHeaderTextStyle*, 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: MainPage.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: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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" />
            </LinearGradientBrush>
        </TextBlock.Foreground>
    </TextBlock>
</Grid>
```

```

        <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." (This does not work in a C++ program. Compound and indexed binding paths are not currently supported.)

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 Chapter 10.

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 pen, 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 pen;
- five events beginning with the word *Manipulation* that combine input from multiple fingers;
- two *Key* events for keyboard input; and
- 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 future chapters. The only other events that *UIElement* defines are also related to user input:

- *GotFocus* and *LostFocus* signal when an element is the target of keyboard input; and

- *DragEnter*, *DragOver*, *DragLeave*, and *Drop* relate to drag-and-drop.

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 pen. To qualify as a *Tapped* event, the finger (or mouse or pen) cannot move very much and must be released in a short period of time.

All the user-input events have a similar pattern. Expressed in C# syntax, *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: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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 *MainPage.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*. You can 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.

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: MainPage.xaml.cs (excerpt)

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

    public MainPage()
    {
        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 pen 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 pen, 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 *MainPage.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 *MainPage.xaml* file and instead attach an event handler in the constructor of the code-behind file:

```
public MainPage()
{
```

```

        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 pen. 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 technique 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: *MainPage.xaml.cs* (excerpt)

```

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

    public MainPage()
    {
        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 that 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: MainPage.xaml (excerpt)

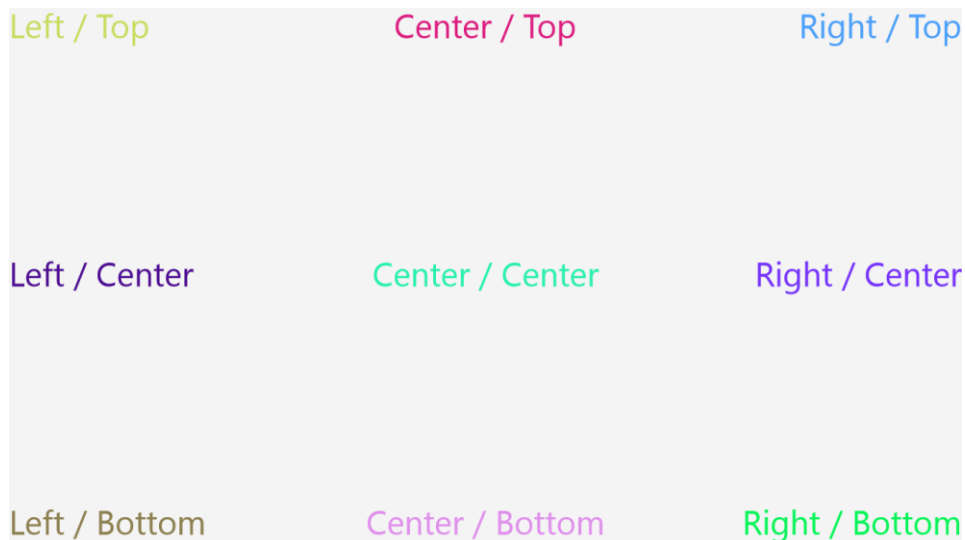
```
<Page
  x:Class="RoutedEvents0.MainPage"
  ...
  FontSize="48">

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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: `MainPage.xaml` (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}"
      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: `MainPage.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);
    }
}
```

Slightly more efficient is performing the cast first and then checking if the result is non-null.

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 elements to process events that originate with their children and other descendants in the visual tree and to know the source of these events. Routed event handling lets a parent know what its children are up to, and *OriginalSource* identifies the particular child involved.

Alternatively, you can set the *Tapped* handler on *MainPage* rather than the *Grid*. But with *MainPage* there's an easier way. I mentioned earlier that *UIElement* defines all the user-input events. These events are inherited by all derived classes, but the *Control* class adds its own event interface consisting of a whole collection of virtual methods corresponding to these events. For example, 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 by the *Page* and *MainPage* classes.

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

Project: *RoutedEvents2* | File: *MainPage.xaml* (excerpt)

```
<Page
  x:Class="RoutedEvents2.MainPage"
  IsTabStop="false"
  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 ApplicationPageBackgroundThemeBrush}">
    <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: *MainPage.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. Depending on the method you're overriding, you might want to call the base method first, last, in the middle, or not at all.

The *On* methods are basically the same as the event handlers, but they have no *sender* argument because it would be redundant: *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: MainPage.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: MainPage.xaml (excerpt)

```

<Grid Name="contentGrid"

```

```

        Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

        <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: MainPage.xaml.cs (excerpt)

```

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

    public MainPage()
    {
        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: MainPage.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: MainPage.xaml.cs (excerpt)

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

    public MainPage()
    {
        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);
    }
}

```

But 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*. (This call raises an exception when used in a C++ program.)

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 is essential.

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: MainPage.xaml (excerpt)

```
<Page ...
    FontSize="48">

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <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: MainPage.xaml.cs (excerpt)

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

    public MainPage()
    {
        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:

Hello, Windows 8!

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 ApplicationPageBackgroundThemeBrush}">
    <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 ApplicationPageBackgroundThemeBrush}">
    <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 layer (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 Windows 8 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 Windows 8 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 Windows 8 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 Windows 8 application—particularly one that does what the old WHATSIZE program did—is named *SizeChanged*. Here’s the XAML file for the Windows 8 WhatSize program. Notice that the root element defines a handler for the *SizeChanged* event:

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

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

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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:

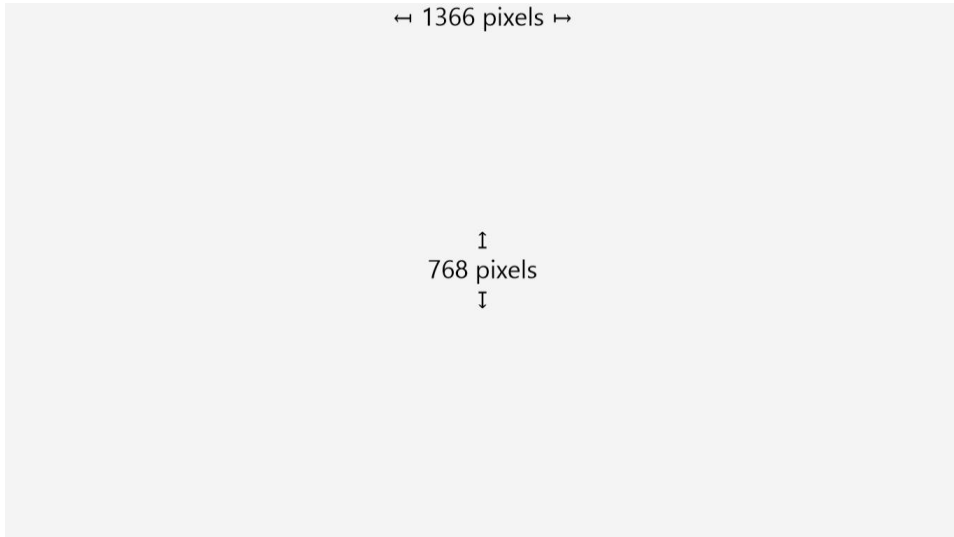
Project: WhatSize | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        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: *MainPage* 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* derivative 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 the *Page* derivative. When that *Page* object 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 MainPage()
{
    this.InitializeComponent();

    Loaded += (sender, args) =>
    {
```



```

    };
}

```

Sometimes Windows 8 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: `MainPage.xaml.cs`

```

public sealed partial class MainPage : Page
{
    public MainPage()
    {
        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 Chapter 12, “Pages and Navigation.”

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 `MainPage.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 `MainPage.xaml` file, remove the `SizeChanged` attribute and give `MainPage` 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 ApplicationPageBackgroundThemeBrush}">
        <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 a discouraging 0.

This is likely to seem odd, particularly when you set the same bindings on the `Text` property of `TextBlock` instead of `Run`:

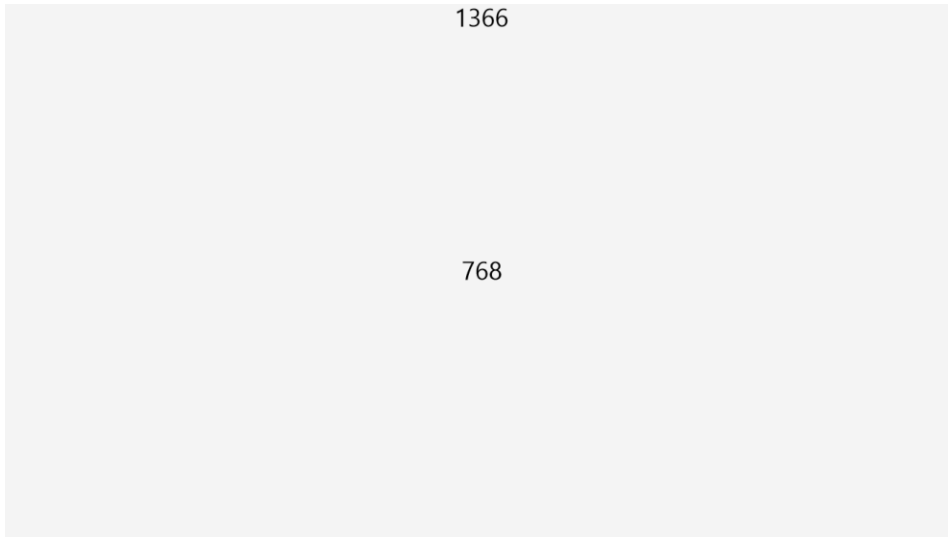
```
<Page ...
    FontSize="36"
    Name="page">

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <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:



At least it appears to work at first. With the version of Windows 8 that I'm using to write this chapter, the numbers are not updated as you change the orientation or size of the page, and they really should be. In theory, 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. This is what is supposed to make data bindings so great.

Unfortunately, by giving up on the bindings to *Run* we've also lost the informative arrows. So why do the data bindings work (or almost work) on the *Text* property of *TextBlock* but not at all 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. The *Run* version of *Text* is 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 Windows 8 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: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        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:



7:43:14 PM

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 Windows 8 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 Windows 8 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 in Chapter 8, "Animation"—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: `MainPage.xaml` (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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: `MainPage.xaml.cs` (excerpt)

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        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 somewhat. 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: `MainPage.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: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    public MainPage()
    {
        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 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: MainPage.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: MainPage.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 *Color* values. 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 at the rate of 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: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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: MainPage.xaml (excerpt)

```
public sealed partial class MainPage : Page
{
    double txtblkBaseSize; // ie, for 1-pixel FontSize

    public MainPage()
    {
        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*, rather like a manual version of *Viewbox*. 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 or 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.

Similar concepts are well known in the Web world. 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.

In contrast, 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. 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 *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: MainPage.xaml (excerpt)

```
<Page ... >

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

        <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:

Hello Windows 8!

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: MainPage.xaml (excerpt)

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

    <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:



Hello Windows 8!

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, 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).

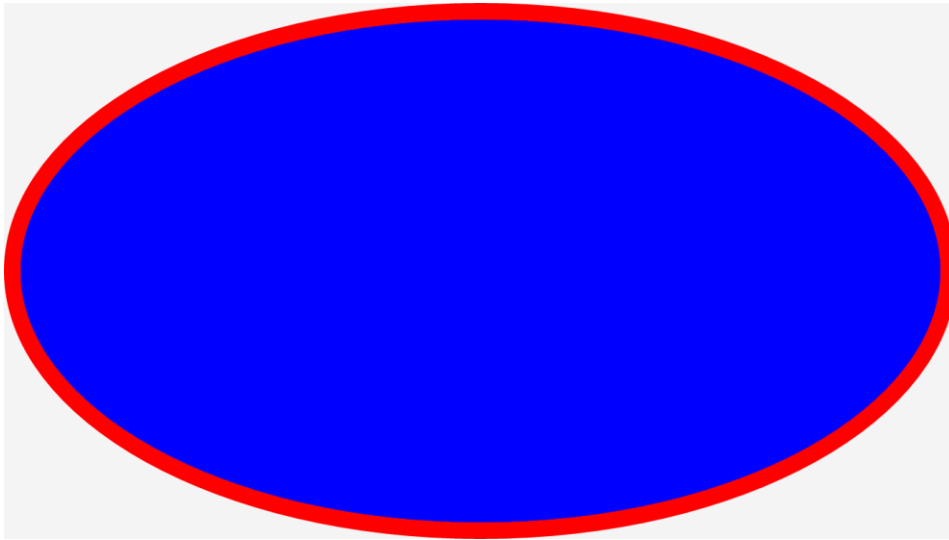
Six classes derive from *Shape*. *Line*, *Polyline*, and *Polygon* render straight lines based on coordinate points, and *Path* uses a series of classes in *Windows.UI.Xaml.Media* for rendering a series of straight lines, arcs, and Bezier curves.

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: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Ellipse Stroke="Red"
        StrokeThickness="24"
        Fill="Blue" />
</Grid>
```

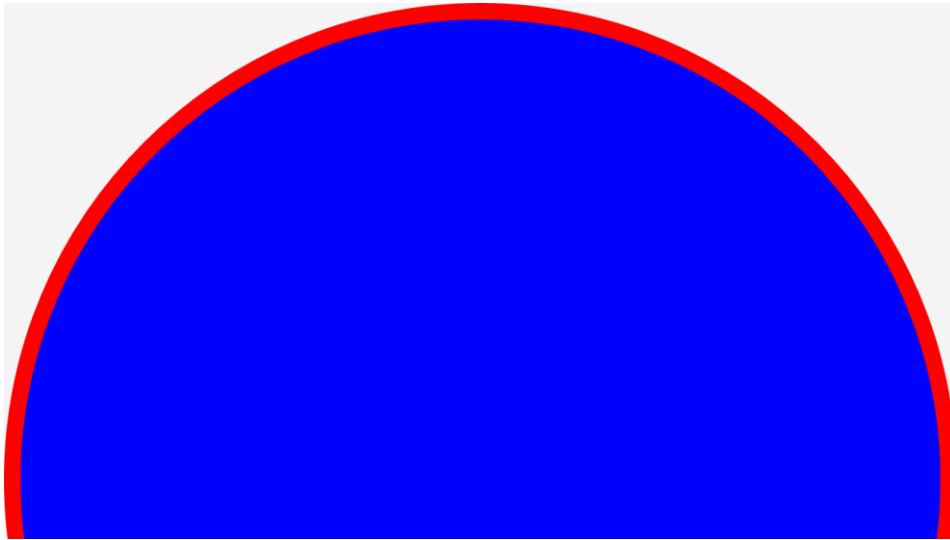
Notice how the ellipse fills its container:



Like all other *FrameworkElement* derivatives, *Ellipse* has default *HorizontalAlignment* and *VerticalAlignment* settings of *Stretch*, but more than most other elements, *Ellipse* 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* and *Viewbox*. 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 Chapter 11, "Collections"). 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: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <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:

Right-Aligned Text



Figure 1. Petzold heading to the basketball court

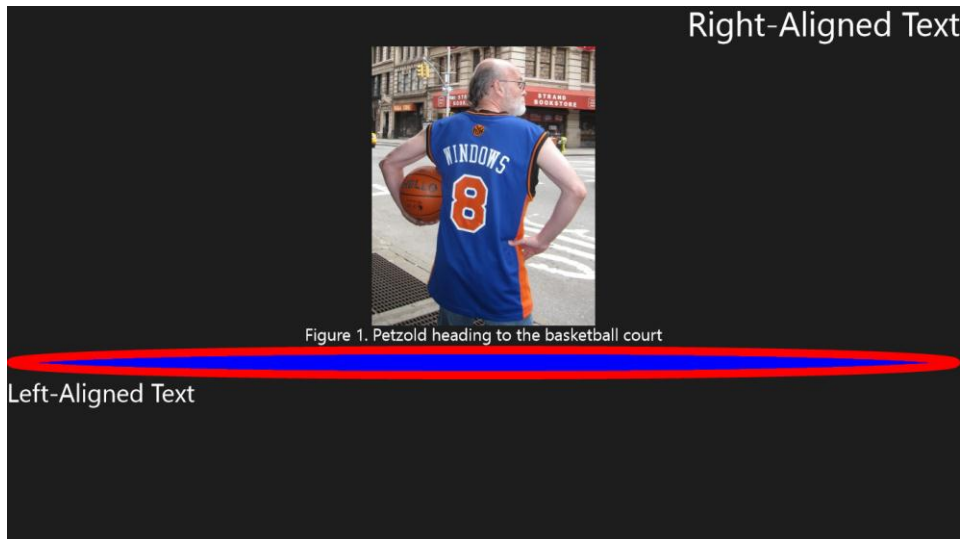
Left-Aligned Text

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 of the screen 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:

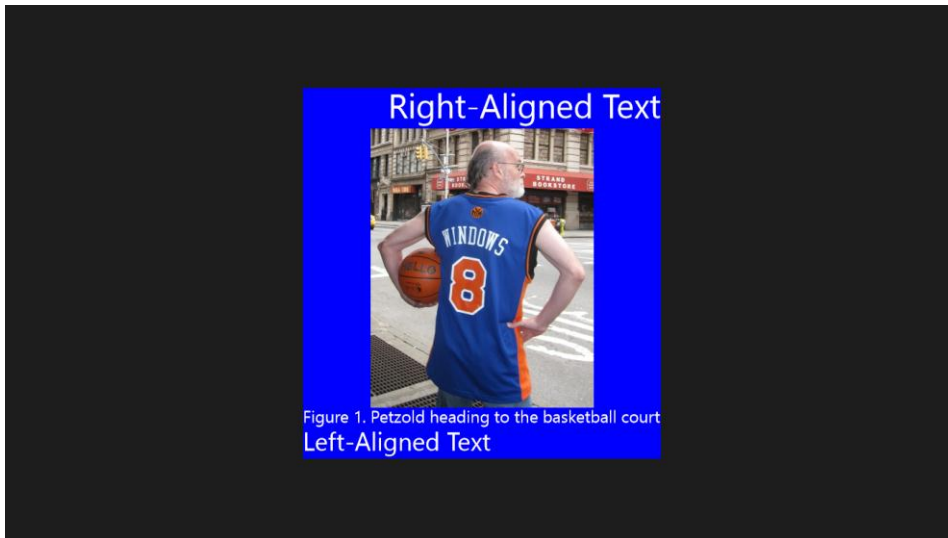


If you also set the *Stretch* property of the *Ellipse* to *Uniform*, you'll get a circle rather than a very wide ellipse.

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 children (so to speak), 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*:



The width of the *StackPanel* is now governed by the width of its widest child, which is the totally honest caption under the photograph.

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: MainPage.xaml (excerpt)

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

```
    <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>

```

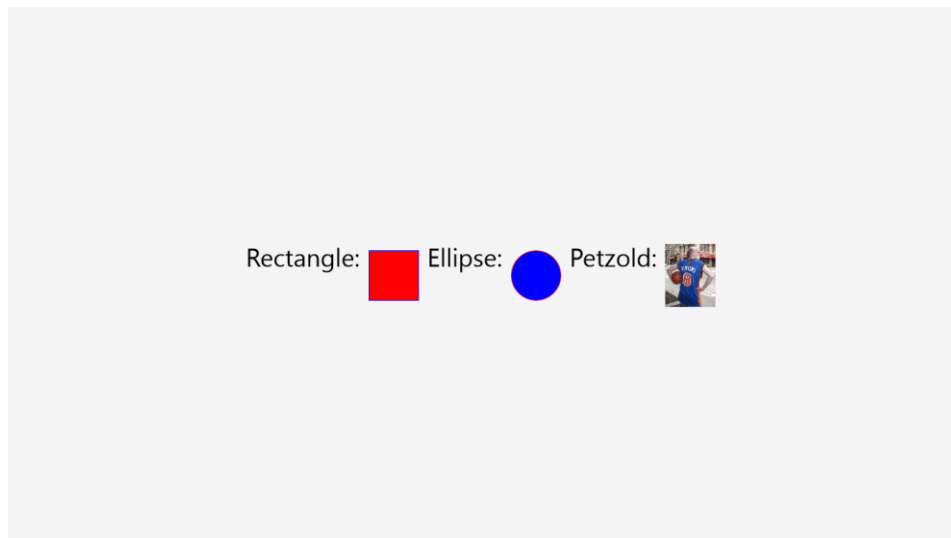
Here it is:

Rectangle:  Ellipse:  Petzold: 

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 occupies 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: MainPage.xaml (excerpt)

```
<Page
  x:Class="WhatSizeWithBindings.MainPage"
  ...
  FontSize="36"
  Name="page">

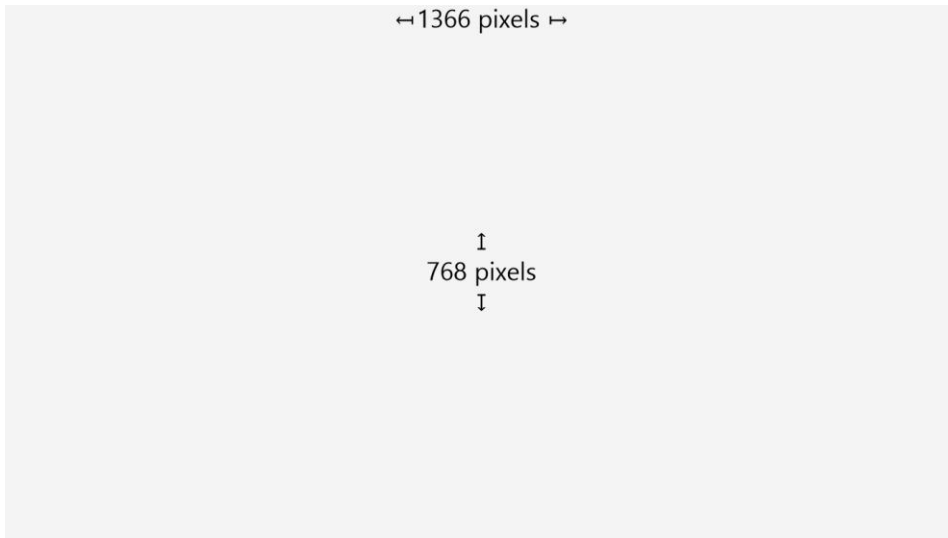
  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel Orientation="Horizontal"
      HorizontalAlignment="Center"
      VerticalAlignment="Top">
      <TextBlock Text="&#x21A4; " />
      <TextBlock Text="{Binding ElementName=page, Path=ActualWidth}" />
      <TextBlock 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:



Although the values are initially correct, the bindings in the Windows 8 build I'm using for this chapter unfortunately don't update the values with a different orientation or snap view.

But supposing they worked correctly, what if you're using bindings to properties of type *double* for some other purpose and you want to specify a number of decimal places? Or perhaps, more appropriate for this example, you want a comma separator to appear in the width so it's 1,366?

Each data binding is converting a *double* to a *string*. It is possible to supply a little piece of code to the *Binding* object so that it performs the 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 comma separators and 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("N0");
}
```

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: MainPage.xaml (excerpt)

```
<Page
  x:Class="WhatSizeWithBindingConverter.MainPage"
  ...
  FontSize="36"
  Name="page">

  <Page.Resources>
    <local:FormattedStringConverter x:Key="stringConverter" />
  </Page.Resources>

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel Orientation="Horizontal"
      HorizontalAlignment="Center"
      VerticalAlignment="Top">
      <TextBlock Text="&#x21A4; " />
      <TextBlock Text="{Binding ElementName=page,
        Path=ActualWidth,
        Converter={StaticResource stringConverter},
        ConverterParameter=N0}" />
      <TextBlock Text=" pixels &#x21A6;" />
    </StackPanel>

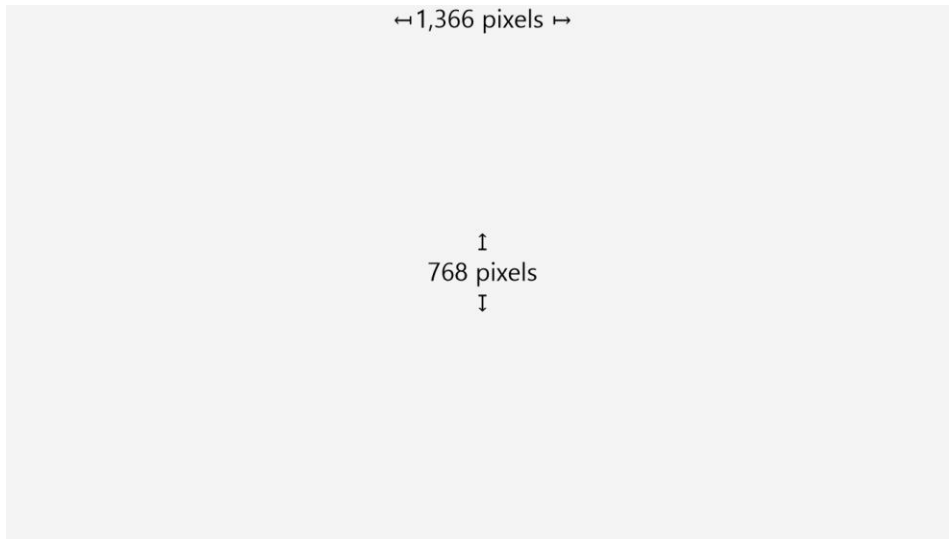
    <StackPanel HorizontalAlignment="Center"
      VerticalAlignment="Center">
      <TextBlock Text="&#x21A5;" TextAlignment="Center" />

      <StackPanel Orientation="Horizontal"
        HorizontalAlignment="Center">
        <TextBlock Text="{Binding ElementName=page,
          Path=ActualHeight,
          Converter={StaticResource stringConverter},
          ConverterParameter=N0}" />
        <TextBlock Text=" pixels" />
      </StackPanel>

      <TextBlock Text="&#x21A7;" TextAlignment="Center" />
    </StackPanel>
  </Grid>
</Page>
```

Take careful note of the Binding syntax. I've spread it 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.

Now the width is formatted a little fancier:



The *ScrollView* 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 *ScrollView*.

The *ScrollView* 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. *ScrollView* provides scrollbars for the mouse-users among us. Otherwise, you can just move the content around with your fingers. By default, *ScrollView* 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*.

ScrollView defines a couple other crucial properties. Most often you'll be using *ScrollView* 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 *ScrollView*, 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 so that it can be inherited throughout the page:

Project: `StackPanelWithScrolling` | File: `MainPage.xaml` (excerpt)

```
<Page
  x:Class="StackPanelWithScrolling.MainPage"
  ...
  FontSize="26">

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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: `MainPage.xaml.cs` (excerpt)

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        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} \u2014 {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-A0-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-EE-00

```

And, of course, you can scroll it with your finger or the mouse.

To simplify the use of reflection in the C++ version of this program, the program references a *ReflectionHelper* library in the solution that I wrote in C#. This library is also referenced in some subsequent projects in this chapter and other chapters. I'll discuss libraries later in this chapter.

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 Chapter 11—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. If you've explored the Windows 8 documentation trying to find these class hierarchies, you've probably discovered that the documentation for each class shows only an ancestor class hierarchy but not derived classes. So how exactly did I create the class hierarchies for this book?. They come 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: *MainPage.xaml* (excerpt)

```

<Page
    x:Class="DependencyObjectClassHierarchy.MainPage"
    ...
    FontSize="{StaticResource ControlContentThemeFontSize}">

```

```

        <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
            <ScrollView>
                <StackPanel Name="stackPanel" />
            </ScrollView>
        </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. Conceptually, the entire Windows Runtime is associated with a single assembly, so to get a reference to that assembly you just need one type. You get the *Assembly* object from the *Assembly* property of the *TypeInfo* object for that type.

Project: DependencyObjectClassHierarchy | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    Type rootType = typeof(DependencyObject);
    TypeInfo rootTypeInfo = typeof(DependencyObject).GetTypeInfo();
    List<Type> classes = new List<Type>();
    Brush highlightBrush;

    public MainPage()
    {
        this.InitializeComponent();
        highlightBrush = this.Resources["ApplicationPressedForegroundThemeBrush"] as Brush;

        // Accumulate all the classes that derive from DependencyObject
        AddToClassList(typeof(Windows.UI.Xaml.DependencyObject));
    }
}

```

```

        // 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
    });

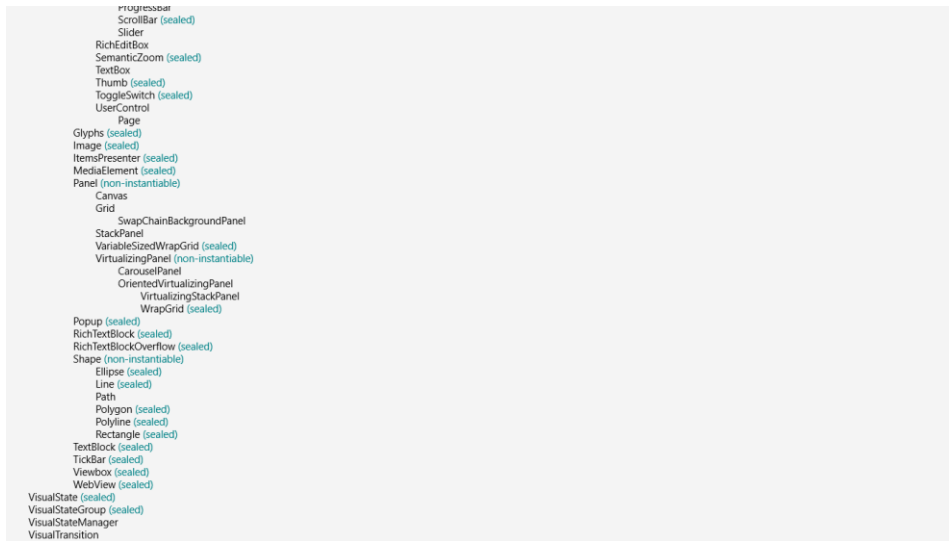
// Add to the StackPanel
stackPanel.Children.Add(txtblk);

// 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?

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, but you can also discover a lot by experimenting.

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 ApplicationPageBackgroundThemeBrush}">
  <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?

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 *ScrollView*, 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 *ScrollView* 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 *ScrollView* as a child of a vertical *StackPanel*. To determine the desired size of this *ScrollView* child, the *StackPanel* offers it an infinite height. How tall does the *ScrollView* really want to be? The height of the *ScrollView* 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 *ScrollView*, 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 *ScrollView* 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 in that project 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 MainPage.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: MainPage.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 *ScrollView*. Here's a little excerpt of the alternating *TextBlock* elements and *Image* elements:

Project: TheTaleOfTomKitten | File: MainPage.xaml (excerpt)

```
<Grid Background="White">
  <ScrollView>
    <StackPanel MaxWidth="640"
      HorizontalAlignment="Center">
      ...
      <!-- pg. 38 -->
      <TextBlock Style="{StaticResource paragraphTextStyle}">
        &#x2003;&#x2003;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}">
        &#x2003;&#x2003;"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}">
        &#x2003;&#x2003;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>
  </ScrollView>
</Grid>
```

The two ` ` characters at the beginning of each paragraph are em-spaces. These provide a first-line indentation, which, unfortunately, is not provided by the *TextBlock* class.

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 *ScrollView*. The XAML file is responsible for centering that *StackPanel*:

Project: ColorList1 | File: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <ScrollView>
        <StackPanel Name="stackPanel"
                    HorizontalAlignment="Center" />
    </ScrollView>
</Grid>
```

Although the *StackPanel* is aligned in the center of the *ScrollView* (and is as wide as its widest child), the *ScrollView* occupies the entire width of the page. Any visible sliders or scrollbars appear on the far right of the page. Alternatively, you can put the *HorizontalAlignment* setting on the *ScrollView*, in which case the contents will still be the center but the *ScrollView* will be only as wide as the *StackPanel*.

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: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        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 Chapter 11, where you'll see not only a better way of doing it, but the *best* way of doing it.

Deriving from *UserControl*

The key to making *ColorList1* 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 *MainPage.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 one common approach. After creating the *ColorList2* 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 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);
  }
}
```

I've redefined the constructor to accept 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 MainPage.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: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        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 it's more common to add a library project to the solution of an existing application project. 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. After developing a library in conjunction with an application, you can then share that library 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, at the left select Visual C# and the option for creating a new Windows 8 project. From the list of templates, select Class Library..

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 ApplicationForegroundThemeBrush}"
            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 `MainPage.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: MainPage.xaml.cs

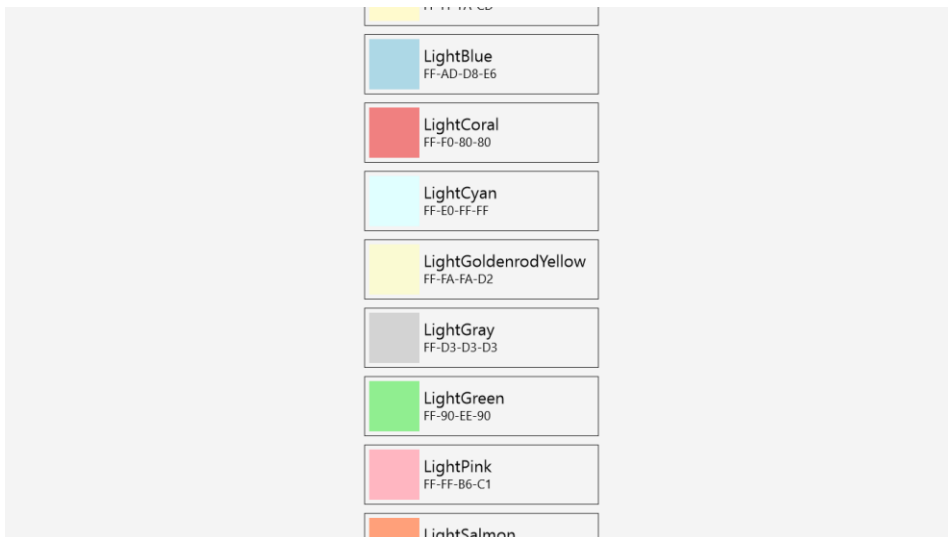
```
using System.Collections.Generic;
using System.Reflection;
using Windows.UI;
using Windows.UI.Xaml.Controls;
using Petzold.Windows8.Controls;

namespace ColorList3
{
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            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:



When creating the Petzold.Windows8.Controls library, I indicated that you should choose Class Library from the Add New Project dialog. There is another option for creating a library labeled Windows Runtime Component. For this particular example, it doesn't matter which one you choose. In

fact, you can right-click the library project name, select Properties, and in the Application screen change the Output type from Class Library to WinMD File (which means “Windows Metadata” but refers to a Windows Runtime Component). The ColorList3 program will run the same.

The big difference is this: The library you create by selecting Class Library can be accessed only from other C# and Visual Basic applications. A Windows Runtime Component can additionally be accessed from C++ and JavaScript. It is the Windows Runtime Component that allows language interoperability for Windows 8 applications.

Consequently, a Windows Runtime Component has some restrictions that a regular Class Library does not. Public classes must be sealed, for example. If you remove the *sealed* keyword from the definition of the *ColorItem* control, that class cannot be part of a Windows Runtime Component. The other major rules involve structures—you can’t have any public members that are not fields—and the restriction of data types passing over the API to Windows Runtime types.

The C++ version of StackPanelWithScrolling includes a Windows Runtime Component named ReflectionHelper written in C# that simplifies the use of reflection by the C++ programs. Chapter 16, “Going Native,” shows the opposite approach: a Windows Runtime Component written in C++ that gives C# programs access to DirectX classes.

The Wrap Alternative

Now let’s use that Petzold.Windows8.Controls 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: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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: MainPage.xaml.cs (excerpt)

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

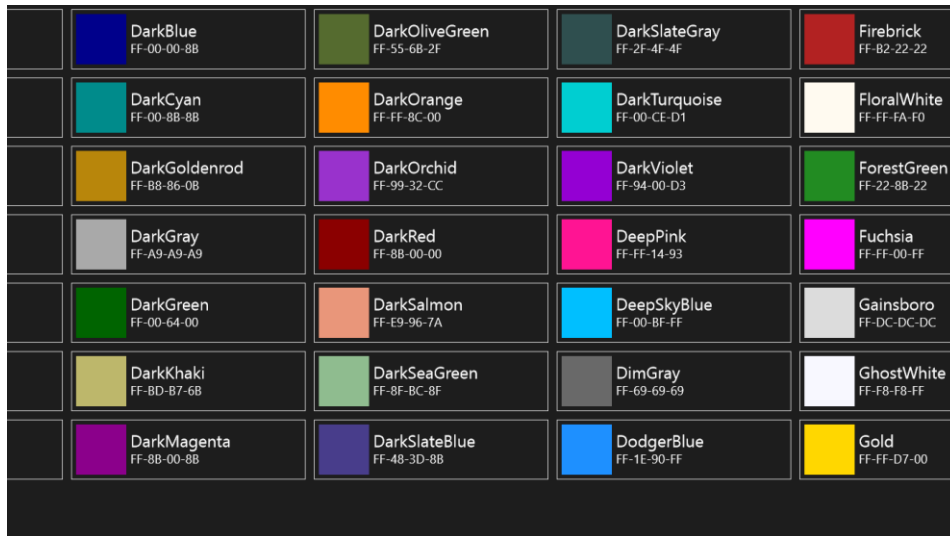
        IEnumerable<PropertyInfo> properties = typeof(Colors).GetTypeInfo().DeclaredProperties;
```

```

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:



Scrolling is horizontal.

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. Properties that let you specify coordinate positions exist for drawing vector graphics but not for other elements because such a property does not have general applicability when you're using a *Grid*, a *StackPanel*, or a *WrapPanel*. 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 property 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: `MainPage.xaml` (excerpt)

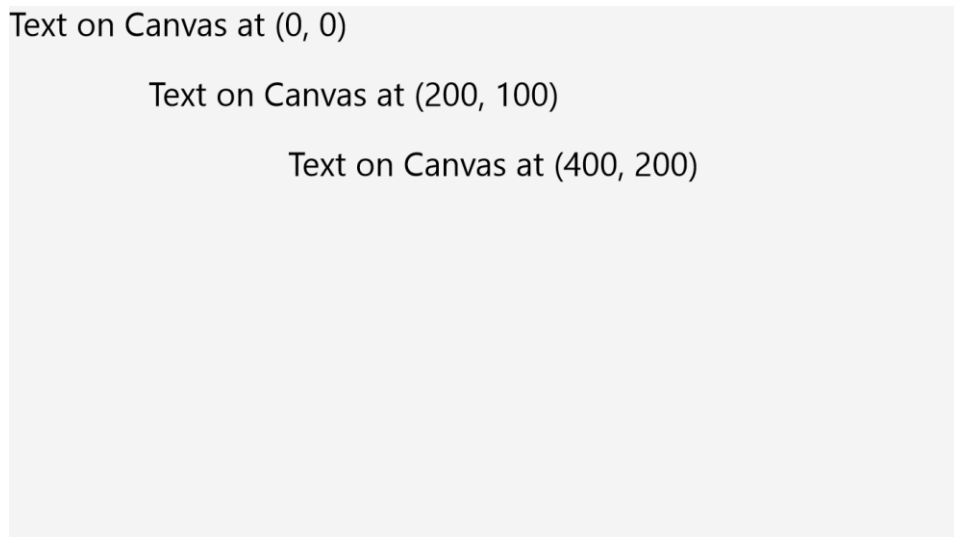
```
<Page
  x:Class="TextOnCanvas.MainPage"
  ...
  FontSize="48">

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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 identified with both 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: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        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),
```

```

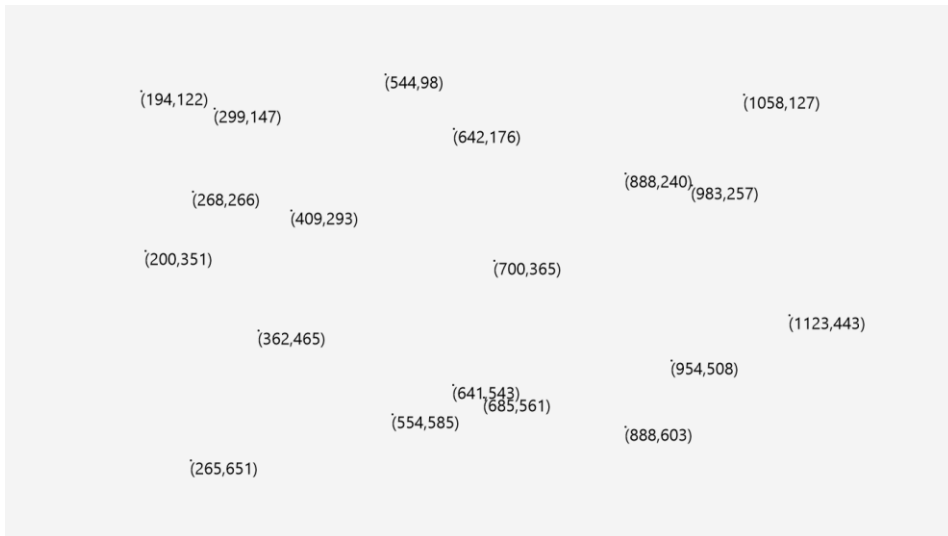
        FontSize = 24,
    };

    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.

However, there's a little flaw in the approach I've used to position the dots. This flaw becomes evident if you make the *Ellipse* a little larger. The program should really be putting the center of the dot at the tapped point, and instead the *Canvas.SetLeft* and *Canvas.SetTop* calls I've used position the upper-left corner of the *Ellipse* there. 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*. Here's how the two *DependencyProperty* objects might be defined if the *Canvas* class were written in C#:

```
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*.

Let me show you something interesting. The *TapAndShowPoint* program calls the static *Canvas.SetLeft* and *Canvas.SetTop* methods like this:

```
Canvas.SetLeft(ellipse, pt.X);
Canvas.SetTop(ellipse, pt.Y);
```

An alternative approach—just as legal, just as valid, and 100% equivalent—involves calling *SetValue* on the child element and referencing the static *DependencyProperty* objects defined by *Canvas*:

```
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, and it doesn't matter which form you use.

You've seen that *SetValue* method before. *SetValue* is defined by *DependencyObject* and inherited by very many classes in the Windows Runtime. A property like *FontSize* is actually defined in terms of the static dependency property referenced with this same *SetValue* method:

```
public double FontSize
{
    set { SetValue(FontSizeProperty, value); }
    get { return (double)GetValue(FontSizeProperty); }
}
```

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 with code that's equivalent to this C#:

```
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 child element rather than the *Canvas*.

Canvas also defines *GetLeft* and *GetTop* methods, defined in code equivalent to this C# code:

```

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.

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 earlier in this chapter 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, and the size of the children of a *Canvas* are ignored in layout.

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. The *TextBlock* essentially exists outside of normal layout.

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 Chapter 10, "The Two Templates," 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.

Control defines a bunch of properties that the *Control* class itself does not need. These are for the use of classes that derive from *Control*, and consist of 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*, *ScrollView*, 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 Chapter 10.

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 *FlipView*, *GridView*, and *ListView*. This is such an important category of controls that the whole of Chapter 11, "Collections," is 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 now familiar spinning circle of dots.

In the quarter-century evolution of Windows, the *ScrollBar* has slipped from its high perch in the

control hierarchy and is commonly seen today only in a *ScrollView* 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: *MainPage.xaml* (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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: *MainPage.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

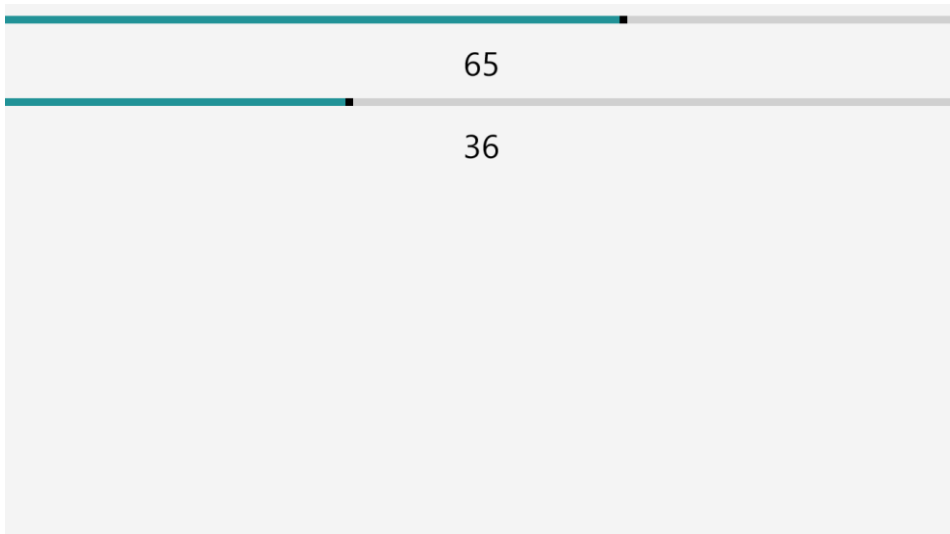
Value property of the *Slider*:

```
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 thickness of the *Slider* cannot be changed except by redefining the visuals with a template. The total thickness 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 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 next project called *SliderBindings*, in which I've moved all

the updating logic to 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 repetitive markup:

Project: SliderBindings | File: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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}" />

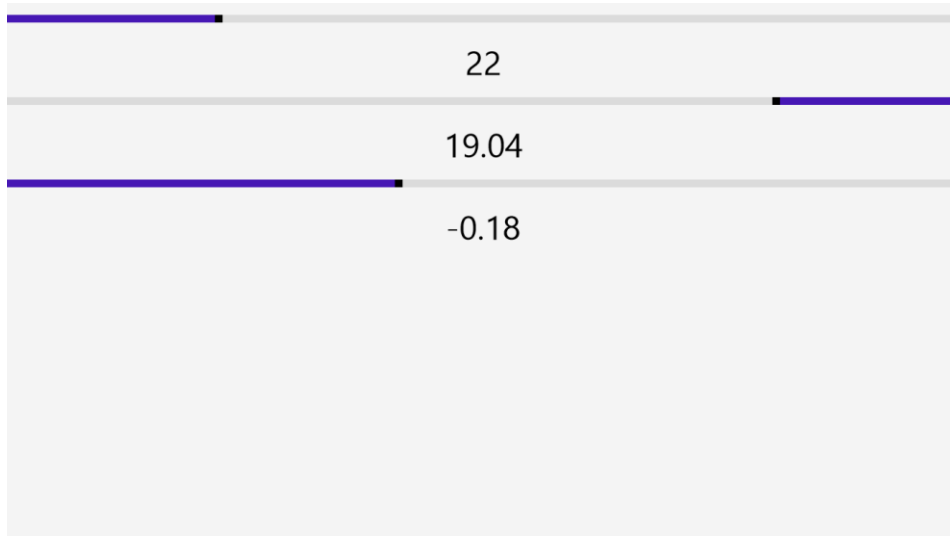
        <Slider Name="slider2"
            IsDirectionReversed="True"
            StepFrequency="0.01" />

        <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. The bindings then keep track of values changed through manipulations:



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 set 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*. 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: You can put the *Grid* 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 specified; 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* that has 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 non-default *VerticalAlignment* without weirdness happening.

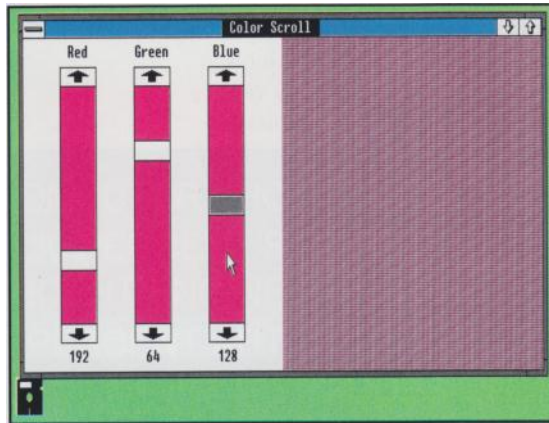
The *Height* property of *RowDefinition* and the *Width* property of *ColumnDefinition* are both of type *GridLength*, a structure defined in *Windows.UI.Xaml* that lets you specify *Auto* or star sizes from code. *RowDefinition* also defines *MinHeight* and *MaxHeight* properties, and *ColumnDefinition* 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 should probably try to avoid 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: MainPage.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 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: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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. When specifying attributes of *Grid* children, I tend to put these attached properties early but after at least one attribute (such as a *Name* or *Text*) that provides a quick visual identification of the element:

Project: SimpleColorScroll | File: MainPage.xaml (excerpt)

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

    ...

    <!-- Red -->
    <TextBlock Text="Red"
        Grid.Column="0"
        Grid.Row="0"
        Foreground="Red" />

    <Slider Name="redSlider"
        Grid.Column="0"
        Grid.Row="1"
        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"
        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"
        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 (or the *Loaded* event) is usually the place to do it.

All three *Slider* controls have the same handler for the *ValueChanged* event. That's in the code-behind file:

Project: SimpleColorScroll | File: MainPage.xaml.cs (excerpt)

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

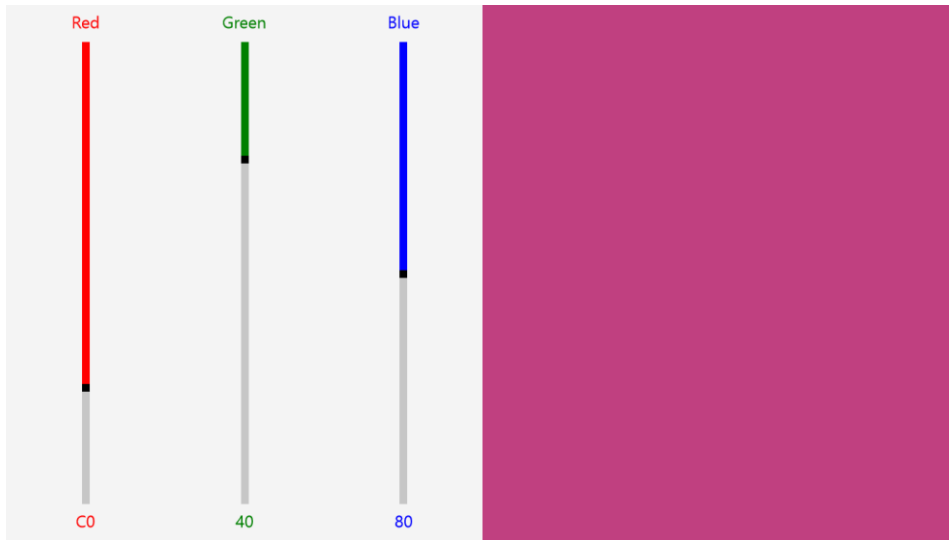
    void OnSliderValueChanged(object sender, RangeBaseValueChangedEventArgs args)
    {
        byte r = (byte)redSlider.Value;
        byte g = (byte)greenSlider.Value;
        byte b = (byte)blueSlider.Value;

        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: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}"
      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* tag 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: MainPage.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);

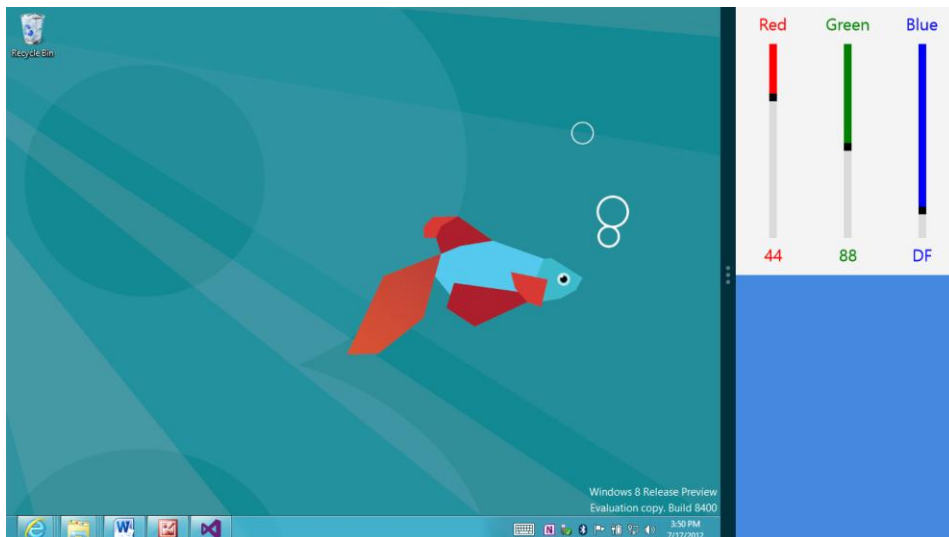
        Grid.SetColumn(rectangleResult, 1);
        Grid.SetRow(rectangleResult, 0);
    }
    // Portrait mode
    else
    {
        secondColDef.Width = new GridLength(0);
        secondRowDef.Height = new GridLength(1, GridUnitType.Star);

        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 this discrepancy results in the *Slider* value being 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 used 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: MainPage.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: MainPage.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}"
        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: MainPage.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);
}
```

It's possible to remove the *ThumbToolTipValueConverter* from the individual *Slider* tags and move it to the *Slider* style:

```
<Style TargetType="Slider">
    <Setter Property="Orientation" Value="Vertical" />
    <Setter Property="IsDirectionReversed" Value="True" />
    <Setter Property="Maximum" Value="255" />
    <Setter Property="HorizontalAlignment" Value="Center" />
    <Setter Property="ThumbToolTipValueConverter" Value="{StaticResource hexConverter}" />
</Style>
```

Is it possible to go another step with the data bindings and eliminate the *ValueChanged* handler entirely? 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: *MainPage.xaml* (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <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 ApplicationForegroundThemeBrush}"
    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: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    public MainPage()
    {
        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: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <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 Chapter 10.

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.

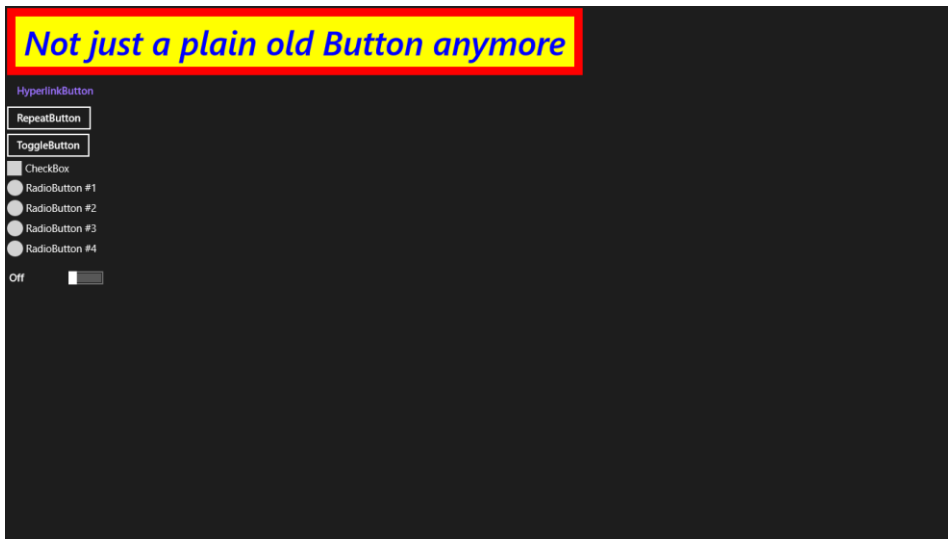
With all that said, you might want to gravitate towards the *ToggleSwitch* control for your toggling needs because it’s specifically designed for touch in Windows 8 applications. Although *ToggleSwitch* does not derive from *ButtonBase*, I’ve included one anyway at the bottom of the list. As you can 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 available 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 Chapter 10, 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 telephone number that you type 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 keypad *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: MainPage.xaml (excerpt)

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

    <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="⌫"
                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 than the width of the overall *Grid* minus the width of the delete button. But within that area the *Border* is aligned to the left. It is sized to fit the *TextBlock*, so despite its *HorizontalAlignment* setting, the *TextBlock* is also positioned 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 12 numeric and symbol 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: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    string inputString = "";
    char[] specialChars = { '*', '#' };

    public MainPage()
    {
        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;

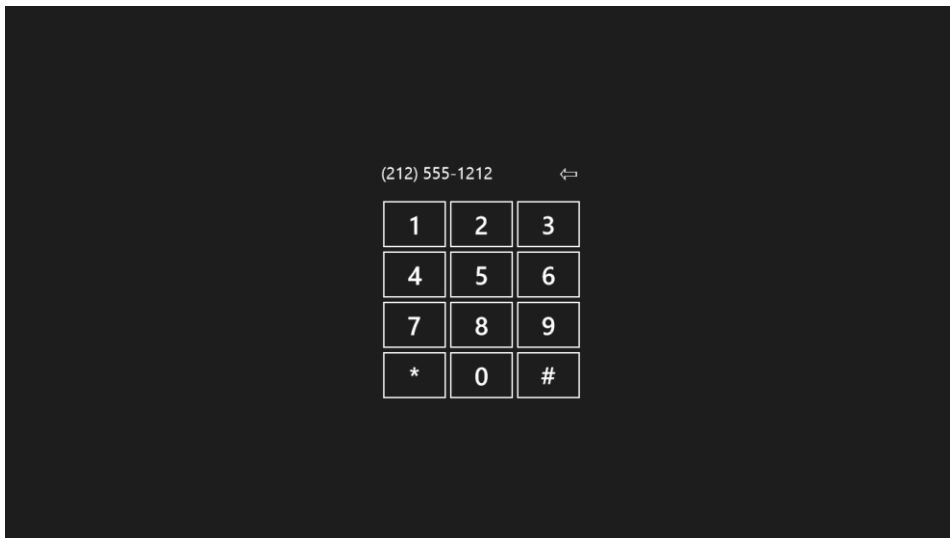
    else if (inputString.Length < 8)
        resultText.Text = String.Format("{0}-{1}", inputString.Substring(0, 3),
                                         inputString.Substring(3));

    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 as a telephone number. 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.

Defining 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 can 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.

An alternative approach is to define *DependencyProperty* objects as private static fields and then return those objects from the public static properties:

```

static readonly DependencyProperty color1Property =
    DependencyProperty.Register("Color1",
        typeof(Color),
        typeof(GradientButton),
        new PropertyMetadata(Colors.White, OnColorChanged));

static readonly DependencyProperty color2Property =
    DependencyProperty.Register("Color2",
        typeof(Color),
        typeof(GradientButton),
        new PropertyMetadata(Colors.Black, OnColorChanged));

public static DependencyProperty Color1Property
{

```

```

    get { return color1Property; }
}

public static DependencyProperty Color2Property
{
    get { return color2Property; }
}

```

The explicit static constructor isn't required. It's also possible to do it WPF or Silverlight style, where you don't have public static properties at all but simply define the static fields as public. Note that the fields are now named *Color1Property* and *Color2Property*:

```

public static readonly DependencyProperty Color1Property =
    DependencyProperty.Register("Color1",
        typeof(Color),
        typeof(GradientButton),
        new PropertyMetadata(Colors.White, OnColorChanged));

public static readonly DependencyProperty Color2Property =
    DependencyProperty.Register("Color2",
        typeof(Color),
        typeof(GradientButton),
        new PropertyMetadata(Colors.Black, OnColorChanged));

```

This approach works with Windows 8, but I tend not to use it because all the public static *DependencyProperty* objects defined by the standard Windows Runtime controls are properties rather than fields.

Regardless of how you define the public static *DependencyProperty* objects, the *GradientButton* class also needs regular .NET property definitions of *Color1* and *Color2*. These properties 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 property that's been changed. 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 of the *GradientButton* class in the class in an order 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.

Where are the actual values of the *Color1* and *Color2* properties stored? I suspect it's some kind of dictionary, perhaps optimized somewhat (one would hope) but otherwise inaccessible through the API. The state of these properties is managed by the operating system, and the only access to their values is through *SetValue* and *GetValue*.

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: MainPage.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>

```

```

</Page.Resources>

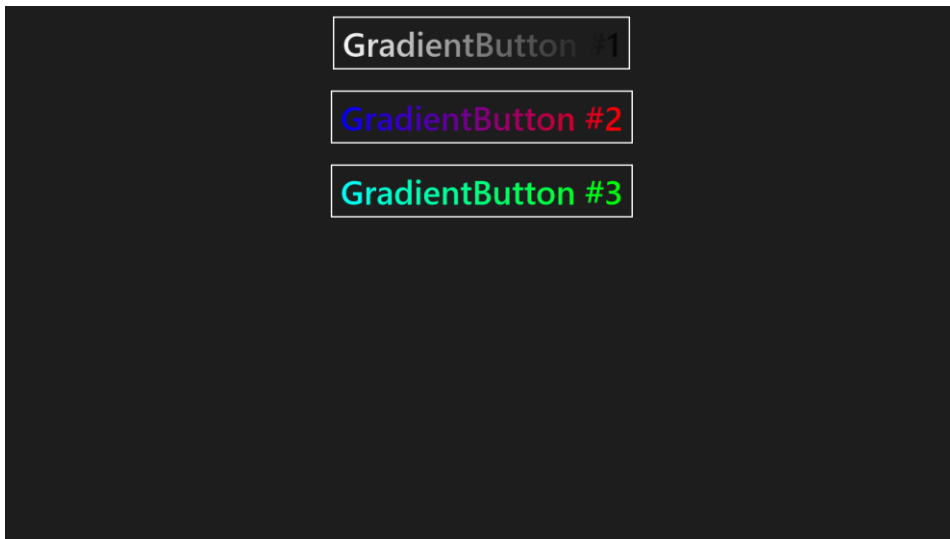
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <StackPanel>
        <local:GradientButton Content="GradientButton #1"
                               Style="{StaticResource baseButtonStyle}" />

        <local:GradientButton Content="GradientButton #2"
                               Style="{StaticResource blueRedButtonStyle}" />

        <local:GradientButton Content="GradientButton #3"
                               Style="{StaticResource baseButtonStyle}"
                               Color1="Aqua"
                               Color2="Lime" />
    </StackPanel>
</Grid>
</Page>

```

The first one gets the default settings of *Color1* and *Color2*, the second one gets the settings defined in the *Style*, and the third gets local settings Here it is:



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, to create the *GradientButton* class, 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 `MainPage.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. You can set *Tag* to anything you want to identify the control. 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 wordy and 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: *MainPage.xaml* (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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>
    </StackPanel>

```

```

        <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 ApplicationForegroundThemeBrush}"
              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 *Loaded* handler defined in the constructor in the code-behind file. (Oddly, when performing the initialization in the constructor, the line-join *RadioButton* gets initialized but not the other two.)

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: MainPage.xaml.cs (excerpt)

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

        Loaded += (sender, args) =>
        {
            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.StrokeEndLineCap;

            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 *Loaded* handler 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:



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: MainPage.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: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();

        Loaded += (sender, args) =>
        {
            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.StrokeEndLineCap;

        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 Windows 8 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. You need to write code that gives the control input focus.

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 lets code set the text in the *TextBox* or obtain 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 just one touch keyboard but several, and some are more suitable for entering numbers or email addresses or URLs. A *TextBox* specifies what type of keyboard it wants with the *InputScope* property.

With the current version of Windows 8, setting the *InputScope* property in XAML is somewhat clumsy. If you want IntelliSense to work, you'll need to use this rather involved syntax:

```
<TextBox ... >
```

```

<TextBox.InputScope>
  <InputScope>
    <InputScope.Names>
      <InputScopeName NameValue="Number" />
    </InputScope.Names>
  </InputScope>
</TextBox.InputScope>
</TextBox>

```

If you don't mind IntelliSense not working—and you don't mind the Visual Studio designer suggesting you've done something terribly wrong—you can simplify it to this:

```

<TextBox ... InputScope="Number" ... />

```

The following `TextBoxInputScopes` 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: `MainPage.xaml` (excerpt)

```

<Page ... >
  <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 ApplicationPageBackgroundThemeBrush}">
    <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" />
      </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"
    InputScope="Default" />

<!-- Email address input scope -->
<TextBlock Text="Email address input scope:"
    Grid.Row="4" Grid.Column="0" />

<TextBox Grid.Row="4" Grid.Column="1"
    InputScope="EmailSMTPAddress" />

<!-- Number input scope -->
<TextBlock Text="Number input scope:"
    Grid.Row="5" Grid.Column="0" />

<TextBox Grid.Row="5" Grid.Column="1"
    InputScope="Number" />

<!-- Search input scope -->
<TextBlock Text="Search input scope:"
    Grid.Row="6" Grid.Column="0" />

<TextBox Grid.Row="6" Grid.Column="1"
    InputScope="Search" />

<!-- Telephone number input scope -->
<TextBlock Text="Telephone number input scope:"
    Grid.Row="7" Grid.Column="0" />

<TextBox Grid.Row="7" Grid.Column="1"

```

```

        InputScope="TelephoneNumber" />

<!-- URL input scope -->
<TextBlock Text="URL input scope:"
           Grid.Row="8" Grid.Column="0" />

<TextBox Grid.Row="8" Grid.Column="1"
         InputScope="Url" />

<!-- 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 Chapter 13, "Touch, Etc.," 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*. *DragCompleted* indicates a lift of a finger or the release of the mouse button.

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 beginning with the word *Manipulation*.

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* shape 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 *MainPage.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: MainPage.xaml (excerpt)

```

<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}"
      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: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    const double BUTTON_SIZE = 60;
    const double BUTTON_FONT = 18;
    string blockChars = "ABCDEFGHJKLMNOPQRSTUVWXYZ0123456789!?.-+*/%=";

```

```

Color[] colors = { Colors.Red, Colors.Green, Colors.Orange, Colors.Blue, Colors.Purple };
Random rand = new Random();

public MainPage()
{
    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);
    }
}

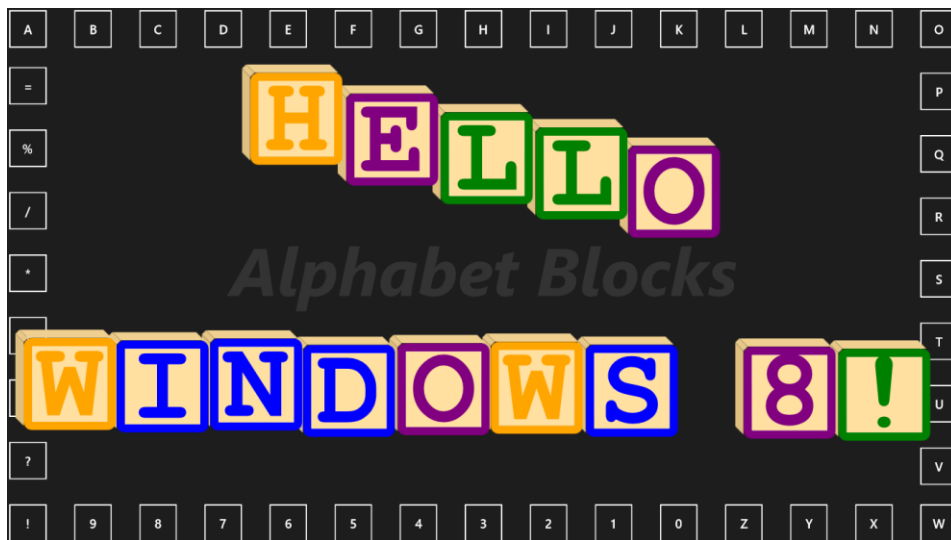
Button MakeButton(int index)
{
    Button button = new Button
    {
        Content = blockChars[index].ToString(),
        Width = BUTTON_SIZE,
        Height = BUTTON_SIZE,
        FontSize = BUTTON_FONT,
        Tag = new SolidColorBrush(colors[index % colors.Length]),
    };
    button.Click += OnButtonClick;
    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:



Chapter 6

WinRT and MVVM

In structuring software, one of the primary 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 somewhat 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, most or all of these method calls and event handling actually occur 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*, converting from *double* to *string* in the process. 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 skim 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. Through data bindings, the controls in the View are connected 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 explore this issue shortly.)

The *RgbViewModel* class is part of the *ColorScrollWithViewModel* project. The *MainPage.xaml* file instantiates the *RgbViewModel* in its *Resources* section.

Project: *ColorScrollWithViewModel* | File: *MainPage.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. That 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: *MainPage.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}"
        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.

Does this seem backwards? Shouldn't the *Slider* be providing a value to *RgbViewModel*?

Yes, but *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

- an updated source value causes a change to the target property (the normal case for a data binding), and
- 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: MainPage.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 usually 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.

Rather than guard against re-entry, try searching for a change in logic that does what you want. 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.

Syntactic Shortcuts

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 projects of type Grid App and Split App. (Don't confuse this class with the *BindingBase* class from which *Binding* derives.)

However, Visual Studio does *not* create this *BindableBase* class in the Blank App project. But let's take a look at it and see if we can learn anything.

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 steals some code from *BindableBase*, and I've given *Color* a public *set* accessor:

Project: *ColorScrollWithDataContext* | File: *RgbViewModel.cs*

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
```



```

using Windows.UI;

namespace ColorScrollWithDataContext
{
    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 (SetProperty<double>(ref red, value, "Red"))
                    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);
}

protected bool SetProperty<T>(ref T storage, T value,
                              [CallerMemberName] string propertyName = null)
{
    if (object.Equals(storage, value))
        return false;

    storage = value;
    OnPropertyChanged(propertyName);
    return true;
}

protected void OnPropertyChanged(string propertyName)
{
    if (PropertyChanged != null)
        PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
}
}
}

```

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*, *RelativeSource*, and *Source*. *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 Chapter 10.) 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 *MainPage.xaml.cs* file of the *ColorScrollWithDataContext* project:

Project: *ColorScrollWithDataContext* | File: *MainPage.xaml.cs*

```
public MainPage()
{
    this.InitializeComponent();
    this.DataContext = new RgbViewModel();

    // Initialize to highlight color
    (this.DataContext as RgbViewModel).Color =
        (this.Resources["ApplicationPressedForegroundThemeBrush"] 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: *MainPage.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}"
        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 in Chapter 11, "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: *MainPage.xaml.cs* (excerpt)

```

public MainPage()
{
    this.InitializeComponent();
    this.DataContext = new RgbViewModel();

    // Initialize to highlight color
    (this.DataContext as RgbViewModel).Color =
        (this.Resources["ApplicationPressedForegroundThemeBrush"] 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: MainPage.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 ApplicationPageBackgroundThemeBrush}">
        <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" />

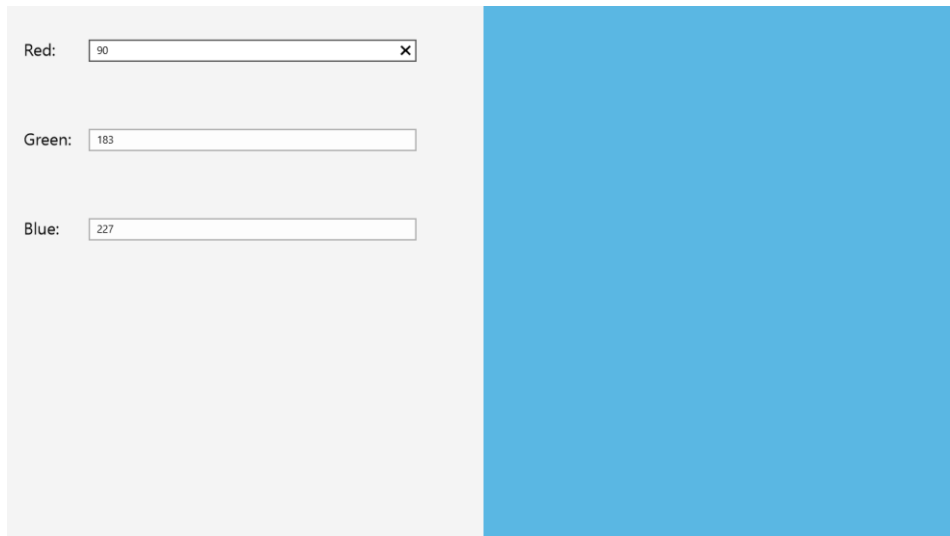
<TextBlock Text="Blue: "
           Grid.Row="2"
           Grid.Column="0" />

<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, all the necessary data conversions being performed behind the scenes:



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, and if you prefer not duplicating *TextBox* logic with a control of your own, 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: *MainPage.xaml* (excerpt)

```
<TextBlock Text="Red: "  
    Grid.Row="0"  
    Grid.Column="0" />  
  
<TextBox Name="redTextBox"  
    Grid.Row="0"  
    Grid.Column="1"  
    Text=""  
    TextChanged="OnTextBoxTextChanged" />  
  
<TextBlock Text="Green: "  
    Grid.Row="1"  
    Grid.Column="0" />  
  
<TextBox Name="greenTextBox"  
    Grid.Row="1"  
    Grid.Column="1"  
    Text=""  
    TextChanged="OnTextBoxTextChanged" />  
  
<TextBlock Text="Blue: "  
    Grid.Row="2"  
    Grid.Column="0" />  
  
<TextBox Name="blueTextBox"  
    Grid.Row="2"  
    Grid.Column="1"  
    Text=""  
    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, but I also decided I wanted to actually validate the text entered by the user:

Project: ColorTextBoxesWithEvents | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    RgbViewModel rgbViewModel;
    Brush textBoxTextBrush;
    Brush textBoxErrorBrush = new SolidColorBrush(Colors.Red);

    public MainPage()
    {
        this.InitializeComponent();

        // Get TextBox brush
        textBoxTextBrush = this.Resources["TextBoxForegroundThemeBrush"] 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["ApplicationPressedForegroundThemeBrush"] 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.

In some cases, data entry validation might more properly be under the jurisdiction of View Model rather than the View.

Buttons and MVVM

At first, the idea that you can use MVVM to eliminate most of a 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 through a data binding. *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 object 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, 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) generic delegates defined in the *System* namespace. These predefined delegates have the names *Action* and *Func* with anything from 1 to 16 arguments. The *Func* delegates returns an object of a particular type; the *Action* delegates do not. The *Action<object>* delegate represents a method with a single *object* argument and a *void* return value; 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. 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 System.ComponentModel;
using System.Runtime.CompilerServices;

namespace KeypadWithViewModel
{
    public class KeypadViewModel : INotifyPropertyChanged
    {
        string inputString = "";
        string displayText = "";
        char[] specialChars = { '*', '#' };

        public event PropertyChangedEventHandler PropertyChanged;

        // 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;
}

protected bool SetProperty<T>(ref T storage, T value,
                              [CallerMemberName] string propertyName = null)
{
    if (object.Equals(storage, value))
        return false;

    storage = value;
    OnPropertyChanged(propertyName);
    return true;
}

```

```

    }

    protected void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
}

```

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 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: MainPage.xaml (excerpt)

```

<Page ... >

    <Page.Resources>
        <local:KeypadViewModel x:Key="viewModel" />
    </Page.Resources>

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}"
          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.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 SimplePad, 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 Windows 8 applications that are part of Windows 8.

The application bars in the Windows 8 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 31 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 31 styles is set to character codes ranging from 0xE100 to 0xE11C, and 0xE141 ("Pin") and 0xE196 ("Unpin"). 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 Windows 8 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 Maps 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 Windows 8 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 Windows 8 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, Start, 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 single 32-bit code to represent any character 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 storing 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 only goes up to character codes of 0x1FFFF. 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: MainPage.xaml (excerpt)

```
<Page ... >

    <Page.Resources>
        <local:DoubleToStringHexByteConverter x:Key="hexByteConverter" />
    </Page.Resources>

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <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: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : 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 MainPage()
    {
        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);
        txtblk.ColumnHeads[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's also that irksome matter of dealing with character codes above 0xFFFF:

Project: SegoeSymbols | File: MainPage.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 <= 0xFFFFF)
            {
                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;
        }
    }
}

```



```

    }
}
}

```

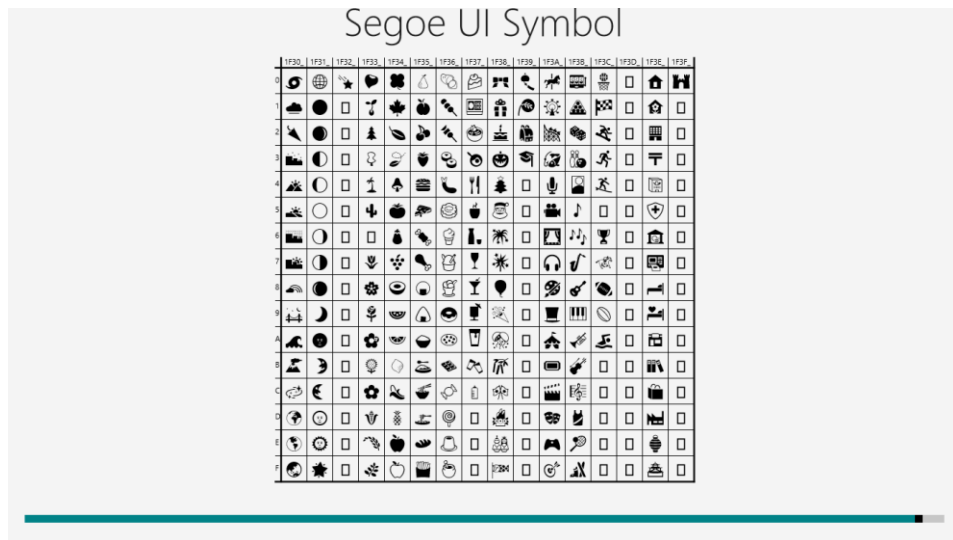
Four statements towards the end of the handler demonstrate the mathematics that separate a Unicode character code between 0x10000 and 0x10FFFF into two 10-bit values to construct leading and trailing surrogates, which in sequence in a string define a single character.

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 *code* value of 0xFFFF and below, *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 Segoe 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 SimplePad programs coming up might represent a first step in creating a new 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. SimplePad1 and SimplePad2 are functionally equivalent but use different methods for asynchronous file I/O. Here's the MainPage.xaml file for SimplePad1:

Project: SimplePad1 | File: MainPage.xaml (excerpt)

```
<Page ... >

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <TextBox Name="textbox"
            FontSize="24"
            AcceptsReturn="True" />
    </Grid>

    <Page.BottomAppBar>
        <AppBar Padding="10 0">
            <Grid>
                <StackPanel Orientation="Horizontal"
                    HorizontalAlignment="Left">

                    <Button Style="{StaticResource AppBarButtonStyle}"
                        Content="⌵"
                        AutomationProperties.Name="Wrap options"
                        Click="OnWrapOptionsAppBarButtonClick" />

                </StackPanel>

                <StackPanel Orientation="Horizontal"
                    HorizontalAlignment="Right">

                    <Button Style="{StaticResource AppBarButtonStyle}"
                        Content="⌵"
                        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: SimplePad1 | File: MainPage.xaml.cs (excerpt)

```

public MainPage()
{
    this.InitializeComponent();

    Loaded += (sender, args) =>
    {
        txtbox.Focus(FocusState.Keyboard);
    };
}

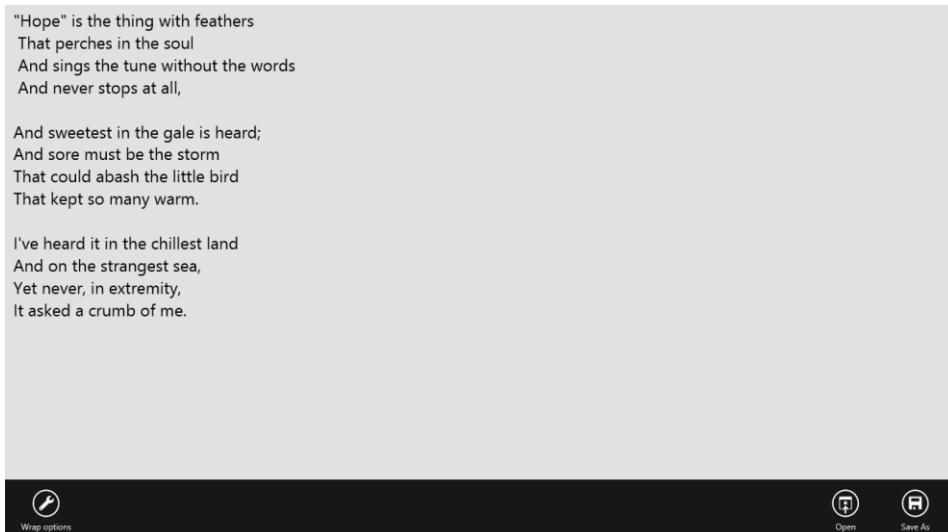
```

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 SimplePad1, 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:



You can use a *RequestedTheme* of *Light* when using an application bar, but it seems to be an uncommon practice.

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 SimplePad1 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: SimplePad1 | File: WrapOptionsDialog.xaml (excerpt)

```
<UserControl ... >

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <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 Windows 8 applications have a white background and black foreground. You can accomplish this with a *RequestedTheme* of *Light* (in which case your application bar will also be white) but apart from that, I've had frustrating experiences trying to flip the colors in the dialog box. You can set the *Grid* background to *ApplicationForegroundThemeBrush* or explicitly to white, but setting the *Foreground* on the root element does not properly 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: SimplePad1 | 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 *MainPage* 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: SimplePad1 | File: MainPage.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
}

```

```

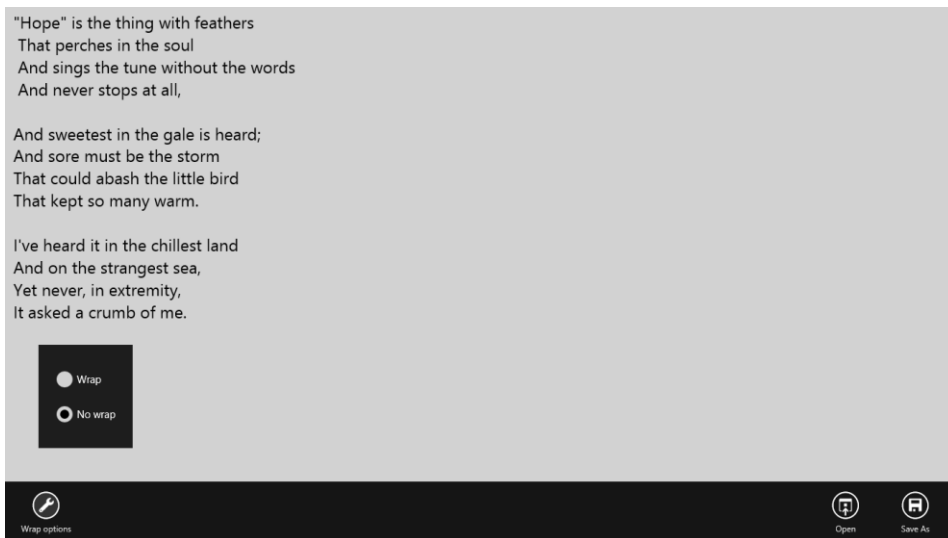
        popup.IsOpen = true;
    }

```

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 down in the lower-left corner:



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

SimplePad1 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 8 program is similar to a traditional Windows Desktop program in being event-driven and structured much like a state machine. Following initialization, a program generally sits dormant 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 SimplePad1 showing the creation and initialization of *FileOpenPicker*, and the *PickSingleFileAsync* call:

Project: SimplePad1 | File: MainPage.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>*. What you set to the *Completed* property is a callback method in your code. Although *Completed* is defined as a property, it functions like an event, in that it signals something of interest to your program. (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 statement that sets the *Completed* property to the callback method, 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 an 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: SimplePad1 | File: MainPage.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: SimplePad1 | File: MainPage.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: SimplePad1 | File: MainPage.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 I've put them 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 *RunAsync* method referencing a method in your code that will run in the proper thread.

Here's the *OnDataReaderLoadCompleted* method calling *RunAsync* 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. Notice that the text read from the *DataReader* is saved as a field so that it can be accessed by the callback method:

Project: SimplePad1 | File: MainPage.xaml.cs (excerpt)

```
string dataReaderText;
...
void OnDataReaderLoadCompleted(IAsyncOperation<uint> asyncInfo, AsyncStatus asyncStatus)
{
    if (asyncInfo.ErrorCode != null)
        return;

    uint length = asyncInfo.GetResults();
    dataReaderText = dataReader.ReadString(length);

    IAsyncAction asyncAction = this.Dispatcher.RunAsync(CoreDispatcherPriority.Normal,
                                                         SetTextBoxText);
    asyncAction.Completed = OnRunAsyncCompleted;
}

void SetTextBoxText()
{
    textbox.Text = dataReaderText;
}
```

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 *RunAsync*, like this:

```
this.Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
{
    textbox.Text = text;
});
```

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 SimplePad1.

Even so, saving to a file is potentially more involved than 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, and that's what I've chosen to use here.

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 *RunAsync* method. Everything goes in the *Click* handler for the button:

Project: SimplePad1 | File: MainPage.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, () =>
        {
            text = textbox.Text;
        })
        .Completed = (asyncInfo2, asyncStatus2) =>
        {
            FileIO.WriteTextAsync(storageFile, text).Completed =
```

```

        (asyncInfo3, asyncStatus3) =>
        {
            };
        };
    };
}

```

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 SimplePad2 program is the same as SimplePad1 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: SimplePad2 | File: MainPage.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, textbox.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, textbox.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: SimplePad2 | File: MainPage.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!
    textbox.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. These details will be discussed 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 MainPage()
{
    this.InitializeComponent();
    ...
    Loaded += OnLoaded;
}
async void OnLoaded(object sender, RoutedEventArgs arg)
{
    ...
}
```

Or, if you prefer defining the *Loaded* handler as an anonymous method:

```
public MainPage()
{
    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 Windows 8 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 result. 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 the *Child1* and *Child2* properties are set 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:

Project: XamlCruncher | File: RulerContainer.xaml.cs (excerpt)

```

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(KeyRoutedEventArgs 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;
        int i = 0;

        while (i < index)
        {
            if (Text[i] == '\n')
            {
                line++;
                col = 0;
            }
            else if (Text[i] == '\r')
            {
                index++;
            }
            else
            {
                col++;
            }
            i++;
        }
    }
}

```

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. (Although the lines in the *Text* property of the *TextBox* are delimited by a carriage return and line feed, the *SelectionStart* index is calculated based on just one end-of-line character.) 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 deal 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 *false* when a new file is loaded or the file is saved and to *true* 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* property. 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.

If you want, you can store program settings in isolated storage, perhaps in XML format serialized from a class typically called *AppSettings*. If you store settings in the *ApplicationData.Current.LocalFolder* directory, you’ll discover that it maps to the following 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.

An easier (though somewhat less versatile) approach involves the *LocalSettings* property of the *ApplicationData.Current* object. This is a dictionary that let you identify individual settings with string keys. This is the approach I’ve used in the *AppSettings* class. *AppSettings* also implements *INotifyPropertyChanged* to let it be used for data binding. It’s basically a View Model, or perhaps (in a larger application) part of a View Model.

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 *AppSettings* showing all the properties that comprise program settings. The constructor loads the settings, and a Save method saves them. 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 : INotifyPropertyChanged
{
    // 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 = 4;

    public event PropertyChangedEventHandler PropertyChanged;

    public AppSettings()
    {
        ApplicationDataContainer appData = ApplicationData.Current.LocalSettings;

        if (appData.Values.ContainsKey("EditOrientation"))
            this.EditOrientation = (EditOrientation)(int)appData.Values["EditOrientation"];

        if (appData.Values.ContainsKey("AutoParsing"))
            this.AutoParsing = (bool)appData.Values["AutoParsing"];

        if (appData.Values.ContainsKey("ShowRuler"))
            this.ShowRuler = (bool)appData.Values["ShowRuler"];

        if (appData.Values.ContainsKey("ShowGridLines"))
            this.ShowGridLines = (bool)appData.Values["ShowGridLines"];

        if (appData.Values.ContainsKey("FontSize"))
            this.FontSize = (double)appData.Values["FontSize"];

        if (appData.Values.ContainsKey("TabSpaces"))
            this.TabSpaces = (int)appData.Values["TabSpaces"];
    }

    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; }
}

public Orientation Orientation
{
    protected set { SetProperty<Orientation>(ref orientation, value); }
    get { return orientation; }
}

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; }
    }

    public void Save()
    {
        ApplicationDataContainer appData = ApplicationData.Current.LocalSettings;
        appData.Values.Clear();
        appData.Values.Add("EditOrientation", (int)this.EditOrientation);
        appData.Values.Add("AutoParsing", this.AutoParsing);
        appData.Values.Add("ShowRuler", this.ShowRuler);
        appData.Values.Add("ShowGridLines", this.ShowGridLines);
        appData.Values.Add("FontSize", this.FontSize);
        appData.Values.Add("TabSpaces", this.TabSpaces);
    }

    protected bool SetProperty<T>(ref T storage, T value,
                                   [CallerMemberName] string propertyName = null)
    {
        if (object.Equals(storage, value))
            return false;

        storage = value;
        OnPropertyChanged(propertyName);
        return true;
    }

    protected void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}

```

Besides *EditOrientation*, *AppSettings* defines two additional properties that more directly correspond to properties of the *SplitContainer*: *Orientation* and *SwapEditAndDisplay*. The *set* accessors are protected, and the properties are set only from the *set* accessor of *EditOrientation*. These two

properties are not saved with the other application settings, but they are easily derived from application settings and make the bindings easier.

The XamlCruncher Page

Sufficient pieces have now been created to let us begin assembling this application. Here's MainPage.xaml:

Project: XamlCruncher | File: MainPage.xaml (excerpt)

<Page ... >

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <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>
</Grid>
```

```

</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="⋮"
                AutomationProperties.Name="Options"
                Click="OnOptionsAppBarButtonClick" />
      </StackPanel>

      <StackPanel Orientation="Horizontal" HorizontalAlignment="Right">
        <Button Style="{StaticResource AppBarButtonStyle}"
                Content="⌕"
                AutomationProperties.Name="Open"
                Click="OnOpenAppBarButtonClick"
                />

        <Button Style="{StaticResource SaveAppBarButtonStyle}"
                AutomationProperties.Name="Save As"
                Click="OnSaveAsAppBarButtonClick" />

        <Button Style="{StaticResource AppBarButtonStyle}"
                Content="💾"
                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 *MainPage* 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: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    ...
    AppSettings appSettings;
    StorageFile loadedStorageFile;

    public MainPage()
    {
        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(Windows.Globalization.Language.CurrentInputLanguageTag);
        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 = new AppSettings();
    }
}
```



```

        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. (In Chapter 16, “Going Native,” I demonstrate how to use *DirectWrite* to obtain a list of fonts installed on the system.)

The remaining initialization occurs in the *Loaded* event handler. The *DataContext* of the page is set to the *AppSettings* instance, as you probably anticipated from the data bindings in the *MainPage.xaml* file.

The *OnLoaded* method continues 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: *MainPage.xaml.cs* (excerpt)

```
public sealed partial class MainPage : Page
{
    Brush textBlockBrush, textBoxBrush, errorBrush;
    ...
    public MainPage()
    {
        ...
        // Set brushes
        textBlockBrush = Resources["ApplicationForegroundThemeBrush"] as SolidColorBrush;
        textBoxBrush = Resources["TextBoxForegroundThemeBrush"] 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: MainPage.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: MainPage.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
{
    editText.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: MainPage.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;
    UICommand 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 ApplicationPageBackgroundThemeBrush}"
    BorderBrush="{StaticResource ApplicationForegroundThemeBrush}"
    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 ApplicationForegroundThemeBrush}" />
                    <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 *MainPage*. 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 SimplePad programs:

Project: XamlCruncher | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : 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;
    }

    void OnPopupClose(object sender, object args)
    {
        appSettings.Save();
        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.

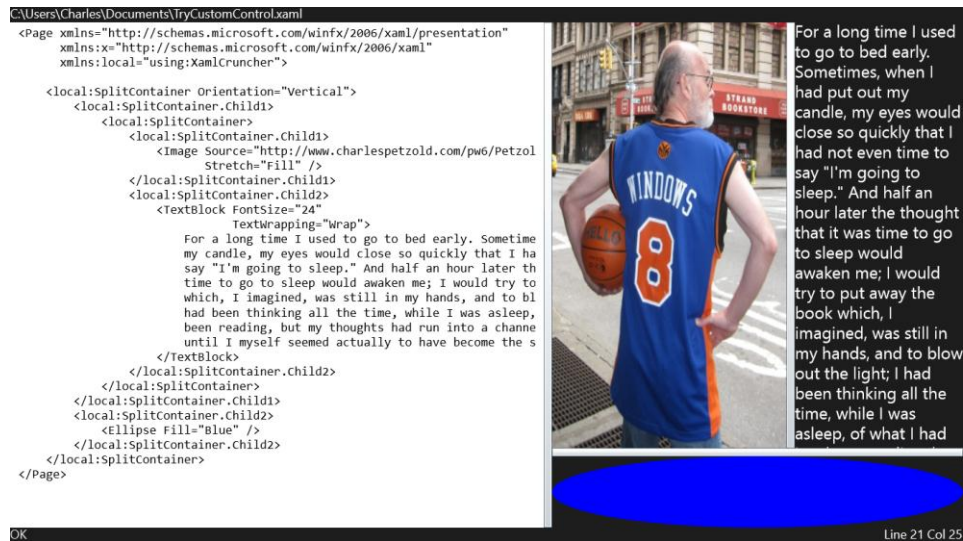
One of the big “to do” items is to handle the *Suspending* event of the *App* object. This event indicates when Windows 8 is suspending an application but also when the application is about to terminate. My thinking now is that the program should save any edited document in the *LocalFolder* area and 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.

Issues such as these will be covered in Chapter 12, “Pages and Navigation.”

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.

Chapter 8

Animation

The topic of animation might at first seem as if it doesn't quite belong in the "Elementals" section of this book. Perhaps the subject is more suited for advanced programmers working on games or physics simulations. Animation just doesn't seem appropriate in sedate and dignified business applications (except perhaps on casual Fridays).

But animation has more of a central role in Windows 8 applications than you might think. You'll discover part of that role in Chapter 10, "The Two Templates," which shows how to use XAML to create *ControlTemplate* objects that entirely redefine the appearances of controls. Although the most important part of a *ControlTemplate* is a visual tree, the template must also indicate how the appearance of the control changes under certain conditions. For example, a *Button* might be highlighted when it's pressed and "grayed out" when it's disabled. All these changes in appearance within the *ControlTemplate* are defined as animations—even if the change is instantaneous and doesn't really seem much like an animation.

Animations also come into play to define transitions between different application views or the movement of items during changes to a collection. Try moving a tile on the Start screen from one location to another and you'll see neighboring tiles shift in response. These are animations, and this is an important part of the fluid nature of Windows 8 aesthetics.

The *Windows.UI.Xaml.Media.Animation* Namespace

In Chapter 3, "Basic Event Handling," I demonstrated how to animate objects by using the *CompositionTarget.Rendering* event, a technique I referred to as a "manual" animation. While a manual animation can be powerful, it has some limitations. The callback method always runs in the user interface thread, which means that the animation can interfere with program responsiveness to user input.

Also, the animations I demonstrated with *CompositionTarget.Rendering* were all linear—that is, they increased or decreased a value linearly over a period of time. Animations are often more pleasant when they have a little variation, usually by speeding up at the beginning and slowing down towards the end, perhaps with a little "bounce" for extra realism. You can certainly perform animations of this sort using *CompositionTarget.Rendering*, but the mathematics can be challenging.

In contrast, in this chapter I'll be demonstrating instead the built-in Windows Runtime animation facility that consists of 71 classes, 4 enumerations, and 2 structures in the *Windows.UI.Xaml.Media.Animation* namespace. These animations often run on background threads and support several features for sophisticated effects. Very often, you can define animations entirely in XAML and then

trigger them from code or (in one particular but common case) directly from XAML.

Of course, the very idea of mastering an animation facility with 71 classes can be intimidating. Fortunately, these classes fall into just a few general categories, and by the end of this chapter, the namespace should be entirely comprehensible.

Animation involves change, and what these animations change is a property of an object. This property is often referred to as the “target” of the animation. The Windows Runtime animations require this target property to be backed by a dependency property and therefore defined in a class that derives from *DependencyObject*.

Some graphical environments have animations that are *frame-based*, meaning that the pacing of the animation is based on the frame rate of the video display. Different video frame rates on different hardware platforms might result in animations of different speeds. The Windows Runtime animations are instead *time-based*, meaning that they are based on actual durations of clock time: seconds and milliseconds.

What happens if the thread running an animation needs to do some work and the animation misses a few ticks? A frame-based animation generally continues where it left off. A Windows Runtime time-based animation adjusts itself based on clock time and catches up to where it should be.

Animation Basics

Let’s begin with the animation of the *FontSize* property of a *TextBlock*, much like the *ExpandingText* program in Chapter 3. The *SimpleAnimation* project has a two-row *Grid* with a *TextBlock* and a *Button* to start the animation going. Very often, animations are defined in the *Resources* section of the root element of the XAML file. A simple animation like this one consists of a *Storyboard* and a *DoubleAnimation*:

Project: SimpleAnimation | File: MainPage.xaml (excerpt)

<Page ... >

```
<Page.Resources>
  <Storyboard x:Key="storyboard">
    <DoubleAnimation Storyboard.TargetName="txtblk"
      Storyboard.TargetProperty="FontSize"
      EnableDependentAnimation="True"
      From="1" To="144" Duration="0:0:3" />
  </Storyboard>
</Page.Resources>

<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
```



```

        <TextBlock Name="txtblk"
            Text="Animated Text"
            Grid.Row="0"
            FontSize="48"
            HorizontalAlignment="Center"
            VerticalAlignment="Center" />

        <Button Content="Trigger!"
            Grid.Row="1"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Click="OnButtonClick" />
    </Grid>
</Page>

```

The name of the *DoubleAnimation* class doesn't mean that it performs two animations! This is an animation that targets properties of type *Double*. As you'll see, the Windows Runtime also supports animations that target properties of type *Point*, *Color*, and *Object*. (An animation that targets properties of type *Object* might seem as if it's the only animation you'd need, but in reality it's limited to switching property values rather than smoothly animating them.)

The Windows Runtime requires that an animation object such as *DoubleAnimation* be a child of a *Storyboard*. A *Storyboard* can have multiple children performing parallel animations, and the job of the *Storyboard* is to ensure that these children are properly synchronized.

Storyboard also defines two attached properties named *TargetName* and *TargetProperty*. You set these properties in the animation object to indicate the name of the object you're targeting, and the property of that object you wish to animate:

```

<Storyboard x:Key="storyboard">
    <DoubleAnimation Storyboard.TargetName="txtblk"
        Storyboard.TargetProperty="FontSize"
        ... />
</Storyboard>

```

By default, animations are performed in a secondary thread so that the user interface thread remains free to respond to user input. However, an animation that targets the *FontSize* property of a *TextBlock* must run in the user interface thread because a change in the font size triggers a layout change. The Windows Runtime is reluctant to run animations in the user interface thread, even to the extent of implementing a default behavior to disallow them! To let the Windows Runtime know that yes, you want the animation to run even if it happens in the user interface thread, you must set the *EnableDependentAnimation* property to *true*:

```

<Storyboard x:Key="storyboard">
    <DoubleAnimation Storyboard.TargetName="txtblk"
        Storyboard.TargetProperty="FontSize"
        EnableDependentAnimation="True"
        ... />
</Storyboard>

```

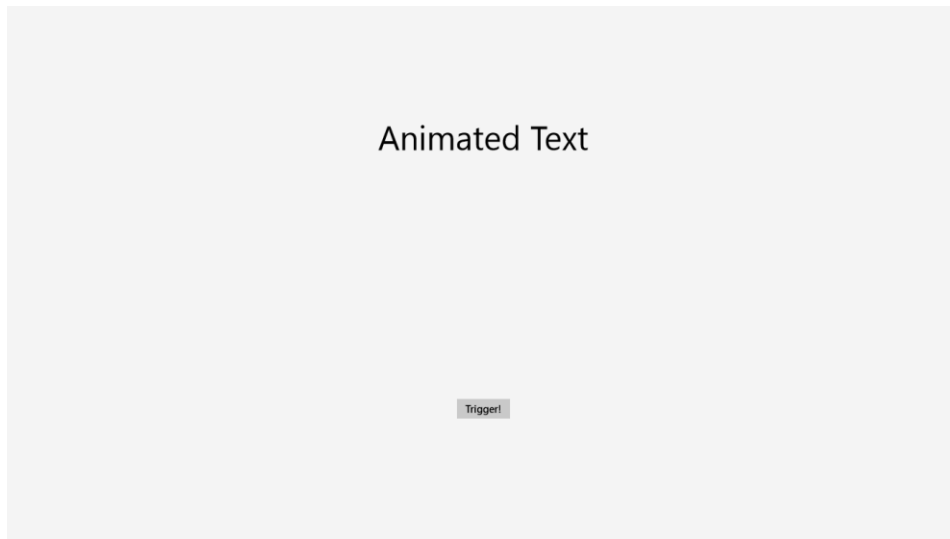
In this context, the word "dependent" means "dependent on the user interface thread."

The remainder of this particular animation indicates that we want to animate the value of the *FontSize* property from 1 to 144 over the course of three seconds:

```
<Storyboard x:Key="storyboard">
    <DoubleAnimation Storyboard.TargetName="txtblk"
        Storyboard.TargetProperty="FontSize"
        EnableDependentAnimation="True"
        From="1" To="144" Duration="0:0:3" />
</Storyboard>
```

The duration of the animation is specified in hours, minutes, and seconds. All three values and two colons are required. If you insist on specifying just one number, it will be interpreted as an integral number of hours; two numbers separated by a colon are interpreted as hours and minutes. The seconds can include fractional seconds. If you need an animation that runs more than a day, you can precede the hours with a number of days and a period.

When you first run this program, the *TextBlock* is displayed with a 48-pixel height, as specified in the *TextBlock* element in the XAML file:



The *Storyboard* doesn't run by itself. It needs to be triggered, usually by something happening in the user interface. In this program, the *Click* handler for the *Button* obtains a reference to the *Storyboard* by accessing the *Resources* collection, and then it calls *Begin*:

Project: SimpleAnimation | File: MainPage.xaml.cs

```
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media.Animation;

namespace SimpleAnimation
{
    public sealed partial class MainPage : Page
```

```

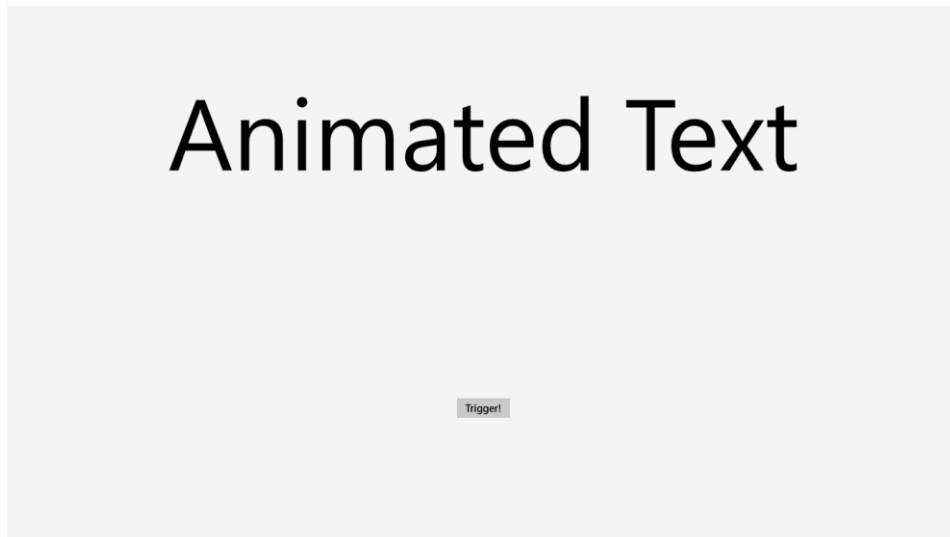
{
    public MainPage()
    {
        this.InitializeComponent();
    }

    void OnButtonClick(object sender, RoutedEventArgs args)
    {
        (this.Resources["storyboard"] as Storyboard).Begin();
    }
}

```

Notice the *using* directive for *Windows.UI.Xaml.Media.Animation*. This is not provided for you automatically by the Visual Studio template.

When the button is clicked, the *TextBlock* immediately jumps to a *FontSize* of 1 (the *From* value in *DoubleAnimation*), and then the *FontSize* increases to 144 over the course of three seconds. The increase is linear: At the one-second mark, the *FontSize* is 48-2/3 pixels, and at two seconds, it's 96-1/3. At the end of three seconds, the animation stops and the *TextBlock* remains at the 144-pixel size:



You can click the button again, and the animation starts over again. In fact, you can click the button repeatedly while the animation is running, and each time it starts over again at the 1-pixel size.

Animation Variation Appreciation

When the animation in the SimpleAnimation program completes, the *FontSize* remains at the value specified by the *To* property of *DoubleAnimation*. This is a result of the setting of the *FillBehavior* property of *DoubleAnimation*, which by default is the enumeration member *HoldEnd*. You can

alternatively set it to *Stop*:

```
<Storyboard x:Key="storyboard">
    <DoubleAnimation Storyboard.TargetName="txtblk"
        Storyboard.TargetProperty="FontSize"
        EnableDependentAnimation="True"
        FillBehavior="Stop"
        From="1" To="144" Duration="0:0:3" />
</Storyboard>
```

Now at the end of the animation, the animation is released from the target property and *FontSize* reverts to its pre-animation value of 48.

Another variation is to leave out the *From* or *To* value. For example,

```
<Storyboard x:Key="storyboard">
    <DoubleAnimation Storyboard.TargetName="txtblk"
        Storyboard.TargetProperty="FontSize"
        EnableDependentAnimation="True"
        From="1" Duration="0:0:3" />
</Storyboard>
```

Now the animation begins at 1 but goes up only to the pre-animation value of 48. The increase in size proceeds at a slower rate because the duration is still three seconds.

This animation causes *FontSize* to go from its current value up to 144 over three seconds:

```
<Storyboard x:Key="storyboard">
    <DoubleAnimation Storyboard.TargetName="txtblk"
        Storyboard.TargetProperty="FontSize"
        EnableDependentAnimation="True"
        To="144" Duration="0:0:3" />
</Storyboard>
```

I say the *FontSize* goes from “its current value” because that value isn’t necessarily the pre-animation value of 48. Click the button, and while the *TextBlock* is still increasing in size, click the button again. Each successive click effectively terminates the existing animation and starts a new animation from the current *FontSize*. Each new click slows down the rate of increase because the length of the animation is still three seconds.

You might assume that the *DoubleAnimation* class defines the *To* and *From* properties as type *double*. That’s *almost* true. They are actually of type nullable *double*, and *null* is the default value. This is how *DoubleAnimation* can determine whether these properties are set.

The other option is *By*:

```
<Storyboard x:Key="storyboard">
    <DoubleAnimation Storyboard.TargetName="txtblk"
        Storyboard.TargetProperty="FontSize"
        EnableDependentAnimation="True"
        By="100" Duration="0:0:3" />
</Storyboard>
```

Now each click of the button triggers an animation that increases the *FontSize* by another 100 pixels over the course of three seconds. The text just gets larger and larger and larger.

Try going back to the original settings and add an attribute that sets *AutoReverse* to *true*:

```
<Storyboard x:Key="storyboard">
  <DoubleAnimation Storyboard.TargetName="txtblk"
    Storyboard.TargetProperty="FontSize"
    EnableDependentAnimation="True"
    From="1" To="144" Duration="0:0:3"
    AutoReverse="True" />
</Storyboard>
```

When this animation is triggered, the *FontSize* jumps down to 1, goes up to 144 over the course of three seconds, and then goes back down to 1 over another three seconds, at which time the animation is completed. The entire animation is six seconds in length. Set *FillBehavior* to *Stop*, and the *FontSize* will jump back to its pre-animation value of 48 at the end of those six seconds.

You can also set a *RepeatBehavior* attribute with or without *AutoReverse*. The following combination indicates that you want to perform three entire cycles of increasing and decreasing the *FontSize*:

```
<Storyboard x:Key="storyboard">
  <DoubleAnimation Storyboard.TargetName="txtblk"
    Storyboard.TargetProperty="FontSize"
    EnableDependentAnimation="True"
    From="1" To="144" Duration="0:0:3"
    AutoReverse="True"
    RepeatBehavior="3x" />
</Storyboard>
```

The entire animation last for 18 seconds.

You can also set *RepeatBehavior* to a duration:

```
<Storyboard x:Key="storyboard">
  <DoubleAnimation Storyboard.TargetName="txtblk"
    Storyboard.TargetProperty="FontSize"
    EnableDependentAnimation="True"
    From="1" To="144" Duration="0:0:3"
    AutoReverse="True"
    RepeatBehavior="0:0:7.5" />
</Storyboard>
```

The total animation lasts 7.5 seconds. The *FontSize* increases from 1 to 144 over the course of three seconds, decreases from 144 to 1 in another three seconds, and then starts to increase again but stops. The final *FontSize* value is 73.5.

You can also set *RepeatBehavior* to *Forever*:

```
<Storyboard x:Key="storyboard">
  <DoubleAnimation Storyboard.TargetName="txtblk"
    Storyboard.TargetProperty="FontSize"
    EnableDependentAnimation="True"
```

```

        From="1" To="144" Duration="0:0:3"
        AutoReverse="True"
        RepeatBehavior="Forever" />
</Storyboard>

```

And it does exactly that (or at least until you get bored and terminate the program).

You can delay the start of an animation with the *BeginTime* property:

```

<Storyboard x:Key="storyboard">
    <DoubleAnimation Storyboard.TargetName="txtblk"
        Storyboard.TargetProperty="FontSize"
        EnableDependentAnimation="True"
        BeginTime="0:0:1.5"
        From="1" To="144" Duration="0:0:3" />
</Storyboard>

```

When you click the button, nothing will seem to happen for a second and a half, and then the *TextBlock* will jump to a 1-pixel size and start to expand. The animation concludes 4.5 seconds after the button click.

Even with all these variations, all the animations so far have been linear. The *FontSize* always increases or decreases linearly by a certain number of pixels per second. One easy way to create a nonlinear animation is by setting the *EasingFunction* property defined by *DoubleAnimation*. Break out the property as a property element, and specify one of the 11 classes that derive from *EasingFunctionBase*. Here's *ElasticEase*:

```

<Storyboard x:Key="storyboard">
    <DoubleAnimation Storyboard.TargetName="txtblk"
        Storyboard.TargetProperty="FontSize"
        EnableDependentAnimation="True"
        From="1" To="144" Duration="0:0:3">
        <DoubleAnimation.EasingFunction>
            <ElasticEase />
        </DoubleAnimation.EasingFunction>
    </DoubleAnimation>
</Storyboard>

```

You really need to try this out to see the effect. As the *TextBlock* gets larger, it actually goes *beyond* the 144-pixel size and then decreases to below 144 and back a forth a couple times, finally settling at the *To* value. (That behavior rather stretches the meaning of the word "ease"!)

EasingFunctionBase defines an *EasingMode* property that is inherited by all 11 derived classes. The default setting is the enumeration member *EasingMode.EaseOut*, which means that the animation begins linearly and the special effect is applied at the end of the animation. You can specify *EaseIn* to apply the effect to the beginning of the animation or *EaseInOut* to the beginning and the end.

Some *EasingFunctionBase* derivatives define their own properties for a little variation. *ElasticEase* defines an *Oscillations* property (an integer with a default value of 3 that indicates how many times the values swings back and forth) and a *Springiness* property, a *double* also with a default setting of 3. The lower the *Springiness* value, the more extreme the effect. Try this:

```

<Storyboard x:Key="storyboard">
  <DoubleAnimation Storyboard.TargetName="txtblk"
    Storyboard.TargetProperty="FontSize"
    EnableDependentAnimation="True"
    From="1" To="144" Duration="0:0:3">
    <DoubleAnimation.EasingFunction>
      <ElasticEase Oscillations="10"
        Springiness="0" />
    </DoubleAnimation.EasingFunction>
  </DoubleAnimation>
</Storyboard>

```

A program to explore the easing functions is coming up soon.

I mentioned earlier that an animation object such as *DoubleAnimation* must be a child of a *Storyboard*. Interestingly, *Storyboard* and *DoubleAnimation* are siblings in the class hierarchy:

```

Object
  DependencyObject
    Timeline
      Storyboard
      DoubleAnimation
      ...

```

Storyboard defines a *Children* property of type *TimelineCollection*, the attached properties *TargetName* and *TargetProperty*, as well as methods to pause and resume the animation. *DoubleAnimation* defines *From*, *To*, *By*, *EnableDependentAnimation*, and *EasingFunction*.

All the other properties you've seen so far—*AutoReverse*, *BeginTime*, *Duration*, *FillBehavior*, and *RepeatBehavior*—are defined by *Timeline*, which means that you can set these properties on *Storyboard* to define behavior for all the children of the *Storyboard*.

Timeline also defines a property named *SpeedRatio*:

```

<Storyboard x:Key="storyboard">
  <DoubleAnimation Storyboard.TargetName="txtblk"
    Storyboard.TargetProperty="FontSize"
    EnableDependentAnimation="True"
    SpeedRatio="10"
    From="1" To="144" Duration="0:0:3" />
</Storyboard>

```

This *SpeedRatio* setting causes the animation to go 10 times faster! Setting *SpeedRatio* on the *DoubleAnimation* is certainly allowed, but it's much more common to set it on a *Storyboard* so that it applies to all the animation children within that *Storyboard*. You can use *SpeedRatio* for fine-tuning the speed of an animation without changing all the individual *Duration* times or for debugging complex collections of animations. For example, set the *SpeedRatio* to 0.1 to slow down the animation so that you can better see what it's doing.

Timeline also defines a *Completed* event, which you can set on either a *Storyboard* or a

DoubleAnimation to be notified when an animation has completed.

It's also possible to define an animation entirely in code. The XAML file for the SimpleAnimationCode project has a *Grid* with nine *Button* elements sharing the same *Click* event handler. No *Storyboard* or *DoubleAnimation* appears in the XAML file:

Project: SimpleAnimationCode | File: MainPage.xaml (excerpt)

```
<Page ... >
  <Page.Resources>
    <Style TargetType="Button">
      <Setter Property="Content" Value="Trigger!" />
      <Setter Property="FontSize" Value="48" />
      <Setter Property="HorizontalAlignment" Value="Center" />
      <Setter Property="VerticalAlignment" Value="Center" />
      <Setter Property="Margin" Value="12" />
    </Style>
  </Page.Resources>

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}"
    <Grid HorizontalAlignment="Center"
      VerticalAlignment="Center">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition Width="Auto" />
      <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>

    <Button Grid.Row="0" Grid.Column="0" Click="OnButtonClick" />
    <Button Grid.Row="0" Grid.Column="1" Click="OnButtonClick" />
    <Button Grid.Row="0" Grid.Column="2" Click="OnButtonClick" />
    <Button Grid.Row="1" Grid.Column="0" Click="OnButtonClick" />
    <Button Grid.Row="1" Grid.Column="1" Click="OnButtonClick" />
    <Button Grid.Row="1" Grid.Column="2" Click="OnButtonClick" />
    <Button Grid.Row="2" Grid.Column="0" Click="OnButtonClick" />
    <Button Grid.Row="2" Grid.Column="1" Click="OnButtonClick" />
    <Button Grid.Row="2" Grid.Column="2" Click="OnButtonClick" />
  </Grid>
</Grid>
</Page>
```

In the code-behind file, you can create the *Storyboard* and *DoubleAnimation* once and reuse the objects whenever you need to trigger the animation, or you can create them anew as needed. The first approach only works when the animation target is always the same object. This program potentially needs nine independent animations for the nine buttons, so it's easiest just creating them on demand. Everything is in the *Click* handler:

Project: SimpleAnimationCode | File: MainPage.xaml.cs (excerpt)

```
void OnButtonClick(object sender, RoutedEventArgs args)
{
    DoubleAnimation anima = new DoubleAnimation
    {
        EnableDependentAnimation = true,
        To = 96,
        Duration = new Duration(new TimeSpan(0, 0, 1)),
        AutoReverse = true,
        RepeatBehavior = new RepeatBehavior(3)
    };
    Storyboard.SetTarget(anima, sender as Button);
    Storyboard.SetTargetProperty(anima, "FontSize");

    Storyboard storyboard = new Storyboard();
    storyboard.Children.Add(anima);
    storyboard.Begin();
}
```

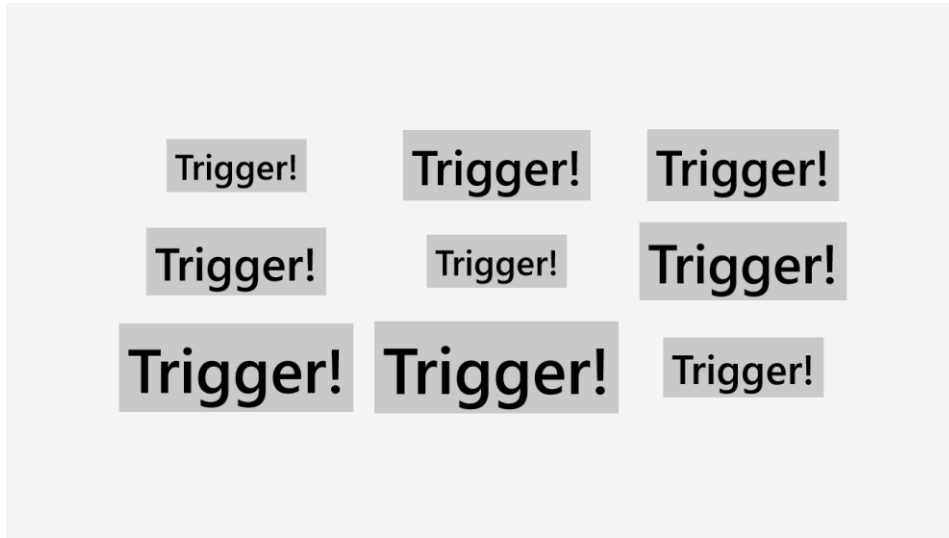
In the earlier XAML definition of *DoubleAnimation*, the attached properties *Storyboard.TargetName* and *Storyboard.TargetProperty* indicate the object and property to animate. In code, it's a little different: You use the static method *Storyboard.SetTargetProperty* to set the property name, but you use *Storyboard.SetTarget*—not *Storyboard.SetTargetName*—to set the target object rather than the XAML name of the target object. If the target object is a *TextBlock* in XAML with the name “txtblk,” the *SetTarget* call would look like this:

```
Storyboard.SetTarget(anima, txtblk);
```

It's the object variable name, not the text name. In this code example I've set the target object to the *Button* generating the *Click* event.

Also notice the how the *Duration* property is set. Using a *TimeSpan* is the most common approach, but *Duration* also has two static properties: *Automatic* (which means one second in this context) and *Forever* (which is deprecated because it makes the animation infinitely slow).

Because the change in each *FontSize* affects the size of each *Button*, the *Grid* needs to recalculate the width and height of its cells. It's fun to get all the animations going at once to watch how the *Grid* changes size:



Other Double Animations

A *DoubleAnimation* can animate any property of type *double* that's backed by a dependency property, for example, *Width* or *Height* (or both):

Project: EllipseBlobAnimation | File: MainPage.xaml (excerpt)

```
<Page ... >

    <Page.Resources>
        <Storyboard x:Key="storyboard"
            RepeatBehavior="Forever">
            <DoubleAnimation Storyboard.TargetName="ellipse"
                Storyboard.TargetProperty="Width"
                EnableDependentAnimation="True"
                From="100" To="600" Duration="0:0:1"
                AutoReverse="True" />

            <DoubleAnimation Storyboard.TargetName="ellipse"
                Storyboard.TargetProperty="Height"
                EnableDependentAnimation="True"
                From="600" To="100" Duration="0:0:1"
                AutoReverse="True" />

        </Storyboard>
    </Page.Resources>

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Ellipse Name="ellipse">
            <Ellipse.Fill>
                <LinearGradientBrush>
                    <GradientStop Offset="0" Color="Red" />
                </LinearGradientBrush>
            </Ellipse.Fill>
        </Ellipse>
    </Grid>
</Page>
```

```

        <GradientStop Offset="1" Color="Blue" />
    </LinearGradientBrush>
</Ellipse.Fill>
</Ellipse>
</Grid>
</Page>

```

The two animations run in parallel. The first animates the *Width* of the *Ellipse* from 100 to 600, and the second animates the *Height* of the *Ellipse* from 600 to 100. The two dimensions only briefly meet up in the middle to make a circle. Both animations have *AutoReverse* set, and the *Storyboard* has the *RepeatBehavior* set to *Forever*. In theory, the *AutoReverse* settings could also be moved to the *Storyboard*, but that didn't work with the release of Windows 8 I've been using for this chapter.

The animation is triggered when the page is loaded, and it runs "forever":

Project: EllipseBlobAnimation | File: MainPage.xaml.cs (excerpt)

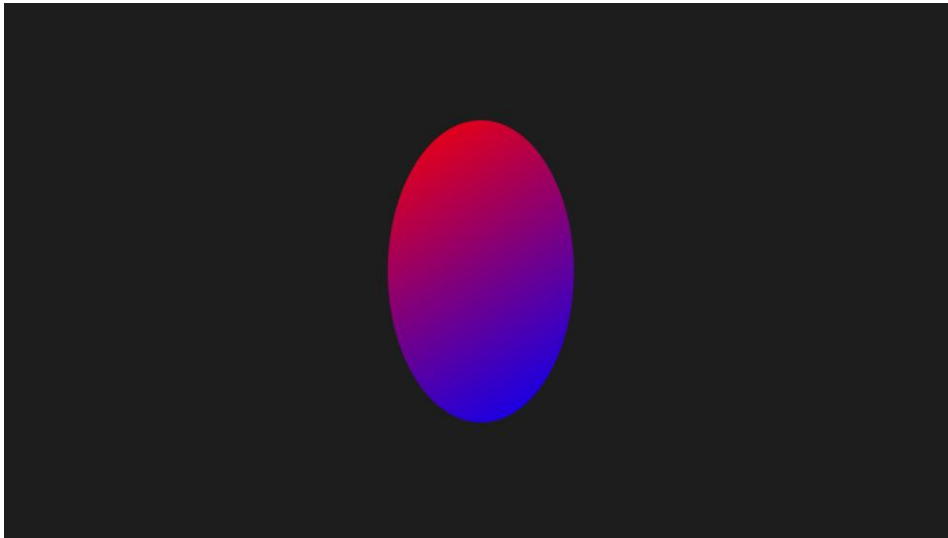
```

public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();

        Loaded += (sender, args) =>
        {
            (this.Resources["storyboard"] as Storyboard).Begin();
        };
    }
}

```

Because the *LinearGradientBrush* that colors the *Ellipse* has a default gradient from the upper-left corner of a bounding rectangle to the lower-right corner, the gradient actually shifts a bit during the animation:



Width and *Height* aren't the only properties of *Ellipse* that can be animated. The *StrokeThickness* property defined by *Shape* is also a double and is backed by a dependency property. Here's an *Ellipse* with a dotted line around its circumference, and the animation targets the *thickness* of that dotted line:

Project *AnimateStrokeThickness* | File: *MainPage.xaml* (excerpt)

```
<Page ... >
  <Page.Resources>
    <Storyboard x:Key="storyboard">
      <DoubleAnimation Storyboard.TargetName="ellipse"
        Storyboard.TargetProperty="StrokeThickness"
        EnableDependentAnimation="True"
        From="1" To="100" Duration="0:0:4"
        AutoReverse="True"
        RepeatBehavior="Forever" />
    </Storyboard>
  </Page.Resources>

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Ellipse Name="ellipse"
      Stroke="Red"
      StrokeDashCap="Round"
      StrokeDashArray="0 2" />
  </Grid>
</Page>
```

The animation is triggered during the *Loaded* event with the same code as the previous program.

The "0 2" value of the *StrokeDashArray* indicates that the dashed line consists of a dash that is zero units long followed by a gap 2 units long, where these units indicate multiples of the *StrokeThickness*. This dash has rounded ends benefit of the *StrokeDashCap* property, and the rounded ends add to the length of the dash, so the dash actually becomes a dot with a diameter equal to the *StrokeThickness*. The centers of these dots are separated by a gap equal to twice the *StrokeThickness*, so the dots

themselves are separated by the *StrokeThickness*.

In this animation, the number of dots actually decreases and then increases as the *StrokeThickness* is increased and decreased by the animation. The dots seem to disappear and reappear at the far right of the *Ellipse*:



Can you find another property of *Ellipse* of type *double*? How about *StrokeDashOffset*, which indicates where the dashes and gaps of a dotted line begin in a dashed line? Here's some XAML that uses a *Path* with Bézier curves to draw an infinity sign with a dotted lines and that then animates *StrokeDashOffset* to make the dots seem to travel around the figure:

Project: AnimateDashOffset | File: MainPage.xaml (excerpt)

```
<Page ... >
  <Page.Resources>
    <Storyboard x:Key="storyboard">
      <DoubleAnimation Storyboard.TargetName="path"
        Storyboard.TargetProperty="StrokeDashOffset"
        EnableDependentAnimation="True"
        From="0" To="1.5" Duration="0:0:1"
        RepeatBehavior="Forever" />
    </Storyboard>
  </Page.Resources>

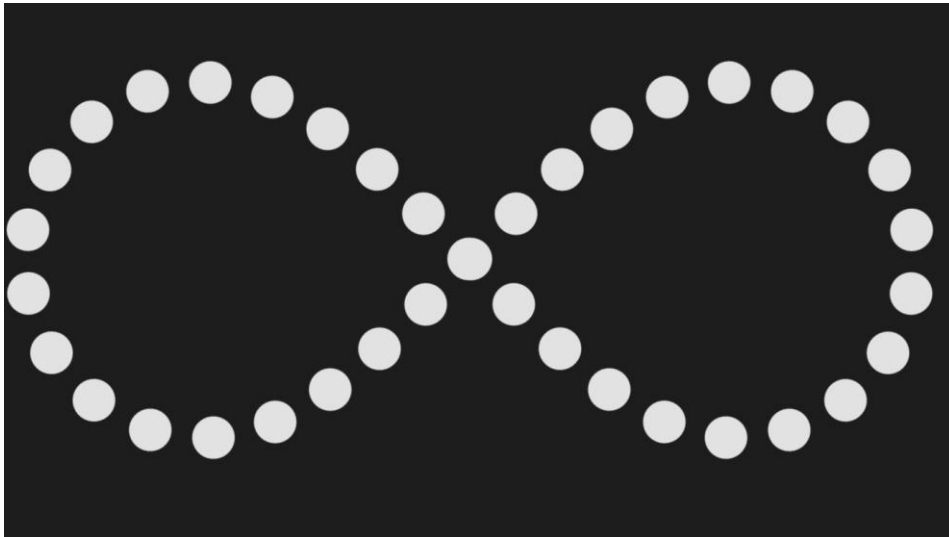
  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Viewbox>
      <Path Name="path"
        Margin="12"
        Stroke="{StaticResource ApplicationForegroundThemeBrush}"
        StrokeThickness="24"
        StrokeDashArray="0 1.5"
        StrokeDashCap="Round"
        Data="M 100 0
```

```

C 45 0, 0 45, 0 100
S 45 200, 100 200
S 200 150, 250 100
S 345 0, 400 0
S 500 45, 500 100
S 455 200, 400 200
S 300 150, 250 100
S 155 0, 100 0" />
</Viewbox>
</Grid>
</Page>

```

Unfortunately, I can't show the dots travelling around the infinity sign on the printed page:



The *Path* incorporates a well-known Bézier approximation to a quarter circle. For a circle centered at the point (0, 0), the lower-right quarter-circle arc begins at (100, 0) and ends at (0, 100). This can be approximated very well with a Bézier curve that also begins at (100, 0) and ends at (0, 100) with two control points (100, 55) and (55, 100). You can draw an entire circle using four of these "Bezier 55" arcs.

Thus, the quarter-circle arc that begins this infinity sign at the upper-left corner starts at (100, 0) and ends at (0, 100), but the center is (100, 100) rather than (0, 0), so the first control point is 55 units to the left of (100, 0), and the second is 55 units above (0, 100), or (45, 0), and (0, 45). The next Bézier should continue the figure around the lower-left corner starting at (0, 100)—the end of the previous Bézier—and ending at (100, 200) with control points (0, 155) and (45, 200). But the remainder of the path markup geometry continues not with figures indicated by C, which stands for "Cubic Bézier," but with S, which stands for "Smooth Bézier." It is well known that two connected Bézier curves will have a smooth connection if their common point and two adjacent control points are collinear (that is, lie on the same line). The S figure in path markup syntax causes the first control point to be automatically derived so that it is collinear with the start point and previous control point and the same distance from the start point as the previous collinear point. Thus, based on the point (0, 45) and (0, 100) in the

first Bézier curve, the first S figure derives the first control point to be (0, 155).

When drawing a dashed line whose end connects back with its beginning, it is very likely that there will be a discontinuity at the start point where only a partial dash will be displayed. The *StrokeThickness* of 24 was derived experimentally and need not necessarily be a whole number. For the Windows Phone version of this program, I settled upon a *StrokeThickness* of 23.98.

When exploring the rest of the *Shapes* library for properties of type double to animate, you'll also discover the *X1*, *Y1*, *X2*, and *Y2* properties of *Line*. Later in this chapter I'll demonstrate how to animate properties of type *Point* that show up in many of the *PathSegment* derivatives.

The *Opacity* property is a very common animation target, and it's used to fade elements in and out. You can set *Opacity* to a value ranging from 0 (transparent) to 1 (opaque). Here's an *Opacity* animation based on John Tenniel's illustrations of the Cheshire Cat for the original edition of Lewis Carroll's *Alice's Adventures in Wonderland* (1865):

Project: CheshireCat | File: MainPage.xaml (excerpt)

<Page ... >

```
<Page.Resources>
    <Storyboard x:Key="storyboard">
        <DoubleAnimation Storyboard.TargetName="image2"
            Storyboard.TargetProperty="Opacity"
            From="0" To="1" Duration="0:0:2"
            AutoReverse="True"
            RepeatBehavior="Forever" />
    </Storyboard>
</Page.Resources>

<!-- Images from Project Gutenberg Book #114
    http://www.gutenberg.org/ebooks/114
    John Tenniel's illustrations for Lewis Carroll's "Alice in Wonderland" -->

<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Viewbox>
        <Grid>
            <Image Source="Images/alice23a.gif"
                Width="640" />

            <TextBlock FontFamily="Century Schoolbook"
                FontSize="24"
                Foreground="Black"
                TextWrapping="Wrap"
                TextAlignment="Justify"
                Width="320"
                Margin="0 0 24 60"
                HorizontalAlignment="Right"
                VerticalAlignment="Bottom">
                &#x2003;&#x2003;"All right," said the Cat; and this
                time it vanished quite slowly, beginning with the end
                of the tail, and ending with the grin, which
                remained some time after the rest of it had gone.
            </TextBlock>
        </Grid>
    </Viewbox>
</Grid>
```

```

<LineBreak />
<LineBreak />
&#x2003;&#x2003;"Well! I've often seen a cat without a
grin," thought Alice; "but a grin without a cat! It's
the most curious thing I ever saw in all my life!"
</TextBlock>

<Image Name="image2"
Source="Images/alice24a.gif"
Stretch="None"
VerticalAlignment="Top">
<Image.Clip>
<RectangleGeometry Rect="320 70 320 240" />
</Image.Clip>
</Image>
</Grid>
</Viewbox>
</Grid>
</Page>

```

In the original edition of *Alice's Adventures in Wonderland*, the two images were both the width of the page, but the first image also extended to the full height of the page to show Alice standing by the tree. The images on the Project Gutenberg site, however, don't have the same width. The first image (alice23a.gif) is 342 × 480 pixels and the second (alice24a.gif) is 640 × 435. When I forced them to have the same rendered width, they seemed to line up very well considering that they're definitely two different drawings. Still, I decided to use a rectangular clipping area to restrict the second image to only the disappearing cat. The text that I added is not the same as that which appeared in this spot in the original edition.



The utility of *DoubleAnimation* increases enormously when you begin animating the classes that derive from *Transform*. This is a subject for the next chapter (Chapter 9, "Transforms"). You might

remember the *RainbowEight* program from Chapter 3 that animated the *Offset* property of 15 *GradientStop* objects in tandem. You can do a similar program using 15 *DoubleAnimation* objects, but in the next chapter I'll show you how to do it with one *DoubleAnimation* animating a *TranslateTransform* set on the *LinearGradientBrush*.

Animating Attached Properties

One of the simple uses of transforms that I'll explore in the next chapter involves moving an object around the screen. But you don't need transforms for that. You can put the object in a *Canvas* and animate the *Canvas.Left* and *Canvas.Top* attached properties. Animating attached properties requires a special syntax for *Storyboard.TargetProperty*, as shown here:

Project: AttachedPropertyAnimation | File: MainPage.xaml (excerpt)

```
<Page ... >

    <Page.Resources>
        <Storyboard x:Key="storyboard">
            <DoubleAnimation Storyboard.TargetName="ellipse"
                Storyboard.TargetProperty="(Canvas.Left)"
                From="0" Duration="0:0:2.51"
                AutoReverse="True"
                RepeatBehavior="Forever" />

            <DoubleAnimation Storyboard.TargetName="ellipse"
                Storyboard.TargetProperty="(Canvas.Top)"
                From="0" Duration="0:0:1.01"
                AutoReverse="True"
                RepeatBehavior="Forever" />
        </Storyboard>
    </Page.Resources>

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Canvas SizeChanged="OnCanvasSizeChanged"
            Margin="0 0 48 48">
            <Ellipse Name="ellipse"
                Width="48"
                Height="48"
                Fill="Red" />
        </Canvas>
    </Grid>
</Page>
```

The *Canvas.Left* and *Canvas.Top* attached properties are simply enclosed in parentheses. The target is an *Ellipse* colored red and hence easily recognizable as a ball.

Notice the absence of an *EnableDependentAnimation* setting. This indicates that these animations do not occur in the user interface thread. If you're unsure whether to use *EnableDependentAnimation*, try leaving it out. If the animation works, it's OK!

This *Storyboard* has two *DoubleAnimation* children that run in synchronization. Notice that each of these *DoubleAnimation* definitions has *AutoReverse* set to *True* and *RepeatBehavior* set to *Forever* and *Duration* values set to 1.01 seconds and 2.51 seconds, respectively. I chose prime numbers here (101 and 251) to avoid repetitive patterns. The two animations include *From* values but no *To* values. That happens in the code-behind file:

Project: AttachedPropertyAnimation | File: MainPage.xaml.cs (excerpt)

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

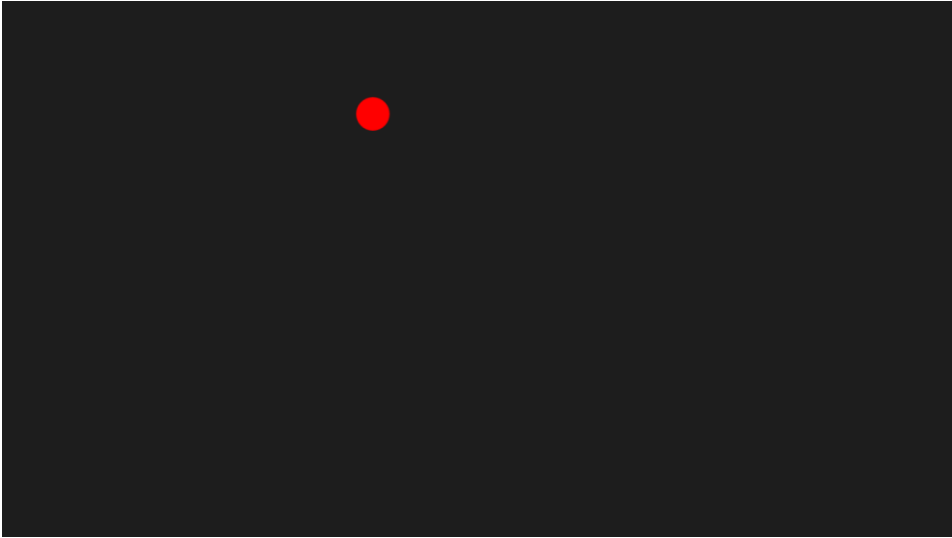
        Loaded += (sender, args) =>
        {
            (this.Resources["storyboard"] as Storyboard).Begin();
        };
    }

    void OnCanvasSizeChanged(object sender, SizeChangedEventArgs args)
    {
        Storyboard storyboard = this.Resources["storyboard"] as Storyboard;

        // Canvas.Left animation
        DoubleAnimation anima = storyboard.Children[0] as DoubleAnimation;
        anima.To = args.NewSize.Width;

        // Canvas.Top animation
        anima = storyboard.Children[1] as DoubleAnimation;
        anima.To = args.NewSize.Height;
    }
}
```

The storyboard is started in the *Loaded* event handler. Whenever the size of the *Canvas* changes (which happens when the size of the window changes), new *To* values are calculated based on the height and width of the *Canvas*, which has a *Margin* setting in the XAML file to compensate for the size of the *Ellipse*. You might assume that you wouldn't be allowed to change values of an ongoing animation, but it seems to work fine. The effect is a ball that seems to bounce between the edges of the screen:



Both *DoubleAnimation* definitions include the same *AutoReverse* and *RepeatBehavior* settings. As I mentioned earlier, these properties are defined by *Timeline*, which is also the parent class to *Storyboard*. Might these two settings be moved to the *Storyboard* tag? Try it:

```
<Storyboard x:Key="storyboard"
    AutoReverse="True"
    RepeatBehavior="Forever">
    <DoubleAnimation Storyboard.TargetName="ellipse"
        Storyboard.TargetProperty="(Canvas.Left)"
        From="0" Duration="0:0:2.51" />

    <DoubleAnimation Storyboard.TargetName="ellipse"
        Storyboard.TargetProperty="(Canvas.Top)"
        From="0" Duration="0:0:1.01" />
</Storyboard>
```

This is perfectly legal, but it doesn't work the same as the previous markup. The duration of a *Storyboard* is the duration of the longest animation child of that *Storyboard*, which in this case is 2.51 seconds. The animation begins by moving the ball both horizontally and vertically. But at the end of 1.01 seconds, the ball hits an edge. In landscape mode, this is the bottom edge. The animation of the *Canvas.Top* property has completed but the animation of *Canvas.Left* continues to move the ball horizontally for another 1.5 seconds. At that point the ball is in the lower-right corner of the screen. Both animations have now completed, so the *Storyboard* reverses the animation we've just seen until the ball is in the upper-left corner again. Then that same pattern repeats forever.

Only if all the animations in a *Storyboard* are the same length can the *AutoReverse* and *RepeatBehavior* properties be moved to the *Storyboard*.

The Easing Functions

Suppose a *DoubleAnimation* has a *From* value of 100, a *To* value of 500, and a *Duration* of five seconds. By default the *DoubleAnimation* is linear, which means that the target property takes on values from 100 through 500 based on a linear relationship with elapsed time:

Time	Value
0 sec	100
1	180
2	260
3	340
4	420
5	500

Or, perhaps more clearly:

$$Value = From + \frac{Time}{Duration} \times (To - From)$$

The purpose of the easing functions is to make this more interesting.

I was originally planning to begin this discussion by demonstrating how to derive from *EasingFunctionBase* to create a custom easing function, but for reasons that I'm sure are very good reasons, you cannot derive from *EasingFunctionBase*. If you were able to, you could create your own easing function by simply overriding the *Ease* method and implementing a transfer function. The *Ease* method has a *double* argument that ranges from 0 to 1. The method returns a *double* value. When the argument is 0, the method returns 0. When the argument is 1, the method returns 1. In between, anything goes. In this way, the easing function effectively bends time and the relationship between elapsed time and the animation value becomes nonlinear.

When an easing function is in effect, the elapsed time is normalized to a value between 0 and 1 by dividing by the *Duration* (just as in the formula above). The *Ease* function is called, and the return value is used to calculate a value:

$$Value = From + Ease\left(\frac{Time}{Duration}\right) \times (To - From)$$

For example, the *ExponentialEase* function with the default *EasingMode* setting of *EaseOut* has this transfer function:

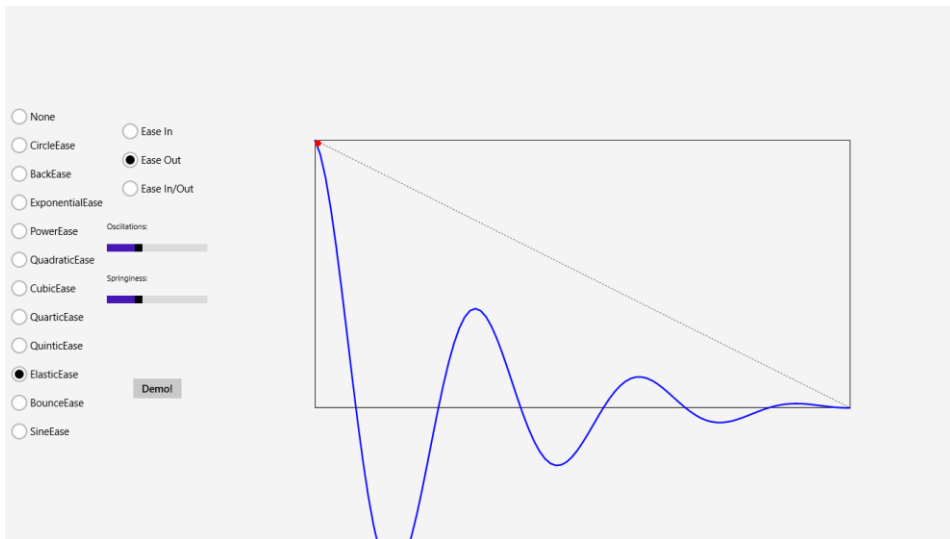
$$t' = \frac{1 - e^{-Nt}}{1 - e^{-N}}$$

where *t* is the argument to the *Ease* function, *t'* is the result, and *N* is the setting of the *Exponent* property. If *N* equals 2 (the default value), the animation shown in the table above is instead like this:

Time	t	t'	Value
0 sec	0.0	0.000	100
1	0.2	0.381	252
2	0.4	0.637	355
3	0.6	0.808	423
4	0.8	0.923	469
5	1.0	1.000	500

It's faster at the beginning and then seems to slow down.

The AnimationEaseGrapher program provides a visual representation of the easing functions and lets you experiment with them:



The graph is the transfer function with the horizontal axis representing t from 0 to 1, and the vertical axis representing t' with 0 at the top and 1 at the bottom. The dotted line from upper left to lower right is a linear transfer function, and the blue line is the selected transfer function. The points of that *Polyline* are assigned from the code-behind file by repeatedly calling the *Ease* method of the selected easing class. When you press the *Demo* button, the little red ball in the upper-left corner is animated horizontally with a regular linear animation and animated vertically with the selected ease function, and—amazingly enough—it follows the graph.

Here's the program's XAML file with the animation for the red ball defined at the top. The easing function for this animation is assigned from the code-behind file. The *To* and *From* values are adjusted based on the 6-pixel radius of the ball (which appears way down at the bottom):

Project: AnimationEaseGrapher | File: MainPage.xaml (excerpt)

```
<Page ... >
    <Page.Resources>
        <Storyboard x:Key="storyboard"
                    FillBehavior="Stop">
            <DoubleAnimation Storyboard.TargetName="redBall"
```

```

        Storyboard.TargetProperty="(Canvas.Left)"
        From="-6" To="994" Duration="0:0:3" />

        <DoubleAnimation x:Name="anima2"
            Storyboard.TargetName="redBall"
            Storyboard.TargetProperty="(Canvas.Top)"
            From="-6" To="494" Duration="0:0:3" />
    </Storyboard>
</Page.Resources>

<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <!-- Control panel -->
    <Grid Grid.Column="0"
        VerticalAlignment="Center">
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto" />
            <ColumnDefinition Width="Auto" />
        </Grid.ColumnDefinitions>

        <!-- Easing function (populated by code) -->
        <StackPanel Name="easingFunctionStackPanel"
            Grid.Row="0"
            Grid.RowSpan="3"
            Grid.Column="0"
            VerticalAlignment="Center">
            <RadioButton Content="None"
                Margin="6"
                Checked="OnEasingFunctionRadioButtonChecked" />
        </StackPanel>

        <!-- Easing mode -->
        <StackPanel Name="easingModeStackPanel"
            Grid.Row="1"
            Grid.Column="1"
            HorizontalAlignment="Center"
            VerticalAlignment="Center">
            <RadioButton Content="Ease In"
                Margin="6"
                Checked="OnEasingModeRadioButtonChecked">
            <RadioButton.Tag>
                <EasingMode>EaseIn</EasingMode>
            </RadioButton.Tag>
        </StackPanel>
    </Grid>

```

```

        <RadioButton Content="Ease Out"
                    Margin="6"
                    Checked="OnEasingModeRadioButtonChecked">
            <RadioButton.Tag>
                <EasingMode>EaseOut</EasingMode>
            </RadioButton.Tag>
        </RadioButton>

        <RadioButton Content="Ease In/Out"
                    Margin="6"
                    Checked="OnEasingModeRadioButtonChecked">
            <RadioButton.Tag>
                <EasingMode>EaseInOut</EasingMode>
            </RadioButton.Tag>
        </RadioButton>
    </StackPanel>

    <!-- Easing properties (populated by code) -->
    <StackPanel Name="propertiesStackPanel"
                Grid.Row="1"
                Grid.Column="1"
                HorizontalAlignment="Center"
                VerticalAlignment="Center" />

    <!-- Demo button -->
    <Button Grid.Row="2"
            Grid.Column="1"
            Content="Demo!"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            Click="OnDemoButtonClick" />
</Grid>

<!-- Graph using arbitrary coordinates and scaled to window -->
<Viewbox Grid.Column="1">
    <Grid Width="1000"
          Height="500"
          Margin="0 250 0 250">

        <!-- Rectangle outline -->
        <Polygon Points="0 0, 1000 0, 1000 500, 0 500"
                 Stroke="{StaticResource ApplicationForegroundThemeBrush}"
                 StrokeThickness="3" />

        <Canvas>
            <!-- Linear transfer -->
            <Polyline Points="0 0, 1000 500"
                     Stroke="{StaticResource ApplicationForegroundThemeBrush}"
                     StrokeThickness="1"
                     StrokeDashArray="3 3" />

            <!-- Points set by code based on easing function -->
            <Polyline Name="polyline"

```

```

        Stroke="Blue"
        StrokeThickness="3" />

        <!-- Animated ball -->
        <Ellipse Name="redBall"
            Width="12"
            Height="12"
            Fill="Red" />
    </Canvas>
</Grid>
</Viewbox>
</Grid>
</Page>

```

The code-behind file uses reflection to obtain all the classes that derive from *EasingFunctionBase* and creates a *RadioButton* element for each one. When one is selected, reflection also comes to the rescue to obtain a parameterless constructor and instantiate the class and to obtain all the public properties it can define on its own. Fortunately, all the public properties defined by the *EasingFunctionBase* derivatives are restricted to *int* or *double* types, so a *Slider* control is created for each.

Project: AnimationEaseGrapher | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    EasingFunctionBase easingFunction;

    public MainPage()
    {
        this.InitializeComponent();
        Loaded += OnMainPageLoaded;
    }

    void OnMainPageLoaded(object sender, RoutedEventArgs args)
    {
        Type baseType = typeof(EasingFunctionBase);
        TypeInfo baseTypeInfo = baseType.GetTypeInfo();
        Assembly assembly = baseTypeInfo.Assembly;

        // Enumerate through all Windows Runtime types
        foreach (Type type in assembly.ExportedTypes)
        {
            TypeInfo typeInfo = type.GetTypeInfo();

            // Create RadioButton for each easing function
            if (typeInfo.IsPublic &&
                baseTypeInfo.IsAssignableFrom(typeInfo) &&
                type != baseType)
            {
                RadioButton radioButton = new RadioButton
                {
                    Content = type.Name,
                    Tag = type,
                    Margin = new Thickness(6),

```



```

    };
    radioButton.Checked += OnEasingFunctionRadioButtonChecked;
    easingFunctionStackPanel.Children.Add(radioButton);
}
}

// Check the first RadioButton in the StackPanel (the one labeled "None")
(easingFunctionStackPanel.Children[0] as RadioButton).IsChecked = true;
}

void OnEasingFunctionRadioButtonChecked(object sender, RoutedEventArgs args)
{
    RadioButton radioButton = sender as RadioButton;
    Type type = radioButton.Tag as Type;
    easingFunction = null;
    propertiesStackPanel.Children.Clear();

    // type is only null for "None" button
    if (type != null)
    {
        TypeInfo typeInfo = type.GetTypeInfo();

        // Find a parameterless constructor and instantiate the easing function
        foreach (ConstructorInfo constructorInfo in typeInfo.DeclaredConstructors)
        {
            if (constructorInfo.IsPublic && constructorInfo.GetParameters().Length == 0)
            {
                easingFunction = constructorInfo.Invoke(null) as EasingFunctionBase;
                break;
            }
        }

        // Enumerate the easing function properties
        foreach (PropertyInfo property in typeInfo.DeclaredProperties)
        {
            // We can only deal with properties of type int and double
            if (property.PropertyType != typeof(int) &&
                property.PropertyType != typeof(double))
            {
                continue;
            }

            // Create a TextBlock for the property name
            TextBlock txtblk = new TextBlock
            {
                Text = property.Name + ":"
            };
            propertiesStackPanel.Children.Add(txtblk);

            // Create a Slider for the property value
            Slider slider = new Slider
            {
                Width = 144,

```

```

        Minimum = 0,
        Maximum = 10,
        Tag = property
    };

    if (property.PropertyType == typeof(int))
    {
        slider.StepFrequency = 1;
        slider.Value = (int)property.GetValue(easingFunction);
    }
    else
    {
        slider.StepFrequency = 0.1;
        slider.Value = (double)property.GetValue(easingFunction);
    }

    // Define the Slider event handler right here
    slider.ValueChanged += (sliderSender, sliderArgs) =>
    {
        Slider sliderChanging = sliderSender as Slider;
        PropertyInfo propertyInfo = sliderChanging.Tag as PropertyInfo;

        if (property.PropertyType == typeof(int))
            property.SetValue(easingFunction, (int)sliderArgs.NewValue);
        else
            property.SetValue(easingFunction, (double)sliderArgs.NewValue);

        DrawNewGraph();
    };
    propertiesStackPanel.Children.Add(slider);
}

// Initialize EasingMode radio buttons
foreach (UIElement child in easingModeStackPanel.Children)
{
    RadioButton easingModeRadioButton = child as RadioButton;
    easingModeRadioButton.IsEnabled = easingFunction != null;

    easingModeRadioButton.IsChecked =
        easingFunction != null &&
        easingFunction.EasingMode == (EasingMode)easingModeRadioButton.Tag;
}

DrawNewGraph();
}

void OnEasingModeRadioButtonChecked(object sender, RoutedEventArgs args)
{
    RadioButton radioButton = sender as RadioButton;
    easingFunction.EasingMode = (EasingMode)radioButton.Tag;
    DrawNewGraph();
}

```

```

void OnDemoButtonClick(object sender, RoutedEventArgs args)
{
    // Set the selected easing function and start the animation
    Storyboard storyboard = this.Resources["storyboard"] as Storyboard;
    (storyboard.Children[1] as DoubleAnimation).EasingFunction = easingFunction;
    storyboard.Begin();
}

void DrawNewGraph()
{
    polyline.Points.Clear();

    if (easingFunction == null)
    {
        polyline.Points.Add(new Point(0, 0));
        polyline.Points.Add(new Point(1000, 500));
        return;
    }

    for (decimal t = 0; t <= 1; t += 0.01m)
    {
        double x = (double)(1000 * t);
        double y = 500 * easingFunction.Ease((double)t);
        polyline.Points.Add(new Point(x, y));
    }
}

```

There is some redundancy in these easing functions: The *QuadraticEase*, *CubicEase*, *QuarticEase*, and *QuinticEase* are all special cases of the *PowerEase* class, and they can be duplicated with *PowerEase* by setting the *Power* property to 2, 3, 4, and 5, respectively.

The above screenshot (a few pages ago) with *ElasticEase* shows that this particular *Ease* function returns values outside the range of 0 and 1. The same is true with *BackEase*. Because the transfer function possibly returns values less than 0 or greater than 1, the animation could take on values outside the range of its *From* and *To* settings.

For many properties, this is no problem. But for some properties an exception could be raised. *Opacity*, for example, can't be set to values less than 0 or greater than 1. *Width* and *Height* can't be set to negative values, and *FontSize* must be greater than 0.

Although the easing functions usually cause animations to slow down and speed up in various ways, it's possible to use the easing functions in somewhat unorthodox ways. For example, *SineEase* has this transfer function when *EasingMode* is set to the default value of *EaseOut*:

$$t' = \sin\left(\frac{\pi}{2}t\right)$$

It's the first quarter of a sine curve, starting fast and slowing down. For *EaseIn*, it's the first quarter of a cosine curve but flipped around to go from 0 to 1:

$$t' = 1 - \cos\left(\frac{\pi}{2}t\right)$$

It starts slow and speeds up.

SineEase with an *EasingMode* setting of *EaseInOut* is the first half of a cosine curve, adjusted to go from 0 to 1:

$$t' = \frac{1 - \cos(\pi t)}{2}$$

It starts slow, speeds up, and then slows down again. If you were to use the *EaseInOut* variation of *SineEase* with a *DoubleAnimation* applied to the *Canvas.Left* property of an *Ellipse* and you set *AutoReverse* equal to *True* and *RepeatBehavior* to *Forever*, you would get motion that resembles a pendulum: slow right before and after the motion reverses, but faster in the middle.

If you apply a similar animation to *Canvas.Top* but offset by half a cycle, you can move an object around in a circle, as the following program demonstrates:

Project: CircleAnimation | File: MainPage.xaml (excerpt)

```
<Page ... >
  <Page.Resources>
    <Storyboard x:Key="storyboard" SpeedRatio="3">
      <DoubleAnimation Storyboard.TargetName="ball"
        Storyboard.TargetProperty="(Canvas.Left)"
        From="-350" To="350" Duration="0:0:2"
        AutoReverse="True"
        RepeatBehavior="Forever">
        <DoubleAnimation.EasingFunction>
          <SineEase EasingMode="EaseInOut" />
        </DoubleAnimation.EasingFunction>
      </DoubleAnimation>

      <DoubleAnimation Storyboard.TargetName="ball"
        Storyboard.TargetProperty="(Canvas.Top)"
        BeginTime="0:0:1"
        From="-350" To="350" Duration="0:0:2"
        AutoReverse="True"
        RepeatBehavior="Forever">
        <DoubleAnimation.EasingFunction>
          <SineEase EasingMode="EaseInOut" />
        </DoubleAnimation.EasingFunction>
      </DoubleAnimation>
    </Storyboard>
  </Page.Resources>

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Canvas HorizontalAlignment="Center"
      VerticalAlignment="Center"
      Margin="0 0 48 48">
      <Ellipse Name="ball"
        Width="48"
        Height="48"
        Fill="Red" />
    </Canvas>
  </Grid>
</Page>
```

```

        </Canvas>
    </Grid>
</Page>

```

The *Canvas* is aligned in the center but offset by the size of the ellipse, which means that the point (0, 0) relative to the *Canvas* is 24 pixels to the left and 24 pixels above the center of the window. The *Ellipse* has default zero values of *Canvas.Left* and *Canvas.Top* and sits in the center. The animations move that *Ellipse* 350 pixels to the left and right and up and down.

Notice that the second animation has a *BeginTime* of one second, so for the first second after the program is loaded, the first animation moves the ellipse horizontally from –350 pixels to 0, and then the second animation kicks in and begins moving the ball vertically from –350 to 0 as its moving horizontally from 0 to 350. Although the easing functions are intended to slow down and speed up animations, the *Ellipse* has a constant angular velocity as it travels around in a circle.

I'll show you a more direct way of implementing revolution by using a *RotateTransform* in the next chapter.

All-XAML Animations

Several of the programs shown so far in this chapter have triggered the *Storyboard* in the handler for the page's *Loaded* event. This is handy when you need to start an animation when a program or page is loaded or for "demo" animations that simply run forever.

Triggering an animation in the *Loaded* event can actually be done entirely in XAML using a legacy property named *Triggers* inherited from the Windows Presentation Foundation. In the long journey from WPF to the Windows Runtime, the *Triggers* property has lost virtually all its earlier functionality, but it can still trigger a storyboard:

```

<Page.Triggers>
    <EventTrigger>
        <BeginStoryboard>
            <Storyboard ... >
                ...
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</Page.Triggers>

```

The *Triggers* property element usually appears on the root element of a XAML file, traditionally towards the bottom of the file, but you can actually define the *Triggers* property element on any ancestor element of the animation target.

Notice *EventTrigger* and *BeginStoryboard*. This is the only context in which you'll see those tags. *EventTrigger* has a *RoutedEvent* property, but if you try setting it to anything (including the reasonable "Loaded" or "Page.Loaded"), you'll generate a run-time error. *BeginStoryboard* can have multiple *Storyboard* children.

Here's a program that's similar to the *ManualColorAnimation* of Chapter 3. The background of the *Grid* and the *Foreground* of a *TextBlock* are animated from black to white in different directions. The two *ColorAnimation* objects target the *Color* properties of two *SolidColorBrush* objects:

Project: ForeverColorAnimation | File: MainPage.xaml (excerpt)

```
<Page ... >
  <Grid>
    <Grid.Background>
      <SolidColorBrush x:Name="gridBrush" />
    </Grid.Background>

    <TextBlock Text="Color Animation"
      FontFamily="Times New Roman"
      FontSize="96"
      FontWeight="Bold"
      HorizontalAlignment="Center"
      VerticalAlignment="Center">
      <TextBlock.Foreground>
        <SolidColorBrush x:Name="txtblkBrush" />
      </TextBlock.Foreground>
    </TextBlock>
  </Grid>

  <Page.Triggers>
    <EventTrigger>
      <BeginStoryboard>
        <Storyboard RepeatBehavior="Forever">
          <ColorAnimation Storyboard.TargetName="gridBrush"
            Storyboard.TargetProperty="Color"
            From="Black" To="White" Duration="0:0:2"
            AutoReverse="True" />

          <ColorAnimation Storyboard.TargetName="txtblkBrush"
            Storyboard.TargetProperty="Color"
            From="White" To="Black" Duration="0:0:2"
            AutoReverse="True" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Page.Triggers>
</Page>
```

ColorAnimation is perhaps the second most common animation class after *DoubleAnimation*. It's pretty much limited to targeting the *Color* property of *SolidColorBrush* and *GradientStop*, but these brushes show up frequently, so it's more versatile than it might seem.

Notice the *RepeatBehavior* setting in the *Storyboard*. Oddly, also moving the *AutoReverse* setting there caused the animation to stop after two seconds.

The code-behind file contains nothing but an *InitializeComponent* call in the page's constructor. What this means is that you can copy the XAML file into the editor in the XamlCruncher program presented in Chapter 7, "Building an Application," remove the *x:Class* attribute, and run the animation

without help from any code. XamlCruncher (or another XAML-editing program) is a fine way to experiment with animations.

It's also possible to animate properties of type *Point*. Properties of type *Point* aren't very common, but *EllipseGeometry* has a *Center* property of type *Point*. If you create a circle or ellipse using *Path* and *EllipseGeometry* rather than the *Ellipse* class, you can move it around the screen by animating the *Center* property. Unlike animating *Canvas.Left* and *Canvas.Top*, this *Path* doesn't need to be in a *Canvas*, and the position of the figure is specified relative to its center rather than the upper-left corner.

However, you can't animate the *X* and *Y* properties of a *Point* value separately. *Point* is a structure rather than a class, which means it doesn't derive from *DependencyObject*, which means that the *X* and *Y* properties aren't backed by dependency properties.

Properties of type *Point* also show up in some *PathSegment* derivatives: *ArcSegment*, *BezierSegment*, *LineSegment*, and *QuadraticBezierSegment* all have properties of type *Point*. Animating these *Point* properties allows you to dynamically alter graphical figures. Here's a program that uses the Bézier approximation to a circle I discussed earlier but then animates all 13 points so that the circle deforms into a square. Just to demonstrate that it works, I've defined the *Triggers* property element with the animation right on the *Path*:

Project: SquaringTheCircle | File: MainPage.xaml (excerpt)

```
<Page ... >
    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Canvas HorizontalAlignment="Center"
                VerticalAlignment="Center">
            <Path Fill="{StaticResource ApplicationPressedForegroundThemeBrush}"
                  Stroke="{StaticResource ApplicationForegroundThemeBrush}"
                  StrokeThickness="3" >
                <Path.Data>
                    <PathGeometry>
                        <PathFigure x:Name="bezier1" IsClosed="True">
                            <BezierSegment x:Name="bezier2" />
                            <BezierSegment x:Name="bezier3" />
                            <BezierSegment x:Name="bezier4" />
                            <BezierSegment x:Name="bezier5" />
                        </PathFigure>
                    </PathGeometry>
                </Path.Data>

                <Path.Triggers>
                    <EventTrigger>
                        <BeginStoryboard>
                            <Storyboard RepeatBehavior="Forever">
                                <PointAnimation Storyboard.TargetName="bezier1"
                                                  Storyboard.TargetProperty="StartPoint"
                                                  EnableDependentAnimation="True"
                                                  From="0 200" To="0 250"
                                                  AutoReverse="True" />
                            </Storyboard>
                        </BeginStoryboard>
                    </EventTrigger>
                </Path.Triggers>
            </Path>
        </Canvas>
    </Grid>
</Page>
```

```

<PointAnimation Storyboard.TargetName="bezier2"
Storyboard.TargetProperty="Point1"
EnableDependentAnimation="True"
From="110 200" To="125 125"
AutoReverse="True" />

<PointAnimation Storyboard.TargetName="bezier2"
Storyboard.TargetProperty="Point2"
EnableDependentAnimation="True"
From="200 110" To="125 125"
AutoReverse="True" />

<PointAnimation Storyboard.TargetName="bezier2"
Storyboard.TargetProperty="Point3"
EnableDependentAnimation="True"
From="200 0" To="250 0"
AutoReverse="True" />

<PointAnimation Storyboard.TargetName="bezier3"
Storyboard.TargetProperty="Point1"
EnableDependentAnimation="True"
From="200 -110" To="125 -125"
AutoReverse="True" />

<PointAnimation Storyboard.TargetName="bezier3"
Storyboard.TargetProperty="Point2"
EnableDependentAnimation="True"
From="110 -200" To="125 -125"
AutoReverse="True" />

<PointAnimation Storyboard.TargetName="bezier3"
Storyboard.TargetProperty="Point3"
EnableDependentAnimation="True"
From="0 -200" To="0 -250"
AutoReverse="True" />

<PointAnimation Storyboard.TargetName="bezier4"
Storyboard.TargetProperty="Point1"
EnableDependentAnimation="True"
From="-110 -200" To="-125 -125"
AutoReverse="True" />

<PointAnimation Storyboard.TargetName="bezier4"
Storyboard.TargetProperty="Point2"
EnableDependentAnimation="True"
From="-200 -110" To="-125 -125"
AutoReverse="True" />

<PointAnimation Storyboard.TargetName="bezier4"
Storyboard.TargetProperty="Point3"
EnableDependentAnimation="True"
From="-200 0" To="-250 0"
AutoReverse="True" />

```



```

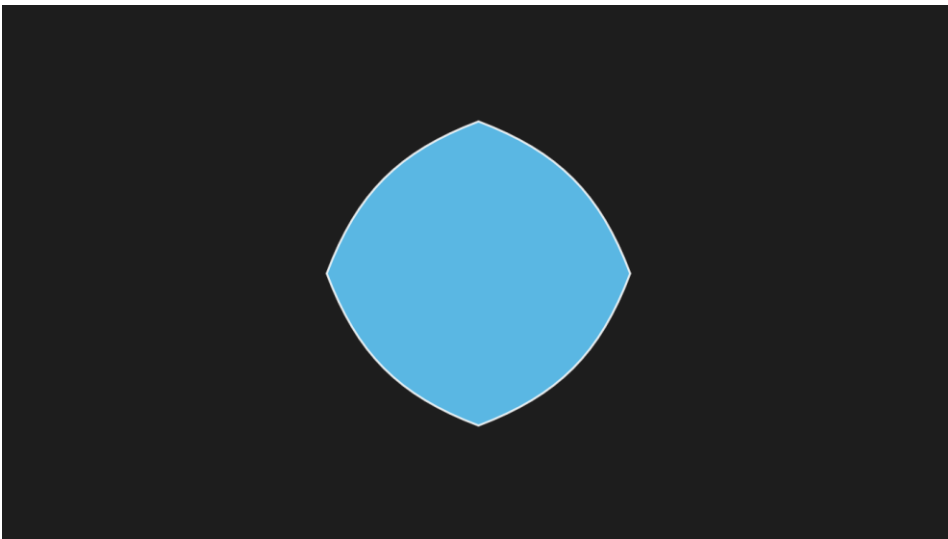
        <PointAnimation Storyboard.TargetName="bezier5"
            Storyboard.TargetProperty="Point1"
            EnableDependentAnimation="True"
            From="-200 110" To="-125 125"
            AutoReverse="True" />

        <PointAnimation Storyboard.TargetName="bezier5"
            Storyboard.TargetProperty="Point2"
            EnableDependentAnimation="True"
            From="-110 200" To="-125 125"
            AutoReverse="True" />

        <PointAnimation Storyboard.TargetName="bezier5"
            Storyboard.TargetProperty="Point3"
            EnableDependentAnimation="True"
            From="0 200" To="0 250"
            AutoReverse="True" />
    </Storyboard>
</BeginStoryboard>
</EventTrigger>
</Path.Triggers>
</Path>
</Canvas>
</Grid>
</Page>

```

Here's the figure halfway between a square and a circle:



Animating Custom Classes

Yes, you can animate properties of custom classes. But the animatable properties must be backed by

dependency properties.

Here's a class named *PieSlice* that derives from *Path* to render a pie slice such as used in pie charts. The custom properties are *Center*, *Radius*, *StartAngle* (in degrees, measured clockwise from 12:00), and *SweepAngle* (in degrees, measured clockwise from *StartAngle*):

Project: AnimatedPieSlice | File: PieSlice.cs

```
using System;
using Windows.Foundation;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Shapes;

namespace AnimatedPieSlice
{
    public class PieSlice : Path
    {
        PathFigure pathFigure;
        LineSegment lineSegment;
        ArcSegment arcSegment;

        public static DependencyProperty CenterProperty { private set; get; }

        public static DependencyProperty RadiusProperty { private set; get; }

        public static DependencyProperty StartAngleProperty { private set; get; }

        public static DependencyProperty SweepAngleProperty { private set; get; }

        static PieSlice()
        {
            CenterProperty = DependencyProperty.Register("Center",
                typeof(Point), typeof(PieSlice),
                new PropertyMetadata(new Point(100, 100), OnPropertyChanged));

            RadiusProperty = DependencyProperty.Register("Radius",
                typeof(double), typeof(PieSlice),
                new PropertyMetadata(100.0, OnPropertyChanged));

            StartAngleProperty = DependencyProperty.Register("StartAngle",
                typeof(double), typeof(PieSlice),
                new PropertyMetadata(0.0, OnPropertyChanged));

            SweepAngleProperty = DependencyProperty.Register("SweepAngle",
                typeof(double), typeof(PieSlice),
                new PropertyMetadata(90.0, OnPropertyChanged));
        }

        public PieSlice()
        {
            pathFigure = new PathFigure { IsClosed = true };
            lineSegment = new LineSegment();
            arcSegment = new ArcSegment { SweepDirection = SweepDirection.Clockwise };
        }
    }
}
```

```

        pathFigure.Segments.Add(lineSegment);
        pathFigure.Segments.Add(arcSegment);

        PathGeometry pathGeometry = new PathGeometry();
        pathGeometry.Figures.Add(pathFigure);

        this.Data = pathGeometry;
        UpdateValues();
    }

    public Point Center
    {
        set { SetValue(CenterProperty, value); }
        get { return (Point)GetValue(CenterProperty); }
    }

    public double Radius
    {
        set { SetValue(RadiusProperty, value); }
        get { return (double)GetValue(RadiusProperty); }
    }

    public double StartAngle
    {
        set { SetValue(StartAngleProperty, value); }
        get { return (double)GetValue(StartAngleProperty); }
    }

    public double SweepAngle
    {
        set { SetValue(SweepAngleProperty, value); }
        get { return (double)GetValue(SweepAngleProperty); }
    }

    static void OnPropertyChanged(DependencyObject obj,
        DependencyPropertyChangedEventArgs args)
    {
        (obj as PieSlice).UpdateValues();
    }

    void UpdateValues()
    {
        pathFigure.StartPoint = this.Center;

        double x = this.Center.X + this.Radius * Math.Sin(Math.PI * this.StartAngle / 180);
        double y = this.Center.Y - this.Radius * Math.Cos(Math.PI * this.StartAngle / 180);
        lineSegment.Point = new Point(x, y);

        x = this.Center.X + this.Radius * Math.Sin(Math.PI * (this.StartAngle +
            this.SweepAngle) / 180);

        y = this.Center.Y - this.Radius * Math.Cos(Math.PI * (this.StartAngle +
            this.SweepAngle) / 180);

        arcSegment.Point = new Point(x, y);
    }

```

```

        arcSegment.IsLargeArc = this.SweepAngle >= 180;

        arcSegment.Size = new Size(this.Radius, this.Radius);
    }
}
}

```

Just about everything in this class is overhead for the dependency properties except that *UpdateValues* method, and that method is critical. *UpdateValues* is called whenever any of the four properties change, and any of those four properties can be the target of an animation, which means that *UpdateValues* might be called 60 times per second for an indefinite period of time.

In methods such as these you should be careful about creating objects that require memory allocations on the heap. Creating new *double* and *Point* values are fine, because those are stored on the stack. But a not very good way to implement this method would be to create new *PathFigure*, *LineSegment*, and *ArcSegment* objects during every call because that generates a lot of activity allocating and then freeing memory.

The *PieSlice* class is part of the *AnimatedPieSlice* project, which includes a *MainPage.xaml* that instantiates, initializes, and animates it:

Project: *AnimatedPieSlice* | File: *MainPage.xaml* (excerpt)

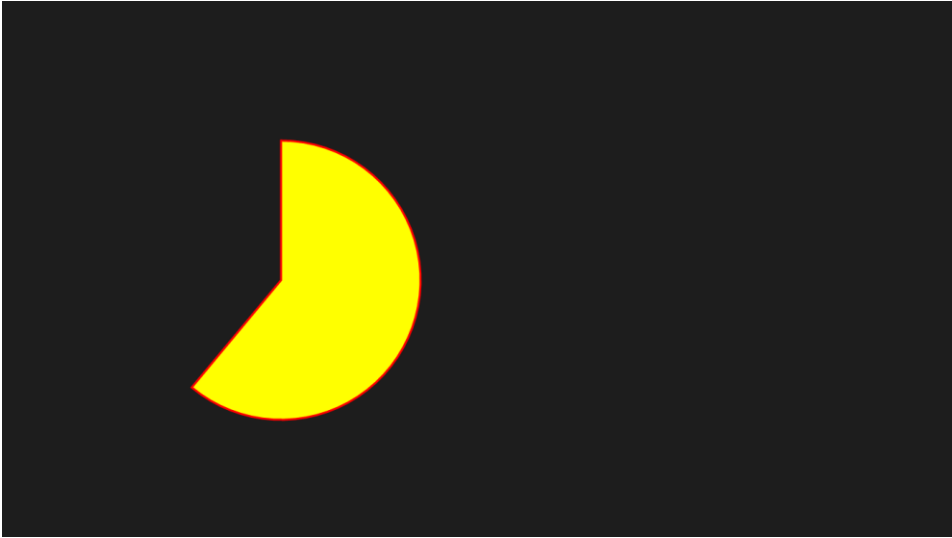
```

<Page ... >
    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <local:PieSlice x:Name="pieSlice"
            Center="400 400"
            Radius="200"
            Stroke="Red"
            StrokeThickness="3"
            Fill="Yellow" />
    </Grid>

    <Page.Triggers>
        <EventTrigger>
            <BeginStoryboard>
                <Storyboard>
                    <DoubleAnimation Storyboard.TargetName="pieSlice"
                        Storyboard.TargetProperty="SweepAngle"
                        EnableDependentAnimation="True"
                        From="1" To="359" Duration="0:0:3"
                        AutoReverse="True"
                        RepeatBehavior="Forever" />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Page.Triggers>
</Page>

```

The result is a pie slice that ranges from 1 degree to 359 degrees, back and forth forever:



Key Frame Animations

All the programs you've seen so far have animated properties from one value to another, usually specified as the *From* and *To* properties of *DoubleAnimation*, *ColorAnimation*, and *PointAnimation* classes, and the only variations have involved nonlinear ways to get from *From* to *To* and then reversing animations to go from *To* to *From*.

What if you need to animate a property from one value to another value and then to a third, and maybe even beyond? A solution that might occur to you is to define several animations in the storyboard targeting the same property and to use *BeginTime* to delay some of those animations so that they don't overlap. But that's illegal. You can't have more than one animation in a storyboard targeting a particular property.

The correct solution is a *key frame* animation, so called because you define the progress of the animation through a series of key frames. Each key frame indicates what the value of the property should be at a particular elapsed time and how to get from the previous key frame value to that key frame.

Here's a simple example of a key frame animation that targets the *Center* property of an *EllipseGeometry* to move the circle around the screen:

Project: SimpleKeyFrameAnimation | File: MainPage.xaml (excerpt)

```
<Page ... >
  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Path Fill="Blue">
      <Path.Data>
        <EllipseGeometry x:Name="ellipse"
```

```

        RadiusX="24"
        RadiusY="24" />
    </Path.Data>
</Path>
</Grid>

<Page.Triggers>
    <EventTrigger>
        <BeginStoryboard>
            <Storyboard>
                <PointAnimationUsingKeyFrames Storyboard.TargetName="ellipse"
                    Storyboard.TargetProperty="Center"
                    EnableDependentAnimation="True"
                    RepeatBehavior="Forever">
                    <DiscretePointKeyFrame KeyTime="0:0:0" Value="100 100" />
                    <LinearPointKeyFrame KeyTime="0:0:2" Value="700 700" />
                    <LinearPointKeyFrame KeyTime="0:0:2.1" Value="700 100" />
                    <LinearPointKeyFrame KeyTime="0:0:4.1" Value="100 700" />
                    <LinearPointKeyFrame KeyTime="0:0:4.2" Value="100 100" />
                </PointAnimationUsingKeyFrames>
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</Page.Triggers>
</Page>

```

Rather than a *PointAnimation*, the *Storyboard* contains a *PointAnimationUsingKeyFrames*. Rather than specifying *From*, *To*, and *Duration* properties in the *PointAnimation*, the *PointAnimationUsingKeyFrames* contains children of type *DiscretePointKeyFrame* and *LinearPointKeyFrame*.

Each key frame in a collection specifies what you want the value of the target property to be at that particular time from the beginning of the animation. Very often a collection of key frames will begin with a *Discrete* item with a *KeyTime* of zero, basically initializing the property to that value:

```
<DiscretePointKeyFrame KeyTime="0:0:0" Value="100 100" />
```

The next key frame in the collection is

```
<LinearPointKeyFrame KeyTime="0:0:2" Value="700 700" />
```

What this means is that the target property is linearly increased from the previous point (100, 100) to the point (700, 700) over the course of two seconds. At an elapsed time of two seconds, the value is (700, 700).

The next key frame specifies a much faster animation:

```
<LinearPointKeyFrame KeyTime="0:0:2.1" Value="700 100" />
```

From an elapsed time of two seconds to 2.1 seconds, the point changes from (700, 700) to (700, 100). The animation then slows up again for the next two seconds:

```
<LinearPointKeyFrame KeyTime="0:0:4.1" Value="100 700" />
```

The last key frame is:

```
<LinearPointKeyFrame KeyTime="0:0:4.2" Value="100 100" />
```

At an elapsed time of 4.2 seconds, the value of the target property is (100, 100) and the animation is finished. At this point, it can reverse (if *AutoReverse* is true) or start over again (if an appropriate value of *RepeatBehavior* is set).

It's possible to programmers to "overthink" key frames, so here are two extraordinary simple rules that might prevent confusion:

- A key frame always indicates the desired value of the property at that elapsed time.
- The duration of an animation is the highest key time in the collection.

For the collection of key frames, the *PointAnimationUsingKeyFrames* class defines a property named *KeyFrames* of type *PointKeyFrameCollection*, which is a collection of *PointKeyFrame* objects. *PointKeyFrame* defines the *KeyTime* and *Value* properties. Four classes derive from *PointKeyFrame*, and you've already seen two of them:

- *DiscretePointKeyFrame* jumps to a particular value.
- *LinearPointKeyFrame* performs a linear animation.
- *SplinePointKeyFrame* can speed up or slow down.
- *EasingPointKeyFrame* animates with an easing function.

Similarly, the Windows Runtime includes a *DoubleAnimationUsingKeyFrames* class, which has children of type *DoubleKeyFrame*, from which similar *Discrete*, *Linear*, *Spline*, and *Easing* classes derive, and *ColorAnimationUsingKeyFrames* with children of type *ColorKeyFrame*, also with *Discrete*, *Linear*, *Spline*, and *Easing* derivatives.

The following project uses *ColorAnimationUsingKeyFrames* to color the background of the grid with colors that animate through the rainbow:

Project: RainbowAnimation | File: MainPage.xaml (excerpt)

```
<Page ... >
  <Grid>
    <Grid.Background>
      <SolidColorBrush x:Name="brush" />
    </Grid.Background>
  </Grid>

  <Page.Triggers>
    <EventTrigger>
      <BeginStoryboard>
        <Storyboard RepeatBehavior="Forever">
          <ColorAnimationUsingKeyFrames Storyboard.TargetName="brush"
            Storyboard.TargetProperty="Color">
            <DiscreteColorKeyFrame KeyTime="0:0:0" Value="#FF0000" />
          </ColorAnimationUsingKeyFrames>
        </Storyboard>
      </EventTrigger>
    </Page.Triggers>
  </Page>
```

```

        <LinearColorKeyFrame KeyTime="0:0:1" Value="#FFFF00" />
        <LinearColorKeyFrame KeyTime="0:0:2" Value="#00FF00" />
        <LinearColorKeyFrame KeyTime="0:0:3" Value="#00FFFF" />
        <LinearColorKeyFrame KeyTime="0:0:4" Value="#0000FF" />
        <LinearColorKeyFrame KeyTime="0:0:5" Value="#FF00FF" />
        <LinearColorKeyFrame KeyTime="0:0:6" Value="#FF0000" />
    </ColorAnimationUsingKeyFrames>
</Storyboard>
</BeginStoryboard>
</EventTrigger>
</Page.Triggers>
</Page>

```

The animation is 6 seconds in length, and it ends up at the same value it started with, which means there won't be any discontinuities when it starts over again from the beginning.

Here's a pair of *PointAnimationUsingKeyFrames* objects that animate the *StartPoint* and *EndPoint* properties of a *LinearGradientBrush* object to make the gradient go around in circles:

Project: GradientBrushPointAnimation | File: MainPage.xaml (excerpt)

```

<Page ... >
    <Grid>
        <Grid.Background>
            <LinearGradientBrush x:Name="gradientBrush">
                <GradientStop Offset="0" Color="Red" />
                <GradientStop Offset="1" Color="Blue" />
            </LinearGradientBrush>
        </Grid.Background>
    </Grid>

    <Page.Triggers>
        <EventTrigger>
            <BeginStoryboard>
                <Storyboard RepeatBehavior="Forever">
                    <PointAnimationUsingKeyFrames Storyboard.TargetName="gradientBrush"
                        Storyboard.TargetProperty="StartPoint"
                        EnableDependentAnimation="True">
                        <LinearPointKeyFrame KeyTime="0:0:0" Value="0 0" />
                        <LinearPointKeyFrame KeyTime="0:0:1" Value="1 0" />
                        <LinearPointKeyFrame KeyTime="0:0:2" Value="1 1" />
                        <LinearPointKeyFrame KeyTime="0:0:3" Value="0 1" />
                        <LinearPointKeyFrame KeyTime="0:0:4" Value="0 0" />
                    </PointAnimationUsingKeyFrames>

                    <PointAnimationUsingKeyFrames Storyboard.TargetName="gradientBrush"
                        Storyboard.TargetProperty="EndPoint"
                        EnableDependentAnimation="True">
                        <LinearPointKeyFrame KeyTime="0:0:0" Value="1 1" />
                        <LinearPointKeyFrame KeyTime="0:0:1" Value="0 1" />
                        <LinearPointKeyFrame KeyTime="0:0:2" Value="0 0" />
                        <LinearPointKeyFrame KeyTime="0:0:3" Value="1 0" />
                        <LinearPointKeyFrame KeyTime="0:0:4" Value="1 1" />
                    </PointAnimationUsingKeyFrames>
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Page.Triggers>
</Page>

```



```

        </BeginStoryboard>
    </EventTrigger>
</Page.Triggers>
</Page>

```

The *SplineDoubleKeyFrame*, *SplineColorKeyFrame*, and *SplinePointKeyFrame* objects are not used as much as they once were because much of their functionality has been replaced by *EasingDoubleKeyFrame*, *EasingColorKeyFrame*, and *EasingPointKeyFrame*. With the Spline variations of the key frame, you use a *KeySpline* object to define two control points of a Bézier spline that begins at the point (0, 0) and ends at (1, 1). This spline performs the same role as an easing function in that it bends time and causes an animation to speed up and slow down. I'll have an example in the next chapter.

The *Object* Animation

The Windows Runtime animation system is also capable of animating properties of type *Object*, which implicitly seems to encompass everything, but there's a catch: There is no *ObjectAnimation* class with *From* and *To* properties. There is only an *ObjectAnimationUsingKeyFrames* class, and the only class that derives from *ObjectKeyFrame* is *DiscreteObjectKeyFrame*.

In other words, you can indeed define an animation to target a property of any type (as long as that property is backed by a dependency property), but you can use the animation only to set that property to discrete values.

In practice, object animations are used mostly for targeting properties of enumeration types or *Brush* types, which allows setting the property to a predefined brush resource. These are mostly used in control templates, as you'll see in Chapter 10.

But here's an example that moves an *Ellipse* around a screen while animating its *Visibility* property with the enumeration members *Visible* and *Collapsed* and its *Fill* property with predefined brushes. Because these animations cause the *Ellipse* to flicker on and off, and with different discrete colors, the project is called FastNotFluid:

Project: FastNotFluid | File: MainPage.xaml (excerpt)

```

<Page ... >

    <Grid Background="Gray">
        <Canvas SizeChanged="OnCanvasSizeChanged"
            Margin="0 0 96 96">
            <Ellipse Name="ellipse"
                Width="96"
                Height="96" />
        </Canvas>
    </Grid>

    <Page.Triggers>
        <EventTrigger>

```

```

<BeginStoryboard>
  <Storyboard>
    <DoubleAnimation x:Name="horzAnima"
      Storyboard.TargetName="ellipse"
      Storyboard.TargetProperty="(Canvas.Left)"
      From="0" Duration="0:0:2.51"
      AutoReverse="True"
      RepeatBehavior="Forever" />

    <DoubleAnimation x:Name="vertAnima"
      Storyboard.TargetName="ellipse"
      Storyboard.TargetProperty="(Canvas.Top)"
      From="0" Duration="0:0:1.01"
      AutoReverse="True"
      RepeatBehavior="Forever" />

    <ObjectAnimationUsingKeyFrames
      Storyboard.TargetName="ellipse"
      Storyboard.TargetProperty="Visibility"
      RepeatBehavior="Forever">
      <DiscreteObjectKeyFrame KeyTime="0:0:0" Value="Visible" />
      <DiscreteObjectKeyFrame KeyTime="0:0:0.2" Value="Collapsed" />
      <DiscreteObjectKeyFrame KeyTime="0:0:0.25" Value="Visible" />
      <DiscreteObjectKeyFrame KeyTime="0:0:0.3" Value="Collapsed" />
      <DiscreteObjectKeyFrame KeyTime="0:0:0.45" Value="Visible" />
    </ObjectAnimationUsingKeyFrames>

    <ObjectAnimationUsingKeyFrames
      Storyboard.TargetName="ellipse"
      Storyboard.TargetProperty="Fill"
      RepeatBehavior="Forever">
      <DiscreteObjectKeyFrame KeyTime="0:0:0"
        Value="{StaticResource ApplicationPageBackgroundThemeBrush}" />
      <DiscreteObjectKeyFrame KeyTime="0:0:0.2"
        Value="{StaticResource ApplicationForegroundThemeBrush}" />
      <DiscreteObjectKeyFrame KeyTime="0:0:0.4"
        Value="{StaticResource ApplicationPressedForegroundThemeBrush}" />
      <DiscreteObjectKeyFrame KeyTime="0:0:0.6"
        Value="{StaticResource ApplicationPageBackgroundThemeBrush}" />
    </ObjectAnimationUsingKeyFrames>
  </Storyboard>
</BeginStoryboard>
</EventTrigger>
</Page.Triggers>
</Page>

```

It is interesting that the *Value* property of the *DiscreteObjectKeyFrame* can be set directly to the name of an enumeration member or a *StaticResource* without causing confusion about the type.

Another advantage of defining the *Storyboard* and animations in a *Triggers* section is accessing the individual animations by name in the code-behind file:

Project: FastNotFluid | File: MainPage.xaml.cs (excerpt)

```
void OnCanvasSizeChanged(object sender, SizeChangedEventArgs args)
{
    horzAnima.To = args.NewSize.Width;
    vertAnima.To = args.NewSize.Height;
}
```

Predefined Animations and Transitions

I said at the outset that the *Windows.UI.Xaml.Media.Animation* contained 71 classes, but if you've been keeping count, you probably haven't reached that number yet.

Besides the classes I've mentioned so far, the namespace also includes 14 predefined animations that derive from *Timeline* with names that end with *ThemeAnimation*. These animations already have all their properties and target properties set and need only a target object that you set with a *TargetName* property. So that you can experiment with these predefined animations, I've created a program where twelve of these animations (excluding *SplitOpenThemeAnimation* and *SplitCloseThemeAnimation*, which don't quite fit in the scheme of this program) are associated with their own *Storyboard* objects where the *TargetName* is set to an element with the name of "button":

Project: PreconfiguredAnimations | MainPage.xaml (excerpt)

```
<Page ... >
  <Page.Resources>
    <Style TargetType="Button">
      <Setter Property="Margin" Value="0 6" />
    </Style>

    <Storyboard x:Key="fadeIn">
      <FadeInThemeAnimation TargetName="button" />
    </Storyboard>

    <Storyboard x:Key="fadeOut">
      <FadeOutThemeAnimation TargetName="button" />
    </Storyboard>

    <Storyboard x:Key="popIn">
      <PopInThemeAnimation TargetName="button" />
    </Storyboard>

    <Storyboard x:Key="popOut">
      <PopOutThemeAnimation TargetName="button" />
    </Storyboard>

    <Storyboard x:Key="reposition">
      <RepositionThemeAnimation TargetName="button" />
    </Storyboard>

    <Storyboard x:Key="pointerUp">
      <PointerUpThemeAnimation TargetName="button" />
    </Storyboard>
```

```

</Storyboard>

<Storyboard x:Key="pointerDown">
    <PointerDownThemeAnimation TargetName="button" />
</Storyboard>

<Storyboard x:Key="swipeBack">
    <SwipeBackThemeAnimation TargetName="button" />
</Storyboard>

<Storyboard x:Key="swipeHint">
    <SwipeHintThemeAnimation TargetName="button" />
</Storyboard>

<Storyboard x:Key="dragItem">
    <DragItemThemeAnimation TargetName="button" />
</Storyboard>

<Storyboard x:Key="dropTargetItem">
    <DropTargetItemThemeAnimation TargetName="button" />
</Storyboard>

<Storyboard x:Key="dragOver">
    <DragOverThemeAnimation TargetName="button" />
</Storyboard>
</Page.Resources>

<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <StackPanel Name="animationTriggersStackPanel"
        Grid.Column="0"
        VerticalAlignment="Center">

        <Button Content="Fade In"
            Tag="fadeIn"
            Click="OnButtonClick" />

        <Button Content="Fade Out"
            Tag="fadeOut"
            Click="OnButtonClick" />

        <Button Content="Pop In"
            Tag="popIn"
            Click="OnButtonClick" />

        <Button Content="Pop Out"
            Tag="popOut"
            Click="OnButtonClick" />

        <Button Content="Reposition"

```

```

        Tag="reposition"
        Click="OnButtonClick" />

<Button Content="Pointer Up"
        Tag="pointerUp"
        Click="OnButtonClick" />

<Button Content="Pointer Down"
        Tag="pointerDown"
        Click="OnButtonClick" />

<Button Content="Swipe Back"
        Tag="swipeBack"
        Click="OnButtonClick" />

<Button Content="Swipe Hint"
        Tag="swipeHint"
        Click="OnButtonClick" />

<Button Content="Drag Item"
        Tag="dragItem"
        Click="OnButtonClick" />

<Button Content="Drop Target Item"
        Tag="dropTargetItem"
        Click="OnButtonClick" />

<Button Content="Drag Over"
        Tag="dragOver"
        Click="OnButtonClick" />
</StackPanel>

<!-- Animation target -->
<Button Name="button"
        Grid.Column="1"
        Content="Big Button"
        FontSize="48"
        HorizontalAlignment="Center"
        VerticalAlignment="Center" />

</Grid>
</Page>

```

Besides the *Button* named "button", the XAML file also defines a *Button* for each of the preconfigured animations. The code-behind file uses the *Tag* property to trigger the corresponding *Storyboard*:

Project: PreconfiguredAnimations | File: MainPage.xaml.cs (excerpt)

```

void OnButtonClick(object sender, RoutedEventArgs args)
{
    Button btn = sender as Button;
    string key = btn.Tag as string;
    Storyboard storyboard = this.Resources[key] as Storyboard;
    storyboard.Begin();
}

```

Watch out! Some of these animations cause the target *Button* to disappear, and others are rather subtle, but you'll get an idea of some of the effects that you might want to add to your own application.

Another set of predefined animations are the 8 classes that derive from *Transition*. These are rather more complex sets of animations that you set to one of the following properties of type *TransitionCollection*:

- *Transitions* property defined by *UIElement*
- *ContentTransitions* property defined by *ContentControl*
- *ChildrenTransitions* property defined by *Panel*
- *ItemContainerTransitions* property defined by *ItemsControl*

For example, try replacing the *StackPanel* tag in the *PreconfiguredAnimations* program with the following:

```
<StackPanel Name="animationTriggersStackPanel"
    Grid.Column="0"
    VerticalAlignment="Center">
    <StackPanel.ChildrenTransitions>
        <TransitionCollection>
            <EntranceThemeTransition />
        </TransitionCollection>
    </StackPanel.ChildrenTransitions>
```

Now, as the page is loaded, the buttons seem to appear a little offset from their actual positions and then shift into place.

I'll have more to say about these transitions in Chapters 10 and 11.

Chapter 9

Transforms

In Chapter 8, “Animation,” you saw how to use animations to move objects around the screen, change their size or color or opacity, and even move the dots in a dotted line. But certain types of animations were missing. What if you want to use an animation to *rotate* a button when the button is clicked? And I don’t necessarily mean to make the button spin around crazy like, but maybe just jiggle a little as if the button is saying, “I simply can’t restrain my enthusiasm to be carrying out the command you desire.”

What you need for this job (and others like it) are *transforms*. Back in the old days, transforms were called *graphics transforms* or even—perhaps to scare away the uninitiated—*matrix transforms*. But in recent years transforms have been liberated from the greedy clutches of the graphics mavens and made available to all programmers.

This is not to imply that transforms no longer have anything to do with mathematics. (Yes, there will be math.) But it’s possible to use transforms in the Windows Runtime without getting involved in the mathematics.

A Brief Overview

A transform is basically a mathematical formula that is applied to a point (x, y) to create a new point (x', y'). If you apply the same formula to all the points of a visual object, you can effectively move the object, or make it a different size, or rotate it, or even distort the object in various ways.

Transforms are supported in the Windows Runtime with three properties defined by *UIElement*: *RenderTransform*, *RenderTransformOrigin*, and *Projection*. Because these properties are defined by *UIElement*, transforms are not limited to vector graphics as they were in the old days. You can apply transforms to any element, including *Image*, *TextBlock*, and *Button*. If you apply a transform to a *Panel* derivative such as a *Grid*, it also applies to all the children of that panel.

To apply a transform to an element, use property-element syntax to set the *RenderTransform* property to an instance of a class that derives from *Transform*, for example *RotateTransform*:

Project: SimpleRotate | File: MainPage.xaml (excerpt)

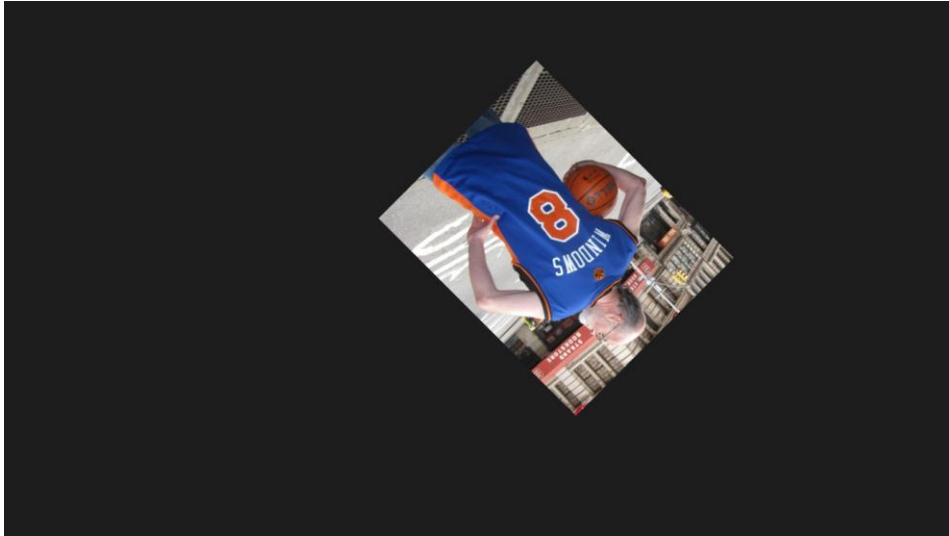
```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
  <Image Source="http://www.charlespetzold.com/pw6/PetzoldJersey.jpg"
        Stretch="None"
        HorizontalAlignment="Right"
        VerticalAlignment="Bottom">
    <Image.RenderTransform>
      <RotateTransform Angle="135" />
    </Image.RenderTransform>
  </Image>
</Grid>
```

```

        </Image.RenderTransform>
    </Image>
</Grid>

```

The *Angle* property of the *RotateTransform* indicates a clockwise rotation of 135 degrees:



But the result only appears reasonable because I knew that the *Image* would be rotated relative to its upper-left corner, so I deliberately positioned the *Image* element in the lower-right corner of the page. Rotation in two dimensions always occurs around a particular point—like a pin that attaches a photograph to a cork board—and setting that point correctly turns out to be one of the trickier aspects of working with transforms.

You can set the *RenderTransform* property to any one of the seven classes that derive from *Transform*, arranged here roughly in order of increasing mathematical complexity:

```

Object
    DependencyObject
        GeneralTransform
            Transform
                TranslateTransform
                ScaleTransform
                RotateTransform
                SkewTransform
                CompositeTransform
                MatrixTransform
                TransformGroup

```

These classes define traditional two-dimensional affine transforms. The word “affine” suggests that the transformed object has certain affinities with the nontransformed object: A straight line is always

transformed to another straight line. The line possibly assumes a different location, size, or orientation, but it is still a straight line. Lines that are parallel prior to an affine transform continue to be parallel after the transform. An affine transform never causes anything to shoot off into infinity. Indeed, the mathematical definition of affine is “preserving finiteness.”

The Windows Runtime also supports a certain type of nonaffine transform commonly used in three-dimensional perspective. You can use the Windows Runtime to achieve three-dimensional effects by setting the *Projection* property defined by *UIElement* to an instance of one of the two classes that derive from *Projection*:

```
Object
    DependencyObject
        Projection
            PlaneProjection
            Matrix3DProjection
```

Rotation in three dimensions is always around an axis. Rotation around the *Y* (vertical) axis is demonstrated in the SimpleProjection project:

Project: SimpleProjection | File: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Image Source="http://www.charlespetzold.com/pw6/PetzoldJersey.jpg"
        HorizontalAlignment="Center"
        VerticalAlignment="Center">
        <Image.Projection>
            <PlaneProjection RotationY="-60" />
        </Image.Projection>
    </Image>
</Grid>
```

This is of a somewhat different sort of rotation in two dimensions:



Obviously, parallel lines are not preserved in this type of transform.

The *Projection* transforms are sometimes called *pseudo-3D* transforms and are intended to provide a little “3Dishness” to the Windows Runtime. You can define an animation to make an element seem to swing into view like a door or flip around like a playing card. But the element itself stays flat. This is why one of the *Projection* classes refers to a “plane.” You’re basically taking a flat element and moving it in 3D space.

Math-oriented programmers might be able to persuade *Matrix3DProjection* to display actual 3D objects in the Windows Runtime. But the Windows Runtime is missing some crucial features of 3D, such as surface shading based on light sources, or clipping when one object is partially hidden behind another. If you need to bring real 3D graphics into your Windows 8 application, you’ll want to use Direct3D, which is only accessible from C++ and (it grieves me to say) beyond the scope of this book.

Rotation (Manual and Animated)

It’s common for tutorials such as this to begin the subject of transforms with the mathematically simple ones: *TranslateTransform* to move objects and *ScaleTransform* to make them larger or smaller. But these aren’t very impressive because you’ve already seen animations that move an object around the screen or change its size. That’s why I’m starting with something you can’t do in other ways.

Although you can set the *Angle* property of *RotateTransform* directly in XAML, it’s much more fun to change the *Angle* property dynamically with a data binding or an animation, and the result can also be more revealing of what’s actually going on. Here’s a XAML file with the *Angle* property of a *RotateTransform* bound to the *Value* property of a *Slider* with a range from 0 through 360:

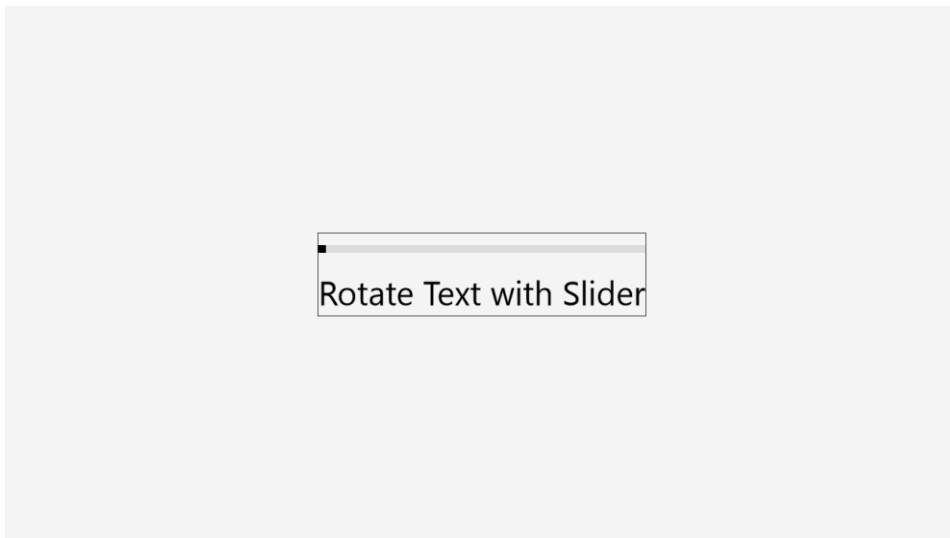
Project: RotateTheText | File: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Border BorderBrush="{StaticResource ApplicationForegroundThemeBrush}"
            BorderThickness="1"
            HorizontalAlignment="Center"
            VerticalAlignment="Center">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>

            <Slider Name="slider"
                    Grid.Row="0"
                    Minimum="0"
                    Maximum="360" />

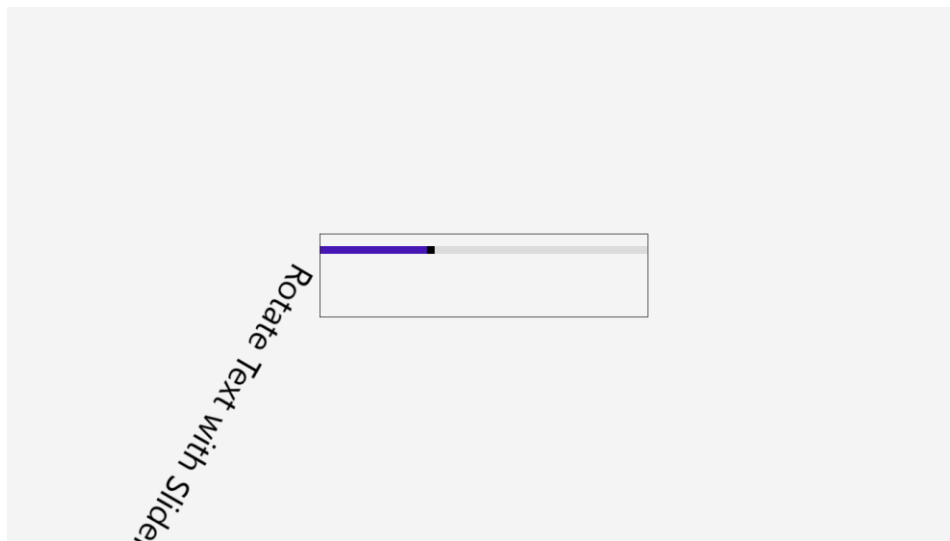
            <TextBlock Text="Rotate Text with Slider"
                      Grid.Row="1"
                      FontSize="48">
                <TextBlock.RenderTransform>
                    <RotateTransform Angle="{Binding ElementName=slider, Path=Value}" />
                </TextBlock.RenderTransform>
            </TextBlock>
        </Grid>
    </Border>
</Grid>
```

The *Slider* and *TextBlock* occupy two rows of a *Grid* that's inside a *Border*. Here's how it looks when the screen first comes up:



The *TextBlock* width determines the *Grid* width, which then determines the *Slider* width and the *Border* width.

As you use the mouse or your fingers to change the *Slider* value, the *TextBlock* rotates in a clockwise direction. Here it is at 120 degrees:



It's obvious that the size of the *Grid* and the *Border* continue to be based on the unrotated *TextBlock*, and the rotated *TextBlock* has broken free of the boundaries of its ancestors in the visual tree.

The property of *UIElement* to which you set the *RotateTransform* is named *RenderTransform*, and you'll want to mull over that property name a little bit. That word *render* means that the transform affects only how the element is *rendered* and *not* how the element appears to the layout system. That's a mix of good news and bad news.

The good news is that this transform occurs at a relatively deep level in the graphics composition system. Rotating the *TextBlock* does not require that the entire visual tree be subjected to an updated layout. Because the layout system doesn't get involved, transform animations can occur in a secondary thread and performance is very good. The layout system is completely unaware that the *TextBlock* is being rotated.

The bad news is that the layout system is completely unaware that the *TextBlock* is being rotated. For example, you might want to display a sideways *TextBlock* by rotating it by 90 degrees, perhaps as a caption for the side of a graph. It would be most convenient if the layout system calculated the dimensions of the rotated *TextBlock* so that you could simply put it in the cell of a *Grid* and have the *Grid* position it properly. But that's not possible in the Windows Runtime in any easy generalized manner.

In contrast, the version of *UIElement* available in the Windows Presentation Foundation (WPF) defined both a *RenderTransform* property (which worked like the Windows Runtime) and a *LayoutTransform* property, which allowed specifying a transform recognized by the layout system. That *LayoutTransform* property was lost in the transition from WPF to Silverlight and the Windows Runtime,

and mimicking it requires a bit of work.

Let's go back to the running *RotateTheText* program. Manipulate the *Slider* so that the *TextBlock* partially lies on top of the *Slider*:



Now remove all fingers from the screen (or release the mouse button), and try touching or clicking the *Slider* in a spot where the *TextBlock* overlaps. The *Slider* doesn't respond because the *TextBlock* is blocking the mouse or touch input. The lesson learned is this: Although the layout system doesn't know that the *TextBlock* has moved, hit-testing logic continues to be aware exactly where it is. (On the other hand, while you're in the actual process of manipulating the *Slider*, the *TextBlock* doesn't interfere because the *Slider* has *captured* this input, which is a concept I'll discuss in Chapter 13, "Touch, Etc.")

You'll also notice that the rotation of the *TextBlock* is relative to its upper-left corner, which conceptually is the origin of the *TextBlock*: the point (0, 0). In many graphics systems, it is common for graphics transforms to be relative to the origin of the canvas on which the graphics object is positioned. In the Windows Runtime, all transforms are relative to the element on which they're applied.

Very often you'll prefer that rotation be relative to some point other than the upper-left corner. This point is sometimes referred to as "the center of rotation," and you can specify it in three different ways:

The first way is the one that is most illuminating of the underlying mathematics of the transform, but I'll save it for later.

The second way involves the *RotateTransform* class itself. The class defines *CenterX* and *CenterY* properties that are 0 by default. If you want this particular *TextBlock* to rotate relative to its center, set *CenterX* to half the width of the *TextBlock* and *CenterY* to half its height. This information can be obtained during the *Loaded* handler, so you can add something like the following to the constructor of the code-behind file. Fortunately, I gave the *TextBlock* a name even though that name isn't used in the

XAML file:

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

    Loaded += (sender, args) =>
    {
        RotateTransform rotate = txtblk.RenderTransform as RotateTransform;
        rotate.CenterX = txtblk.ActualWidth / 2;
        rotate.CenterY = txtblk.ActualHeight / 2;
    };
}
```

You might think that this approach is a bit of a hassle, so you'll be pleased to discover that the third approach is much simpler. It involves the *RenderTransformOrigin* property defined by *UIElement*. This property is of type *Point* but you set it to *relative* coordinates, where the *X* and *Y* values normally range from 0 to 1. The default is the point (0, 0), which is the upper-left corner. The point (1, 0) is the upper-right corner, (0, 1) is the lower-left corner, and (1, 1) is the lower-right corner. To specify an origin at the center of the element, use the point (0.5, 0.5):

```
<TextBlock Name="txtblk"
    Text="Rotate Text with Slider"
    Grid.Row="1"
    FontSize="48"
    RenderTransformOrigin="0.5 0.5">
    <TextBlock.RenderTransform>
        <RotateTransform Angle="{Binding ElementName=slider, Path=Value}" />
    </TextBlock.RenderTransform>
</TextBlock>
```

Notice that *CenterX* and *CenterY* are properties of *RotateTransform* but the *RenderTransformOrigin* property is defined by *UIElement* and common to all elements. If you set *RenderTransformOrigin* in addition to *CenterX* and *CenterY*, the effects are compounded. In this example, the compounded effect of both examples would cause rotation to be around the lower-right corner of the *TextBlock*.

You can specify a center of rotation that is outside the element. Here's a XAML file that positions a *TextBlock* in the top center of the page and then starts up a "forever" animation to rotate it:

Project: RotateAroundCenter | File: MainPage.xaml (excerpt)

```
<Page ... >
    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <TextBlock Name="txtblk"
            Text="Rotated Text"
            FontSize="48"
            HorizontalAlignment="Center"
            VerticalAlignment="Top">
            <TextBlock.RenderTransform>
                <RotateTransform x:Name="rotate" />
            </TextBlock.RenderTransform>
        </TextBlock>
    </Grid>
```

```

<Page.Triggers>
  <EventTrigger>
    <BeginStoryboard>
      <Storyboard RepeatBehavior="Forever">
        <DoubleAnimation Storyboard.TargetName="rotate"
          Storyboard.TargetProperty="Angle"
          From="0" To="360" Duration="0:0:2" />
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger>
</Page.Triggers>
</Page>

```

Without any additional code, this program would rotate the *TextBlock* around its upper-left corner, and it would sweep right off the screen at certain times during the animation. But the constructor of the code-behind file defines two event handlers to set the *CenterX* and *CenterY* properties of the *RotateTransform*:

Project: RotateAroundCenter | File: MainPage.xaml.cs (excerpt)

```

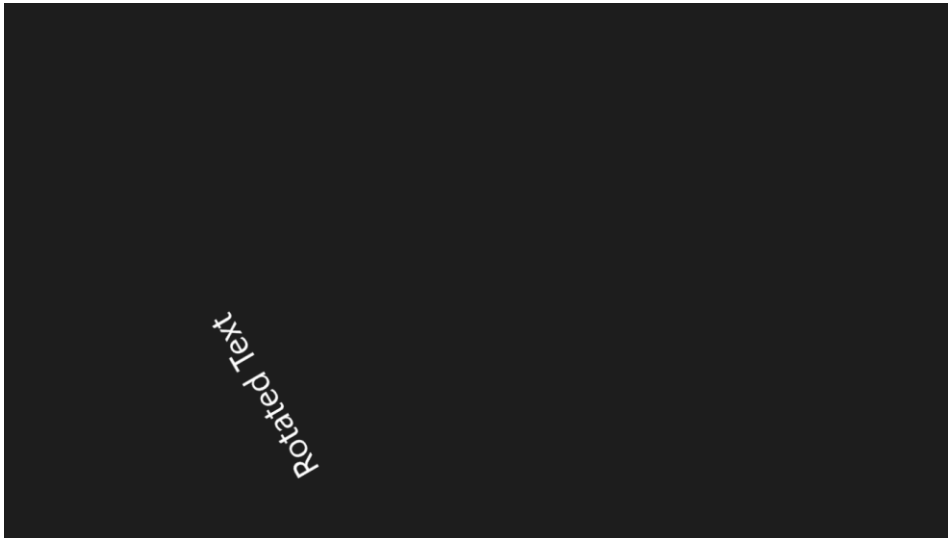
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();

        Loaded += (sender, args) =>
        {
            rotate.CenterX = txtblk.ActualWidth / 2;
        };

        SizeChanged += (sender, args) =>
        {
            rotate.CenterY = args.NewSize.Height / 2;
        };
    }
}

```

The center of rotation is set to a point aligned with the horizontal center of the *TextBlock* but a distance below the *TextBlock* equal to half the height of the page. The result is that the *TextBlock* rotates in a circle around the page center:



Visual Feedback

An animated transform can be effective for alerting the user to something on the screen that requires attention or for confirming that an operation has been initiated. In the `JiggleButtonDemo` program, I added a new `UserControl` item that I named *JiggleButton*, but then I changed the base class in the XAML and C# files from `UserControl` to `Button`. Here's the complete `JiggleButton.xaml` file:

Project: `JiggleButtonDemo` | `JiggleButton.xaml`

```
<Button
    x:Class="JiggleButtonDemo.JiggleButton"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    RenderTransformOrigin="0.5 0.5"
    Click="OnJiggleButtonClick">

    <Button.Resources>
        <Storyboard x:Key="jiggleAnimation">
            <DoubleAnimation Storyboard.TargetName="rotate"
                            Storyboard.TargetProperty="Angle"
                            From="0" To="10" Duration="0:0:0.33"
                            AutoReverse="True">
                <DoubleAnimation.EasingFunction>
                    <ElasticEase EasingMode="EaseIn" />
                </DoubleAnimation.EasingFunction>
            </DoubleAnimation>
        </Storyboard>
    </Button.Resources>

    <Button.RenderTransform>
        <RotateTransform x:Name="rotate" />
    </Button.RenderTransform>
</Button>
```



```

        </Button.RenderTransform>
    </Button>

```

The content of the *Button* isn't defined in this XAML file but three *Button* properties are set: *RenderTransformOrigin* (in the root tag), *Resources*, and *RenderTransform*. Normally if you wanted to jiggle an element with a rotation, you'd need to use key frames, because you first want to rotate from 0 to 10 degrees (for example), then from 10 degrees to -10 degrees several times, and then back to 0 degrees. But *ElasticEase* with an *EasingMode* of *EaseIn* is a great alternative. The *DoubleAnimation* is defined to rotate the button 10 degrees and then back to zero, but the *ElasticEase* function incorporates wide negative swing, so the animation actually ranges from -10 to 10 degrees.

The code-behind file for the *JiggleButton* simply triggers the animation in a *Click* event handler:

Project: JiggleButtonDemo | JiggleButton.xaml.cs

```

using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media.Animation;

namespace JiggleButtonDemo
{
    public sealed partial class JiggleButton : Button
    {
        public JiggleButton()
        {
            this.InitializeComponent();
        }

        void OnJiggleButtonClick(object sender, RoutedEventArgs args)
        {
            (this.Resources["jiggleAnimation"] as Storyboard).Begin();
        }
    }
}

```

The *MainPage.xaml* file instantiates a *JiggleButton* so that you can play with it:

Project: JiggleButtonDemo | File: MainPage.xaml (excerpt)

```

<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <local:JiggleButton Content="JiggleButton Demo"
        FontSize="24"
        HorizontalAlignment="Center"
        VerticalAlignment="Center" />
</Grid>

```

Keep in mind that *JiggleButton* derives from *Button*, so you can use it just like any other *Button*, except that you shouldn't set the *RenderTransform* or *RenderTransformOrigin* properties on it because doing so would interfere with the jiggle animation.

Translation

TranslateTransform defines two properties *X* and *Y* that cause an element to be rendered offset to its original position. One simple application of the *TranslateTransform* is to display text with an “embossed” or “engraved” appearance, or with a drop shadow like this:



Because light normally comes from above—and perhaps also because we’re accustomed to the convention that 3D-ish objects on the computer screen are illuminated with a light source from the upper-left—the text on the top appears as if it has shadows at the right and bottom, and hence the letters are projecting outwards from the screen. The engraved effect is opposite that: The shadows are on the left and top, and so the letters appear to be carved out.

The page that displays those three text strings actually consists of six *TextBlock* elements. In the first two pairs, a *TextBlock* colored with the default foreground brush is covered by another *TextBlock* colored with the default background brush but offset by 2 pixels in the horizontal and vertical directions:

Project: TextEffects | File: MainPage.xaml (excerpt)

```
<Page ... >
  <Page.Resources>
    <Style TargetType="TextBlock">
      <Setter Property="FontFamily" Value="Times New Roman" />
      <Setter Property="FontSize" Value="192" />
      <Setter Property="HorizontalAlignment" Value="Center" />
      <Setter Property="VerticalAlignment" Value="Center" />
    </Style>
  </Page.Resources>

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

```

<Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
    <RowDefinition Height="*" />
</Grid.RowDefinitions>

<TextBlock Text="EMBOSS"
    Grid.Row="0" />

<TextBlock Text="EMBOSS"
    Grid.Row="0"
    Foreground="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <TextBlock.RenderTransform>
        <TranslateTransform X="-2" Y="-2" />
    </TextBlock.RenderTransform>
</TextBlock>

<TextBlock Text="ENGRAVE"
    Grid.Row="1" />

<TextBlock Text="ENGRAVE"
    Grid.Row="1"
    Foreground="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <TextBlock.RenderTransform>
        <TranslateTransform X="2" Y="2" />
    </TextBlock.RenderTransform>
</TextBlock>

<TextBlock Text="Drop Shadow"
    Grid.Row="2"
    Foreground="Gray">
    <TextBlock.RenderTransform>
        <TranslateTransform X="6" Y="6" />
    </TextBlock.RenderTransform>
</TextBlock>

<TextBlock Text="Drop Shadow"
    Grid.Row="2" />
</Grid>
</Page>

```

Notice that the embossing effect requires negative offsets (so that the *TextBlock* on top is shifted to the left and up) whereas the engraving effect has positive offsets. You can use these same effects just slightly less successfully with a dark theme, but you'll have to switch the signs of the *X* and *Y* values.

A drop-shadow effect is similar except that the text on top is colored normally and a gray shadow is offset underneath.

I don't recommend using the following technique on a regular basis, but you can give your on-screen text a little bit of depth—that's *visual* depth and not *intellectual* depth, alas—using a bunch of *TextBlock* elements offset from each other by one pixel:



The generation of these elements is handled entirely in the code-behind file:

Project: DepthText | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    const int COUNT = 48;    // ~1/2 inch

    public MainPage()
    {
        this.InitializeComponent();

        Grid grid = this.Content as Grid;
        Brush foreground = this.Resources["ApplicationForegroundThemeBrush"] as Brush;
        Brush grayBrush = new SolidColorBrush(Colors.Gray);

        for (int i = 0; i < COUNT; i++)
        {
            bool firstOrLast = i == 0 || i == COUNT - 1;

            TextBlock txtblk = new TextBlock
            {
                Text = "DEPTH",
                FontSize = 192,
                FontWeight = FontWeights.Bold,
                HorizontalAlignment = HorizontalAlignment.Center,
                VerticalAlignment = VerticalAlignment.Center,
                RenderTransform = new TranslateTransform
                {
                    X = COUNT - i - 1,
                    Y = i - COUNT + 1,
                },
                Foreground = firstOrLast ? foreground : grayBrush
            };
            grid.Children.Add(txtblk);
        }
    }
}
```

```

    }
}
}

```

A *TranslateTransform* is a great way to move something a little bit from the position determined by the layout system. You'll see a couple examples of *TranslateTransform* used in this way in the *StandardStyles.xaml* file.

In Chapter 8 I showed an example of animating *Canvas.Left* and *Canvas.Top* attached properties to move an object around the screen. You can do the same type of animation by defining a *TranslateTransform* on the element you wish to move and using the animation to target the *X* and *Y* properties. One advantage is that the element being animated need not be a child of a *Canvas*, but there doesn't seem to be a performance difference. Both types of animations are performed in secondary threads.

Transform Groups

I mentioned earlier that there are three ways to set a center of rotation but I was going to save the first way for a later discussion. Now is the time. It's a little more complicated because it involves a transform that is constructed from other transforms.

One of the classes that derives from *Transform* is *TransformGroup*, which has a property named *Children* of type *TransformCollection*, with which you can construct a composite transform from multiple *Transform* derivatives.

You might define a *RotateTransform* like this:

```
<RotateTransform Angle="A" CenterX="CX" CenterY="CY" />
```

where *A*, *CX*, and *CY* are actual numbers or perhaps data bindings. That transform is equivalent to the following *TransformGroup*:

```
<TransformGroup>
  <TranslateTransform X="-CX" Y="-CY" />
  <RotateTransform Angle="A" />
  <TranslateTransform X="CX" Y="CY" />
</TransformGroup>
```

The two *TranslateTransform* tags seem to cancel each other out, but they surround a *RotateTransform*. Let me demonstrate in two ways that this transform group is equivalent to the first *RotateTransform* by itself.

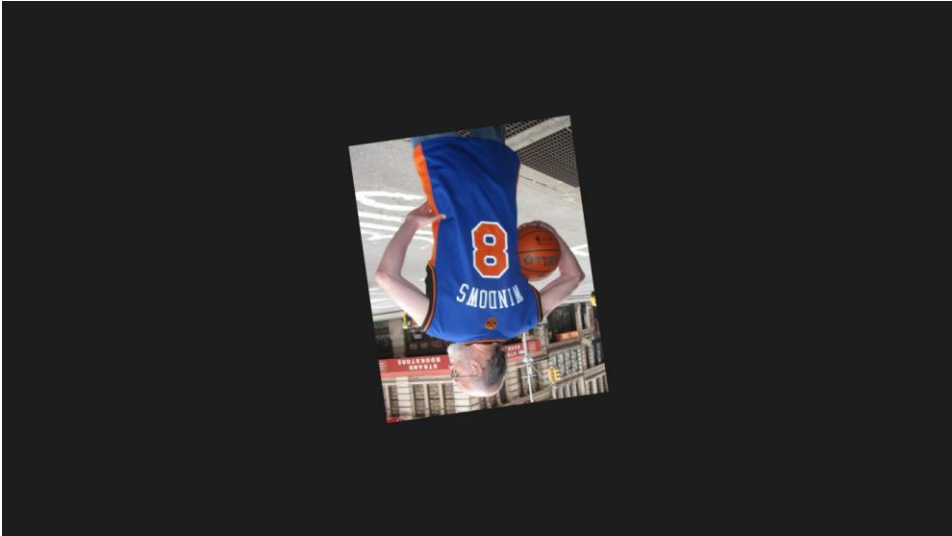
The following *ImageRotate* program references a bitmap on my website that I know is 320 pixels wide and 400 pixels tall. To rotate that bitmap around its center the *RotateTransform* would normally have *CenterX* and *CenterY* set to half those values (160 and 200), but I've instead used a pair of *TranslateTransform* objects:

Project: ImageRotate | File: MainPage.xaml (excerpt)

```
<Page ... >
  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Image Source="http://www.charlespetzold.com/pw6/PetzoldJersey.jpg"
      Stretch="None"
      HorizontalAlignment="Center"
      VerticalAlignment="Center">
      <Image.RenderTransform>
        <TransformGroup>
          <TranslateTransform X="-160" Y="-200" />
          <RotateTransform x:Name="rotate" />
          <TranslateTransform X="160" Y="200" />
        </TransformGroup>
      </Image.RenderTransform>
    </Image>
  </Grid>

  <Page.Triggers>
    <EventTrigger>
      <BeginStoryboard>
        <Storyboard RepeatBehavior="Forever">
          <DoubleAnimation Storyboard.TargetName="rotate"
            Storyboard.TargetProperty="Angle"
            From="0" To="360" Duration="0:0:3">
            <DoubleAnimation.EasingFunction>
              <ElasticEase EasingMode="EaseInOut" />
            </DoubleAnimation.EasingFunction>
          </DoubleAnimation>
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Page.Triggers>
</Page>
```

The *ElasticEase* animation with a mode of *EaseInOut* causes the image to rock back and forth crazily before and after it actually spins around, but you can see that the rotation is clearly around the image's center:



The following screenshot shows the process in the individual steps: The lightest *TextBlock* is positioned in the center of the page. The next darkest *TextBlock* shows the effect of a *TranslateTransform* that shifts the *TextBlock* left by half its width and up by half its height. The next darkest *TextBlock* is rotated relative to its origin—the upper-left corner of the original *TextBlock*. The final black *TextBlock* is then shifted by half its width and height. The final result is the original *TextBlock* rotated around its center:



Here's the XAML file that created that image:

Project: RotationCenterDemo | File: MainPage.xaml (excerpt)

<Page ... >

```

<Page.Resources>
    <Style TargetType="TextBlock">
        <Setter Property="Text" Value="Rotate around Center" />
        <Setter Property="FontSize" Value="48" />
        <Setter Property="HorizontalAlignment" Value="Center" />
        <Setter Property="VerticalAlignment" Value="Center" />
    </Style>
</Page.Resources>

<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <TextBlock Name="txtblk"
        Foreground="#D0D0D0" />

    <TextBlock Foreground="#A0A0A0">
        <TextBlock.RenderTransform>
            <TranslateTransform x:Name="translateBack1" />
        </TextBlock.RenderTransform>
    </TextBlock>

    <TextBlock Foreground="#707070">
        <TextBlock.RenderTransform>
            <TransformGroup>
                <TranslateTransform x:Name="translateBack2" />
                <RotateTransform Angle="45" />
            </TransformGroup>
        </TextBlock.RenderTransform>
    </TextBlock>

    <TextBlock Foreground="{StaticResource ApplicationForegroundThemeBrush}">
        <TextBlock.RenderTransform>
            <TransformGroup>
                <TranslateTransform x:Name="translateBack3" />
                <RotateTransform Angle="45" />
                <TranslateTransform x:Name="translate" />
            </TransformGroup>
        </TextBlock.RenderTransform>
    </TextBlock>
</Grid>
</Page>

```

The *X* and *Y* values for all the *TranslateTransform* tags are set from the *Loaded* handler:

Project: RotationCenterDemo | File: MainPage.xaml.cs (excerpt)

```

public MainPage()
{
    this.InitializeComponent();

    Loaded += (sender, args) =>
    {
        translateBack1.X =
        translateBack2.X =
        translateBack3.X = -(translate.X = txtblk.ActualWidth / 2);

        translateBack1.Y =

```



```

        translateBack2.Y =
            translateBack3.Y = -(translate.Y = txtblk.ActualHeight / 2);
    };
}

```

Transforms can be combined for some very interesting effects that might initially seem beyond the scope of the nonmathematical, nongraphics programmer. Here's a XAML file that uses a *Polygon* element to define a simple propeller shape, and then applies three transforms to it, a *RotateTransform*, a *TranslateTransform*, and another *RotateTransform*:

Project: Propeller | File: MainPage.xaml (excerpt)

```

<Page ... >
    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Polygon Points="40 0, 60 0, 53 47,
                        100 40, 100 60, 53 53,
                        60 100, 40 100, 47 53,
                        0 60, 0 40, 47 47"
                Stroke="{StaticResource ApplicationForegroundThemeBrush}"
                Fill="SteelBlue"
                HorizontalAlignment="Center"
                VerticalAlignment="Center"
                RenderTransformOrigin="0.5 0.5">
            <Polygon.RenderTransform>
                <TransformGroup>
                    <RotateTransform x:Name="rotate1" />
                    <TranslateTransform X="300" />
                    <RotateTransform x:Name="rotate2" />
                </TransformGroup>
            </Polygon.RenderTransform>
        </Polygon>
    </Grid>

    <Page.Triggers>
        <EventTrigger>
            <BeginStoryboard>
                <Storyboard>
                    <DoubleAnimation Storyboard.TargetName="rotate1"
                                    Storyboard.TargetProperty="Angle"
                                    From="0" To="360" Duration="0:0:0.5"
                                    RepeatBehavior="Forever" />

                    <DoubleAnimation Storyboard.TargetName="rotate2"
                                    Storyboard.TargetProperty="Angle"
                                    From="0" To="360" Duration="0:0:6"
                                    RepeatBehavior="Forever" />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Page.Triggers>
</Page>

```

The *Storyboard* contains two *DoubleAnimation* objects. The first *DoubleAnimation* targets the first *RotateTransform* object to rotate the propeller itself around its center at the speed of 2 cycles per

second. The *TranslateTransform* moves this rotating propeller 300 pixels to the right of the center of the page, and the second *DoubleAnimation* targets the second *RotateTransform* to rotate the propeller again. But this rotation is relative to the original center of the propeller, which means that the propeller circles the center of the page with a radius of 300 pixels at the rate of 10 revolutions per minute.



Now it's perhaps clear how *RenderTransformOrigin* works: *RenderTransformOrigin* is equivalent to performing a *TranslateTransform* with negative *X* and *Y* values prior to the transform specified as the *RenderTransform* property, and performing another *TranslateTransform* with positive *X* and *Y* values after the *RenderTransform*.

The Scale Transform

The *ScaleTransform* class defines properties named *ScaleX* and *ScaleY* that increase or decrease the size of an element independently in the horizontal and vertical directions. If you want to preserve the correct aspect ratio of a target, you'll need to use the same values for *ScaleX* and *ScaleY*. If it's an animation, you need two animation objects.

The *ScaleTransform* does not affect the *ActualWidth* and *ActualHeight* properties of an element.

You've seen how to use a *Viewbox* to stretch a *TextBlock* in ways that violate its typographically correct aspect ratio. Here's how to do it with a *ScaleTransform*:

Project: OppositelyScaledText | File: MainPage.xaml (excerpt)

```
<Page ... >
    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <TextBlock Text="Scaled Text"
            FontSize="144"
            HorizontalAlignment="Center"
```

```

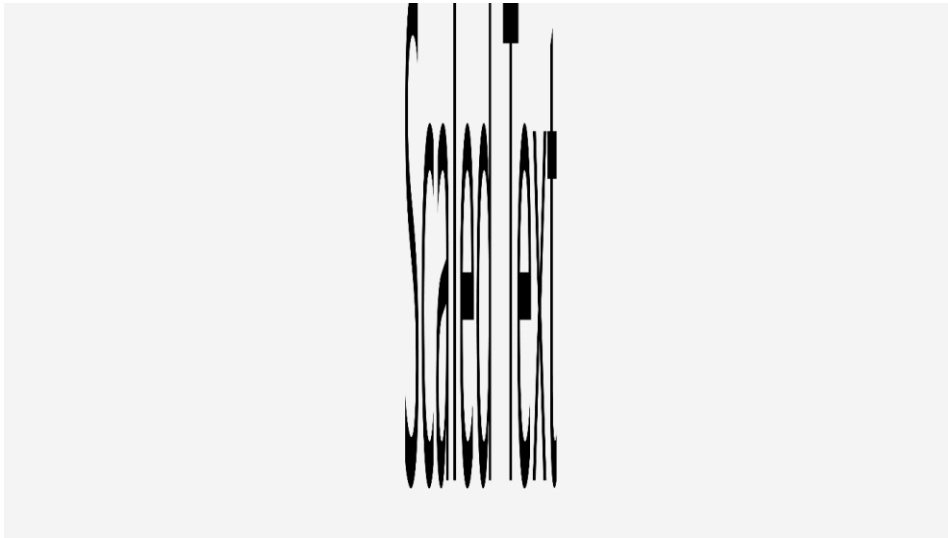
        VerticalAlignment="Center"
        RenderTransformOrigin="0.5 0.5">
        <TextBlock.RenderTransform>
            <ScaleTransform x:Name="scale" />
        </TextBlock.RenderTransform>
    </TextBlock>
</Grid>

<Page.Triggers>
    <EventTrigger>
        <BeginStoryboard>
            <Storyboard>
                <DoubleAnimation Storyboard.TargetName="scale"
                    Storyboard.TargetProperty="ScaleX"
                    BeginTime="0:0:2"
                    From="1" To="0.01" Duration="0:0:2"
                    AutoReverse="True"
                    RepeatBehavior="Forever" />

                <DoubleAnimation Storyboard.TargetName="scale"
                    Storyboard.TargetProperty="ScaleY"
                    From="10" To="0.1" Duration="0:0:2"
                    AutoReverse="True"
                    RepeatBehavior="Forever" />
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</Page.Triggers>
</Page>

```

This is actually not quite the way I wanted to write this program. I wanted to give the *TextBlock* a *FontSize* of 1 and then to animate *ScaleX* from 1 to 144 and *ScaleY* from 144 to 1, both reversed and repeated forever. That should have worked, but a flaw in the beta version of Windows 8 I've been using for this chapter apparently resulted in the 1-pixel high font being increased in size by a factor of 144 rather than becoming a 144-pixel high font. To get the program to work in a way I wanted, I gave the *TextBlock* a 144-pixel size and started the animations at the maximum values, one offset from the other. The *TextBlock* alternately stretches out horizontally and vertically:



Scaling is like rotation in that it is always in reference to a center point. The *ScaleTransform* class defines *CenterX* and *CenterY* properties just like *RotateTransform*, or you can set *RenderTransformOrigin* as I've done in the *OppositelyScaledText* program. The scaling center is the point that remains in the same location when the scaling occurs.

Scaling and rotation centers play a big role in manipulating on-screen objects (such as photographs) with your fingers. As you stretch, pinch, and rotate a photograph, the scaling and rotation center changes as your fingers move relative to each other. I'll discuss the technique for calculating this rotation center in Chapter 13.

Negative scaling factors flip an element around the horizontal or vertical axis. This technique is particularly useful for creating reflection effects. Unfortunately, the Windows Runtime is missing an important contributor to this effect: a *UIElement* property named *OpacityMask* of type *Brush* that allows defining a graduated opacity based on the alpha channel of the colors of a gradient brush. In the Windows Runtime, you'll have to mimic a graduated fade-out by covering up the element with another element that has a gradient brush incorporating transparency and the background color.

This is demonstrated in the *ReflectedFadeOutImage* project. The upper half of a *Grid* is shared by two items: an *Image* and another *Grid*. That second *Grid* contains the same *Image* covered by a *Rectangle* with a *LinearGradientBrush* that fades from the background color at top to transparent at the bottom:

Project: *ReflectedFadeOutImage* | File: *MainPage.xaml* (excerpt)

```
<Page ... >
    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Grid.RowDefinitions>
            <RowDefinition Height="*" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>
```

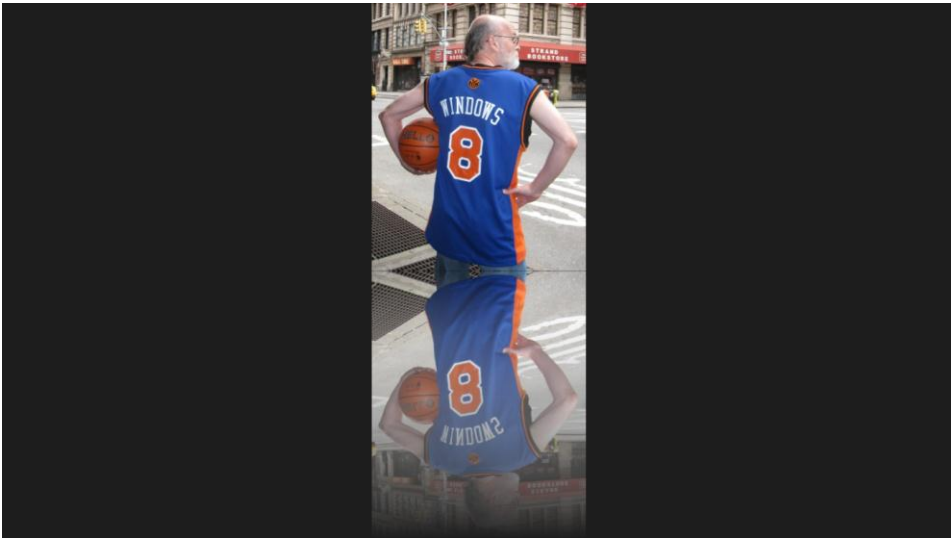
```

<Image Source="http://www.charlespetzold.com/pw6/PetzoldJersey.jpg"
      HorizontalAlignment="Center" />

<Grid RenderTransformOrigin="0 1"
      HorizontalAlignment="Center">
    <Grid.RenderTransform>
        <ScaleTransform ScaleY="-1" />
    </Grid.RenderTransform>
    <Image Source="http://www.charlespetzold.com/pw6/PetzoldJersey.jpg" />
    <Rectangle>
        <Rectangle.Fill>
            <LinearGradientBrush StartPoint="0 0" EndPoint="0 1" >
                <GradientStop Offset="0"
                    Color="{Binding
                        Source={StaticResource ApplicationPageBackgroundThemeBrush},
                        Path=Color}" />
                <GradientStop Offset="1" Color="Transparent" />
            </LinearGradientBrush>
        </Rectangle.Fill>
    </Rectangle>
</Grid>
</Grid>
</Page>

```

That inner *Grid* is also reflected around its bottom edge. The *RenderTransformOrigin* assigns a transform center at the lower left, and the *ScaleTransform* sets *ScaleY* to -1 , which flips the element around the horizontal axis:



In Chapter 14, “Bitmaps,” I’ll demonstrate another way to achieve this effect accessing the pixels of a bitmap and setting the transparency appropriately.

Building an Analog Clock

An analog clock is round. This simple fact implies that drawing the clock would probably be mathematically easy if you use arbitrary coordinates—that is, coordinates not in units of pixels but in units you choose for convenience—with the origin in the center. Putting the origin in the center also means you probably won't need to mess around with *CenterX* or *CenterY* settings for the *RotateTransform* objects that position the hands of the clock because the origin is also the center of rotation.

The traditional analog clock in a graphical environment adapts itself to whatever size it's given. It is tempting to use a *Viewbox* for this job, but with an analog clock that could be a problem. The layout system (and *Viewbox*) perceives the size of a vector graphics object to be the maximum *X* and *Y* values of its coordinate points. Negative coordinates are ignored, including those in three-quarters of an analog clock with an origin in the center.

The layout system (and *Viewbox*) will not determine the size of graphics involving negative coordinates properly, and a little “help” is required. Fortunately, transforms cascade from parent to child. You can set a transform on a *Grid*, and it will apply to everything in that *Grid*. The contents of the *Grid* can then have their own transforms.

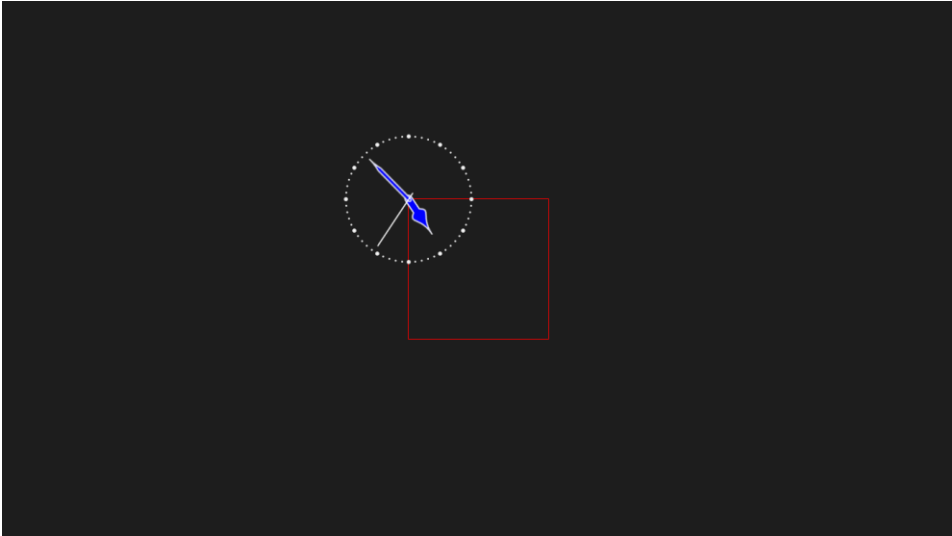
That's what I've done in the AnalogClock program. All the graphics are in a *Grid* that is fixed in size with a 200-pixel *Width* and *Height* implying a 100-pixel radius:

```
<Grid Width="200" Height="200">
```

```
    ... // clock graphics go here
```

```
</Grid>
```

Within that *Grid* are five *Path* elements that render the tick marks around the circumference of the clock, as well as the hour, minute, and second hand. These are all based on a coordinate system with *X* and *Y* values ranging from -100 to 100. If you could see that *Grid* (outlined here in red) and the clock, it would look like this:

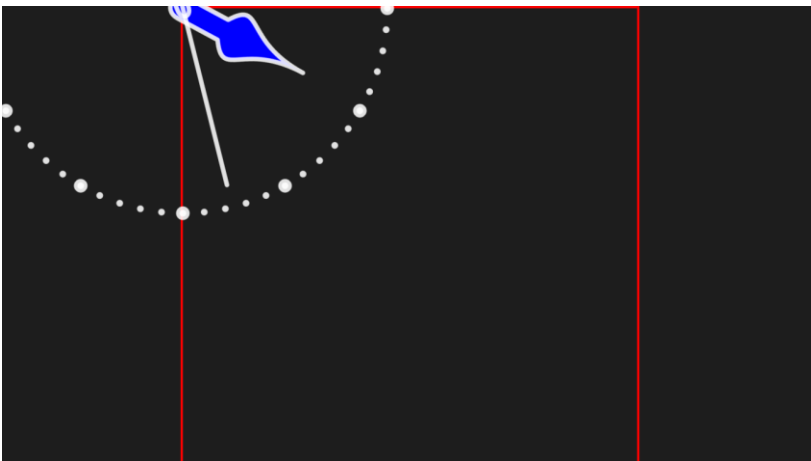


The *Grid* is positioned in the center of the page thanks its default alignment, but the center of the clock is positioned at the upper-left corner of the *Grid* because that's where the point (0, 0) is.

Now let's put that *Grid* in a *Viewbox*, like so:

```
<Viewbox>
  <Grid Width="200" Height="200">
    ... // clock graphics go here
  </Grid>
</Viewbox>
```

The *Viewbox* can correctly handle elements that have an origin at the upper-left corner but not graphics with negative coordinates:



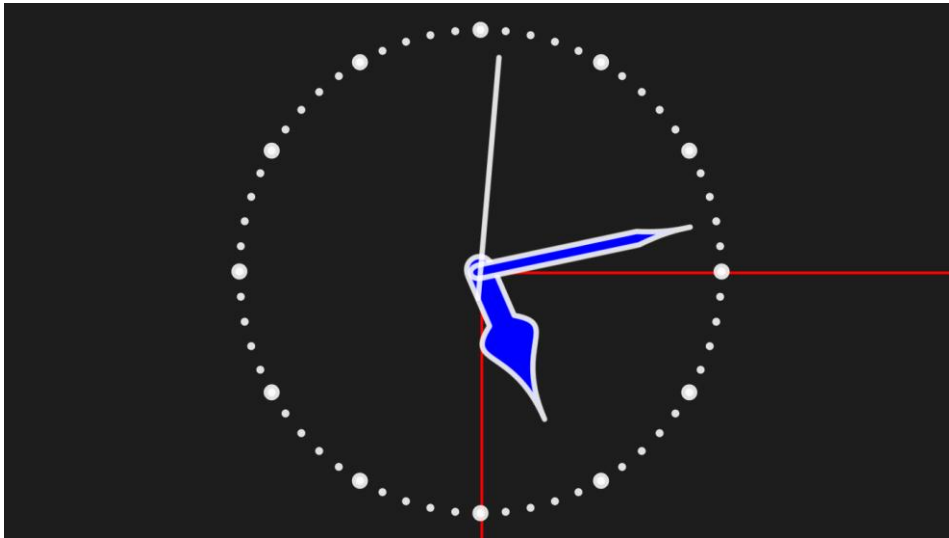
Fortunately, the fix is fairly easy. All that's necessary is to shift the *Grid* and the clock. This transform occurs before the *Viewbox* gets ahold of the element, so it's merely by 100 pixels:

```
<Viewbox>
  <Grid Width="200" Height="200">
    <Grid.RenderTransform>
      <TranslateTransform X="100" Y="100" />
    </Grid.RenderTransform>

    ... // clock graphics go here

  </Grid>
</Viewbox>
```

And here it is:



Now all that's necessary is to get rid of that red border.

The clock consists of five *Path* elements. Each of the three hands is defined by path markup syntax consisting of straight lines and Bézier curves. Here's the hour hand pointing to the 12:00 position. Because the hand is initially mostly on the top half of this clock, most of the hand has negative *Y* coordinates with only a few positive *Y* coordinates as it loops around the center.

```
<Path Data="M 0 -60 C 0 -30, 20 -30, 5 -20 L 5 0
           C 5 7.5, -5 7.5, -5 0 L -5 -20
           C -20 -30, 0 -30, 0 -60">
  <Path.RenderTransform>
    <RotateTransform x:Name="rotateHour" />
  </Path.RenderTransform>
</Path>
```

The tick marks are actually dotted lines. Here's the *Path* element for the small tick marks:


```

<Path Fill="{x:Null}"
      StrokeThickness="3"
      StrokeDashArray="0 3.14159">
  <Path.Data>
    <EllipseGeometry RadiusX="90" RadiusY="90" />
  </Path.Data>
</Path>

```

This creates a circle with a radius of 90, so the circumference is $2\pi 90$, which means that the 60 tick marks are separated by 3π , which not coincidentally is the product of the *StrokeThickness* and number in the *StrokeDashArray* indicating the distance between the dots in units of the *StrokeThickness*.

Since you enjoyed that one, here's the *Path* for the large tick marks:

```

<Path Fill="{x:Null}"
      StrokeThickness="6"
      StrokeDashArray="0 7.854">
  <Path.Data>
    <EllipseGeometry RadiusX="90" RadiusY="90" />
  </Path.Data>
</Path>

```

Again, the circumference is $2\pi 90$, but there are only 12 tick marks, so they are separated by 15π , which is close enough to the product of 6 and 7.854. Here's everything put together:

Project: AnalogClock | File: MainPage.xaml (excerpt)

```

<Page ... >

  <Page.Resources>
    <Style TargetType="Path">
      <Setter Property="Stroke" Value="{StaticResource ApplicationForegroundThemeBrush}" />
      <Setter Property="StrokeThickness" Value="2" />
      <Setter Property="StrokeStartLineCap" Value="Round" />
      <Setter Property="StrokeEndLineCap" Value="Round" />
      <Setter Property="StrokeLineJoin" Value="Round" />
      <Setter Property="StrokeDashCap" Value="Round" />
      <Setter Property="Fill" Value="Blue" />
    </Style>
  </Page.Resources>

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

    <Viewbox>
      <!-- Grid containing all graphics based on (0, 0) origin, 100-pixel radius -->
      <Grid Width="200" Height="200">

        <!-- Transform for entire clock -->
        <Grid.RenderTransform>
          <TranslateTransform X="100" Y="100" />
        </Grid.RenderTransform>

        <!-- Small tick marks -->
        <Path Fill="{x:Null}"
              StrokeThickness="3"

```

```

        StrokeDashArray="0 3.14159">
        <Path.Data>
            <EllipseGeometry RadiusX="90" RadiusY="90" />
        </Path.Data>
    </Path>

    <!-- Large tick marks -->
    <Path Fill="{x:Null}"
        StrokeThickness="6"
        StrokeDashArray="0 7.854">
        <Path.Data>
            <EllipseGeometry RadiusX="90" RadiusY="90" />
        </Path.Data>
    </Path>

    <!-- Hour hand pointing straight up -->
    <Path Data="M 0 -60 C 0 -30, 20 -30, 5 -20 L 5 0
        C 5 7.5, -5 7.5, -5 0 L -5 -20
        C -20 -30, 0 -30, 0 -60">
        <Path.RenderTransform>
            <RotateTransform x:Name="rotateHour" />
        </Path.RenderTransform>
    </Path>

    <!-- Minute hand pointing straight up -->
    <Path Data="M 0 -80 C 0 -75, 0 -70, 2.5 -60 L 2.5 0
        C 2.5 5, -2.5 5, -2.5 0 L -2.55 -60
        C 0 -70, 0 -75, 0 -80">
        <Path.RenderTransform>
            <RotateTransform x:Name="rotateMinute" />
        </Path.RenderTransform>
    </Path>

    <!-- Second hand pointing straight up -->
    <Path Data="M 0 10 L 0 -80">
        <Path.RenderTransform>
            <RotateTransform x:Name="rotateSecond" />
        </Path.RenderTransform>
    </Path>
</Grid>
</Viewbox>
</Grid>
</Page>

```

The code-behind file is responsible for calculating angles measured clockwise from 12:00 for the three *RotateTransform* objects:

Project: AnalogClock | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();
        CompositionTarget.Rendering += OnCompositionTargetRendering;
    }
}

```

```

    }

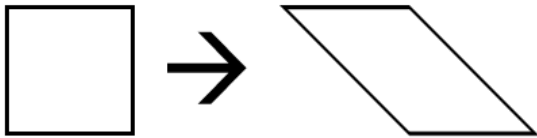
    void OnCompositionTargetRendering(object sender, object args)
    {
        DateTime dt = DateTime.Now;
        rotateSecond.Angle = 6 * (dt.Second + dt.Millisecond / 1000.0);
        rotateMinute.Angle = 6 * dt.Minute + rotateSecond.Angle / 60;
        rotateHour.Angle = 30 * (dt.Hour % 12) + rotateMinute.Angle / 12;
    }
}

```

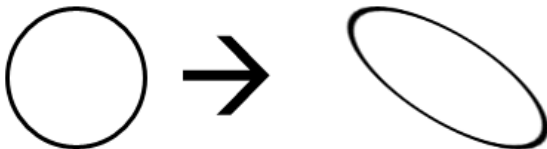
This clock has a “sweep” second hand that seems to move continuously. If you prefer a “tick” second hand that jumps by seconds, you can simply remove the milliseconds from the calculation. But a better solution is using a *DispatcherTimer* with an interval of 1 second rather than *CompositionTarget.Rendering*, which always goes at the video refresh rate.

Skew

I discussed earlier that the classes that derive from *Transform* are capable of representing two-dimensional affine transforms, and one of the characteristics of an affine transform is the preservation of parallel lines. However, an affine transform does not necessarily preserve angles between lines. For example, an affine transform is capable of transforming a square to a parallelogram:



In the Windows Runtime, this type of transform is known as a *skew*, but in other graphics environments it might be called a *shear*. The figure is progressively shifted positively or negatively in the horizontal or vertical direction. In a sense, the skew is the most extreme of the affine transforms, but it still preserves a great deal of the original geometry. A skew transform applied to a circle or ellipse never results in anything other than an ellipse:



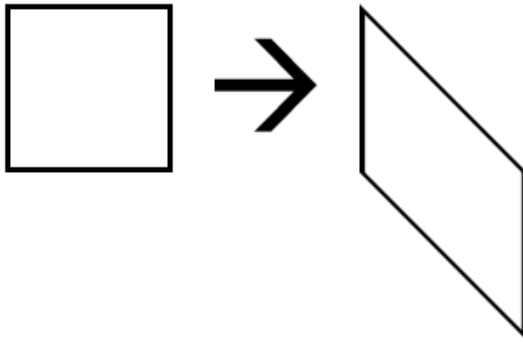
Similarly, a skewed Bézier curve remains a Bézier curve.

The *SkewTransform* has *AngleX* and *AngleY* properties that you set to an angle in degrees. The examples shown were created with a *SkewTransform* with *AngleX* set to 45 degrees, which skews the bottom to the right. Set the angle negative to skew the bottom to the left. For text, negative *AngleX*

values create an oblique effect (similar to italic but without any typographical changes to the characters). Here's *AngleX* set to -30 degrees:

Text \rightarrow *Text*

Nonzero settings of *AngleY* cause skew in the vertical direction. Positive values of *AngleY* cause the right side of figures to skew down:



Negative values cause the right sides to skew up. By default, the upper-left corner of the figure stays in the same location with the skew, but you can change that with *CenterX* and *CenterY* properties or with *RenderTransformOrigin*.

The following program demonstrates what happens when you combine *AngleX* and *AngleY* skewing:

Project: SkewPlusSkew | File: MainPage.xaml (excerpt)

```
<Page ... >
  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <TextBlock Text="SKEW"
      FontSize="288"
      FontWeight="Bold"
      HorizontalAlignment="Center"
      VerticalAlignment="Center"
      RenderTransformOrigin="0.5 0.5">
      <TextBlock.RenderTransform>
        <SkewTransform x:Name="skew" />
      </TextBlock.RenderTransform>
    </TextBlock>
  </Grid>

  <Page.Triggers>
    <EventTrigger>
      <BeginStoryboard>
        <Storyboard SpeedRatio="0.5" RepeatBehavior="Forever">
          <DoubleAnimationUsingKeyFrames Storyboard.TargetName="skew"
```

```

Storyboard.TargetProperty="AngleX">

<!-- Back and forth for 4 seconds -->
<DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="0" />
<LinearDoubleKeyFrame KeyTime="0:0:1" Value="90" />
<LinearDoubleKeyFrame KeyTime="0:0:2" Value="0" />
<LinearDoubleKeyFrame KeyTime="0:0:3" Value="-90" />
<LinearDoubleKeyFrame KeyTime="0:0:4" Value="0" />

<!-- Do nothing for 4 seconds -->
<DiscreteDoubleKeyFrame KeyTime="0:0:8" Value="0" />

<!-- Back and forth for 4 seconds -->
<LinearDoubleKeyFrame KeyTime="0:0:9" Value="90" />
<LinearDoubleKeyFrame KeyTime="0:0:10" Value="0" />
<LinearDoubleKeyFrame KeyTime="0:0:11" Value="-90" />
<LinearDoubleKeyFrame KeyTime="0:0:12" Value="0" />
</DoubleAnimationUsingKeyFrames>

<DoubleAnimationUsingKeyFrames Storyboard.TargetName="skew"
Storyboard.TargetProperty="AngleY">

<!-- Do nothing for 4 seconds -->
<DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="0" />
<DiscreteDoubleKeyFrame KeyTime="0:0:4" Value="0" />

<!-- Back and forth for 4 seconds -->
<LinearDoubleKeyFrame KeyTime="0:0:5" Value="-90" />
<LinearDoubleKeyFrame KeyTime="0:0:6" Value="0" />
<LinearDoubleKeyFrame KeyTime="0:0:7" Value="90" />
<LinearDoubleKeyFrame KeyTime="0:0:8" Value="0" />

<!-- Back and forth for 4 seconds -->
<LinearDoubleKeyFrame KeyTime="0:0:9" Value="-90" />
<LinearDoubleKeyFrame KeyTime="0:0:10" Value="0" />
<LinearDoubleKeyFrame KeyTime="0:0:11" Value="90" />
<LinearDoubleKeyFrame KeyTime="0:0:12" Value="0" />
</DoubleAnimationUsingKeyFrames>
</Storyboard>
</BeginStoryboard>
</EventTrigger>
</Page.Triggers>
</Page>

```

I've set the *SpeedRatio* on the Storyboard to 0.5 so you can better relish the effects, but I'll use the key frame times to discuss what's going on. During the first four seconds, the first animation animates the *AngleX* property to 90 degrees, back to zero, to -90 degrees, and back to zero. During the next four seconds the second animation animates the *AngleY* property between -90 and 90. During the final four seconds, the two animations go together.

You may or may not be surprised that combining *AngleX* and *AngleY* in this way results in rotation:



However, due to the mathematics, the figure gets larger as well.

Skew is often used to give a little 3D-like depth to elements, but it works best in combination with an unskewed element, as I'll demonstrate later in this chapter.

Making an Entrance

Sometimes you want an animated transform to occur on an element when a page is first loaded. For example, an element might slide in from the side and then come to rest, or expand in size, or spin in from above.

It's generally easiest to begin by positioning the element in its final location with no transforms. You can then define the transforms and animations so that the element ends up in that spot. Often you can simply leave out the *To* value of a *DoubleAnimation* on a transform because the *To* value is the same as the pre-animation default value.

This is demonstrated in the *SkewSlideInText* project. As you can see, the *TextBlock* has some transforms defined, but with default values the element simply sits in the center of the display. That's the final location and orientation of the *TextBlock*, and the animations conclude at that spot.

Project: *SkewSlideInText* | File: *MainPage.xaml* (excerpt)

```
<Page ... >
    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <TextBlock Text="Hello!"
            FontSize="192"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            RenderTransformOrigin="0.5 1">
            <TextBlock.RenderTransform>
```

```

        <TransformGroup>
            <SkewTransform x:Name="skew" />
            <TranslateTransform x:Name="translate" />
        </TransformGroup>
    </TextBlock.RenderTransform>
</TextBlock>
</Grid>

<Page.Triggers>
    <EventTrigger>
        <BeginStoryboard>
            <Storyboard>
                <DoubleAnimation Storyboard.TargetName="translate"
                                Storyboard.TargetProperty="X"
                                From="-1000" Duration="0:0:1" />

                <DoubleAnimationUsingKeyFrames
                                Storyboard.TargetName="skew"
                                Storyboard.TargetProperty="AngleX">
                    <DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="15" />
                    <LinearDoubleKeyFrame KeyTime="0:0:1" Value="30" />
                    <EasingDoubleKeyFrame KeyTime="0:0:1.5" Value="0">
                        <EasingDoubleKeyFrame.EasingFunction>
                            <ElasticEase />
                        </EasingDoubleKeyFrame.EasingFunction>
                    </EasingDoubleKeyFrame>
                </DoubleAnimationUsingKeyFrames>
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</Page.Triggers>
</Page>

```

The *DoubleAnimation* applied to the *TranslateTransform* has a *From* value that starts the *TextBlock* 1000 pixels to the left of its final location. The absence of a *To* value means that the animation ends at the pre-animation value, which is 0.

As that's happening, a *DoubleAnimationUsingKeyFrames* makes the skew progress from an *AngleX* value of 15 degrees to 30 degrees, as if the *TextBlock* is being pulled into the center of the screen. The final key frame then animates the *AngleX* back to its pre-animation value of 0, shaking it back and forth in the process.

Transform Mathematics

A transform is a formula that converts a point (x, y) into (x', y') and performs that conversion for all the points of an element.

Suppose a *TranslateTransform* has its *X* and *Y* properties set to *TX* and *TY*. The transform formulas add these translation factors to x and y :

$$\begin{aligned}x' &= x + TX \\y' &= y + TY\end{aligned}$$

If the *ScaleX* and *ScaleY* properties of a *ScaleTransform* are set to *SX* and *SY*, the transform formulas are also fairly obvious:

$$\begin{aligned}x' &= SX \cdot x \\y' &= SY \cdot y\end{aligned}$$

Now that we have the basics down, let's start combining transforms, such as in a *TransformGroup*. If the *ScaleTransform* occurs first, followed by the *TranslateTransform*, the formulas are:

$$\begin{aligned}x' &= SX \cdot x + TX \\y' &= SY \cdot y + TY\end{aligned}$$

But if the translate transform is applied first, followed by the scale transform, it's a little different:

$$\begin{aligned}x' &= SX \cdot (x + TX) \\y' &= SY \cdot (y + TY)\end{aligned}$$

The translation factors are now effectively multiplied by the scaling factors.

The *ScaleTransform* defines *ScaleX* and *ScaleY* properties but also *CenterX* and *CenterY*. I discussed earlier how the center point is used to construct two translations. The first translation is negative, which is then followed by the scale or rotation, followed by positive translation. Suppose *CenterX* and *CenterY* are set to the values *CX* and *CY*. The composite scaling formulas are:

$$\begin{aligned}x' &= SX \cdot (x - CX) + CX \\y' &= SY \cdot (y - CY) + CY\end{aligned}$$

You can easily confirm that the point (*CX*, *CY*) is transformed to the point (*CX*, *CY*), which is the characteristic of the center of scaling: the point that the transform leaves unchanged.

In all the cases so far, *x'* has depended solely on constants multiplied and added to *x*, and *y'* has depended only on constants multiplied and added to *y*. With rotation, it gets a bit messier because *x'* depends on both *x* and *y*, and *y'* also depends on both *x* and *y*. If the *Angle* property of a *RotateTransform* is set to *A*, the transform formulas are:

$$\begin{aligned}x' &= \cos(A) \cdot x - \sin(A) \cdot y \\y' &= \sin(A) \cdot x + \cos(A) \cdot y\end{aligned}$$

These formulas are pretty easy to confirm for simple cases. If *A* is zero, the formulas are just:

$$\begin{aligned}x' &= x \\y' &= y\end{aligned}$$

If *A* is 90 degrees, the sine is 1, and the cosine is 0, so

$$x' = -y$$

$$y' = x$$

For example, the point (1, 0) is transformed to (0, 1), and (0, 1) is transformed to (−1, 0). When A is 180 degrees, the sine is 0 and the cosine is −1, so

$$x' = -x$$

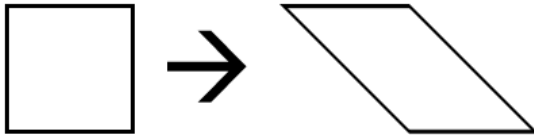
$$y' = -y$$

It's a reflection around the origin, and you can get the same effect with a *ScaleTransform* with *ScaleX* and *ScaleY* both set to −1. When A is 270 degrees,

$$x' = y$$

$$y' = -x$$

Here's the first diagram of a skew transform shown earlier:



The transform formulas for this particular skew (*AngleX* set to 45 degrees) are

$$x' = x + y$$

$$y' = y$$

When *y* equals 0 (at the top of the figure), *x'* simply equals *x* and *y'* equals *y*. But as you move down the figure, *y* gets larger, so *x'* becomes increasingly greater than *x*. The generalized formulas for *SkewTransform* when *AngleX* is set to *AX* and *AngleY* is set to *AY* are

$$x' = x + \sin(AX) \cdot y$$

$$y' = \sin(AY) \cdot x + y$$

When you begin exploring combinations of rotation with other transforms, this type of notation starts to become rather clumsy. Fortunately, matrix algebra comes to the rescue. If individual transforms can be represented by matrices, transforms can be combined through the well-established process of matrix multiplication.

Let's represent a point (*x*, *y*) as a 2×1 matrix:

$$\begin{bmatrix} x & y \end{bmatrix}$$

And let's represent the transform as a 2×2 matrix:

$$\begin{bmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{bmatrix}$$

Applying the transform can then be represented with a matrix multiplication. The result is the transformed point:

$$\begin{bmatrix} x & y \end{bmatrix} \times \begin{bmatrix} M11 & M12 \\ M21 & M22 \end{bmatrix} = \begin{bmatrix} x' & y' \end{bmatrix}$$

The rules of matrix multiplication imply the following formulas:

$$\begin{aligned} x' &= M11 \cdot x + M21 \cdot y \\ y' &= M12 \cdot x + M22 \cdot y \end{aligned}$$

This process works for scaling if *M11* is the *ScaleX* value and *M22* is the *ScaleY* value, and *M21* and *M12* are zero. It also works for rotation and skewing, which both involve factors that are multiplied by *x* and *y*.

But it does not work for translation. The translation formulas look like this:

$$\begin{aligned} x' &= x + TX \\ y' &= y + TY \end{aligned}$$

These translation factors are added in by themselves, not multiplied by *x* or *y*. How can we represent a generalized transform by a matrix if it doesn't allow for translation, which is the arguably the simplest type of transform of them all?

The interesting solution is to introduce a third dimension. In addition to the *X* and *Y* axes on the plane of the computer screen, a conceptual *Z* axis extends out from the screen. Let's assume that we're still drawing on a two-dimensional plane, but that plane exists in 3D space with a constant *Z* coordinate equal to 1.

This means that the point (*x*, *y*) is actually the point (*x*, *y*, 1) and we can represent it as a 3×1 matrix:

$$\begin{bmatrix} x & y & 1 \end{bmatrix}$$

The matrix transform is now a 3×3 matrix, and the multiplication looks like this:

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} M11 & M12 & M13 \\ M21 & M22 & M23 \\ M31 & M32 & M33 \end{bmatrix} = \begin{bmatrix} x' & y' & z' \end{bmatrix}$$

The formulas implied by the matrix multiplication are:

$$\begin{aligned} x' &= M11 \cdot x + M21 \cdot y + M31 \\ y' &= M12 \cdot x + M22 \cdot y + M32 \\ z' &= M13 \cdot x + M23 \cdot y + M33 \end{aligned}$$

This is partial success because the transform formulas now include translation factors of *M31* and *M32*. These two numbers aren't multiplied by *x* or *y*.

But it's not a total success because *z'* is generally not equal to 1, which means that we've shifted off the plane where *z* always equals 1. One way to get back to that plane is simply to set all those errant *z* values to 1. But shouldn't points that are transformed a long distance away from the plane where *z* equals 1 be distinguished from those that end up close to it?

One clever way to get the *z* values to 1 without simply ignoring them is to take the 3×1 matrix

result and divide all three coordinates by z' :

$$\left(\frac{x'}{z'}, \frac{y'}{z'}, \frac{z'}{z'}\right) = \left(\frac{x'}{z'}, \frac{y'}{z'}, 1\right)$$

But now another problem has been introduced. What happens when z' is zero? We get coordinates of infinity. If we want to avoid infinite coordinates, z' cannot be allowed to be zero. Indeed, we can avoid dividing by z' entirely if we ensure that z' is always equal to 1.

It's possible to do that by setting $M13$ and $M23$ in the matrix to 0 and $M33$ to 1. Now the transform is represented by formulas that remain entirely in the same plane:

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} M11 & M12 & 0 \\ M21 & M22 & 0 \\ M31 & M32 & 1 \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix}$$

This is the standard matrix representation of the two-dimensional affine transform. (Allowing other values in the third column results in a nonaffine transform. Because such a matrix is capable of transforming parallel lines to nonparallel lines, it is sometimes also called a *taper* transform.)

With the notation I was using earlier, the *ScaleTransform* where *ScaleX* is set to SX and *ScaleY* is set to SY is

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} SX & 0 & 0 \\ 0 & SY & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix}$$

A *TranslateTransform* with TX and TY factors is

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ TX & TY & 1 \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix}$$

A *ScaleTransform* with center (CX, CY) is effectively a multiplication of three 3×3 transforms:

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -CX & -CY & 1 \end{bmatrix} \times \begin{bmatrix} SX & 0 & 0 \\ 0 & SY & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ CX & CY & 1 \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix}$$

Similarly, a *RotateTransform* with angle A and center (CX, CY) also informs three transforms:

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -CX & -CY & 1 \end{bmatrix} \times \begin{bmatrix} \cos(A) & \sin(A) & 0 \\ -\sin(A) & \cos(A) & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ CX & CY & 1 \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix}$$

And here's the *SkewTransform* with angles AX and AY and a center:

$$\begin{bmatrix} x & y & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -CX & -CY & 1 \end{bmatrix} \times \begin{bmatrix} 1 & \sin(AX) & 0 \\ \sin(AY) & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ CX & CY & 1 \end{bmatrix} = \begin{bmatrix} x' & y' & 1 \end{bmatrix}$$

A well-known property of matrix multiplication is that it is not commutative. The order of multiplication makes a difference. This has already been demonstrated with translation and scaling. If the translation comes first, the translation factors themselves are also scaled by the scaling factors.

However, certain types of transforms can be safely multiplied in any order:

- Multiple *TranslateTransform* objects. The total translation is the sum of the individual translation factors.
- Multiple *ScaleTransform* objects with the same scaling center. The total scaling is the product of the individual scaling factors.
- Multiple *RotateTransforms* with the same rotation center. The total rotation is the sum of the angles of the individual rotations.

In addition, if a *ScaleTransform* has equal *ScaleX* and *ScaleY* properties, it can be multiplied by a *RotateTransform* or a *SkewTransform* in either order.

The Windows Runtime defines a *Matrix* structure that has six properties that correspond to the cells of the matrix like this:

$$\begin{vmatrix} M11 & M12 & 0 \\ M21 & M22 & 0 \\ OffsetX & OffsetY & 1 \end{vmatrix}$$

The last row of this matrix is fixed. You cannot use this *Matrix* structure to define a taper transform or anything “crazier” than what you’ve already seen. *OffsetX* and *OffsetY* are the translation properties. The default values for *M11* and *M22* are 1, and the default values for the other four properties are zero. That’s the identity matrix with a diagonal of 1s:

$$\begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

The *Matrix* structure has a static *Identity* property that returns this value and an *IsIdentity* property that returns *true* if the *Matrix* value is the identity matrix.

Along with the “easy” *Transform* derivatives like *ScaleTransform* and *RotateTransform*, there is also the low-level alternative *MatrixTransform*, which has a property of type *Matrix*. If you know the matrix transform you want, you can specify it directly in six numbers in the order *M11*, *M12*, *M21*, *M22*, *OffsetX*, *OffsetY*. Here’s one way to set this transform:

```
<TextBlock ... >
  <TextBlock.RenderTransform>
    <MatrixTransform Matrix="10 0 0 5 0 100" />
  </TextBlock.RenderTransform>
</TextBlock>
```

This transform scales in the horizontal direction by a factor of 10 (*M11*) and in the vertical direction by a factor of 5 (*M22*), and then it shifts the *TextBlock* down by 100 pixels (*OffsetY*). But you can also set the transform directly to the *RenderTransform* property:

```
<TextBlock ...
  RenderTransform="10 0 0 5 0 100"
... />
```

The preview design view in the version of Microsoft Visual Studio I'm using for this chapter isn't particularly amenable to this syntax, but it's no problem for the compiler.

Using this implicit form of *MatrixTransform* is handy for several common rotation transforms that are shown in the following program. Each *TextBlock* displays the transform applied to it:

Project: CommonMatrixTransforms | File: MainPage.xaml (excerpt)

```
<Page ... >
  <Page.Resources>
    <Style TargetType="TextBlock">
      <Setter Property="FontSize" Value="24" />
      <Setter Property="RenderTransformOrigin" Value="0 0.5" />
    </Style>
  </Page.Resources>

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

    <!-- Move origin to center -->
    <Canvas HorizontalAlignment="Center"
      VerticalAlignment="Center">

      <TextBlock Text="  RenderTransform='1 0 0 1 0 0'"
        RenderTransform="1 0 0 1 0 0" />

      <TextBlock Text="  RenderTransform='.7 .7 -.7 .7 0 0'"
        RenderTransform=".7 .7 -.7 .7 0 0" />

      <TextBlock Text="  RenderTransform='0 1 -1 0 0 0'"
        RenderTransform="0 1 -1 0 0 0" />

      <TextBlock Text="  RenderTransform='-.7 .7 -.7 -.7 0 0'"
        RenderTransform="-.7 .7 -.7 -.7 0 0" />

      <TextBlock Text="  RenderTransform='-1 0 0 -1 0 0'"
        RenderTransform="-1 0 0 -1 0 0" />

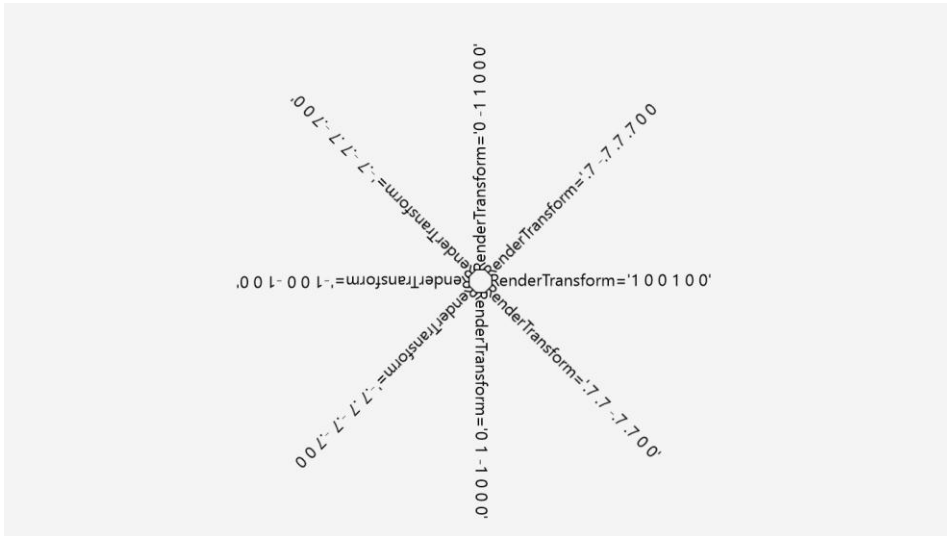
      <TextBlock Text="  RenderTransform='-.7 -.7 .7 -.7 0 0'"
        RenderTransform="-.7 -.7 .7 -.7 0 0" />

      <TextBlock Text="  RenderTransform='0 -1 1 0 0 0'"
        RenderTransform="0 -1 1 0 0 0" />

      <TextBlock Text="  RenderTransform='.7 -.7 .7 .7 0 0'"
        RenderTransform=".7 -.7 .7 .7 0 0" />

    </Canvas>
  </Grid>
</Page>
```

The frequent references to .7 should more accurately be .707, the sine and cosine of 45 degrees and (not coincidentally) half the square root of 2. These eight transforms result in each *TextBlock* being rotated an additional 45 degrees from the previous one:



If you're working in code, the *Matrix* structure has a *Transform* method that applies the transform to a *Point* value and returns the transformed *Point*.

However, the *Matrix* structure is missing many amenities. It's missing a multiplication operator that would allow you to easily perform your own matrix multiplications in code. You could write the multiplication code yourself, or you can use *TransformGroup*, which internally performs matrix multiplications and makes the result available in a read-only *Value* property of type *Matrix*. If you need to perform matrix multiplications, you can create a *TransformGroup* in code, add a couple initialized *Transform* derivatives to it, and access the *Value* property.

I'll have an important example in Chapter 13. Matrix transform calculations become essential in computing scaling and rotation centers when using touch to manipulate on-screen objects.

The CompositeTransform

When combining transforms of various types, the order makes a difference. In practical use, however, it turns out that you usually want various transforms to be applied in a fairly standard order.

For example, suppose you want to rotate, scale, and translate an element. *ScaleTransform* usually comes first because generally you want to specify the scaling in terms of the unrotated element. The *TranslateTransform* comes last because generally you don't want scaling or rotation to affect the translation factors. That means the *RotateTransform* is in the middle. The order is: scale, rotate, translate.

If that's the order you want, you can use *CompositeTransform*. *CompositeTransform* has a bunch of properties defined to perform transforms in the order:

- Scale
- Skew
- Rotate
- Translate

The properties are

- *CenterX* and *CenterY* for the center of scaling, skewing, and rotation
- *ScaleX* and *ScaleY*
- *SkewX* and *SkewY*
- *Rotation*
- *TranslateX* and *TranslateY*

Here's a little program that uses a *CompositeTransform* as a convenient way to combine scaling and skewing:

Project: TiltedShadow | File: MainPage.xaml (excerpt)

```
<Page ... >
  <Page.Resources>
    <Style TargetType="TextBlock">
      <Setter Property="Text" Value="quirky" />
      <Setter Property="FontFamily" Value="Times New Roman" />
      <Setter Property="FontSize" Value="192" />
      <Setter Property="HorizontalAlignment" Value="Center" />
      <Setter Property="VerticalAlignment" Value="Center" />
    </Style>
  </Page.Resources>

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <!-- Shadow TextBlock -->
    <TextBlock Foreground="Gray"
      RenderTransformOrigin="0 1">
      <TextBlock.RenderTransform>
        <CompositeTransform ScaleY="1.5" SkewX="-60" />
      </TextBlock.RenderTransform>
    </TextBlock>

    <!-- TextBlock with all styled properties -->
    <TextBlock />
  </Grid>
</Page>
```

The XAML instantiates two *TextBlock* elements with mostly the same properties specified in the *Style*, including the *Text* property, and as far as the layout system is concerned, they both occupy the same space. The bottom one is gray, however, and has scale and skew transforms applied:

The word "quirky" is displayed in a black serif font. Behind the text is a semi-transparent, skewed shadow of the same word. The shadow is tilted upwards and to the right, creating a sense of depth and movement. The background is a light gray.

Notice that the *RenderTransformOrigin* is set to the point (0, 1), which means that the transform is relative to the lower-left corner. However, that point could be specified as (1, 1) or anything in between, and it would work the same. All that's required is that the two *TextBlock* elements share the same bottom edge. A *ScaleY* of 1.5 is applied to increase the height of the shadow by 50%. The *SkewX* value of -60 degrees should shift the bottom to the left, but since the bottom is the center of scaling and skewing, the top is skewed to the right.

Look close and you'll notice that the bottoms of the descenders don't quite meet up. That's because the *TextBlock* actually extends a little below the bottom of the descenders. Change the *RenderTransformOrigin* to (0, 0.96) for a somewhat better match.

What if you wanted a similar effect with text with no descenders? Here's an example:



The problem is that you need to come up with a *RenderTransformOrigin* with a *Y* value equal to the relative height of the text above the baseline. That's dependent on the font. For this particular screenshot, I experimented until I came up with (0, 0.78), but that's appropriate only for the Times New Roman font. To do something like this in a generalized way, you'd need access to font metrics, which are available to a Windows 8 application only through DirectX. I'll show you how to do that in a later chapter.

Geometry Transforms

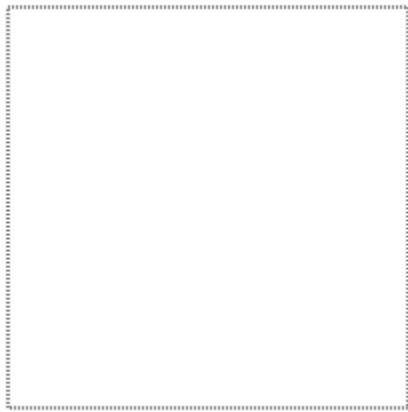
The *Geometry* class defines a *Transform* property, which naturally raises the question: What is the difference between applying a transform to a *Path* element and applying a transform to a *Geometry* object that is set to the *Data* property of a *Path*?

The big difference is that a *Transform* applied to the *RenderTransform* property of a *Path* increases the width of the strokes, whereas a *Transform* applied to the *Geometry* does not.

Here's a *Path* element based on a *RectangleGeometry* with a height and width of 10 but with a transform applied to the geometry to increase it by a factor of 20:

```
<Path Stroke="Black"
      StrokeThickness="1"
      StrokeDashArray="1 1">
  <Path.Data>
    <RectangleGeometry Rect="0 0 10 10"
                      Transform="20 0 0 20 0 0" />
  </Path.Data>
</Path>
```

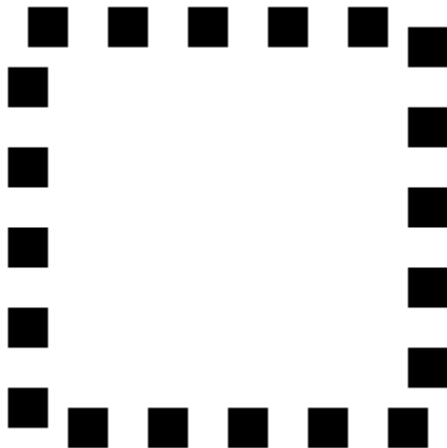
The result is as if the *Rect* value in the *RectangleGeometry* had a height and width of 200:



This XAML has the same initial *RectangleGeometry* but the transform is applied to the *Path*:

```
<Path Stroke="Black"
      StrokeThickness="1"
      StrokeDashArray="1 1"
      RenderTransform="20 0 0 20 0 0">
  <Path.Data>
    <RectangleGeometry Rect="0 0 10 10" />
  </Path.Data>
</Path>
```

The result is quite different:



To the layout system, however, these elements appear to be identical. Both *Path* elements are perceived to have a width and height of 10.

Brush Transforms

The *Brush* class defines two transform-related properties: *Transform* and *RelativeTransform*, which are distinguished by letting you specify translation factors based on the pixel size of the brush or relative to its size. *RelativeTransform* is often easier to use unless you've given the brushed element a specific pixel size.

Here's a program that replicates the *RainbowEight* program from Chapter 3, "Basic Event Handling," but using an animated brush transform. I've substituted a *Path* rendition of the '8' rather than using a *TextBlock* because I couldn't get the brush to repeat with the *SpreadMethod* property of *Repeat* for a *TextBlock*.

Project: *RainbowEightTransform* | File: *MainPage.xaml* (excerpt)

```
<Page ... >
  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Viewbox>
      <Path StrokeThickness="50"
            Margin="0 25 0 0">
        <Path.Data>
          <PathGeometry>
            <PathFigure StartPoint="110 0">
              <ArcSegment Size="90 90" Point="110 180"
                          SweepDirection="Clockwise" />
              <ArcSegment Size="110 110" Point="110 400"
                          SweepDirection="Counterclockwise" />
              <ArcSegment Size="110 110" Point="110 180"
                          SweepDirection="Counterclockwise" />
              <ArcSegment Size="90 90" Point="110 0"
                          SweepDirection="Clockwise" />
            </PathFigure>
          </PathGeometry>
        </Path.Data>
        <Path.Stroke>
          <LinearGradientBrush StartPoint="0 0" EndPoint="1 1"
                              SpreadMethod="Repeat">
            <LinearGradientBrush.RelativeTransform>
              <TranslateTransform x:Name="translate" />
            </LinearGradientBrush.RelativeTransform>

            <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" />
          </LinearGradientBrush>
        </Path.Stroke>
      </Path>
    </Viewbox>
  </Grid>
</Page>
```

```

</Grid>

<Page.Triggers>
  <EventTrigger>
    <BeginStoryboard>
      <Storyboard>
        <DoubleAnimation Storyboard.TargetName="translate"
          Storyboard.TargetProperty="Y"
          EnableDependentAnimation="True"
          From="0" To="-1.36" Duration="0:0:10"
          RepeatBehavior="Forever" />
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger>
</Page.Triggers>
</Page>

```

Here's the image:



There's a "magic number" in the markup. It's the *To* value of the *DoubleAnimation*. That's the value that is applied to the *Y* property of the *TranslateTransform*, and it was chosen so that the translated brush with that value is identical to the untranslated brush. The magic number, you can see, is -1.36 , and I'm sure you want to know where it came from.

If the *LinearGradientBrush* went from top to bottom—with *StartPoint* of (0, 0) and an *EndPoint* of (0, 1)—this *To* value would simply be -1 . If the gradient went from left to right—with a *StartPoint* of (0, 0) and an *EndPoint* of (1, 0)—the *X* property of the *TranslateTransform* would be the animation target, and again a *To* value of 1 or -1 would be used.

But when the gradient goes from one corner to the opposite—with the default *StartPoint* of (0, 0) and *EndPoint* of (1, 1)—then that's not quite right. When covering a *Path* element with a brush, the

Windows Runtime computes a bounding rectangle that includes the geometric size of the element plus the stroke width. The brush is then stretched to this bounding rectangle:



The gradient line runs along the diagonal, which means that lines of constant color are at right angles to this gradient line.

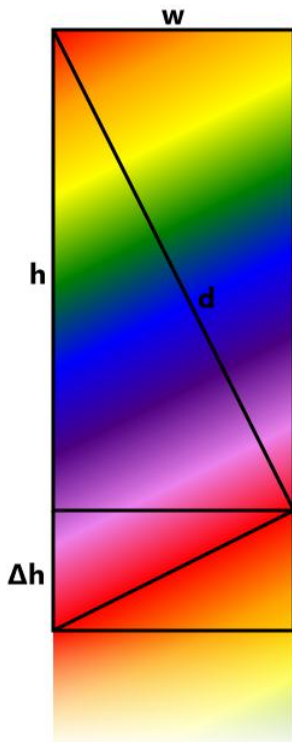
When the brush has a *SpreadMethod* of *Repeat*, the brush conceptually repeats beyond the specified offsets. This *SpreadMethod* setting is useful when applying a *TranslateTransform* to the brush because the brush seems to repeat regardless how it's shifted.

If you shift this brush up by the height of the element (that is, a Y value of -1 in the *TranslateTransform*), the bottom edge of the untransformed brush becomes the top edge of the transformed brush, but you can see the result in the following image, and it's not the same as the previous image:



To get a smooth animation, you need to shift it up some more. But by how much?

Let's extend this figure to show part of the repeating brush, and let's label the width of the element with ' w ', the height with ' h ', the diagonal with ' d ', and the increase in height with ' Δh '.



You can figure out Δh in a variety of ways, but perhaps the most straightforward is based on similar triangles:

$$\frac{d}{h} = \frac{h + \Delta h}{d}$$

from which it's easy to derive

$$\Delta h = \frac{w^2}{h}$$

or, the number we really want:

$$\frac{h + \Delta h}{h} = 1 + \left(\frac{w}{h}\right)^2$$

Try plugging in the numbers from the *Path* shown above. You'll need to add the *StrokeThickness* to the width and heights of the geometry. With a width of 270 and a height of 450, then Δh is 162. Add that to h and divide by h , and that's the magic number of 1.36.

Would you like to hear about an easier approach? Simply use two *DoubleAnimation* objects in the *Storyboard*, where one targets the *Y* property and the other targets *X*. Set the *To* value of both to -1 and the brush shifts both up and left with every cycle.

Dude, Where's My Element?

Earlier I mentioned that a computed *Matrix* value is available from *TransformGroup*, but it's not available from other sources where you might expect it. For example, *GeneralTransform*—from which *Transform* and all the other transform classes derives—might be expected to have a *Matrix* property, but it does not.

However, the *GeneralTransform* class has a *TransformPoint* method and a *TransformBounds* method, which applies the transform to a *Rect* value, and these actually come in handy in some circumstances.

Suppose an element is a child of a panel. The panel is responsible for positioning the element relative to itself, but the element could also have a *RenderTransform* applied with translation, scale, rotation, or skewing. For purposes of hit-testing, the location and orientation of that element is known internal to the system. But can your own program find where the element is actually located?

Yes! The essential (but obscure) method is defined by *UIElement* and called *TransformToVisual*. Generally you'll call this method on an element with an argument that is the element's parent or some other ancestor:

```
GeneralTransform xform = element.TransformToVisual(parent);
```

The *GeneralTransform* object returned from the method maps from *element* coordinates to *parent*

coordinates. But you can't actually see what this transform is! It won't give you a *Matrix* value. All you can do with it is call *TransformPoint* or *TransformBounds* or use the *Inverse* property. But this is often all you need.

Here's a XAML file that animates properties of a *CompositeTransform* to make a *TextBlock* go crazy all over the screen:

Project: WheresMyElement | File: MainPage.xaml (excerpt)

```
<Page ... >
    <Grid Name="contentGrid"
        Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <TextBlock Name="txtblk"
            Text="Tap to Find"
            FontSize="96"
            HorizontalAlignment="Center"
            VerticalAlignment="Center"
            RenderTransformOrigin="0.5 0.5">
            <TextBlock.RenderTransform>
                <CompositeTransform x:Name="transform" />
            </TextBlock.RenderTransform>
        </TextBlock>

        <Polygon Name="polygon" Stroke="Blue" />
        <Path Name="path" Stroke="Red" />
    </Grid>

    <Page.Triggers>
        <EventTrigger>
            <BeginStoryboard>
                <Storyboard x:Name="storyboard">
                    <DoubleAnimation Storyboard.TargetName="transform"
                        Storyboard.TargetProperty="TranslateX"
                        From="-300" To="300" Duration="0:0:2.11"
                        AutoReverse="True" RepeatBehavior="Forever" />
                    <DoubleAnimation Storyboard.TargetName="transform"
                        Storyboard.TargetProperty="TranslateY"
                        From="-300" To="300" Duration="0:0:2.23"
                        AutoReverse="True" RepeatBehavior="Forever" />
                    <DoubleAnimation Storyboard.TargetName="transform"
                        Storyboard.TargetProperty="Rotation"
                        From="0" To="360" Duration="0:0:2.51"
                        AutoReverse="True" RepeatBehavior="Forever" />
                    <DoubleAnimation Storyboard.TargetName="transform"
                        Storyboard.TargetProperty="ScaleX"
                        From="1" To="2" Duration="0:0:2.77"
                        AutoReverse="True" RepeatBehavior="Forever" />
                    <DoubleAnimation Storyboard.TargetName="transform"
                        Storyboard.TargetProperty="ScaleY"
                        From="1" To="2" Duration="0:0:3.07"
                        AutoReverse="True" RepeatBehavior="Forever" />
                    <DoubleAnimation Storyboard.TargetName="transform"
                        Storyboard.TargetProperty="SkewX"
                        From="-30" To="30" Duration="0:0:3.31"
```



```

                                AutoReverse="True" RepeatBehavior="Forever" />
        <DoubleAnimation Storyboard.TargetName="transform"
                        Storyboard.TargetProperty="SkewY"
                        From="-30" To="30" Duration="0:0:3.53"
                        AutoReverse="True" RepeatBehavior="Forever" />
    </Storyboard>
</BeginStoryboard>
</EventTrigger>
</Page.Triggers>
</Page>

```

Notice that the *Grid* also contains a blue *Polygon* and a red *Path*, but with no actual coordinate points.

The code-behind file uses the *Tapped* event to take a “snapshot” of the *TextBlock* by calling *TransformToVisual* and pausing the *Storyboard* (resumed on the next tap). *TransformToVisual* returns a *GeneralTransform* object that describes the relationship between the *TextBlock* and the *Grid*. The program uses this to transform the four corners of the *TextBlock* to *Grid* coordinates for the *Polygon*, which effectively draws a rectangle around the *TextBlock*:

Project: WheresMyElement | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    bool storyboardPaused;

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

    protected override void OnTapped(TappedRoutedEventArgs args)
    {
        if (storyboardPaused)
        {
            storyboard.Resume();
            storyboardPaused = false;
            return;
        }

        GeneralTransform xform = txtblk.TransformToVisual(contentGrid);

        // Draw blue polygon around element
        polygon.Points.Clear();
        polygon.Points.Add(xform.TransformPoint(new Point(0, 0)));
        polygon.Points.Add(xform.TransformPoint(new Point(txtblk.ActualWidth, 0)));
        polygon.Points.Add(xform.TransformPoint(new Point(txtblk.ActualWidth,
                                                            txtblk.ActualHeight)));
        polygon.Points.Add(xform.TransformPoint(new Point(0, txtblk.ActualHeight)));

        // Draw red bounding box
        path.Data = new RectangleGeometry
        {
            Rect = xform.TransformBounds(new Rect(new Point(0, 0), txtblk.DesiredSize))
        };
    }
}

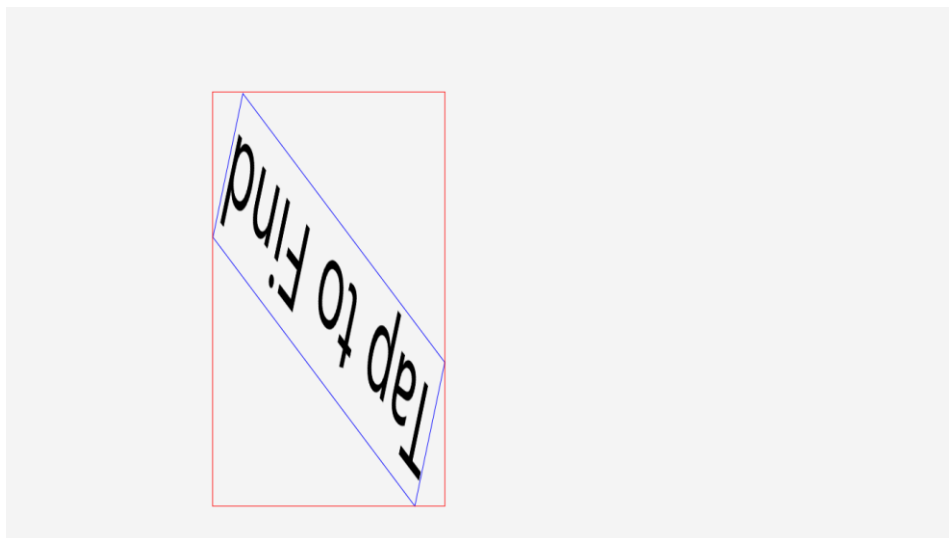
```

```

        storyboard.Pause();
        storyboardPaused = true;
        base.OnTapped(args);
    }
}

```

The call to *TransformBounds* obtains something a little different: a rectangle describing a boundary box with sides parallel to the horizontal and vertical large enough to encompass the element. This is drawn in red:



That boundary rectangle is easily calculable from the maximum and minimum X and Y coordinates of the transformed four corners, but it's nice to have it conveniently available.

Projection Transforms

Earlier in this chapter I discussed why a two-dimensional graphics transform is mathematically described by a 3×3 matrix and requires a flirtation with the third dimension. By a similar analogy, a three-dimensional graphics transform is expressed by a 4×4 matrix, and the Windows Runtime has one.

The *Windows.UI.Xaml.Media.Media3D* namespace contains exactly two items: a *Matrix3D* structure available for all programmers, and a *Matrix3DHelper* class that's mostly of value to C++ programmers because they can't access any of the methods defined by *Matrix3D*. The properties of *Matrix3D* are analogous to those in the regular *Matrix* structure except that every cell of the matrix is available:

$$\begin{bmatrix} M11 & M12 & M13 & M14 \\ M21 & M22 & M23 & M24 \\ M31 & M32 & M33 & M34 \\ OffsetX & OffsetY & OffsetZ & M44 \end{bmatrix}$$

However, few programmers ever really get close to this matrix. Most of them are content to use the *PlaneProjection* class that I briefly demonstrated at the beginning of this chapter.

PlaneProjection is intended mostly to let you rotate two-dimensional elements in three-dimensional space. Rotation in 3D space is always around an axis, and *PlaneProjection* lets you rotate an element around a horizontal axis (using the *RotationX* property), the vertical axis (with *RotationY*), or the Z axis that conceptually pokes out of the screen. Rotation around the Z axis is simply two-dimensional rotation, so that's not nearly as exciting as the other two.

You can anticipate the direction of rotation using the right-hand rule: Point the thumb of your right hand in the direction of the positive axis. That's right for the X axis, down for the Y axis, and out of the screen for Z. The curve of your fingers indicates the direction of rotation for positive angles. *PlaneProjection* applies the rotations in the order X, Y, and Z, but generally you'll be using only one of them.

With a little discreet use of *PlaneProjection*, you can have elements swing into view or even conceptually flip over to reveal something on the "other side" (as I'll demonstrate shortly).

And then there's the not-so-discreet uses. The ThreeDeeSpinningText program lets you independently animate the *RotationX*, *RotationY*, and *RotationZ* properties to spin a *TextBlock* around in 3D space. Here's the XAML file with a group of Begin/Stop and Play/Pause buttons at the bottom:

Project: ThreeDeeSpinningText | File: MainPage.xaml (excerpt)

```
<Page ... >
  <Page.Resources>
    <Storyboard x:Key="xAxisAnimation" RepeatBehavior="Forever">
      <DoubleAnimation Storyboard.TargetName="projection"
        Storyboard.TargetProperty="RotationX"
        From="0" To="360" Duration="0:0:1.9" />
    </Storyboard>

    <Storyboard x:Key="yAxisAnimation" RepeatBehavior="Forever">
      <DoubleAnimation Storyboard.TargetName="projection"
        Storyboard.TargetProperty="RotationY"
        From="0" To="360" Duration="0:0:3.1" />
    </Storyboard>

    <Storyboard x:Key="zAxisAnimation" RepeatBehavior="Forever">
      <DoubleAnimation Storyboard.TargetName="projection"
        Storyboard.TargetProperty="RotationZ"
        From="0" To="360" Duration="0:0:4.3" />
    </Storyboard>
  </Page.Resources>

  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
      <RowDefinition Height="*" />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
```

```

<TextBlock Text="3D-ish"
    FontSize="384"
    HorizontalAlignment="Center"
    VerticalAlignment="Center">
    <TextBlock.Projection>
        <PlaneProjection x:Name="projection" />
    </TextBlock.Projection>
</TextBlock>

<!-- Control Panel -->
<Grid Grid.Row="1" HorizontalAlignment="Center">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>

    <Grid.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="FontSize"
                Value="{StaticResource ControlContentThemeFontSize}" />
            <Setter Property="VerticalAlignment" Value="Center" />
        </Style>

        <Style TargetType="Button">
            <Setter Property="Width" Value="120" />
            <Setter Property="Margin" Value="12" />
        </Style>
    </Grid.Resources>

    <TextBlock Text="X Axis: " Grid.Row="0" Grid.Column="0"
        Tag="xAxisAnimation" />
    <Button Content="Begin" Grid.Row="0" Grid.Column="1"
        Click="OnBeginStopButton" />
    <Button Content="Pause" Grid.Row="0" Grid.Column="2"
        IsEnabled="False"
        Click="OnPauseResumeButton" />

    <TextBlock Text="Y Axis: " Grid.Row="1" Grid.Column="0"
        Tag="yAxisAnimation" />
    <Button Content="Begin" Grid.Row="1" Grid.Column="1"
        Click="OnBeginStopButton" />
    <Button Content="Pause" Grid.Row="1" Grid.Column="2"
        IsEnabled="False"
        Click="OnPauseResumeButton" />

    <TextBlock Text="Z Axis: " Grid.Row="2" Grid.Column="0"
        Tag="zAxisAnimation" />

```

```

        <Button Content="Begin" Grid.Row="2" Grid.Column="1"
            Click="OnBeginStopButton" />
        <Button Content="Pause" Grid.Row="2" Grid.Column="2"
            IsEnabled="False"
            Click="OnPauseResumeButton" />
    </Grid>
</Grid>
</Page>

```

The durations of the individual *DoubleAnimation* objects all have somewhat different times to avoid patterns when they're all going at once. The buttons in the code-behind file use the *Begin*, *Stop*, *Pause*, and *Resume* methods of *Storyboard* to control the activity:

Project: ThreeDeeSpinningText | File: MainPage.xaml.cs (excerpt)

```

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

    void OnBeginStopButton(object sender, RoutedEventArgs args)
    {
        Button btn = sender as Button;
        string key = GetSibling(btn, -1).Tag as string;
        Storyboard storyboard = this.Resources[key] as Storyboard;
        Button pauseResumeButton = GetSibling(btn, 1) as Button;
        pauseResumeButton.Content = "Pause";

        if (btn.Content as string == "Begin")
        {
            storyboard.Begin();
            btn.Content = "Stop";
            pauseResumeButton.IsEnabled = true;
        }
        else
        {
            storyboard.Stop();
            btn.Content = "Begin";
            pauseResumeButton.IsEnabled = false;
        }
    }

    void OnPauseResumeButton(object sender, RoutedEventArgs args)
    {
        Button btn = sender as Button;
        string key = GetSibling(btn, -2).Tag as string;
        Storyboard storyboard = this.Resources[key] as Storyboard;

        if (btn.Content as string == "Pause")
        {
            storyboard.Pause();
            btn.Content = "Resume";
        }
    }
}

```

```

        else
        {
            storyboard.Resume();
            btn.Content = "Pause";
        }
    }

FrameworkElement GetSibling(FrameworkElement element, int relativeIndex)
{
    Panel parent = element.Parent as Panel;
    int index = parent.Children.IndexOf(element);
    return parent.Children[index + relativeIndex] as FrameworkElement;
}
}

```

And here's a sample image:



The *PlaneProjection* class has a bunch of additional properties. The *CenterOfRotationX* and *CenterOfRotationY* properties are both in coordinates relative to the element. The default values are 0.5, which is the center of the element and usually what you want. The *CenterOfRotationZ* property is in pixels with a default value of 0, corresponding to the surface of the screen. For purposes of internal calculations, it is assumed that the "camera" (or you, the user) is viewing the screen from a distance of 1000 pixels, or about 10 inches.

PlaneProjection also defines three *LocalOffset* properties for the X, Y, and Z dimensions and three *GlobalOffset* properties. These are translation factors in pixels. The *LocalOffset* values are applied before the rotation, and the *GlobalOffset* values are applied after the rotation. Most often, you'll be setting the *GlobalOffset* properties.

Here's a little example of a "flip panel," a technique that was once quite difficult and involved real 3D programming. The idea is that you have a little collection of controls on a panel and a way to flip

that panel over to use a different set of controls. In this example, I've represented the front and "back" of this panel with two *Grid* panels with different background colors containing a *TextBlock* each:

Project: TapToFlip | File: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid HorizontalAlignment="Center"
          VerticalAlignment="Center"
          Tapped="OnGridTapped">

        <Grid Name="grid1"
              Background="Cyan"
              Canvas.ZIndex="1">
            <TextBlock Text="Hello"
                      HorizontalAlignment="Center"
                      FontSize="192" />
        </Grid>

        <Grid Name="grid2"
              Background="Yellow"
              Canvas.ZIndex="0">
            <TextBlock Text="Windows 8"
                      FontSize="192" />
        </Grid>

        <Grid.Projection>
            <PlaneProjection x:Name="projection" />
        </Grid.Projection>
    </Grid>
</Grid>
```

Notice the *Canvas.ZIndex* settings. These ensure that the *grid1* is visually on top of *grid2* even though it comes earlier in the children collection of their mutual parent.

The *Resources* section contains two *Storyboard* definitions, one to flip and the other to flip back:

Project: TapToFlip | File: MainPage.xaml (excerpt)

```
<Page.Resources>
    <Storyboard x:Key="flipStoryboard">
        <DoubleAnimationUsingKeyFrames
            Storyboard.TargetName="projection"
            Storyboard.TargetProperty="RotationY">
            <DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="0" />
            <LinearDoubleKeyFrame KeyTime="0:0:0.99" Value="90" />
            <DiscreteDoubleKeyFrame KeyTime="0:0:1.01" Value="-90" />
            <LinearDoubleKeyFrame KeyTime="0:0:2" Value="0" />
        </DoubleAnimationUsingKeyFrames>

        <DoubleAnimation Storyboard.TargetName="projection"
            Storyboard.TargetProperty="GlobalOffsetZ"
            From="0" To="-1000" Duration="0:0:1"
            AutoReverse="True" />

        <ObjectAnimationUsingKeyFrames
```

```

        Storyboard.TargetName="grid1"
        Storyboard.TargetProperty="(Canvas.ZIndex)">
        <DiscreteObjectKeyFrame KeyTime="0:0:0" Value="1" />
        <DiscreteObjectKeyFrame KeyTime="0:0:1" Value="0" />
    </ObjectAnimationUsingKeyFrames>

    <ObjectAnimationUsingKeyFrames
        Storyboard.TargetName="grid2"
        Storyboard.TargetProperty="(Canvas.ZIndex)">
        <DiscreteObjectKeyFrame KeyTime="0:0:0" Value="0" />
        <DiscreteObjectKeyFrame KeyTime="0:0:1" Value="1" />
    </ObjectAnimationUsingKeyFrames>
</Storyboard>

<Storyboard x:Key="flipBackStoryboard">
    <DoubleAnimationUsingKeyFrames
        Storyboard.TargetName="projection"
        Storyboard.TargetProperty="RotationY">
        <DiscreteDoubleKeyFrame KeyTime="0:0:0" Value="0" />
        <LinearDoubleKeyFrame KeyTime="0:0:0.99" Value="-90" />
        <DiscreteDoubleKeyFrame KeyTime="0:0:1.01" Value="90" />
        <LinearDoubleKeyFrame KeyTime="0:0:2" Value="0" />
    </DoubleAnimationUsingKeyFrames>

    <DoubleAnimation Storyboard.TargetName="projection"
        Storyboard.TargetProperty="GlobalOffsetZ"
        From="0" To="-1000" Duration="0:0:1"
        AutoReverse="True" />

    <ObjectAnimationUsingKeyFrames
        Storyboard.TargetName="grid1"
        Storyboard.TargetProperty="(Canvas.ZIndex)">
        <DiscreteObjectKeyFrame KeyTime="0:0:0" Value="0" />
        <DiscreteObjectKeyFrame KeyTime="0:0:1" Value="1" />
    </ObjectAnimationUsingKeyFrames>

    <ObjectAnimationUsingKeyFrames
        Storyboard.TargetName="grid2"
        Storyboard.TargetProperty="(Canvas.ZIndex)">
        <DiscreteObjectKeyFrame KeyTime="0:0:0" Value="1" />
        <DiscreteObjectKeyFrame KeyTime="0:0:1" Value="0" />
    </ObjectAnimationUsingKeyFrames>
</Storyboard>
</Page.Resources>

```

These two storyboards are very similar. Each of them contains a *DoubleAnimationUsingKeyFrames* to target the *RotationY* property of the *PlaneProjection* object. The property is rotated from 0 to either 90 or –90 degrees (at which point it's at right angles to the user), and then it's switched 180 degrees so that the animation can continue in the same direction back to 0.

At the same time, the *GlobalOffsetZ* property is animated from 0 to –1000 and back to 0. This makes it seem as if the panel is dropping behind the screen to make the flip (perhaps so that the flipping panel won't smack the user in the nose).

Halfway through each *Storyboard*, the *Canvas.ZIndex* indices are swapped. The *Canvas.ZIndex* property is another appropriate target of an *ObjectAnimationUsingKeyFrames*.

The animations are triggered by a tap, which is handled in the code-behind file:

Project: TapToFlip | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    Storyboard flipStoryboard, flipBackStoryboard;
    bool flipped = false;

    public MainPage()
    {
        this.InitializeComponent();
        flipStoryboard = this.Resources["flipStoryboard"] as Storyboard;
        flipBackStoryboard = this.Resources["flipBackStoryboard"] as Storyboard;
    }

    void OnGridTapped(object sender, TappedRoutedEventArgs args)
    {
        if (flipStoryboard.GetCurrentState() == ClockState.Active ||
            flipBackStoryboard.GetCurrentState() == ClockState.Active)
        {
            return;
        }

        Storyboard storyboard = flipped ? flipBackStoryboard : flipStoryboard;
        storyboard.Begin();
        flipped ^= true;
    }
}
```

Much of the logic here is to prevent one *Storyboard* from starting when the previous one hasn't yet finished. With the way these storyboards are defined, that would cause discontinuities. (Try removing the *return* statement from *OnGridTapped* to see the unsatisfactory result.) I would prefer that a tap while an animation is in progress would simply reverse the operation, but that would require somewhat more complex logic.

Deriving a Matrix3D

Let's get into some hairy math, OK?

As you discovered earlier, two-dimensional graphics requires a 3×3 transform matrix to accommodate translation as well as scaling, rotation, and skew. Conceptually, a point (x, y) is treated as if it exists in 3D space with the coordinates (x, y, 1).

The generalized two-dimensional matrix transform looks like this:

$$|x \ y \ 1| \times \begin{vmatrix} M11 & M12 & 0 \\ M21 & M22 & 0 \\ OffsetX & OffsetY & 1 \end{vmatrix} = |x' \ y' \ 1|$$

Those are the actual fields of the *Matrix* structure provided for this purpose. The fixed third column restricts it to affine transforms. The transform formulas implied by the matrix multiplication are

$$x' = M11 \cdot x + M21 \cdot y + OffsetX$$

$$y' = M12 \cdot x + M22 \cdot y + OffsetY$$

The last row is fixed, which means that this is an affine transform. A square is always transformed into a parallelogram. This parallelogram is defined by three corners, and the fourth corner is determined by the other three.

Is it possible to derive an affine transform that maps a unit square into an arbitrary parallelogram? What we want is a mapping like this:

$$(0, 0) \rightarrow (x_0, y_0)$$

$$(0, 1) \rightarrow (x_1, y_1)$$

$$(1, 0) \rightarrow (x_2, y_2)$$

If you begin substituting these points into the transform formulas, it is easy to derive the following cells of the required matrix:

$$M11 = x_2 - x_0$$

$$M12 = y_2 - y_0$$

$$M21 = x_1 - x_0$$

$$M22 = y_1 - y_0$$

$$OffsetX = x_0$$

$$OffsetY = y_0$$

In 3D graphics programming, a 4×4 transform matrix is required and a point (x, y, z) is treated as if it exists in 4D space with coordinates (x, y, z, 1). Because there are no remaining letters after x, y, and z, that fourth dimension is usually referred to with the letter w. Application of a transform looks like this:

$$|x \ y \ z \ 1| \times \begin{vmatrix} M11 & M12 & M13 & M14 \\ M21 & M22 & M23 & M24 \\ M31 & M32 & M33 & M34 \\ OffsetX & OffsetY & OffsetZ & M44 \end{vmatrix} = |x' \ y' \ z' \ w'|$$

Those are the actual fields of the *Matrix3D* structure.

That resultant 4×1 matrix is then converted back to a point in three-dimensional space by dividing all the coordinates by w':

$$|x' \ y' \ z' \ w'| \rightarrow \left(\frac{x'}{w'}, \frac{y'}{w'}, \frac{z'}{w'} \right)$$

In conventional 2D graphics, a potential division by zero is generally undesired. But in 3D graphics,

division by a value that might equal zero is essential because this is how perspective is achieved. You want parallel lines to meet at infinity because that's how the world looks in real life.

The only purpose of this *Matrix3D* structure in the Windows Runtime is to set to the *ProjectionMatrix* property of a *Matrix3DProjection* object, which you can then set to the *Projection* property of an element as an alternative to *PlaneProjection*. In XAML, it might look like this:

```
<Image ... >
  <Image.Projection>
    <Matrix3DProjection>
      <Matrix3DProjection.ProjectionMatrix>
        1 0 0 0, 0 1 0 0, 0 0 1 0, 0 0 0 1
      </Matrix3DProjection.ProjectionMatrix>
    </Matrix3DProjection>
  </Image.Projection>
</Image>
```

You can't actually instantiate a *Matrix3D* value in XAML because *Matrix3D* is a structure, so instead you need to specify the 16 numbers that make up the matrix, starting with the first row. This example shows the identity matrix, with its characteristic diagonal of 1s.

This full-blown 4×4 matrix isn't entirely used in this context because the element that it's applied to is flat and has a Z coordinate of zero, so the application of the matrix really looks like this:

$$\begin{bmatrix} x & y & 0 & 1 \end{bmatrix} \times \begin{bmatrix} M11 & M12 & M13 & M14 \\ M21 & M22 & M23 & M24 \\ M31 & M32 & M33 & M34 \\ OffsetX & OffsetY & OffsetZ & M44 \end{bmatrix} = \begin{bmatrix} x' & y' & z' & w' \end{bmatrix}$$

This means that the cells that make up the entire third row—the values of *M31*, *M32*, *M33*, and *M34*—are irrelevant. They are multiplied by 0 and hence do not enter the calculation.

Moreover, the 3D point derived from this process is collapsed on the Z axis to obtain a 2D point for mapping to the video display:

$$\left(\frac{x'}{w'}, \frac{y'}{w'}, \frac{z'}{w'} \right) \rightarrow \left(\frac{x'}{w'}, \frac{y'}{w'} \right)$$

This is a process that happens in standard 3D graphics as well, but there's usually much more work involved because the Z values also indicate what's visible to the camera and what's obscured. In the Windows Runtime context, however, this collapse of Z values means that the entire third column—the values *M13*, *M23*, *M33*, and *OffsetZ*—are also irrelevant.

The transform formulas are therefore

$$x' = M11 \cdot x + M21 \cdot y + OffsetX$$

$$y' = M12 \cdot x + M22 \cdot y + OffsetY$$

$$w' = M14 \cdot x + M24 \cdot y + M44$$

If $M14$ and $M24$ are zero and $M44$ is 1, this is simply a two-dimensional affine transform. Nonzero values of $M14$ and $M24$ are the nonaffine parts of these formulas. $M44$ can be something other than 1, but if it's not zero you can always find an equivalent transform where $M44$ equals 1. Just multiply all the fields by $1/M44$.

With a nonaffine transform, a square is not necessarily transformed to a parallelogram. However, a nonaffine matrix transform still has limitations. It can't transform a square to any arbitrary quadrilateral. The transformed lines cannot cross each other and the four angles must be convex.

Let's attempt to derive a nonaffine transform that maps the four corners of a square to four arbitrary points:

$$\begin{aligned}(0, 0) &\rightarrow (x_0, y_0) \\ (0, 1) &\rightarrow (x_1, y_1) \\ (1, 0) &\rightarrow (x_2, y_2) \\ (1, 1) &\rightarrow (x_3, y_3)\end{aligned}$$

This exercise will be easier if we break this down into two transforms:

$$\begin{aligned}(0, 0) &\rightarrow (0, 0) \rightarrow (x_0, y_0) \\ (0, 1) &\rightarrow (0, 1) \rightarrow (x_1, y_1) \\ (1, 0) &\rightarrow (1, 0) \rightarrow (x_2, y_2) \\ (1, 1) &\rightarrow (a, b) \rightarrow (x_3, y_3)\end{aligned}$$

The first transform is obviously a nonaffine transform that I'll call **B**. The second is something that we'll force to be an affine transform called **A** (for affine). The way we'll force it to be an affine transform is by deriving values of a and b . The composite transform is **B**×**A**.

I've already shown you the derivation of the affine transform, and I don't even need to change notation when switching from the 3×3 matrix to the 4×4 matrix. But we also want this affine transform to map the point (a, b) to the arbitrary point (x_3, y_3) . By applying the derived affine transform to (a, b) and solving for a and b , we get this:

$$\begin{aligned}a &= \frac{M22 \cdot x_3 - M21 \cdot y_3 + M21 \cdot \text{OffsetY} - M22 \cdot \text{OffsetX}}{M11 \cdot M22 - M12 \cdot M21} \\ b &= \frac{M11 \cdot y_3 - M12 \cdot x_3 + M12 \cdot \text{OffsetX} - M11 \cdot \text{OffsetY}}{M11 \cdot M22 - M12 \cdot M21}\end{aligned}$$

Now let's take a shot at the nonaffine transform, which needs to yield the following mappings:

$$\begin{aligned}(0, 0) &\rightarrow (0, 0) \\ (0, 1) &\rightarrow (0, 1) \\ (1, 0) &\rightarrow (1, 0) \\ (1, 1) &\rightarrow (a, b)\end{aligned}$$

Here are the transform formulas from above:

$$x' = M11 \cdot x + M21 \cdot y + OffsetX$$

$$y' = M12 \cdot x + M22 \cdot y + OffsetY$$

$$w' = M14 \cdot x + M24 \cdot y + M44$$

Keep in mind that x' and y' must be divided by w' to get the transformed point.

If (0, 0) maps to (0, 0), then *OffsetX* and *OffsetY* are zero and *M44* is nonzero. Let's go out on a limb and set *M44* to 1.

If (0, 1) maps to (0, 1), then *M21* must be zero (to calculate a zero value of x') and y' divided by w' must equal 1, which means *M24* equals *M22* minus 1.

If (1, 0) maps to (1, 0), then *M12* is zero (for the zero value of y') and x' divided by w' must equal 1, or *M14* equals *M11* minus 1.

If (1, 1) maps to (a , b), then a bit of algebra derives

$$M11 = \frac{a}{a + b - 1}$$

$$M22 = \frac{b}{a + b - 1}$$

And a and b have already been derived.

Now let's code it up. I want to display the actual matrix that's derived from this process. That's the purpose of a *UserControl* derivative named *DisplayMatrix3D*. The XAML file consists of little more than a 4×4 *Grid* of *TextBlock* elements:

Project: NonAffineStretch | File: DisplayMatrix3D.xaml (excerpt)

```
<UserControl ... >
    <UserControl.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="TextAlignment" Value="Right" />
            <Setter Property="Margin" Value="6 0" />
        </Style>
    </UserControl.Resources>

    <Border BorderBrush="{StaticResource ApplicationForegroundThemeBrush}"
            BorderThickness="1 0">
        <Grid>
            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>

            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="Auto" />
            </Grid.ColumnDefinitions>
        </Grid>
    </Border>
</UserControl>
```

```

        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="Auto" />
    </Grid.ColumnDefinitions>

    <TextBlock Name="m11" Grid.Row="0" Grid.Column="0" />
    <TextBlock Name="m12" Grid.Row="0" Grid.Column="1" />
    <TextBlock Name="m13" Grid.Row="0" Grid.Column="2" />
    <TextBlock Name="m14" Grid.Row="0" Grid.Column="3" />

    <TextBlock Name="m21" Grid.Row="1" Grid.Column="0" />
    <TextBlock Name="m22" Grid.Row="1" Grid.Column="1" />
    <TextBlock Name="m23" Grid.Row="1" Grid.Column="2" />
    <TextBlock Name="m24" Grid.Row="1" Grid.Column="3" />

    <TextBlock Name="m31" Grid.Row="2" Grid.Column="0" />
    <TextBlock Name="m32" Grid.Row="2" Grid.Column="1" />
    <TextBlock Name="m33" Grid.Row="2" Grid.Column="2" />
    <TextBlock Name="m34" Grid.Row="2" Grid.Column="3" />

    <TextBlock Name="m41" Grid.Row="3" Grid.Column="0" />
    <TextBlock Name="m42" Grid.Row="3" Grid.Column="1" />
    <TextBlock Name="m43" Grid.Row="3" Grid.Column="2" />
    <TextBlock Name="m44" Grid.Row="3" Grid.Column="3" />
</Grid>
</Border>
</UserControl>

```

The code-behind file defines a dependency property of type *Matrix3D*, so it receives a notification whenever the property is changed. Watch out: The notification will not occur if a *property* of the existing *Matrix3D* structure is changed. The entire structure must be replaced.

Project: NonAffineStretch | File: DisplayMatrix3D.xaml.cs (excerpt)

```

public sealed partial class DisplayMatrix3D : UserControl
{
    static DependencyProperty matrix3DProperty =
        DependencyProperty.Register("Matrix3D",
            typeof(Matrix3D), typeof(DisplayMatrix3D),
            new PropertyMetadata(Matrix3D.Identity, OnPropertyChanged));

    public DisplayMatrix3D()
    {
        this.InitializeComponent();
    }

    public static DependencyProperty Matrix3DProperty
    {
        get { return matrix3DProperty; }
    }

    public Matrix3D Matrix3D
    {
        set { SetValue(Matrix3DProperty, value); }
        get { return (Matrix3D)GetValue(Matrix3DProperty); }
    }
}

```

```

static void OnPropertyChanged(DependencyObject obj,
                              DependencyPropertyChangedEventArgs args)
{
    (obj as DisplayMatrix3D).OnPropertyChanged(args);
}

void OnPropertyChanged(DependencyPropertyChangedEventArgs args)
{
    m11.Text = this.Matrix3D.M11.ToString("F3");
    m12.Text = this.Matrix3D.M12.ToString("F3");
    m13.Text = this.Matrix3D.M13.ToString("F3");
    m14.Text = this.Matrix3D.M14.ToString("F6");

    m21.Text = this.Matrix3D.M21.ToString("F3");
    m22.Text = this.Matrix3D.M22.ToString("F3");
    m23.Text = this.Matrix3D.M23.ToString("F3");
    m24.Text = this.Matrix3D.M24.ToString("F6");

    m31.Text = this.Matrix3D.M31.ToString("F3");
    m32.Text = this.Matrix3D.M32.ToString("F3");
    m33.Text = this.Matrix3D.M33.ToString("F3");
    m34.Text = this.Matrix3D.M34.ToString("F6");

    m41.Text = this.Matrix3D.OffsetX.ToString("F0");
    m42.Text = this.Matrix3D.OffsetY.ToString("F0");
    m43.Text = this.Matrix3D.OffsetZ.ToString("F0");
    m44.Text = this.Matrix3D.M44.ToString("F0");
}
}

```

The formatting specifications were chosen based on a bit of experience with the common ranges of these cells.

The XAML file for *MainPage* includes an instance of the *DisplayMatrix3D* control, but it also references an image from my website and adorns it with four *Thumb* controls. These *Thumb* controls allow us to drag any corner to an arbitrary location. The prefixes “ul”, “ur”, “ll”, and “lr” stand for “upper-left”, “upper-right”, “lower-left”, and “lower-right.”

Project: NonAffineStretch | File: MainPage.xaml (excerpt)

```

<Page ... >
    <Page.Resources>
        <Style TargetType="Thumb">
            <Setter Property="Width" Value="48" />
            <Setter Property="Height" Value="48" />
            <Setter Property="HorizontalAlignment" Value="Left" />
            <Setter Property="VerticalAlignment" Value="Top" />
        </Style>
    </Page.Resources>

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Image Source="http://www.charlespetzold.com/pw6/PetzoldJersey.jpg"
              Stretch="None"

```

```

        HorizontalAlignment="Left"
        VerticalAlignment="Top">
        <Image.Projection>
            <Matrix3DProjection x:Name="matrixProjection" />
        </Image.Projection>
    </Image>

    <Thumb DragDelta="OnThumbDragDelta">
        <Thumb.RenderTransform>
            <TransformGroup>
                <TranslateTransform X="-24" Y="-24" />
                <TranslateTransform x:Name="ulTranslate" X="100" Y="100" />
            </TransformGroup>
        </Thumb.RenderTransform>
    </Thumb>

    <Thumb DragDelta="OnThumbDragDelta">
        <Thumb.RenderTransform>
            <TransformGroup>
                <TranslateTransform X="-24" Y="-24" />
                <TranslateTransform x:Name="urTranslate" X="420" Y="100" />
            </TransformGroup>
        </Thumb.RenderTransform>
    </Thumb>

    <Thumb DragDelta="OnThumbDragDelta">
        <Thumb.RenderTransform>
            <TransformGroup>
                <TranslateTransform X="-24" Y="-24" />
                <TranslateTransform x:Name="llTranslate" X="100" Y="500" />
            </TransformGroup>
        </Thumb.RenderTransform>
    </Thumb>

    <Thumb DragDelta="OnThumbDragDelta">
        <Thumb.RenderTransform>
            <TransformGroup>
                <TranslateTransform X="-24" Y="-24" />
                <TranslateTransform x:Name="lrTranslate" X="420" Y="500" />
            </TransformGroup>
        </Thumb.RenderTransform>
    </Thumb>

    <local:DisplayMatrix3D HorizontalAlignment="Right"
        VerticalAlignment="Bottom"
        FontSize="24"
        Matrix3D="{Binding ElementName=matrixProjection,
            Path=ProjectionMatrix}" />
</Grid>
</Page>

```

The code-behind file implements the math I just showed you, except that another matrix is needed for mapping from the actual size and location of the image to a unit square. That's the matrix called *S* in the *CalculateNewTransform* code:

Project: NonAffineStretch | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    // Location and Size of Image with no transform
    Rect imageRect = new Rect(0, 0, 320, 400);

    public MainPage()
    {
        this.InitializeComponent();

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

    void OnThumbDragDelta(object sender, DragDeltaEventArgs args)
    {
        Thumb thumb = sender as Thumb;
        TransformGroup xformGroup = thumb.RenderTransform as TransformGroup;
        TranslateTransform translate = xformGroup.Children[1] as TranslateTransform;
        translate.X += args.HorizontalChange;
        translate.Y += args.VerticalChange;
        CalculateNewTransform();
    }

    void CalculateNewTransform()
    {
        Matrix3D matrix = CalculateNewTransform(imageRect,
                                                new Point(ulTranslate.X, ulTranslate.Y),
                                                new Point(urTranslate.X, urTranslate.Y),
                                                new Point(llTranslate.X, llTranslate.Y),
                                                new Point(lrTranslate.X, lrTranslate.Y));

        matrixProjection.ProjectionMatrix = matrix;
    }

    // The returned transform maps the points (0, 0),
    // (0, 1), (1, 0), and (1, 1) to the points
    // ptUL, ptUR, ptLL, and ptLR normalized based on rect.
    static Matrix3D CalculateNewTransform(Rect rect, Point ptUL, Point ptUR,
                                         Point ptLL, Point ptLR)
    {
        // Scale and translate normalization transform
        Matrix3D S = new Matrix3D()
        {
            M11 = 1 / rect.Width,
            M22 = 1 / rect.Height,
            OffsetX = -rect.Left / rect.Width,
            OffsetY = -rect.Top / rect.Height,
            M44 = 1
        };

        // Affine transform: Maps
```

```

//      (0, 0) --> ptUL
//      (1, 0) --> ptUR
//      (0, 1) --> ptLL
//      (1, 1) --> (x2 + x1 + x0, y2 + y1 + y0)
Matrix3D A = new Matrix3D()
{
    OffsetX = ptUL.X,
    OffsetY = ptUL.Y,
    M11 = (ptUR.X - ptUL.X),
    M12 = (ptUR.Y - ptUL.Y),
    M21 = (ptLL.X - ptUL.X),
    M22 = (ptLL.Y - ptUL.Y),
    M44 = 1
};

// Nonaffine transform
Matrix3D B = new Matrix3D();
double den = A.M11 * A.M22 - A.M12 * A.M21;
double a = (A.M22 * ptLR.X - A.M21 * ptLR.Y +
            A.M21 * A.OffsetY - A.M22 * A.OffsetX) / den;
double b = (A.M11 * ptLR.Y - A.M12 * ptLR.X +
            A.M12 * A.OffsetX - A.M11 * A.OffsetY) / den;

B.M11 = a / (a + b - 1);
B.M22 = b / (a + b - 1);
B.M14 = B.M11 - 1;
B.M24 = B.M22 - 1;
B.M44 = 1;

// Product of three transforms
return S * B * A;
}
}

```

Unlike the two-dimensional *Matrix* structure, the *Matrix3D* structure implements the multiplication operator, which makes array manipulation much easier.

It is certainly possible to drag one of the thumbs to a position where the image disappears because at least one of the angles is concave or the lines cross each other. But under those restrictions you can indeed stretch the image to a nonaffine shape:



Chapter 13

Touch, Etc.

One of the most forward-looking aspects of the Windows Runtime is the consolidation of touch, mouse, and pen input. No longer is it necessary to add touch to an existing mouse-oriented application, or add some mouse support to a touch application. From the very beginning, the programmer treats all these forms of input in a fairly interchangeable manner. In accordance with the Windows Runtime programming interface, I will be using the word *pointer* to refer to input from touch, mouse, and the pen (also known as the stylus) when it's not necessary to distinguish the actual input device.

The best way to handle pointer input is through the existing Windows Runtime controls. As you've seen, standard controls such as *Button*, *Slider*, *ScrollView*, and *Thumb* all respond to pointer input and use that to deliver higher-level input to your application.

In some cases, however, the programmer needs to obtain actual pointer input, and for that purpose *UIElement* defines three different families of events:

- Eight low-level events beginning with the word *Pointer*.
- Five higher-level events beginning with the word *Manipulation*.
- *Tapped*, *RightTapped*, *DoubleTapped*, and *Holding* events.

The *Control* class supplements these events with virtual protected methods beginning with the word *On* and followed by the event name.

To receive pointer input, a *FrameworkElement* derivative must have its *IsHitTestVisible* property set to *true* and its *Visibility* property set to *Visible*. A *Control* derivative must have its *IsEnabled* property set to *true*. The element must have some kind of graphical representation on the screen; a *Panel* derivative can have a *Transparent* background but not a *null* background.

All these events are associated with the element that is underneath your finger or mouse or pen at the time of the event. The only exception is when a pointer has been "captured" by an element, as you'll see later in this chapter.

If you need to track individual fingers, you'll want to use the *Pointer* events. Each event is accompanied by an ID number that uniquely identifies either an individual finger touching the screen, or the mouse or pen. In this chapter I'll demonstrate how to use *Pointer* events for a finger-paint program and a piano keyboard (unfortunately without sound). Both these programs obviously need to handle simultaneous input from multiple fingers.

In a sense, the *Pointer* events are the only events you need. For example, if you wish to implement a feature that allows the user to stretch a photograph with two fingers, you can track *Pointer* events for

those two fingers and measure how far they're moving apart. But calculations of this sort are provided for you in the *Manipulation* events. The *Manipulation* events consolidate multiple fingers into a single action, and they're ideal for moving, stretching, pinching, and rotating visual objects.

For some applications you might be puzzled whether to use *Pointer* or *Manipulation* events. The *Manipulation* events should probably be your first choice. Certainly, if you think to yourself "I hope the user's not going to start using a second finger because I'll just have to ignore it," you probably want to use the *Manipulation* events. Then, if the user does use two or more fingers when only one finger is necessary, the multiple fingers will be averaged. The *XYSlider* control shown in this chapter is an example of a program that could be done with *Pointer* events, but it wouldn't know what to do with extra fingers.

You'll also discover that the *Manipulation* events have an intrinsic lag. A finger touching the screen needs to move a bit before that finger is interpreted as contributing to a manipulation. *Manipulation* events are not fired if a finger taps or holds. Sometimes this lag will be enough to persuade you to use the *Pointer* events instead. *XYSlider* (as you'll see) is a case in point.

Pointer events are generated on a window level by the *CoreWindow* object, and you can derive *Manipulation* events on your own using the *GestureRecognizer*, but I'll be ignoring those facilities in this chapter and sticking with the events defined by *UIElement* and the virtual methods defined by *Control*. I also won't get into information about hardware input devices available from classes in the *Windows.Devices.Input* namespace.

Input from the pen has some special considerations involving the selection, erasing, and storage of pen strokes, as well as handwriting recognition. Those topics will be saved for a future chapter.

A *Pointer* Roadmap

Of the eight *Pointer* events, five are very common. If you touch a finger to an enabled and visible *UIElement* derivative, move it, and lift it, these five *Pointer* events are generated in the following order:

- *PointerEntered*
- *PointerPressed*
- *PointerMoved* (multiple occurrences in the general case)
- *PointerReleased*
- *PointerExited*

A finger generates *Pointer* events only when the finger is touching the screen or when it has just been removed. There is no such thing as "hover" with touch.

The mouse is a little different. The mouse generates *PointerMoved* events even without the mouse button pressed. Suppose you move the mouse pointer over a particular element, press the button,

move the mouse some more, release the button, and then move the mouse off the element. The element generates the following series of events:

- *PointerEntered*
- *PointerMoved* (multiple)
- *PointerPressed*
- *PointerMoved* (multiple)
- *PointerReleased*
- *PointerMoved* (multiple)
- *PointerExited*

Now let's try a pen. The element begins reacting to the pen before it actually touches the screen, so you'll first see a *PointerEntered* event followed by *PointerMoved*. As the pen touches the screen, a *PointerPressed* event is generated. Move the pen, and lift it. The element continues to fire *PointerMoved* events after *PointerReleased*, but it culminates with a *PointerExited* when the pen is moved further away from the screen. It's the same sequence of events as the mouse.

When the user spins the mouse wheel, the following event is generated:

- *PointerWheelChanged* (mouse wheel)

The remaining two events are rarer:

- *PointerCaptureLost*
- *PointerCanceled*

I'll discuss pointer capture later in this chapter, at which time the *PointerCaptureLost* event becomes much more important.

I have never seen a *PointerCanceled* event even when I've unplugged the mouse from the computer, but the event exists to report an error of that sort.

All these events are accompanied by an instance of *PointerRoutedEventArgs*, defined in the *Windows.UI.Xaml.Input* namespace. (Watch out: There's also a *PointerEventArgs* class in the *Windows.UI.Core* namespace, but that's used for the processing of pointer input on the window level.) As the name of this class indicates, these *Pointer* events are all routed events that travel up the visual tree.

PointerRoutedEventArgs defines two properties common for routed events:

- *OriginalSource* indicates the element that raised the event.
- *Handled* lets you stop further routing of the event further up the visual tree.

Lots of other information is available from the *PointerRoutedEventArgs* object. The following description covers only the highlights. The class also defines these:

- *Pointer* property of type *Pointer*.
- *KeyModifiers* property indicating the status of the Shift, Control, Menu (otherwise known as Alt), and Windows keys.
- *GetCurrentPoint* method that returns a *PointerPoint* object.

Watch out: Already we're dealing with classes named *Pointer* (defined in the *Windows.UI.Xaml.Input* namespace) and *PointerPoint* (defined in *Windows.UI.Input*).

The *Pointer* class has just four properties:

- ***PointerId*** An unsigned integer identifying the mouse, or a series of events involving a single finger or pen.
- ***PointerDeviceType*** An enumeration value *Touch*, *Mouse*, or *Pen*
- ***IsInRange*** A *bool* that indicates if the device is in range of the screen
- ***IsInContact*** A *bool* indicating if the finger or pen is touching the screen, or the mouse button is down.

The *PointerId* property is extremely important. This is what you use to track the movement of individual fingers. Almost always, a program that handles *Pointer* events will define a dictionary in which this *PointerId* property serves as a key.

The *GetCurrentPoint* method of *PointerRoutedEventArgs* sounds as if it returns the current coordinate location of the pointer, and it does, except that it also provides a whole lot more. Because it's convenient to get the location relative to a particular element, *GetCurrentPoint* accepts an argument of type *UIElement*. The *PointerPoint* object returned from this method duplicates some information from *Pointer* (the *PointerId* and *IsInContact* properties) and provides some other information:

- *Position* of type *Point*, the (x, y) location of the pointer at the time of the event.
- *Timestamp* of type *ulong*.
- *Properties* of type *PointerPointProperties* (defined in *Windows.UI.Input*).

The *Position* property is always relative to the upper-left corner of the element you pass to the *GetCurrentPoint* method.

The *PointerPointProperties* class defines 22 properties that provide detailed information about the event, including which mouse buttons are pressed, whether the button on the pen barrel is pressed, how the pen is tilted, the contact rectangle of the finger with the screen (if that's available), the pressure of a finger or pen against the screen (if that's available), and *MouseWheelDelta*.

You can use as little or as much of this information as you need. Obviously, some of it will not be applicable to every pointer device and will therefore have default values.

A First Dab at Finger Painting

Perhaps the archetypal multitouch application is one that lets you paint with your fingers on the screen. You can write such a program handling just three *Pointer* events and examining just two properties from the event arguments, but I'm afraid the result has a flaw not quite compensated for by its simplicity.

The MainPage.xaml file simply provides a name for the standard *Grid*:

Project: FingerPaint1 | File: MainPage.xaml (excerpt)

```
<Page ... >
    <Grid Name="contentGrid"
          Background="{StaticResource ApplicationPageBackgroundThemeBrush}" />
</Page>
```

The very first thing that the code-behind file does is define a *Dictionary* with a key of type *uint*. I mentioned earlier that virtually every program that handles *Pointer* events has a *Dictionary* of this sort. The type of the items you store in the *Dictionary* are dependent on the application; sometimes an application will define a class or structure specifically for this purpose. In a rudimentary finger-painting application, each finger touching the screen will be drawing a unique *Polyline*, so the *Dictionary* can store that *Polyline* instance:

Project: FingerPaint1 | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    Dictionary<uint, Polyline> pointerDictionary = new Dictionary<uint, Polyline>();
    Random rand = new Random();
    byte[] rgb = new byte[3];

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

    protected override void OnPointerPressed(PointerRoutedEventArgs args)
    {
        // Get information from event arguments
        uint id = args.Pointer.PointerId;
        Point point = args.GetCurrentPoint(this).Position;

        // Create random color
        rand.NextBytes(rgb);
        Color color = Color.FromArgb(255, rgb[0], rgb[1], rgb[2]);

        // Create Polyline
        Polyline polyline = new Polyline
```



```

    {
        Stroke = new SolidColorBrush(color),
        StrokeThickness = 24,
    };
    polyline.Points.Add(point);

    // Add to Grid
    contentGrid.Children.Add(polyline);

    // Add to dictionary
    pointerDictionary.Add(id, polyline);
    base.OnPointerPressed(args);
}

protected override void OnPointerMoved(PointerRoutedEventArgs args)
{
    // Get information from event arguments
    uint id = args.Pointer.PointerId;
    Point point = args.GetCurrentPoint(this).Position;

    // If ID is in dictionary, add the point to the Polyline
    if (pointerDictionary.ContainsKey(id))
        pointerDictionary[id].Points.Add(point);

    base.OnPointerMoved(args);
}

protected override void OnPointerReleased(PointerRoutedEventArgs args)
{
    // Get information from event arguments
    uint id = args.Pointer.PointerId;

    // If ID is in dictionary, remove it
    if (pointerDictionary.ContainsKey(id))
        pointerDictionary.Remove(id);

    base.OnPointerReleased(args);
}
}

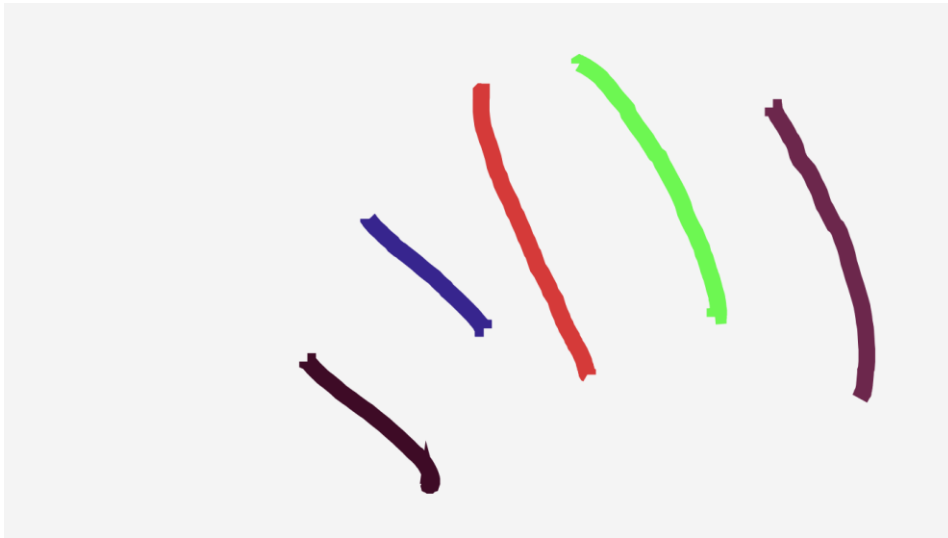
```

In the *OnPointerPressed* override, the program creates a *Polyline* and gives it a random color. The first point is the location of the pointer. The *Polyline* is added to the *Grid* and also to the dictionary.

When subsequent *OnPointerMoved* calls occur, the *PointerId* property identifies the finger, so the particular *Polyline* associated with that finger can be accessed from the dictionary and the new *Point* value can be added to the *Polyline*. Because it's the same instance as the *Polyline* in the *Grid*, the on-screen object will seem to grow in length as the finger moves.

The *OnPointerReleased* processing simply removes the entry from the dictionary. That particular *Polyline* is completed.

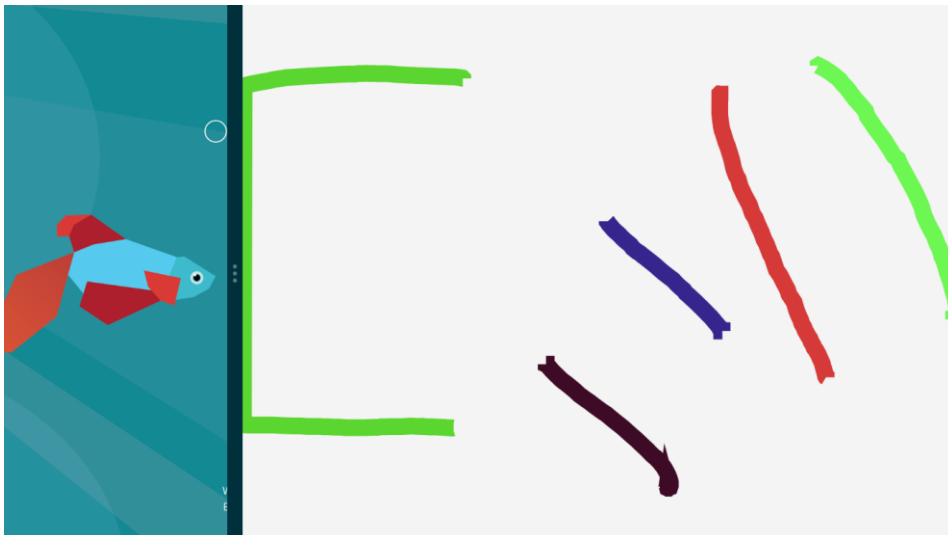
When you run the program, of course the first thing you'll want to do is sweep your whole hand across the screen like the glaciers that created the Finger Lakes in upstate New York.



Each finger paints its own polyline as a single series of connected points of a particular color, and you'll discover that you can use the mouse and pen as well.

I mentioned that this code has a flaw. The *OnPointerMoved* and *OnPointerReleased* overrides are very careful to check that the particular ID exists as a key in the dictionary before using it to access the dictionary. This is very important for mouse and pen processing because these devices generate *PointerMoved* events prior to *OnPointerPressed*.

But try this: Put the program in a snap mode, and with your finger, draw a line that goes outside the page and then back in.



Look at that straight line down the left side. That line is drawn when the finger reenters the page, and it indicates that the program doesn't get *PointerMoved* events during the time the finger strays outside. Try it with the mouse. Same thing.

Now try this: Using a finger, draw a line from the inside of the page to the outside and lift your finger. Now use your finger to draw inside the page again. This seems to work OK.

Now try it with the mouse. Press the mouse button over the *FingerPaint1* page, move the mouse to outside the page, and release the mouse button. Now move the mouse to the *FingerPaint1* page again. The program continues to draw the line even with the mouse button released! This is obviously wrong (but I'm sure you've seen programs get "confused" like this). Now press the mouse button, and you'll generate an exception when the *OnPointerPressed* method attempts to add an entry to the dictionary using a key that already exists in the dictionary. Unlike touch or the pen, all mouse events have the same ID.

Let's fix these problems.

Capturing the Pointer

To allow me (and you) to get a better sense of the sequence of *Pointer* events, I wrote a program called *PointerLog* that logs all the events on the screen. The core of the program is a *UserControl* called *LoggerControl*. The *Grid* in the *LoggerControl.xaml* file has been given a name but is otherwise initially empty:

Project: *PointerLog* | File: *LoggerControl.xaml* (excerpt)

```
<UserControl ... >

    <Grid Name="contentGrid" Background="Transparent" />

</UserControl>
```

The code-behind file has overrides of all eight *Pointer* events, all of which call a method named *Log* with the event name and event arguments. Like all *Pointer* programs, a *Dictionary* is defined, but the values in this one are not simple objects. Instead, I defined a nested class named *PointerInfo* right at the top of the *LoggerControl* class for storing per-finger information in this dictionary.

Project: *PointerLog* | File: *LoggerControl.xaml.cs* (excerpt)

```
public sealed partial class LoggerControl : UserControl
{
    class PointerInfo
    {
        public StackPanel stackPanel;
        public string repeatEvent;
        public TextBlock repeatTextBlock;
    };
};
```

```

Dictionary<uint, PointerInfo> pointerDictionary = new Dictionary<uint, PointerInfo>();

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

public bool CaptureOnPress { set; get; }

protected override void OnPointerEntered(PointerRoutedEventArgs args)
{
    Log("Entered", args);
    base.OnPointerEntered(args);
}

protected override void OnPointerPressed(PointerRoutedEventArgs args)
{
    if (this.CaptureOnPress)
        CapturePointer(args.Pointer);

    Log("Pressed", args);
    base.OnPointerPressed(args);
}

protected override void OnPointerMoved(PointerRoutedEventArgs args)
{
    Log("Moved", args);
    base.OnPointerMoved(args);
}

protected override void OnPointerReleased(PointerRoutedEventArgs args)
{
    Log("Released", args);
    base.OnPointerReleased(args);
}

protected override void OnPointerExited(PointerRoutedEventArgs args)
{
    Log("Exited", args);
    base.OnPointerExited(args);
}

protected override void OnPointerCaptureLost(PointerRoutedEventArgs args)
{
    Log("CaptureLost", args);
    base.OnPointerCaptureLost(args);
}

protected override void OnPointerCanceled(PointerRoutedEventArgs args)
{
    Log("Canceled", args);
    base.OnPointerCanceled(args);
}

```

```

protected override void OnPointerWheelChanged(PointerRoutedEventArgs args)
{
    Log("WheelChanged", args);
    base.OnPointerWheelChanged(args);
}

void Log(string eventName, PointerRoutedEventArgs args)
{
    uint id = args.Pointer.PointerId;
    PointerInfo pointerInfo;

    if (pointerDictionary.ContainsKey(id))
    {
        pointerInfo = pointerDictionary[id];
    }
    else
    {
        // New ID, so new StackPanel and header
        TextBlock header = new TextBlock
        {
            Text = args.Pointer.PointerId + " - " + args.Pointer.PointerDeviceType,
            FontWeight = FontWeights.Bold
        };
        StackPanel stackPanel = new StackPanel();
        stackPanel.Children.Add(header);

        // New PointerInfo for dictionary
        pointerInfo = new PointerInfo
        {
            stackPanel = stackPanel
        };
        pointerDictionary.Add(id, pointerInfo);

        // New column in the Grid for the StackPanel
        ColumnDefinition coldef = new ColumnDefinition
        {
            Width = new GridLength(1, GridUnitType.Star)
        };
        contentGrid.ColumnDefinitions.Add(coldef);
        Grid.SetColumn(stackPanel, contentGrid.ColumnDefinitions.Count - 1);
        contentGrid.Children.Add(stackPanel);
    }

    // Don't repeat PointerMoved and PointerWheelChanged events
    TextBlock txtblk = null;

    if (eventName == pointerInfo.repeatEvent)
    {
        txtblk = pointerInfo.repeatTextBlock;
    }
    else
    {
        txtblk = new TextBlock();
    }
}

```

```

        pointerInfo.stackPanel.Children.Add(txtblk);
    }

    txtblk.Text = eventName + " ";

    if (eventName == "WheelChanged")
    {
        txtblk.Text += args.GetCurrentPoint(this).Properties.MouseWheelDelta;
    }
    else
    {
        txtblk.Text += args.GetCurrentPoint(this).Position;
    }

    txtblk.Text += args.Pointer.IsInContact ? " C" : "";
    txtblk.Text += args.Pointer.IsInRange ? " R" : "";

    if (eventName == "Moved" || eventName == "WheelChanged")
    {
        pointerInfo.repeatEvent = eventName;
        pointerInfo.repeatTextBlock = txtblk;
    }
    else
    {
        pointerInfo.repeatEvent = null;
        pointerInfo.repeatTextBlock = null;
    }
}

public void Clear()
{
    contentGrid.ColumnDefinitions.Clear();
    contentGrid.Children.Clear();
    pointerDictionary.Clear();
}
}

```

The *Log* method seems rather complicated, but every time it encounters a new *PointerId* value in the event arguments, it adds a new column to the *Grid*, puts a *TextBlock* at the top indicating the ID and device type, and adds an entry to the dictionary. All subsequent events with that ID go in that column, except that consecutive *PointerMoved* and *PointerWheelChanged* events don't get extra entries. There's no scrolling facility and eventually there will be too many columns, but a public *Clear* method restores everything to a pristine condition.

The *LoggerControl* only gets *Pointer* events for that control. To ease the examination of what happens when fingers move between controls, I made *LoggerControl* part of a larger page with the program name at the top and three buttons at the bottom:

Project: PointerLog | File: MainPage.xaml (excerpt)

```

<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />

```

```

        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <TextBlock Text="Pointer Event Log"
        Grid.Row="0"
        Style="{StaticResource HeaderTextStyle}"
        HorizontalAlignment="Center"
        Margin="12" />

    <local:LoggerControl x:Name="logger"
        Grid.Row="1"
        FontSize="{StaticResource ControlContentThemeFontSize}" />

    <Grid Grid.Row="2">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="*" />
        </Grid.ColumnDefinitions>

        <Button Content="Clear"
            Grid.Column="0"
            HorizontalAlignment="Center"
            Click="OnClearButtonClick" />

        <ToggleButton Name="captureButton"
            Content="Capture on Press"
            Grid.Column="1"
            HorizontalAlignment="Center"
            Checked="OnCaptureToggleButtonChecked"
            Unchecked="OnCaptureToggleButtonChecked" />

        <Button Content="Release Captures in 5 seconds"
            Grid.Column="2"
            IsEnabled="{Binding ElementName=captureButton, Path=IsChecked}"
            HorizontalAlignment="Center"
            Click="OnReleaseCapturesButtonClick" />
    </Grid>
</Grid>

```

Notice the final *Button* is enabled only when the *ToggleButton* is toggled on.

The code-behind file just handles the buttons (which I'll discuss shortly):

Project: PointerLog | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    DispatcherTimer timer;

    public MainPage()
    {
        this.InitializeComponent();
        timer = new DispatcherTimer { Interval = TimeSpan.FromSeconds(5) };
    }
}

```

```

        timer.Tick += OnTimerTick;
    }

    void OnClearButtonClick(object sender, RoutedEventArgs args)
    {
        logger.Clear();
    }

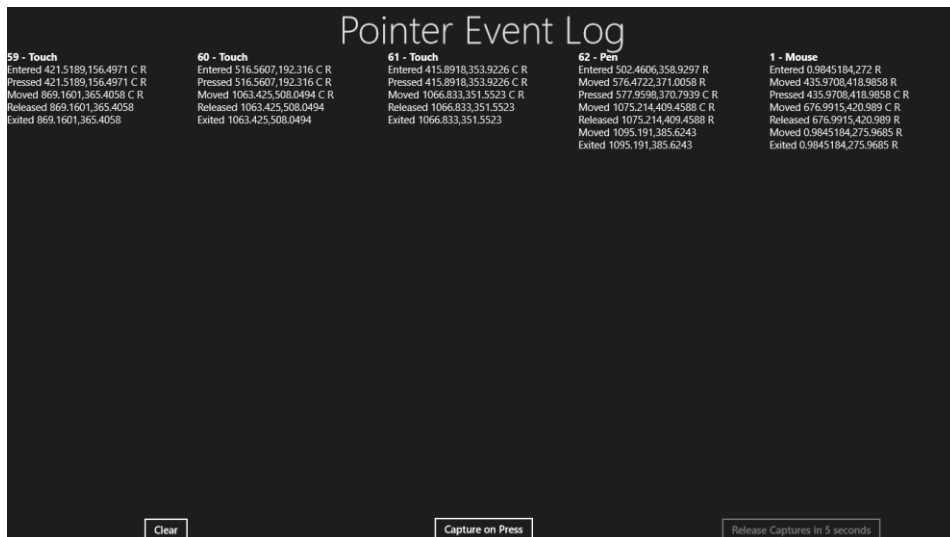
    void OnCaptureToggleButtonChecked(object sender, RoutedEventArgs args)
    {
        ToggleButton toggle = sender as ToggleButton;
        logger.CaptureOnPress = toggle.IsChecked.Value;
    }

    void OnReleaseCapturesButtonClick(object sender, RoutedEventArgs args)
    {
        timer.Start();
    }

    void OnTimerTick(object sender, object args)
    {
        logger.ReleasePointerCaptures();
        timer.Stop();
    }
}

```

You can see from the screen that each new finger press gets a unique ID and only five events. Each new series of pen events also gets its own ID (using the same numbering sequence as touch) with a few more events. The mouse always has an ID of 1:



The letters C and R indicate *true* values of the *IsInContact* and *IsInRange* properties of the *Pointer* object. As you can see, for the pen and mouse you can use the *IsInRange* property to distinguish

between *PointerMoved* events that occur when the pen is touching the screen or when the mouse button is pressed.

By default, an element gets *Pointer* input only when the pointer is within the boundaries of the element. This can sometimes result in a loss of information. To demonstrate this, I deliberately designed the program so that *LoggerControl* does not extend to the full height of the screen. Above it is an area for the program title, and below is the button area. These areas are the domain of *MainPage*. This configuration allows you to experiment with input that moves from one element to another.

For example, touch the *PointerLog* screen somewhere in the middle, move your finger around, and then move the finger to the top title area or the bottom button area. Lift it off the screen. The program does not receive that *PointerReleased* event, and it has no idea that the pointer has been released. It will never get another event with that particular ID number, but it's living in a state of ignorance. The entry in the dictionary is never removed.

Similarly, touch the screen in the top or bottom area and move your finger to the central area. The program registers *PointerEntered* and *PointerMoved* events but not a *PointerPressed* event.

Often while tracking a particular pointer, you want to continue getting input even if it drifts outside the element. Not getting that pointer input accounts for the flaws in the *FingerPaint1* program.

You can get what you want with a process called "capturing the pointer," which you do with a call to the *CapturePointer* method defined by *UIElement*. The method has an argument of type *Pointer* and returns a *bool* indicating if the pointer capture has been successful. When will it not be successful? If you call *CapturePointer* during an event prior to *PointerPressed* or during *PointerReleased* or later.

For this reason—and for the sake of program politeness—it really only makes sense to call *CapturePointer* during a *PointerPressed* event. By pressing a finger (or pen or mouse button) on a particular element, the user is generally indicating a desire to interact with that element even if the finger sometimes drifts outside the element.

If you toggle on the "Capture on Press" button at the bottom of the *PointerLog* screen, the program calls

```
CapturePointer(args.Pointer);
```

during the *OnPointerPressed* override.

Now if you press in the central area of the *PointerLog* program, move your finger to the top or bottom, and then release, the program logs the *PointerReleased* event as well as a final *PointerCaptureLost* event following *PointerExited*.

A program can get a list of all the captured pointers with a call to *PointerCaptures* and release a particular capture with a *ReleasePointerCapture* call or release all pointer captures with *ReleasePointerCaptures*.

In a real-life application it is tempting to simply ignore the *PointerCaptureLost* event, but it's not a

good idea. If Windows needs to communicate something urgent to the user, it's possible that pointer capture will be snatched from a program involuntarily. I have not actually seen this happen under Windows 8, but historically it occurs upon the display of a system modal dialog box—a dialog box that considers itself so important that it gets all user input until it's dismissed.

To demonstrate what happens in such a case, I've defined the third button to set a *DispatcherTimer* for five seconds and then conclude by calling *ReleasePointerCaptures* for the *LoggerControl*. When that happens, a pointer that has been captured receives a *PointerCaptureLost* event. It will continue to receive other *Pointer* events if the pointer is still over the element but not if it drifts outside the element.

What an application should do when it receives an unexpected *PointerCaptureLost* depends on the application. A finger-paint program might want to move *PointerReleased* logic into *PointerCaptureLost*, for example, and treat both expected and unexpected losses of capture as the same.

Or, it might make sense to entirely discard that particular drawing event.

In fact, you might want to build this feature into your program. Suppose you decide that the user should be able to press the Escape key to jettison a drawing event that's in progress. You could then implement Escape key processing with a simple call to *ReleasePointerCaptures*.

The FingerPaint2 program does precisely that. The XAML file is the same as FingerPaint1, and so is the code-behind file with the following exceptions:

Project: FingerPaint2 | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    ...
    public MainPage()
    {
        this.InitializeComponent();
        this.IsTabStop = true;
    }

    protected override void OnPointerPressed(PointerRoutedEventArgs args)
    {
        ...
        // Capture the Pointer
        CapturePointer(args.Pointer);

        // Set input focus
        Focus(FocusState.Programmatic);

        base.OnPointerPressed(args);
    }
    ...
    protected override void OnPointerCaptureLost(PointerRoutedEventArgs args)
    {
        // Get information from event arguments
        uint id = args.Pointer.PointerId;
```

```

        // If ID is in dictionary, abandon the drawing operation
        if (pointerDictionary.ContainsKey(id))
        {
            contentGrid.Children.Remove(pointerDictionary[id]);
            pointerDictionary.Remove(id);
        }

        base.OnPointerCaptureLost(args);
    }

    protected override void OnKeyDown(KeyRoutedEventArgs args)
    {
        if (args.Key == VirtualKey.Escape)
            ReleasePointerCaptures();

        base.OnKeyDown(args);
    }
}

```

In the constructor, the *IsTabStop* property must be set to *true* for the element to receive keyboard input. Only one element can receive keyboard input at any time. This is called the element with keyboard “focus,” and some controls indicate they have keyboard focus with a special appearance, such as a dotted line. Often an element can give itself keyboard focus by calling the *Focus* method when the element is tapped or (in this case) during the *OnPointerPressed* event. That override concludes its processing by calling the *Focus* method as well as *CapturePointer*:

The *OnPointerCaptureLost* removes the *Polyline* in progress from the *Grid* and removes the ID from the dictionary. However, the *OnPointerCaptureLost* event can occur normally after a finger has been released from the screen, so this ID will still be in the dictionary only if the page didn’t get a call to *OnPointerReleased*.

The *OnKeyDown* method gets keystrokes and calls *ReleasePointerCaptures* for the Escape key. This call has no effect if no pointers are captured.

Try the problematic actions identified with the *FingerPaint1*, and you’ll find that they’re gone in this version. Moreover, now you can be drawing on the screen and press the Escape key, and what you’re currently drawing will disappear and the finger will have no further effect until it’s released and pressed again. (Let’s hope that’s what you want.)

Editing with a Popup Menu

Let’s add an editing feature to this program. If you click an existing *Polyline* with the right mouse button—or you do something equivalent with a finger or pen—a little menu pops up with the options “Change color” and “Delete.”

In the previous two *FingerPaint* programs, the *Polyline* was created, initialized, and added to the

content *Grid* and touch dictionary like so:

```
// Create Polyline
Polyline polyline = new Polyline
{
    Stroke = new SolidColorBrush(color),
    StrokeThickness = 24,
};
polyline.Points.Add(point);

// Add to Grid
contentGrid.Children.Add(polyline);

// Add to dictionary
pointerDictionary.Add(id, polyline);
```

For *FingerPaint3* let's add some additional code that sets two event handlers on this *Polyline*. The goal here is to use the handler for the *RightTapped* event of the *Polyline* to display a popup menu:

Project: *FingerPaint3* | File: *MainPage.xaml.cs* (excerpt)

```
protected override void OnPointerPressed(PointerRoutedEventArgs args)
{
    ...
    // Create Polyline
    Polyline polyline = new Polyline
    {
        Stroke = new SolidColorBrush(color),
        StrokeThickness = 24,
    };
    polyline.PointerPressed += OnPolylinePointerPressed;
    polyline.RightTapped += OnPolylineRightTapped;
    polyline.Points.Add(point);
    ...
}
```

Although we're interested only in the *RightTapped* event for the *Polyline*, I've also set a handler for the *PointerPressed* event. That handler is not very interesting, but it's very important:

Project: *FingerPaint3* | File: *MainPage.xaml.cs* (excerpt)

```
void OnPolylinePointerPressed(object sender, PointerRoutedEventArgs args)
{
    args.Handled = true;
}
```

You'll definitely want to try this program without this particular handler, and here's why: When a *PointerPressed* event is fired, that event is associated with the topmost element that is enabled for user input. If you're clicking or right-clicking a *Polyline* rather than the surface of *MainPage*, the *PointerPressed* event is fired for that *Polyline*.

However, *PointerPressed* is a routed event, and you'll recall from Chapter 3, "Basic Event Handling," that routed events travel up the visual tree, which means that if the *Polyline* isn't interested in this event, it will go to *MainPage*, which will assume that you want to begin drawing a new figure. To

prevent that from happening in this program, the *Polyline* handles the *PointerPressed* event by setting the *Handled* property on the event arguments to *true*. This prevents the event from reaching *MainPage*.

The popup menu logic occurs in the *RightTapped* event:

Project: FingerPaint3 | File: MainPage.xaml.cs (excerpt)

```
async void OnPolylineRightTapped(object sender, RightTappedRoutedEventArgs args)
{
    Polyline polyline = sender as Polyline;
    PopupMenu popupMenu = new PopupMenu();
    popupMenu.Commands.Add(new UICommand("Change color", OnMenuChangeColor, polyline));
    popupMenu.Commands.Add(new UICommand("Delete", OnMenuDelete, polyline));
    await popupMenu.ShowAsync(args.GetPosition(this));
}
```

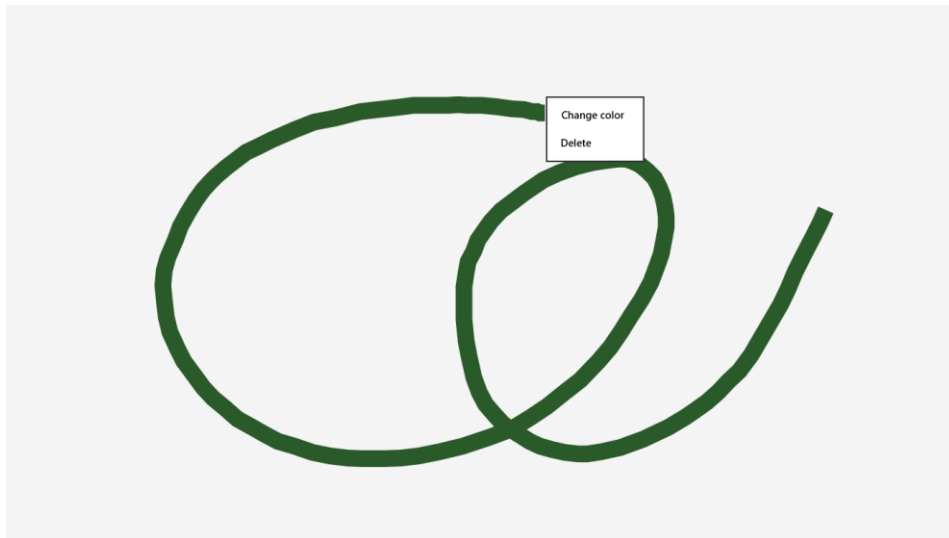
It's fairly easy to use *PopupMenu*. After creating the object, you can add up to six items to the menu. Each item consists of a text label, a callback, and an optional object to help the callback identify the event. The *ShowAsync* method displays the menu at a particular location.

The handlers can obtain that last argument passed to the *UICommand* constructor by casting the *Id* property of the callback method's *UICommand* argument:

```
void OnMenuChangeColor(UICommand command)
{
    Polyline polyline = command.Id as Polyline;
    rand.NextBytes(rgb);
    Color color = Color.FromArgb(255, rgb[0], rgb[1], rgb[2]);
    (polyline.Stroke as SolidColorBrush).Color = color;
}

void OnMenuDelete(UICommand command)
{
    Polyline polyline = command.Id as Polyline;
    contentGrid.Children.Remove(polyline);
}
```

I'm sure you already know how to use the mouse to right-click a *Polyline*. With touch, you'll need to hold your finger steady on a *Polyline* for a moment, and then release. You'll see a square form when you've held long enough. Similarly with a pen, hold it until you see a circle form, and then release. The menu appears:



The square and circle you see when you hold your finger or the pen to the screen are actually associated with the *Holding* event. If you set the *IsHoldingEnabled* property on the *Polyline* to *false*, they won't appear and the user might be a little stymied how long to press. The *RightTapped* event isn't fired until the user lifts the finger or pen from the screen.

The *OnMenuDelete* method in *FingerPaint3* actually has a subtle bug. If one finger is drawing a line while another finger invokes the menu for that line, *OnMenuDelete* removes the *Polyline* from the screen but not the dictionary entry with that *Polyline*. Nothing bad will happen, but the dictionary might accumulate some abandoned entries. Logic to fix this would have to search through the dictionary for the deleted *Polyline* and then remove the key for that entry.

As I demonstrated with routed events in Chapter 3, whenever you're dealing with events generated by different elements, you can structure your event handling in various ways. For example, an *OnPointerPressed* override in *MainPage* can incorporate the logic that I put in *OnPolylinePointerPressed*, and you can perform all the *RightTapped* handling in an *OnRightTapped* override. All you need do is check the *OriginalSource* property on the event arguments to determine whether the input is coming from the *Polyline* or *MainPage*.

The program now has a little drawback. You can't draw a new line if you want to begin that line on a point occupied by an existing line. Any *PointerPressed* event received by the *Polyline* is flagged as *Handled* and essentially discarded.

What if you wanted to give the user both options? If the user presses an existing *Polyline* and starts moving, a new figure is started. If the user presses and holds, that's a menu.

Probably the easiest approach is abandoning the use of the *RightTapped* event and handling everything through the *Pointer* logic. When *OnPointerPressed* occurs on an existing *Polyline*, set a *DispatcherTimer* for one second, but cancel that timer (and start a drawing operation) if

OnPointerMoved occurs, indicating that the finger has moved a distance greater than some preset criteria. If the timer fires, display the menu.

Pressure Sensitivity

The lines drawn by the various FingerPaint programs are of a uniform stroke thickness—24 pixels to be precise—but I probably would have done it differently if the tablet I’m using was sensitive to touch pressure. But it is not.

There are two properties that might influence line thickness in a finger-painting program, and both are defined by the *PointerPointProperties* object returned from the *Properties* property of the *PointerPoint* class (which in turn is obtained by a call to the *GetCurrentPoint* method of the *PointerRoutedEventArgs* event arguments).

The first property is *ContactRect*, a *Rect* value that is intended to report the rectangular bounding box of the contact area of a finger (or pen point) on the screen. On my tablet, this *Rect* always has a *Width* and *Height* of zero regardless of the pointer device.

The second is *Pressure*, which is a *float* value that can take on values between 0 and 1. On my tablet, this *Pressure* value is the default value of 0.5 for fingers and the mouse, but it is variable for the pen, and so we have the opportunity to try it out.

For purposes of simplicity, the FingerPaint4 program does not include Esc key processing or the editing feature, but it does implement pointer capturing. The big difference is that the *Polyline* approach to drawing must be abandoned because a *Polyline* has only a single *StrokeThickness* property. In this new program each stroke must instead be composed of very short individual lines, each a unique *StrokeThickness* that is calculated from the *Pressure* value, but all in the same color. This implies that the dictionary needs to contain values of type *Color* (or better yet, a *Brush*) and the previous *Point*. This is now two items, so let’s define a custom structure for that purpose that I called *PointerInfo*:

Project: FingerPaint4 | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    struct PointerInfo
    {
        public Brush Brush;
        public Point PreviousPoint;
    }

    Dictionary<uint, PointerInfo> pointerDictionary = new Dictionary<uint, PointerInfo>();
    Random rand = new Random();
    byte[] rgb = new byte[3];

    public MainPage()
    {
```

```

        this.InitializeComponent();
    }

    protected override void OnPointerPressed(PointerRoutedEventArgs args)
    {
        // Get information from event arguments
        uint id = args.Pointer.PointerId;
        Point point = args.GetCurrentPoint(this).Position;

        // Create random color
        rand.NextBytes(rgb);
        Color color = Color.FromArgb(255, rgb[0], rgb[1], rgb[2]);

        // Create PointerInfo
        PointerInfo pointerInfo = new PointerInfo
        {
            PreviousPoint = point,
            Brush = new SolidColorBrush(color)
        };

        // Add to dictionary
        pointerDictionary.Add(id, pointerInfo);

        // Capture the Pointer
        CapturePointer(args.Pointer);

        base.OnPointerPressed(args);
    }

    protected override void OnPointerMoved(PointerRoutedEventArgs args)
    {
        // Get information from event arguments
        uint id = args.Pointer.PointerId;
        PointerPoint pointerPoint = args.GetCurrentPoint(this);
        Point point = pointerPoint.Position;
        float pressure = pointerPoint.Properties.Pressure;

        // If ID is in dictionary, create a new Line element and add to Grid
        if (pointerDictionary.ContainsKey(id))
        {
            PointerInfo pointerInfo = pointerDictionary[id];

            Line line = new Line
            {
                X1 = pointerInfo.PreviousPoint.X,
                Y1 = pointerInfo.PreviousPoint.Y,
                X2 = point.X,
                Y2 = point.Y,
                Stroke = pointerInfo.Brush,
                StrokeThickness = pressure * 24,
                StrokeStartLineCap = PenLineCap.Round,
                StrokeEndLineCap = PenLineCap.Round
            };
            contentGrid.Children.Add(line);
        }
    }

```



```

        // Update PointerInfo and store back in dictionary
        pointerInfo.PreviousPoint = point;
        pointerDictionary[id] = pointerInfo;
    }

    base.OnPointerMoved(args);
}

protected override void OnPointerReleased(PointerRoutedEventArgs args)
{
    // Get information from event arguments
    uint id = args.Pointer.PointerId;

    // If ID is in dictionary, remove it
    if (pointerDictionary.ContainsKey(id))
        pointerDictionary.Remove(id);

    base.OnPointerReleased(args);
}

protected override void OnPointerCaptureLost(PointerRoutedEventArgs args)
{
    // Get information from event arguments
    uint id = args.Pointer.PointerId;

    // If ID is still in dictionary, remove it
    if (pointerDictionary.ContainsKey(id))
        pointerDictionary.Remove(id);

    base.OnPointerCaptureLost(args);
}
}

```

The previous *PointerPressed* handling created a *Polyline*, gave it an initial point, and added it to the *Grid* and *Dictionary*. In this program, only a *PointerInfo* value is created and added to the dictionary. Much more work occurs in the *PointerMoved* handler. Using the new point and the previous point from the dictionary, a *Line* element is constructed and added to the *Grid*. The new point then replaces the previous point in the *PointerInfo* value.

Notice that the *StrokeThickness* is set to 24 times the *Pressure* value. This results in a maximum stroke thickness of 24 and a stroke thickness of 12 for non-pressure-sensitive devices. Notice also that the *StrokeStartLineCap* and *StrokeEndLineCap* properties are set to *Round*. Try commenting out these property settings and see what happens when a stroke has sharp turns: little gaps appear because two short lines are at an angle to each other. The line caps cover those gaps.

Here's a little, umm, artwork I did entirely with the pen:



Notice the graceful subtlety of the strokes when rendered with a pressure-sensitive input device.

If you draw very quickly, you might notice that the resultant strokes are not as curvy as you like and wonder if the *GetIntermediatePoints* method of *PointerPoint* might provide some more detail. It is my experience that the collection of points returned from this call always contains only one item, but I'm glad the feature is built in to obtain a more detailed flow of points.

It is also my experience that *PointerMoved* events are fired over 100 times per second, which is faster than the frame rate of the video display but not quite fast enough for extremely energetic fingers.

How Do I Save My Drawings?

None of the finger-painting programs has any facility to save the drawings, but how would you implement such a thing?

Each program draws by adding *Polyline* or *Line* elements to a *Grid*. One way to save your drawing would be to access those objects and save all the points in a file, perhaps in an XML format. You could then add a feature to load them back in and create new *Polyline* or *Line* elements from each collection of points.

But you might be more inclined to save a *bitmap* of your drawing. (Traditionally, "draw" programs work with vectors while "paint" programs work with bitmaps.) Indeed, it makes sense for a *FingerPaint* program do perform *all* its painting on a bitmap.

This is possible, but it's not as easy as you might think. The easiest approach is to use *WritableBitmap*, but you'd have to implement your own line-drawing logic to render lines on that

bitmap. I'll show you how in Chapter 14, "Bitmaps" (but not in the version of that chapter in this Release Preview ebook). It's also possible using DirectX with some C++ coding. That's coming in Chapter 16, "Going Native."

A Touch Piano

Not all touch applications fall into the same pattern. For example, consider an on-screen piano keyboard. Obviously you want to be able to play chords with your fingers, so this is a job for the *Pointer* events rather than the *Manipulation* events.

But what you also *really* want to do with an on-screen piano keyboard is run your fingers up and down the keys making glissandi. If you couldn't do that with an on-screen keyboard, you would undoubtedly consider it broken. What that implies, however, is that you're probably not exclusively concerned with *PointerPressed* and *PointerReleased*. Yes, you can press down on one key and release on another, but in between you could be playing many other keys just by sweeping your finger.

There are basically two ways to construct this piano keyboard. You can use one control for the whole keyboard, or you can use many controls (and by "many" I really mean one control for each key).

A single control must draw all the keys and also evaluate *PointerMoved* events by comparing pointer positions with the boundaries of these keys. You'll be tracking each finger to determine when a *PointerMoved* event indicates a key coming within a key boundary and when it leaves a key boundary. This is classic "hit testing"—you're examining pointer positions to determine if they lie within a boundary.

However, if each key is a separate control, that key doesn't need to perform hit testing. If it's getting a *Pointer* event, the *Pointer* is within the boundaries of that control (unless the control has captured the pointer, but pointer capturing makes no sense in this application).

What *Pointer* events are necessary to implement a piano key? Don't start by thinking about presses and releases. Think about glissandi. If we're talking about a keyboard that reacts solely to touch, the only two *Pointer* events that are necessary are *PointerEntered* and *PointerExited*.

However, you probably want the keyboard to respond reasonably to the mouse and pen as well. A piano key will get *PointerEntered* and *PointerExited* events for a mouse when the mouse button is not pressed, and that's a problem. The *PointerEntered* handler will need to examine the *IsInContact* property to correct handle the mouse and pen. That property is always *true* for touch but only *true* for a mouse if the button is down or for the pen if it's in contact with the screen.

Moreover, when considering a single element, the mouse and pen generate *PointerEntered* events before *PointerPressed* and *PointerExited* after *PointerReleased*, so *PointerPressed* and *PointerReleased* must be handled as well.

Let's construct a two-octave piano keyboard from the bottom up, starting with the keys. The

following *Key* class is a *Control* derivative without a default template, so it has no default visible appearance. But it does define two visual states called Normal and Pressed, an *IsPressed* dependency property, and a property changed handler for *IsPressed* that toggles between those two visual states.

Project: SilentPiano | File: Key.cs (excerpt)

```
namespace SilentPiano
{
    [TemplateVisualStateAttribute(Name = "Normal", GroupName = "CommonStates")]
    [TemplateVisualStateAttribute(Name = "Pressed", GroupName = "CommonStates")]
    public class Key : Control
    {
        static readonly DependencyProperty isPressedProperty =
            DependencyProperty.Register("IsPressed",
                typeof(bool), typeof(Key),
                new PropertyMetadata(false, OnIsPressedChanged));

        List<uint> pointerList = new List<uint>();

        public static DependencyProperty IsPressedProperty
        {
            get { return isPressedProperty; }
        }

        public bool IsPressed
        {
            set { SetValue(IsPressedProperty, value); }
            get { return (bool)GetValue(IsPressedProperty); }
        }

        protected override void OnPointerEntered(PointerRoutedEventArgs args)
        {
            if (args.Pointer.IsInContact)
                AddToList(args.Pointer.PointerId);
            base.OnPointerEntered(args);
        }

        protected override void OnPointerPressed(PointerRoutedEventArgs args)
        {
            AddToList(args.Pointer.PointerId);
            base.OnPointerPressed(args);
        }

        protected override void OnPointerReleased(PointerRoutedEventArgs args)
        {
            RemoveFromList(args.Pointer.PointerId);
            base.OnPointerReleased(args);
        }

        protected override void OnPointerExited(PointerRoutedEventArgs args)
        {
            RemoveFromList(args.Pointer.PointerId);
            base.OnPointerExited(args);
        }
    }
}
```

```

void AddToList(uint id)
{
    if (!pointerList.Contains(id))
        pointerList.Add(id);

    CheckList();
}

void RemoveFromList(uint id)
{
    if (pointerList.Contains(id))
        pointerList.Remove(id);

    CheckList();
}

void CheckList()
{
    this.IsPressed = pointerList.Count > 0;
}

static void OnIsPressedChanged(DependencyObject obj,
                               DependencyPropertyChangedEventArgs args)
{
    VisualStateManager.GoToState(obj as Key,
                                   (bool)args.NewValue ? "Pressed" : "Normal", false);
}
}
}

```

Because you can use two fingers to play the same key, this control still needs to track individual fingers. But it doesn't need a *Dictionary* to retain information for each ID. It can simply use a *List*. IDs are put into this *List* in the *OnPointerEntered* override (but only if *IsInContact* is true) and in *OnPointerPressed*, and removed in *OnPointerReleased* and *OnPointerExited*, and that triggers the change in visual state. The *IsPressed* property is *true* if the *List* contains at least one entry. The *PointerPressed* and *PointerReleased* event handlers are only for the benefit of the mouse and pen.

Two templates—one for white keys and one for black keys—are defined in the *Octave.xaml* file. The two templates differ only by the size of a *Polygon* that defines the key shape and the default color. (The shape is a rectangle for both keys. Originally I wanted to make the various white keys different shapes as they are on a real piano, but the uniform approach was a lot easier and required far fewer templates.) Both templates switch the color to red during a Pressed state:

Project: SilentPiano | File: Octave.xaml (excerpt)

```

<UserControl ... >
    <UserControl.Resources>
        <ControlTemplate x:Key="whiteKey" TargetType="local:Key">
            <Grid Width="80">
                <Polygon Points="2 0, 78 0, 78 320, 0 320">
                    <Polygon.Fill>
                        <SolidColorBrush x:Name="brush" Color="White" />

```

```

        </Polygon.Fill>
    </Polygon>

    <VisualStateManager.VisualStateGroups>
        <VisualStateGroup x:Name="CommonStates">
            <VisualState x:Name="Normal"/>
            <VisualState x:Name="Pressed">
                <Storyboard>
                    <ColorAnimationUsingKeyFrames Storyboard.TargetName="brush"
                                                    Storyboard.TargetProperty="Color">
                        <DiscreteColorKeyFrame KeyTime="0" Value="Red" />
                    </ColorAnimationUsingKeyFrames>
                </Storyboard>
            </VisualState>
        </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
</Grid>
</ControlTemplate>

<ControlTemplate x:Key="blackKey" TargetType="local:Key">
    <Grid>
        <Polygon Points="0 0, 40 0, 40 220, 0 220">
            <Polygon.Fill>
                <SolidColorBrush x:Name="brush" Color="Black" />
            </Polygon.Fill>
        </Polygon>

        <VisualStateManager.VisualStateGroups>
            <VisualStateGroup x:Name="CommonStates">
                <VisualState x:Name="Normal"/>
                <VisualState x:Name="Pressed">
                    <Storyboard>
                        <ColorAnimationUsingKeyFrames Storyboard.TargetName="brush"
                                                        Storyboard.TargetProperty="Color">
                            <DiscreteColorKeyFrame KeyTime="0" Value="Red" />
                        </ColorAnimationUsingKeyFrames>
                    </Storyboard>
                </VisualState>
            </VisualStateGroup>
        </VisualStateManager.VisualStateGroups>
    </Grid>
</ControlTemplate>
</UserControl.Resources>

<Grid>
    <StackPanel Orientation="Horizontal">
        <local:Key Template="{StaticResource whiteKey}" />
        <local:Key Template="{StaticResource whiteKey}" />
        <local:Key Template="{StaticResource whiteKey}" />
        <local:Key Template="{StaticResource whiteKey}" />
        <local:Key Template="{StaticResource whiteKey}" />
        <local:Key Template="{StaticResource whiteKey}" />
        <local:Key Template="{StaticResource whiteKey}" />
        <local:Key x:Name="lastKey"

```

```

        Template="{StaticResource whiteKey}"
        Visibility="Collapsed" />
</StackPanel>
<Canvas>
    <local:Key Template="{StaticResource blackKey}"
        Canvas.Left="60" Canvas.Top="0" />
    <local:Key Template="{StaticResource blackKey}"
        Canvas.Left="140" Canvas.Top="0" />
    <local:Key Template="{StaticResource blackKey}"
        Canvas.Left="300" Canvas.Top="0" />
    <local:Key Template="{StaticResource blackKey}"
        Canvas.Left="380" Canvas.Top="0" />
    <local:Key Template="{StaticResource blackKey}"
        Canvas.Left="460" Canvas.Top="0" />
</Canvas>
</Grid>
</UserControl>

```

Eight white keys are arranged horizontally in a *StackPanel*, but the five black keys are in a *Canvas*. This configuration allows the white keys to define the size of the control but lets the black keys sit on top of the white keys and cover parts of them.

The eight white keys go from C to C. Very often small keyboards start with C and end with C as well, but you don't want a pair of adjacent C keys where two octaves meet up. That's the reason why the last key has a *Visibility* of *Collapsed*. That *Visibility* property is set to *Visible* or *Collapsed* by the code-behind file based on the setting of the *LastKeyVisible* dependency property:

Project: SilentPiano | File: Octave.xaml.cs (excerpt)

```

public sealed partial class Octave : UserControl
{
    static readonly DependencyProperty lastKeyVisibleProperty =
        DependencyProperty.Register("LastKeyVisible",
            typeof(bool), typeof(Octave),
            new PropertyMetadata(false, OnLastKeyVisibleChanged));

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

    public static DependencyProperty LastKeyVisibleProperty
    {
        get { return lastKeyVisibleProperty; }
    }

    public bool LastKeyVisible
    {
        set { SetValue(LastKeyVisibleProperty, value); }
        get { return (bool)GetValue(LastKeyVisibleProperty); }
    }

    static void OnLastKeyVisibleChanged(DependencyObject obj,
        DependencyPropertyChangedEventArgs args)
    {
    }
}

```

```

{
    (obj as Octave).lastKey.Visibility =
        (bool)args.NewValue ? Visibility.Visible : Visibility.Collapsed;
}
}

```

All that's left is to instantiate two *Octave* objects in the MainPage.xaml file, the second one with *LastKeyVisible* set to *true*:

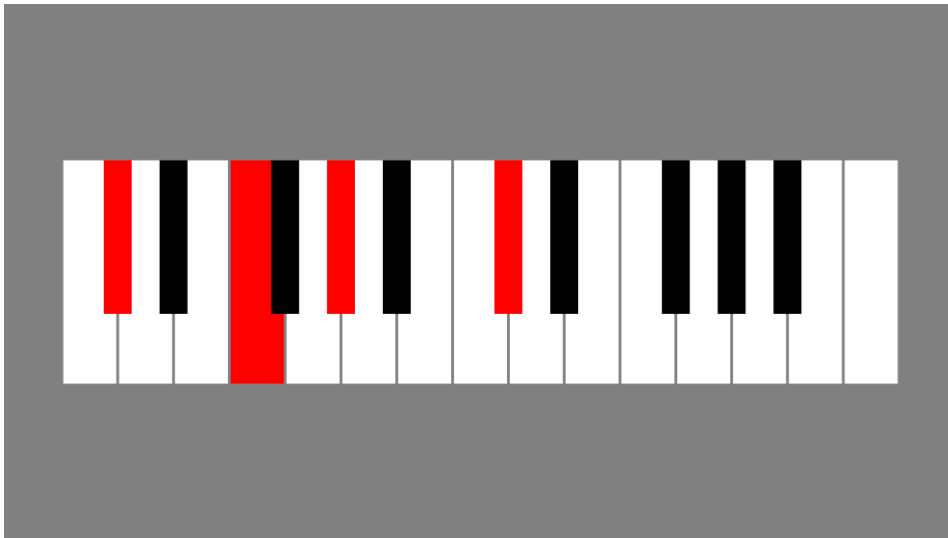
Project: SilentPiano | File: MainPage.xaml (excerpt)

```

<Page ... >
    <Grid Background="Gray">
        <StackPanel Orientation="Horizontal"
                    HorizontalAlignment="Center"
                    VerticalAlignment="Center">
            <local:Octave />
            <local:Octave LastKeyVisible="True" />
        </StackPanel>
    </Grid>
</Page>

```

And here I am playing my favorite chord (consonant with a major programming language):



Manipulation, Fingers, and Elements

The great advantage of the *Pointer* events is that you can track individual fingers. The great advantage of the *Manipulation* events is that you can't track individual fingers.

The *Manipulation* events combine multiple fingers—and by “multiple” we’re really often talking about “two”—into higher-level gestures such as pinch and rotate. These gestures correspond to

common graphics transforms: translation, scaling (although limited to equal scaling in the horizontal and vertical directions), and rotation. Capture is intrinsic to manipulation. As a bonus, inertia is also available.

Keep in mind that multiple fingers are combined into a single series of *Manipulation* event not for the entire window, but for each element handling these events. What this means is that you can use a finger or a pair of fingers to manipulate one element, while using another couple fingers to manipulate a second element.

UIElement defines five *Manipulation* events that an element generally receives in the following order (and take heed that the first two have extremely similar names):

- *ManipulationStarting*
- *ManipulationStarted*
- *ManipulationDelta* (many)
- *ManipulationInertiaStarting*
- *ManipulationDelta* (more)
- *ManipulationCompleted*

The *Control* class defines virtual methods corresponding to these five events named *OnManipulationStarting*, and so forth.

Although the mouse or pen can generate *Manipulation* events, these occur only when a mouse button is pressed or when the pen is touching the screen. A *ManipulationStarting* event occurs when a finger first touches an element, or the mouse button is pressed over an element, or the pen is touched to an element.

The *ManipulationStarted* event generally occurs soon after *ManipulationStarting* (but, as I'll discuss shortly, the key word here is "generally"). What follows is usually a bunch of *ManipulationDelta* events as the fingers move on the screen. When all fingers leave an element, *ManipulationInertiaStarting* is fired. The element continues to generate *ManipulationDelta* events representing inertia, but *ManipulationCompleted* indicates that the sequence is over.

Although the *ManipulationStarting* event occurs when a finger first touches an element (or a mouse click or pen press occurs), this event is not necessarily followed by a *ManipulationStarted* event and *ManipulationStarted* might be delayed a little. The problem is that the system must distinguish between a tap or a hold and an actual manipulation. *ManipulationStarted* is fired when the finger (or mouse or pen) moves a little bit.

For example, if you touch an element with a sweeping motion, *ManipulationStarting* is followed very quickly by *ManipulationStarted* and multiple *ManipulationDelta* events. But put a finger down in one place and hold it, and the *ManipulationStarted* event can be delayed quite some time.

If the user taps, or right-taps, or double-taps the screen, a *ManipulationStarted* event won't occur at all. However, it's possible for a *Holding* event to be fired after *ManipulationStarting* and for the user to then move the finger and get *ManipulationStarted* and the rest of the events. Another *Holding* event is then fired with a *HoldingState* property indicating *Canceled*.

By default, however, an element doesn't generate any *Manipulation* events whatsoever! The *Manipulation* events must first be enabled on a per-element basis. To allow a program to specify exactly what types of manipulation it wants, *UIElement* defines a *ManipulationMode* property of the enumeration type *ManipulationModes*. (The property name is singular; the enumeration name is plural.) The default setting of *ManipulationMode* is *ManipulationModes.System*, which for an application is equivalent to *ManipulationModes.None*. To enable an element for manipulation you'll need to set it to at least one other *ManipulationModes* member. The enumeration members are defined as bit flags, so you can combine them with the bitwise OR operator (`|`).

Although some applications need to handle all five *Manipulation* events, it's possible to write code that only examines *ManipulationDelta*.

This is the case with the *ManipulationTracker* program. The program displays a bunch of *CheckBox* controls for the members of the *ManipulationModes* enumeration and three *Rectangle* elements that you can manipulate. To ease some of the code and markup, a custom *CheckBox* derivative is used to store and display the *ManipulationModes* members:

Project: *ManipulationTracker* | File: *ManipulationModeCheckBox.cs*

```
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Input;

namespace ManipulationTracker
{
    public class ManipulationModeCheckBox : CheckBox
    {
        public ManipulationModes ManipulationModes { set; get; }
    }
}
```

Ten instances of this custom *CheckBox* are arranged in a *StackPanel* in *MainPage.xaml*, each identified both with the name of the enumeration member (with spaces inserted in the name to be more readable) and the integer value:

Project: *ManipulationTracker* | File: *MainPage.xaml* (excerpt)

```
<Page ... >
    <Page.Resources>
        <Style TargetType="local:ManipulationModeCheckBox">
            <Setter Property="Margin" Value="12 6 24 6" />
        </Style>

        <Style TargetType="Rectangle">
            <Setter Property="Width" Value="144" />
            <Setter Property="Height" Value="144" />
            <Setter Property="HorizontalAlignment" Value="Left" />
        </Style>
    </Page.Resources>
    <StackPanel>
```

```

        <Setter Property="VerticalAlignment" Value="Top" />
        <Setter Property="RenderTransformOrigin" Value="0.5 0.5" />
    </Style>
</Page.Resources>

<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" />
        <ColumnDefinition Width="*" />
    </Grid.ColumnDefinitions>

    <StackPanel Name="checkBoxPanel"
        Grid.Column="0">
        <local:ManipulationModeCheckBox Checked="OnManipulationModeCheckBoxChecked"
            Unchecked="OnManipulationModeCheckBoxChecked"
            Content="Translate X (1)"
            ManipulationModes="TranslateX" />

        <local:ManipulationModeCheckBox Checked="OnManipulationModeCheckBoxChecked"
            Unchecked="OnManipulationModeCheckBoxChecked"
            Content="Translate Y (2)"
            ManipulationModes="TranslateY" />

        <local:ManipulationModeCheckBox Checked="OnManipulationModeCheckBoxChecked"
            Unchecked="OnManipulationModeCheckBoxChecked"
            Content="Translate Rails X (4)"
            ManipulationModes="TranslateRailsX" />

        <local:ManipulationModeCheckBox Checked="OnManipulationModeCheckBoxChecked"
            Unchecked="OnManipulationModeCheckBoxChecked"
            Content="Translate Rails Y (8)"
            ManipulationModes="TranslateRailsY" />

        <local:ManipulationModeCheckBox Checked="OnManipulationModeCheckBoxChecked"
            Unchecked="OnManipulationModeCheckBoxChecked"
            Content="Rotate (16)"
            ManipulationModes="Rotate" />

        <local:ManipulationModeCheckBox Checked="OnManipulationModeCheckBoxChecked"
            Unchecked="OnManipulationModeCheckBoxChecked"
            Content="Scale (32)"
            ManipulationModes="Scale" />

        <local:ManipulationModeCheckBox Checked="OnManipulationModeCheckBoxChecked"
            Unchecked="OnManipulationModeCheckBoxChecked"
            Content="Translate Inertia (64)"
            ManipulationModes="TranslateInertia" />

        <local:ManipulationModeCheckBox Checked="OnManipulationModeCheckBoxChecked"
            Unchecked="OnManipulationModeCheckBoxChecked"
            Content="Rotate Inertia (128)"
            ManipulationModes="RotateInertia" />

        <local:ManipulationModeCheckBox Checked="OnManipulationModeCheckBoxChecked"

```

```

        Unchecked="OnManipulationModeCheckBoxChecked"
        Content="Scale Inertia (256)"
        ManipulationModes="ScaleInertia" />

<local:ManipulationModeCheckBox Checked="OnManipulationModeCheckBoxChecked"
    Unchecked="OnManipulationModeCheckBoxChecked"
    Content="All (0xFFFF)"
    ManipulationModes="All" />

</StackPanel>

<Grid Name="rectanglePanel"
    Grid.Column="1">
    <Rectangle Fill="Red">
        <Rectangle.RenderTransform>
            <CompositeTransform />
        </Rectangle.RenderTransform>
    </Rectangle>

    <Rectangle Fill="Green">
        <Rectangle.RenderTransform>
            <CompositeTransform />
        </Rectangle.RenderTransform>
    </Rectangle>

    <Rectangle Fill="Blue">
        <Rectangle.RenderTransform>
            <CompositeTransform />
        </Rectangle.RenderTransform>
    </Rectangle>
</Grid>
</Page>

```

In the larger cell of the *Grid* are three *Rectangle* elements, with the three colors of the state flag of Computerstan: red, green, and blue.

In the code-behind file, any checking or unchecking of the custom *CheckBox* controls causes a calculation of a new *ManipulationModes* value by combining enumeration members associated with the checked check boxes with the bitwise OR operator. This composite *ManipulationModes* value is then set to the *ManipulationMode* property of the three *Rectangle* elements:

Project: ManipulationTracker | File: MainPage.xaml.cs (excerpt)

```

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

    void OnManipulationModeCheckBoxChecked(object sender, RoutedEventArgs args)
    {
        // Get composite ManipulationModes value of checked CheckBoxes
        ManipulationModes manipulationModes = ManipulationModes.None;
    }
}

```

```

        foreach (UIElement child in checkBoxPanel.Children)
        {
            ManipulationModeCheckBox checkBox = child as ManipulationModeCheckBox;

            if ((bool)checkBox.IsChecked)
                manipulationModes |= checkBox.ManipulationModes;
        }

        // Set ManipulationMode property of each Rectangle
        foreach (UIElement child in rectanglePanel.Children)
            child.ManipulationMode = manipulationModes;
    }

    protected override void OnManipulationDelta(ManipulationDeltaRoutedEventArgs args)
    {
        // OriginalSource is always Rectangle because nothing else has its
        // ManipulationMode set to anything other than ManipulationModes.None
        Rectangle rectangle = args.OriginalSource as Rectangle;
        CompositeTransform transform = rectangle.RenderTransform as CompositeTransform;

        transform.TranslateX += args.Delta.Translation.X;
        transform.TranslateY += args.Delta.Translation.Y;

        transform.ScaleX *= args.Delta.Scale;
        transform.ScaleY *= args.Delta.Scale;

        transform.Rotation += args.Delta.Rotation;

        base.OnManipulationDelta(args);
    }
}

```

The final part of the program is the *OnManipulationDelta* override, which is a virtual method defined by the *Control* class that provides easier access to the *ManipulationDelta* event defined by *UIElement*. *ManipulationDelta* is the primary *Manipulation* event and indicates in what kind of manipulation the user's fingers are engaged.

Notice that the *OnManipulationDelta* override casts the *OriginalSource* property of the event arguments to *Rectangle* without even checking if the cast is successful. In theory, the *OriginalSource* property can be *MainPage* or any child of *MainPage*. However, only the *Rectangle* elements are enabled for manipulation, so only the *Rectangle* elements can generate *OnManipulationDelta* events.

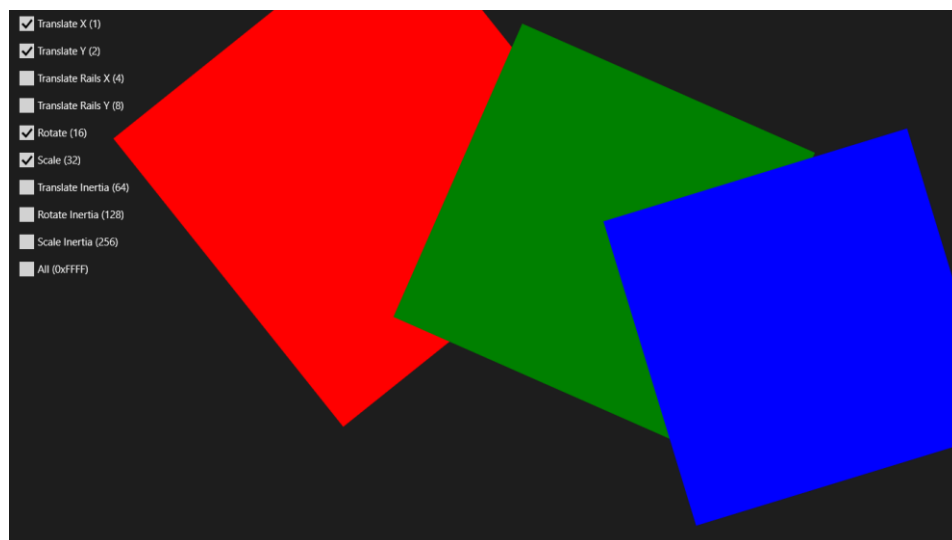
The override obtains the *CompositeTransform* set to the *RenderTransform* property of that particular *Rectangle* and adjusts five properties of the transform based on the *Delta* property of the event arguments. This *Delta* property is of type *ManipulationDelta*, a structure with four properties. (Watch out! This structure has the same name as the event that delivers it!) The values indicate change since the last *ManipulationDelta* event.

Three of the four *ManipulationDelta* properties are accessed by this code. The fourth is *Expansion*, and it's similar to *Scale* except expressed in pixels rather than a multiplicative scaling factor. The

Translation property of the *ManipulationDelta* structure indicates the average distance the fingers have moved since the last *ManipulationDelta* event, so these are just added to the *TranslateX* and *TranslateY* properties of the *CompositeTransform*. If there is no movement, these values are zero.

Similarly (but handled rather differently) the *Scale* property of the *ManipulationDelta* structure indicates the increase in the distance between the fingers since the last event. The *ScaleX* and *ScaleY* properties of the *CompositeTransform* are multiplied by this factor. (Because the *Manipulation* events don't provide separate scaling factors for horizontal and vertical scaling, all manipulation scaling is necessarily isotropic—equal in both directions.) If there is no scaling (or scaling has not been enabled), the *Scale* value is 1. The *Rotate* property of *ManipulationDelta* is a change in the rotation angle caused by turning the fingers relative to each other, and this is added to the *Rotation* property of *CompositeTransform*.

Check a few checkboxes, and you can indeed move the rectangles with the mouse or pen or use multiple fingers to move, scale, and rotate the rectangles pretty much as you might expect, even manipulating two or three at once:



For a program using *Manipulation* events, the rules are very simple: Always set the *ManipulationMode* property to a nondefault value on the element or elements that you want to generate *Manipulation* events. Each element you do this to generates its own independent stream of *Manipulation* events. You can set a handler for the *ManipulationDelta* event of the element itself, or you can handle that event by an ancestor in the visual tree.

I said that this manipulation works pretty much as you might expect, but it's not entirely correct. You'll notice that neither the code nor XAML has any reference to centers of scaling or rotation, except that the *RenderTransformOrigin* is set to the relative point (0.5, 0.5). Hence all scaling and rotation is relative to the center of each particular rectangle.

This is not correct behavior. For example, suppose you put one finger near a corner of a rectangle and hold it steady. You use a second finger to grab the opposite edge and pull it or rotate it. The scaling and rotation that results should be relative to the first finger. In other words, the part of the rectangle under that first finger should remain in place while the rest of the rectangle is scaled or rotated around it.

It turns out fixing this problem takes rather more complex logic, so I'm going to ignore it until later in this chapter.

Meanwhile, you can play with some of the other types of manipulation. There are three types of inertia—for translation, scaling, and rotation—and you can indeed flick or spin a rectangle right off the screen. There are ways to control the extent of inertia that I'll discuss later.

You can set an equivalent *ManipulationMode* property shown in the above screenshot like this in code:

```
rectangle.ManipulationMode = ManipulationModes.TranslateX |  
                             ManipulationModes.TranslateY |  
                             ManipulationModes.Scale |  
                             ManipulationModes.Rotate;
```

But not in XAML. Setting the *ManipulationMode* property in XAML is limited to just a single enumeration member, and in a real-life application, that would probably be *All*.

If you want to restrict manipulation to horizontal movement only, you can specify the *ManipulationModes* member *TranslateX* but not *TranslateY*:

```
rectangle.ManipulationMode = ManipulationModes.TranslateX;
```

Similarly, to restrict movement to the vertical, specify *TranslateY* but not *TranslateX*.

Two of the members of the *ManipulationModes* enumeration are called *TranslateRailsX* and *TranslateRailsY*. These only work as they are intended if you also specify both *TranslateX* and *TranslateY*. For example,

```
rectangle.ManipulationMode = ManipulationModes.TranslateX |  
                             ManipulationModes.TranslateY |  
                             ManipulationModes.TranslateRailsX;
```

This configuration still allows you to freely move the element in the horizontal and vertical directions. *However*, if the manipulation *begins* with movement in the horizontal direction, the element gets stuck in the rails (so to speak) and all further movement is restricted to the horizontal until you lift off your finger and start over again.

Similarly, this configuration restricts movement to the vertical if the manipulation begins with vertical movement:

```
rectangle.ManipulationMode = ManipulationModes.TranslateX |  
                             ManipulationModes.TranslateY |  
                             ManipulationModes.TranslateRailsY;
```

You can also specify both:

```
rectangle.ManipulationMode = ManipulationModes.TranslateX |  
                             ManipulationModes.TranslateY |  
                             ManipulationModes.TranslateRailsX |  
                             ManipulationModes.TranslateRailsY;
```

Begin dragging the element diagonally, and you can move it any which way. But begin with horizontal or vertical movement, and the element gets stuck in the rails.

The ManipulationTracker program uses the *Delta* property of the *ManipulationDeltaRoutedEventArgs* argument to make changes to a *CompositeTransform*:

```
transform.TranslateX += args.Delta.Translation.X;  
transform.TranslateY += args.Delta.Translation.Y;  
  
transform.ScaleX *= args.Delta.Scale;  
transform.ScaleY *= args.Delta.Scale;  
  
transform.Rotation += args.Delta.Rotation;
```

If you've examined the properties of *ManipulationDeltaRoutedEventArgs*, you'll have discovered that besides the *Delta* property there is a *Cumulative* property, also of type *ManipulationDelta*. The *Delta* property indicates change since the last *ManipulationDelta* event, but *Cumulative* indicates change since *ManipulationStarted*.

You might suspect that this *Cumulative* property is easier to work with than the *Delta* property because you can just transfer the values to the corresponding properties of the *CompositeTransform*, like this:

```
transform.TranslateX = args.Cumulative.Translation.X;  
transform.TranslateY = args.Cumulative.Translation.Y;  
  
transform.ScaleX = args.Cumulative.Scale;  
transform.ScaleY = args.Cumulative.Scale;  
  
transform.Rotation = args.Cumulative.Rotation;
```

With this code, the first time you manipulate an element, it seems to work just fine. But lift your fingers off and try another manipulation on the same element. The element jumps back to its original position in the upper-left corner of the screen!

The *Cumulative* property is not cumulative from the beginning of the program but only from a particular *ManipulationStarted* event.

Working with Inertia

The *Manipulation* events support inertia for translation, scaling, and rotation, but if you don't want inertia, simply don't specify those *ManipulationModes*.

If at any time you want to stop the manipulation or the inertia, the event arguments accompanying the *ManipulationStarted* and *ManipulationDelta* events have a *Complete* method, which causes a firing of the *ManipulationCompleted* event.

If you'd like to handle inertia on your own, you can do that as well. The event arguments accompanying the *ManipulationDelta* and *ManipulationInertiaStarting* events have a *Velocities* property that indicates the linear, scaling, and rotational velocities. For linear movement, the *Velocities* property is in pixels per millisecond, which I suspect aren't exactly intuitive units. As I experimented with giving on-screen objects a good flick with my finger, I came close to 10 pixels per millisecond but could never get it higher than that. That's 10,000 pixels per second, which is equivalent to about 100 inches per second, or about 8 feet per second, or not quite 6 miles per hour.

Default deceleration is provided, but if you'd like to set your own you need to handle the *ManipulationInertiaStarting* event. The *ManipulationInertiaStartingRoutedEventArgs* class defines these three properties:

- *TranslationBehavior* of type *InertiaTranslationBehavior*
- *ExpansionBehavior* of type *InertiaExpansionBehavior*
- *RotationBehavior* of type *InertiaRotationBehavior*

The *InertiaTranslationBehavior* class (for example) lets you set linear deceleration in two ways: with a *DesiredDisplacement* property in units of pixels (which is how much further you want the object to travel) or a *DesiredDeceleration* property in units of pixels per millisecond squared. Both properties have default values of NaN (not a number).

The *DesiredDeceleration* values are generally very small, but perhaps a physics review is in order here.

From basic physics, we know that with a constant acceleration applied to an object at rest, the distance the object travels in time t is

$$x = \frac{1}{2}at^2$$

For example, an object in free fall near the surface of the Earth without air resistance experiences a constant acceleration of 32 feet per second per second, or 32 feet per second squared. The object falls 16 feet at the end of 1 second, a total of 64 feet at the end of 2 seconds, and a total of 144 feet at the end of 3 seconds.

The velocity v is calculated as the first derivative of the distance with respect to time:

$$v = \frac{dx}{dt} = at$$

Again, for an object in free fall, the velocity is 32 feet per second at the end of 1 second, 64 feet per second at the end of 2 seconds, and 96 feet per second at the end of 3 seconds. Every second the velocity increases by 32 feet per second.

Deceleration is the same process in reverse. From that second formula we know that

$$a = \frac{v}{t}$$

If an object is travelling at velocity v , a constant deceleration a will bring it to rest in t seconds. If an on-screen object is travelling at the rate of 5 pixels per millisecond, you can use this formula to calculate a deceleration necessary to stop it in a fixed number of seconds, for example 5 seconds or 5000 milliseconds:

$$a = \frac{5}{5000} = 0.001 \text{ pixels/msec}^2$$

The FlickAndBounce project makes a similar calculation, except that the deceleration time is set via a *Slider* and can range from 1 second to 60 seconds. The XAML file includes that *Slider* and also an *Ellipse* with a *ManipulationMode* setting and three *Manipulation* events. Although *ManipulationMode* is set to *All* (because there's not much of an alternative in XAML), the program uses translation only and moves the *Ellipse* by setting *Canvas.Left* and *Canvas.Top* attached properties rather than a transform:

Project: FlickAndBounce | File: MainPage.xaml (excerpt)

```
<Page ... >
    <Grid Name="contentGrid"
        Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Canvas>
            <Ellipse Name="ellipse"
                Fill="Red"
                Width="144"
                Height="144"
                ManipulationMode="All"
                ManipulationStarted="OnEllipseManipulationStarted"
                ManipulationDelta="OnEllipseManipulationDelta"
                ManipulationInertiaStarting="OnEllipseManipulationInertiaStarting" />
        </Canvas>

        <Slider x:Name="slider"
            Value="5" Minimum="1" Maximum="60"
            VerticalAlignment="Bottom"
            Margin="24 0" />
    </Grid>
</Page>
```

Of course, any deceleration would be wasted if the object just skittered off past the edge of the screen. For that reason, the *ManipulationDelta* event detects when the *Ellipse* has moved past the edges of the screen. It moves the *Ellipse* back into view as if it's bounced off the edge and reverses further movement using the *xDirection* and *yDirection* fields.

Notice that this logic uses the *IsInertial* property for the bounce logic. It doesn't stop you from dragging the *Ellipse* past the edges of the screen:

Project: FlickAndBounce | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
```

```

{
    int xDirection;
    int yDirection;

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

    void OnEllipseManipulationStarted(object sender, ManipulationStartedRoutedEventArgs args)
    {
        // Initialize directions
        xDirection = 1;
        yDirection = 1;
    }

    void OnEllipseManipulationDelta(object sender, ManipulationDeltaRoutedEventArgs args)
    {
        // Find new position of ellipse regardless of edges
        double x = Canvas.GetLeft(ellipse) + xDirection * args.Delta.Translation.X;
        double y = Canvas.GetTop(ellipse) + yDirection * args.Delta.Translation.Y;

        if (args.IsInertial)
        {
            // Bounce it off the edges
            Size playground = new Size(contentGrid.ActualWidth - ellipse.Width,
                                       contentGrid.ActualHeight - ellipse.Height);

            while (x < 0 || y < 0 || x > playground.Width || y > playground.Height)
            {
                if (x < 0)
                {
                    x = -x;
                    xDirection *= -1;
                }
                if (x > playground.Width)
                {
                    x = 2 * playground.Width - x;
                    xDirection *= -1;
                }
                if (y < 0)
                {
                    y = -y;
                    yDirection *= -1;
                }
                if (y > playground.Height)
                {
                    y = 2 * playground.Height - y;
                    yDirection *= -1;
                }
            }
        }

        Canvas.SetLeft(ellipse, x);
    }
}

```

```

        Canvas.SetTop(ellipse, y);
    }

    void OnEllipseManipulationInertiaStarting(object sender,
                                             ManipulationInertiaStartingRoutedEventArgs args)
    {
        double maxVelocity = Math.Max(Math.Abs(args.Velocities.Linear.X),
                                       Math.Abs(args.Velocities.Linear.Y));

        args.TranslationBehavior.DesiredDeceleration = maxVelocity / (1000 * slider.Value);
    }
}

```

In the *ManipulationInertiaStarting* handler down at the bottom, the maximum of the absolute values of the horizontal and vertical velocities is used to calculate a deceleration based on a *Slider* value in seconds.

An XYSlider Control

An *XYSlider* control is similar to a *Slider* except that it allows you to select a point in a two-dimensional surface by changing the location of a crosshair (or something similar). At first, it seems like the *Pointer* events would be fine for this control, until you realize that the control really doesn't want to deal with multiple fingers. If it used the *Manipulation* events instead, it could avoid all that.

That was my original thought, anyway. But I'll let you decide if it works or not.

I derived *XYSlider* from *ContentControl* so that it could display whatever you wanted as a background simply by setting the *Content* property. Sitting on top of that is a crosshair that you move around with a finger, mouse, or pen. The control has one property, *Value* of type *Point*, and a *ValueChanged* event. The *X* and *Y* coordinates of the *Point* property are normalized to the range 0 to 1 relative to the content, which relieves the control of defining *Minimum* and *Maximum* values like *RangeBase* or an *IsDirectionReversed* property like *Slider*. (Actually it would need a pair of *IsDirectionReversed* properties for the *X* and *Y* axis.)

The control definition itself is templateless but indicates that it wants two parts in the template: the customary *ContentPresenter* normally found in a *ContentControl* template, and something that visually resembles a cross-hair. This cross-hair is moved around by code using *Canvas.Left* and *Canvas.Top* attached properties, strongly suggesting that the template needs to define this cross-hair in a *Canvas*.

Project: XYSliderDemo | File: XYSlider.cs

```

namespace XYSliderDemo
{
    [TemplatePartAttribute(Name = "ContentPresenterPart", Type = typeof(ContentPresenter))]
    [TemplatePartAttribute(Name = "CrossHairPart", Type = typeof(FrameworkElement))]
    public class XYSlider : ContentControl
    {
        ContentPresenter contentPresenter;
    }
}

```

```

FrameworkElement crossHairPart;

static readonly DependencyProperty valueProperty =
    DependencyProperty.Register("Value",
        typeof(Point), typeof(XYSlider),
        new PropertyMetadata(new Point(0.5, 0.5), OnValueChanged));

public event EventHandler<Point> ValueChanged;

public XYSlider()
{
    this.DefaultStyleKey = typeof(XYSlider);
}

public static DependencyProperty ValueProperty
{
    get { return valueProperty; }
}

public Point Value
{
    set { SetValue(ValueProperty, value); }
    get { return (Point)GetValue(ValueProperty); }
}

protected override void OnApplyTemplate()
{
    // Detach event handlers
    if (contentPresenter != null)
    {
        contentPresenter.ManipulationStarted -= OnContentPresenterManipulationStarted;
        contentPresenter.ManipulationDelta -= OnContentPresenterManipulationDelta;
        contentPresenter.SizeChanged -= OnContentPresenterSizeChanged;
    }

    // Get new parts
    crossHairPart = GetTemplateChild("CrossHairPart") as FrameworkElement;
    contentPresenter = GetTemplateChild("ContentPresenterPart") as ContentPresenter;

    // Attach event handlers
    if (contentPresenter != null)
    {
        contentPresenter.ManipulationMode = ManipulationModes.TranslateX |
                                           ManipulationModes.TranslateY;
        contentPresenter.ManipulationStarted += OnContentPresenterManipulationStarted;
        contentPresenter.ManipulationDelta += OnContentPresenterManipulationDelta;
        contentPresenter.SizeChanged += OnContentPresenterSizeChanged;
    }

    // Make cross-hair transparent to touch
    if (crossHairPart != null)
    {
        crossHairPart.IsHitTestVisible = false;
    }
}

```

```

        base.OnApplyTemplate();
    }

    void OnContentPresenterManipulationStarted(object sender,
                                              ManipulationStartedRoutedEventArgs args)
    {
        RecalculateValue(args.Position);
    }

    void OnContentPresenterManipulationDelta(object sender,
                                              ManipulationDeltaRoutedEventArgs args)
    {
        RecalculateValue(args.Position);
    }

    void OnContentPresenterSizeChanged(object sender, SizeChangedEventArgs args)
    {
        SetCrossHair();
    }

    void RecalculateValue(Point absolutePoint)
    {
        double x = Math.Max(0, Math.Min(1, absolutePoint.X / contentPresenter.ActualWidth));
        double y = Math.Max(0, Math.Min(1, absolutePoint.Y / contentPresenter.ActualHeight));
        this.Value = new Point(x, y);
    }

    void SetCrossHair()
    {
        if (contentPresenter != null && crossHairPart != null)
        {
            Canvas.SetLeft(crossHairPart, this.Value.X * contentPresenter.ActualWidth);
            Canvas.SetTop(crossHairPart, this.Value.Y * contentPresenter.ActualHeight);
        }
    }

    static void OnValueChanged(DependencyObject obj, DependencyPropertyChangedEventArgs args)
    {
        (obj as XYSlider).SetCrossHair();
        (obj as XYSlider).OnValueChanged((Point)args.NewValue);
    }

    protected void OnValueChanged(Point value)
    {
        if (ValueChanged != null)
            ValueChanged(this, value);
    }
}

```

When the *Value* property is set programmatically, the class must set the cross-hair to the correct position by multiplying the width and height of the *ContentPresenter* by the relative coordinates. This happens in the *SetCrossHair* method. The *ManipulationStarted* and *ManipulationDelta* event handlers

are set on the *ContentPresenter* object. Both call the *RecalculateValue* method to convert the absolute coordinates of the pointer to relative coordinates for the *Value* property.

The *ManipulationStarted* and *ManipulationDelta* handlers both reference a property of the event arguments named *Position*, which I haven't mentioned yet. For a mouse or pen, this *Position* property is simply the location of the mouse pointer or pen tip relative to the control generating these *Manipulation* events—the *ContentPresenter* in this case. For touch, the *Position* property is the average location of all the fingers involved in the manipulation. It provides a convenient way to deal with multiple fingers when you really want the position of only one finger.

The *MainPage.xaml* file instantiates an *XYSlider* and references a flattened map of the earth that I obtained from a NASA website. But most of the XAML file is dedicated to defining a template for the *XYSlider* and particularly the cross-hair. Notice that I put the *ContentPresenter* and the *Canvas* in a *Grid* and assigned some properties to the *Grid* normally assigned to the *ContentPresenter*. This means that the upper-left corners of the *ContentPresenter* and *Canvas* are aligned, which makes it easier to convert between *ContentPresenter* coordinates and relative coordinates:

Project: XYSliderDemo | File: *MainPage.xaml* (excerpt)

```
<Page ... >
  <Page.Resources>
    <ControlTemplate x:Key="xySliderTemplate" TargetType="local:XYSlider">
      <Border BorderBrush="{TemplateBinding BorderBrush}"
        BorderThickness="{TemplateBinding BorderThickness}"
        Background="{TemplateBinding Background}">

        <Grid Margin="{TemplateBinding Padding}"
          HorizontalAlignment="{TemplateBinding HorizontalContentAlignment}"
          VerticalAlignment="{TemplateBinding VerticalContentAlignment}">

          <ContentPresenter Name="ContentPresenterPart"
            Content="{TemplateBinding Content}"
            ContentTemplate="{TemplateBinding ContentTemplate}" />

          <Canvas>
            <Path Name="CrossHairPart"
              Stroke="{TemplateBinding Foreground}"
              StrokeThickness="3"
              Fill="Transparent">
              <Path.Data>
                <GeometryGroup FillRule="Nonzero">
                  <EllipseGeometry RadiusX="48" RadiusY="48" />
                  <EllipseGeometry RadiusX="6" RadiusY="6" />
                  <LineGeometry StartPoint="-48 0" EndPoint="-6 0" />
                  <LineGeometry StartPoint="48 0" EndPoint="6 0" />
                  <LineGeometry StartPoint="0 -48" EndPoint="0 -6" />
                  <LineGeometry StartPoint="0 48" EndPoint="0 6" />
                </GeometryGroup>
              </Path.Data>
            </Path>
          </Canvas>
        </Grid>
      </Border>
    </ControlTemplate>
  </Page.Resources>
</Page>
```

```

</ControlTemplate>

<Style TargetType="local:XYSlider">
    <Setter Property="Template" Value="{StaticResource xySliderTemplate}" />
</Style>
</Page.Resources>

<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <local:XYSlider x:Name="xySlider"
        Grid.Row="0"
        Margin="48"
        ValueChanged="OnXYSliderValueChanged">
        <!-- Image courtesy of NASN/JPL-Caltech (http://maps.jpl.nasa.gov) -->
        <Image Source="Images/ear0xuu2.jpg" />
    </local:XYSlider>

    <TextBlock Name="label"
        Grid.Row="1"
        Style="{StaticResource SubheaderTextStyle}"
        HorizontalAlignment="Center" />
</Grid>
</Page>

```

The code-behind file has a handler for the *ValueChanged* event of *XYSlider* and uses that to display the corresponding longitude and latitude. Just to check that the code works the other way, it also uses the *Geolocator* class to obtain the current geographical location of the computer on which the program is running:

Project: XYSliderDemo | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    bool manualChange = false;

    public MainPage()
    {
        this.InitializeComponent();

        // Initialize position of cross-hair in XYSlider
        Loaded += async (sender, args) =>
        {
            Geolocator geolocator = new Geolocator();

            // Might not have permission!
            try
            {
                Geoposition position = await geolocator.GetGeopositionAsync();

                if (!manualChange)

```



```

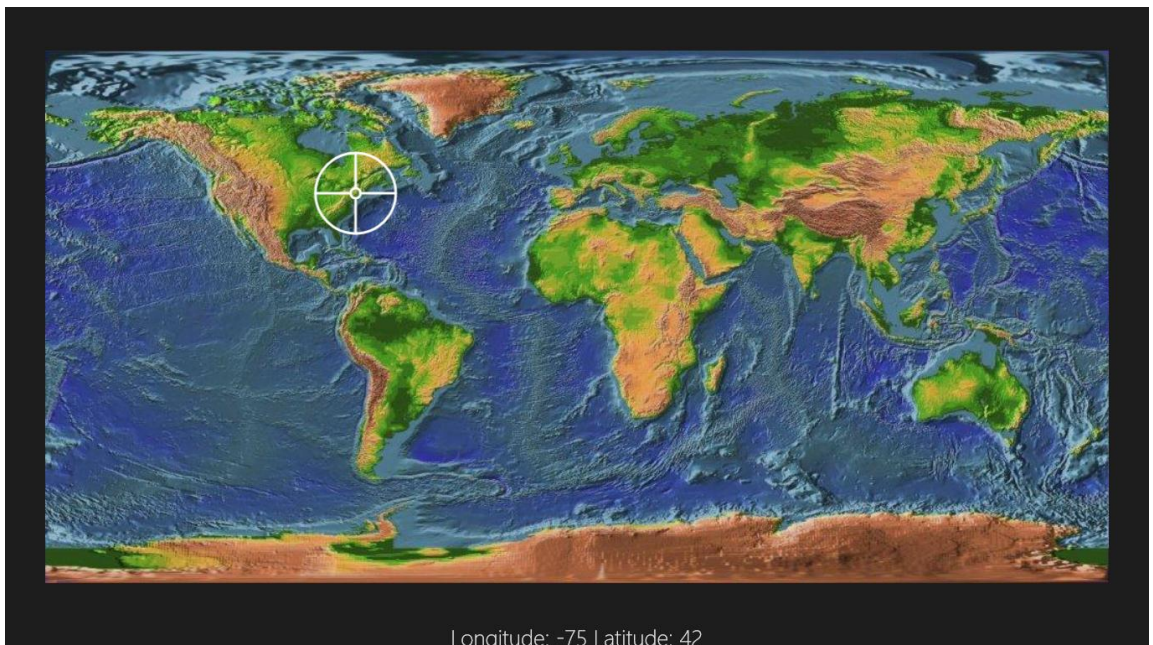
        {
            double x = (position.Coordinate.Longitude + 180) / 360;
            double y = (90 - position.Coordinate.Latitude) / 180;
            xySlider.Value = new Point(x, y);
        }
    }
    catch
    {
    }
};
}

void OnXYSliderValueChanged(object sender, Point point)
{
    double longitude = 360 * point.X - 180;
    double latitude = 90 - 180 * point.Y;
    label.Text = String.Format("Longitude: {0:F0} Latitude: {1:F0}",
                                longitude, latitude);
    manualChange = true;
}
}

```

Using the *Geolocator* class requires that you edit the *Package.appxmanifest* class to request Location capabilities. In Visual Studio, select the *Package.appxmanifest* file, select the Capabilities tab, and click Location. At run time, Windows 8 will then ask the user if it's OK for the program to know the computer's location. If the user denies permission, the *GetGeopositionAsync* call raises an exception.

Here's how it looks:



In an earlier version of this control that I wrote for Windows Phone 7, I used a templated *Thumb* for the cross-hair. I wasn't happy with that version because it required the user to drag the *Thumb* from its current location to a new location. For this new version, I wanted the cross-hair to snap to a new position with a simple touch.

But I'm not sure this version entirely succeeds either. As I mentioned earlier (and as you'll experience), simply touching a location does not snap the cross-hair to that point because some movement is required before the *ManipulationStarted* event is fired.

At first I thought I could make it respond faster by substituting a *PointerPressed* event for the *ManipulationStarted* event. However, apparently the simple act of calling the *GetCurrentPoint* on the *PointerRoutedEventArgs* object inhibits *Manipulation* events.

Perhaps this is a case where the *Pointer* events are really best, and if there are multiple fingers attempting to move the crosshair they should just be averaged together. I wouldn't be surprised if there's a better version of *XYSlider* in the next chapter, when it's used for a color-selection control in a bitmap-based finger-painting program.

Centered Scaling and Rotation

When I first introduced the scaling and rotation features of the *Manipulation* events, I mentioned that applying these transforms with reference to a center point was a little tricky. Yet, in many cases it's important. The satisfaction of a touch interface depends a lot on how close the connection feels between a user's fingers and on-screen objects.

There is a technique to determine the scaling and rotation center involving the *Position* property that I used in the last section. This property is the average of the positions of all the fingers relative to the element being manipulated. It is not the center of scaling and rotation, but it can be used to derive that center.

The CenteredTransforms project has a XAML file that references a bitmap on my website:

Project: CenteredTransforms | File: MainPage.xaml (excerpt)

```
<Page ... >
  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Image Name="image"
      Source="http://www.charlespetzold.com/pw6/PetzoldJersey.jpg"
      Stretch="None"
      HorizontalAlignment="Left"
      VerticalAlignment="Top">
      <Image.RenderTransform>
        <TransformGroup x:Name="xformGroup">
          <MatrixTransform x:Name="matrixXform" />
          <CompositeTransform x:Name="compositeXform" />
        </TransformGroup>
      </Image.RenderTransform>
    </Image>
  </Grid>
</Page>
```

```

        </Grid>
    </Page>

```

Notice that the *RenderTransform* property is now set to a *TransformGroup* containing both a *MatrixTransform* and a *CompositeTransform*.

The code-behind file enables all forms of *Manipulation* except those involving rails:

Project: CenteredTransforms | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();
        image.ManipulationMode = ManipulationModes.All &
                                ~ManipulationModes.TranslateRailsX &
                                ~ManipulationModes.TranslateRailsY;
    }

    protected override void OnManipulationDelta(ManipulationDeltaRoutedEventArgs args)
    {
        // Make this the entire transform to date
        matrixXform.Matrix = xformGroup.Value;

        // Use that to transform the Position property
        Point center = matrixXform.TransformPoint(args.Position);

        // That becomes the center of the new incremental transform
        compositeXform.CenterX = center.X;
        compositeXform.CenterY = center.Y;

        // Set the other properties
        compositeXform.TranslateX = args.Delta.Translation.X;
        compositeXform.TranslateY = args.Delta.Translation.Y;
        compositeXform.ScaleX = args.Delta.Scale;
        compositeXform.ScaleY = args.Delta.Scale;
        compositeXform.Rotation = args.Delta.Rotation;

        base.OnManipulationDelta(args);
    }
}

```

The *OnManipulationDelta* override juggles around the three transform objects defined in the XAML file. At any time, the *Value* property of the *TransformGroup* (which is a *Matrix* value) represents the entire transform, which is the product of the transforms represented by the *MatrixTransform* and *CompositeTransform* objects. The *ManipulationDelta* handler first sets the *Matrix* value from the *TransformGroup* to the *MatrixTransform*, which means that the *MatrixTransform* is now the entire transform up to this point. This transform is also applied to the *Position* property, and that becomes the *CenterX* and *CenterY* properties for the *CompositeTransform*. The new values from the *ManipulationDelta* structure can then be set directly to the other properties of the *CompositeTransform*.

Does it work? You'll definitely want to try it out since you can't tell from this screenshot:



Try holding one finger still on a corner and pulling the opposite corner away or rotate it, and you'll see that the image follows your fingers—given the restriction of the isotropic scaling, of course.

To make this technique a little easier to use, I wrote a tiny class called *ManipulationManager* that performs this calculation in its own private collection of transforms created in the constructor and saved in fields:

Project: ManipulationManagerDemo | File: ManipulationManager.cs

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

namespace ManipulationManagerDemo
{
    public class ManipulationManager
    {
        TransformGroup xformGroup;
        MatrixTransform matrixXform;
        CompositeTransform compositeXform;

        public ManipulationManager()
        {
            xformGroup = new TransformGroup();
            matrixXform = new MatrixTransform();
            xformGroup.Children.Add(matrixXform);
            compositeXform = new CompositeTransform();
            xformGroup.Children.Add(compositeXform);
            this.Matrix = Matrix.Identity;
        }
    }
}
```

```

public Matrix Matrix { private set; get; }

public void AccumulateDelta(Point position, ManipulationDelta delta)
{
    matrixXform.Matrix = xformGroup.Value;
    Point center = matrixXform.TransformPoint(position);
    compositeXform.CenterX = center.X;
    compositeXform.CenterY = center.Y;
    compositeXform.TranslateX = delta.Translation.X;
    compositeXform.TranslateY = delta.Translation.Y;
    compositeXform.ScaleX = delta.Scale;
    compositeXform.ScaleY = delta.Scale;
    compositeXform.Rotation = delta.Rotation;
    this.Matrix = xformGroup.Value;
}
}
}

```

The public *AccumulateDelta* method accepts a *ManipulationDelta* value directly and calculates a new *Matrix* property. This allows elements that must be manipulated in this way to have only a single transform:

Project: ManipulationManagerDemo | File: MainPage.xaml (excerpt)

```

<Page ... >
    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Image Name="image"
            Source="http://www.charlespetzold.com/pw6/PetzoldJersey.jpg"
            Stretch="None"
            HorizontalAlignment="Left"
            VerticalAlignment="Top">
            <Image.RenderTransform>
                <MatrixTransform x:Name="matrixXform" />
            </Image.RenderTransform>
        </Image>
    </Grid>
</Page>

```

The code-behind file creates an instance of *ManipulationManager* and uses that to calculate a new transform for the *Image*:

Project: ManipulationManagerDemo | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    ManipulationManager manipulationManager = new ManipulationManager();

    public MainPage()
    {
        this.InitializeComponent();

        image.ManipulationMode = ManipulationModes.All &
                                ~ManipulationModes.TranslateRailsX &
                                ~ManipulationModes.TranslateRailsY;
    }
}

```

```
protected override void OnManipulationDelta(ManipulationDeltaRoutedEventArgs args)
{
    manipulationManager.AccumulateDelta(args.Position, args.Delta);
    matrixXform.Matrix = manipulationManager.Matrix;
    base.OnManipulationDelta(args);
}
}
```

If you had multiple manipulable objects on the screen, you'd need an instance of *ManipulationManager* for each one. In the next chapter I'll use a variation of *ManipulationManager* in a *PhotoScatter* project that displays the images in your *Pictures* directory and lets you pore through them with your fingers.

Single-Finger Rotation

Although the *ManipulationStarting* event doesn't necessarily signal that a manipulation will actually occur, it offers a few ways for a program to initialize the manipulation, all involving properties of *ManipulationStartingRoutedEventArgs*:

- The *Mode* property is of the familiar enumeration type *ManipulationModes*, and here it lets you set the types of manipulation you want to handle. But keep in mind that you'll get a *ManipulationStarting* event only if the element has its *ManipulationMode* property set to something other than *ManipulationModes.None* or *ManipulationModes.System*.
- The *Container* property is read-only in all the other *Manipulation* events but writeable in the *ManipulationStarting* event. By default, the *Container* property is the same as the *OriginalSource* property, but in later events it's the element that the *Position* property is relative to. If you want the *Position* property to be relative to an element other than *OriginalSource*, set the *Container* property to that element.
- The *Pivot* property enables single-finger rotation, and that's what I'll show you here.

Suppose a photograph is sitting on a table. You touch your finger to a corner and pull it towards you. Does the photograph stay in the same orientation? Not necessarily. If you're touching it fairly lightly, friction between the table and photograph causes the photograph to rotate a bit and the rest of it drags behind the corner that you're pulling.

You get a similar effect with single-finger rotation, but you need to use the technique I just showed you for rotating objects around a center. Indeed, this XAML file is basically the same as the *CenteredTransforms* project:

Project: *SingleFingerRotate* | File: *MainPage.xaml* (excerpt)

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

```

        Stretch="None"
        HorizontalAlignment="Left"
        VerticalAlignment="Top"
        RenderTransformOrigin="0 0">
        <Image.RenderTransform>
            <TransformGroup x:Name="xformGroup">
                <MatrixTransform x:Name="matrixXform" />
                <CompositeTransform x:Name="compositeXform" />
            </TransformGroup>
        </Image.RenderTransform>
    </Image>
</Grid>
</Page>

```

The code-behind file is nearly identical as well with the exception of the *OnManipulationStarting* override:

Project: SingleFingerRotate | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();

        image.ManipulationMode = ManipulationModes.All &
            ~ManipulationModes.TranslateRailsX &
            ~ManipulationModes.TranslateRailsY;
    }

    protected override void OnManipulationStarting(ManipulationStartingRoutedEventArgs args)
    {
        args.Pivot = new ManipulationPivot(new Point(image.ActualWidth / 2,
                                                    image.ActualHeight / 2),
                                           50);
        base.OnManipulationStarting(args);
    }

    protected override void OnManipulationDelta(ManipulationDeltaRoutedEventArgs args)
    {
        // Make this the entire transform to date
        matrixXform.Matrix = xformGroup.Value;

        // Use that to transform the Position property
        Point center = matrixXform.TransformPoint(args.Position);

        // That becomes the center of the new incremental transform
        compositeXform.CenterX = center.X;
        compositeXform.CenterY = center.Y;

        // Set the other properties
        compositeXform.TranslateX = args.Delta.Translation.X;
        compositeXform.TranslateY = args.Delta.Translation.Y;
        compositeXform.ScaleX = args.Delta.Scale;
        compositeXform.ScaleY = args.Delta.Scale;
    }
}

```

```

        compositeXform.Rotation = args.Delta.Rotation;

        base.OnManipulationDelta(args);
    }
}

```

The key here is setting the *Pivot* property of the *ManipulationStartingRoutedEventArgs* object to a *ManipulationPivot* object. This object provides two things:

- A center of rotation, almost always the center of the object being manipulated.
- A protection radius around the center, here set to 50 pixels.

Without that second item your finger can get very close to the center of the element, whereupon just a little movement can give it a big spin.

This is one of those programs you really have to try out for yourself to get a feel for how single-finger rotation adds some realism to the dragging operation.

Remember the *SliderSketch* program from Chapter 5, “Control Interaction”? Remember how you said “Shouldn’t these be dials rather than sliders?”? The *DialSketch* program that concludes this chapter uses a *Dial* control that incorporates single-finger rotation.

To make the *Dial* class a little easier to define, I decided it should derive from *RangeBase* just like *Slider*. This gives the control *Minimum*, *Maximum*, and *Value* properties all of type *double*, as well as a *ValueChanged* event. The *double* values in this control, however, are rotation angles, and the only enabled manipulation mode is rotation:

Project: *DialSketch* | File: *Dial.cs*

```

using System;
using Windows.Foundation;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Input;

namespace DialSketch
{
    public class Dial : RangeBase
    {
        public Dial()
        {
            ManipulationMode = ManipulationModes.Rotate;
        }

        protected override void OnManipulationStarting(ManipulationStartingRoutedEventArgs args)
        {
            args.Pivot = new ManipulationPivot(new Point(this.ActualWidth / 2,
                                                         this.ActualHeight / 2),
                                              48);
            base.OnManipulationStarting(args);
        }

        protected override void OnManipulationDelta(ManipulationDeltaRoutedEventArgs args)

```



```

    {
        this.Value = Math.Max(this.Minimum,
                               Math.Min(this.Maximum, this.Value + args.Delta.Rotation));

        base.OnManipulationDelta(args);
    }
}
}

```

That's it! Of course, it doesn't have a template yet, nor does it touch any transforms. It just sets a new *Value* property (which causes *RangeBase* to fire a *ValueChanged* event), and it expects everything else to be implemented elsewhere.

Two of these *Dial* controls are instantiated in the XAML file for *DialSketch*. The Resources section is devoted to supplying a *Style* for these two controls, including a *ControlTemplate*. The *Dial* control requires visuals that let the user know it's rotating, so the template uses a dashed line with very short dashes to simulate tick marks.

Notice the *Minimum* and *Maximum* values set on the *Dial*. These imply that the *Dial* can be rotated 10 full times between its minimum and maximum positions. To draw a line from one edge of the *DialSketch* canvas to the opposite edge, you need to turn the dial 10 times.:

Project: *DialSketch* | File: *MainPage.xaml* (excerpt)

```

<Page ... >
    <Page.Resources>
        <Style TargetType="local:Dial">
            <Setter Property="Minimum" Value="-1800" />
            <Setter Property="Maximum" Value="1800" />
            <Setter Property="RenderTransformOrigin" Value="0.5 0.5" />
            <Setter Property="Width" Value="144" />
            <Setter Property="Height" Value="144" />
            <Setter Property="Margin" Value="24" />
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate>
                        <Grid>
                            <Ellipse Fill="DarkRed" />
                            <Ellipse Stroke="Black"
                                StrokeThickness="12"
                                StrokeDashArray="0.1 1"
                                Margin="3" />
                            <Ellipse Fill="Black"
                                Width="6"
                                Height="6" />
                        </Grid>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </Page.Resources>

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

```

```

<Grid.RowDefinitions>
    <RowDefinition Height="*" />
    <RowDefinition Height="Auto" />
</Grid.RowDefinitions>

<Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto" />
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="Auto" />
</Grid.ColumnDefinitions>

<Border Grid.Row="0"
        Grid.Column="0"
        Grid.ColumnSpan="3"
        BorderBrush="{StaticResource ApplicationForegroundThemeBrush}"
        BorderThickness="3 0 0 3"
        Background="#C0C0C0"
        Padding="24">

    <Grid Name="drawingGrid">
        <Polyline Name="polyline"
                Stroke="#404040"
                StrokeThickness="3" />
    </Grid>
</Border>

<local:Dial x:Name="horzDial"
            Grid.Row="1"
            Grid.Column="0"
            Maximum="1800"
            ValueChanged="OnDialValueChanged">
    <local:Dial.RenderTransform>
        <RotateTransform />
    </local:Dial.RenderTransform>
</local:Dial>

<Button Content="Clear"
        Grid.Row="1"
        Grid.Column="1"
        HorizontalAlignment="Center"
        VerticalAlignment="Center"
        Click="OnClearButtonClick" />

<local:Dial x:Name="vertDial"
            Grid.Row="1"
            Grid.Column="2"
            Maximum="1800"
            ValueChanged="OnDialValueChanged">
    <local:Dial.RenderTransform>
        <RotateTransform />
    </local:Dial.RenderTransform>
</local:Dial>
</Grid>
</Page>

```

You'll notice that the *Maximum* settings are repeated on the individual *Dial* controls. In the version of Windows 8 that I'm using, the settings in the *Style* didn't seem to "take." Also notice that each *Dial* control has a *RotateTransform* attached to it.

The code-behind file initializes the *Polyline* to a point in the center. For each *ValueChanged* event from a *Dial*, the *RotateTranform* on the control is set and a new *Point* is added to the *Polyline*:

Project: DialSketch | File: MainPage.xaml.cs (excerpt)

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

        Loaded += (sender, args) =>
        {
            polyline.Points.Add(new Point(drawingGrid.ActualWidth / 2,
                                           drawingGrid.ActualHeight / 2));
        };
    }

    void OnDialValueChanged(object sender, RangeBaseValueChangedEventArgs args)
    {
        Dial dial = sender as Dial;
        RotateTransform rotate = dial.RenderTransform as RotateTransform;
        rotate.Angle = args.NewValue;

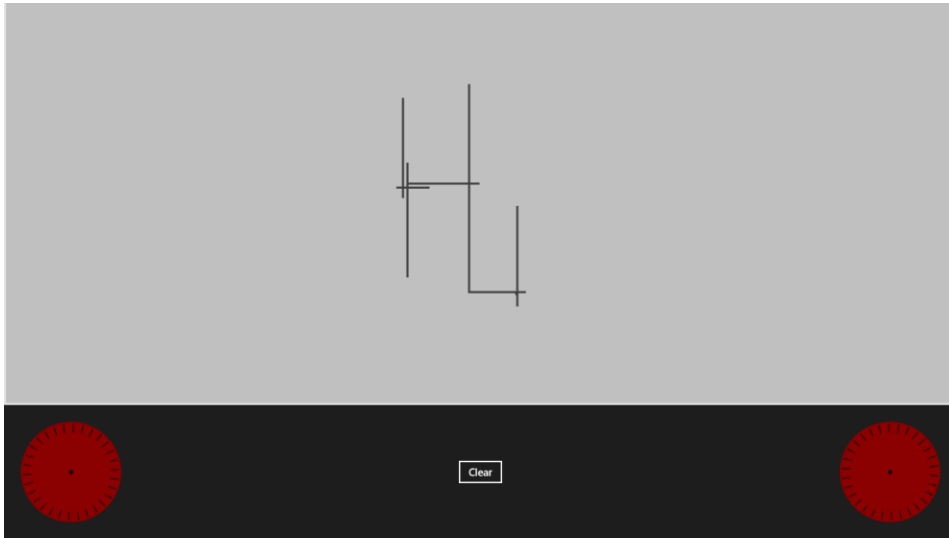
        double xFraction = (horzDial.Value - horzDial.Minimum) /
                           (horzDial.Maximum - horzDial.Minimum);

        double yFraction = (vertDial.Value - vertDial.Minimum) /
                           (vertDial.Maximum - vertDial.Minimum);

        double x = xFraction * drawingGrid.ActualWidth;
        double y = yFraction * drawingGrid.ActualHeight;
        polyline.Points.Add(new Point(x, y));
    }

    void OnClearButtonClick(object sender, RoutedEventArgs args)
    {
        polyline.Points.Clear();
    }
}
```

Of course, the program is still impossible to use, but it says "Hi" the best it can:



Chapter 14

Bitmaps

We've been working with bitmap images since the early pages of this book: displaying them, using them for brushes, stretching them, skewing them, and rotating them. But this chapter is all about reaching into the inner soul of bitmaps and manipulating their pixel bits. Every program in this chapter makes use of the *WriteableBitmap* class, which derives from *ImageSource* and therefore can be used with *Image* and *ImageBrush*:

Object

DependencyObject

ImageSource

BitmapSource

BitmapImage

WriteableBitmap

From *BitmapSource*, *WriteableBitmap* inherits a *SetSource* method that you can use to load a bitmap file through an object that implements *IRandomAccessStream*.

What makes *WriteableBitmap* different is that it defines a *PixelBuffer* property that gives you access to the pixel bits. You can manipulate the pixels of an existing image or create an entire image from scratch. This chapter also discusses reading and writing various formats of image files (such as PNG and JPEG).

If you're familiar with the Silverlight version of *WriteableBitmap*, you might be disappointed to learn that the Windows Runtime version does not implement the *Render* method that allows you to render any *UIElement* on the surface of the image. This greatly limits *WriteableBitmap* for several common purposes.

For example, if you need to draw regular graphics primitives (such as lines) on a bitmap such as in a finger-painting program, the absence of a *Render* method is a real problem. In the final version of this chapter (that is, when the final version of this book is published in fall 2012), I'll show you how to draw lines on a bitmap algorithmically, and in Chapter 16, "Going Native," I'll show you how to use *SurfaceImageSource*, which also derives from *ImageSource* and can be drawn upon using DirectX drawing operations from C++ code.

It is not my policy to discuss third-party libraries in books about APIs such as the Windows Runtime, but if you need to draw complex graphics on bitmaps, you might find *WriteableBitmapEx* to be useful. This is available at <http://writeablebitmapex.codeplex.com>.

Pixel Bits

A bitmap image has an integral number of rows and columns. For any instance of a class that derives from *BitmapSource*, these dimensions are available from the *PixelHeight* and *PixelWidth* properties.

To work with the actual pixel bits, you need to know how each pixel is stored in the bitmap. This is sometimes referred to as the bitmap's "color format" and could range from 1 bit per pixel (in a bitmap capable of only black and white) to 1 byte per pixel (in a gray-shade bitmap or a bitmap with a 256-color palette) to 3 or 4 bytes per pixel (for full-color with or without transparency), or even higher.

However, when working with *WriteableBitmap*, a uniform color format has been established. In every *WriteableBitmap*, each pixel consists of four bytes. The total number of bytes in the bitmap is therefore

*PixelHeight * PixelWidth * 4*

The image begins with the topmost row, and each row goes from left to right. There is no row padding. For each pixel, the bytes are in this order:

Blue, Green, Red, Alpha

The bytes range from 0 to 255 just as in a *Color* value. The *WriteableBitmap* color values are assumed to be in accordance with sRGB ("standard RGB") and hence compatible with the Windows Runtime *Color* value.

The pixels are in a premultiplied alpha format. I'll discuss what that means shortly.

The order Blue, Green, Red, Alpha might seem backwards from how we usually refer to these color bytes (and their order in the *Color.FromArgb* method), but it makes more sense if you consider that a pixel is really a 32-bit unsigned integer with the Alpha value stored in the high byte and the Blue value in the low byte. That integer is stored in the bitmap in the little-endian order (lowest byte first) common in operating systems built around Intel microprocessors.

Let's make a custom image by creating a *WriteableBitmap* and filling it with pixels. Just to make the math easy, this *WriteableBitmap* will have 256 rows and 256 columns. The upper-left corner will be black, the upper-right corner will be blue, the lower-left corner will be red, and the lower-right corner will be magenta. This is a form of gradient, but it's not like any gradient available in the Windows Runtime.

Here's the XAML file with an *Image* element ready to receive an *ImageSource* derivative:

Project: CustomGradient | File: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Image Name="image" />
</Grid>
```

The code-behind file builds the *WriteableBitmap* in a handler for the *Loaded* event. Here's the

complete file so that you can see the *using* directives as well. *WriteableBitmap* itself is defined in the *Windows.UI.Xaml.Media.Imaging* namespace:

Project: CustomGradient | File: MainPage.xaml.cs (excerpt)

```
using System.IO;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Media.Imaging;

namespace CustomGradient
{
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
            Loaded += OnMainPageLoaded;
        }

        async void OnMainPageLoaded(object sender, RoutedEventArgs args)
        {
            WriteableBitmap bitmap = new WriteableBitmap(256, 256);
            byte[] pixels = new byte[4 * bitmap.PixelWidth * bitmap.PixelHeight];

            for (int y = 0; y < bitmap.PixelHeight; y++)
                for (int x = 0; x < bitmap.PixelWidth; x++)
                {
                    int index = 4 * (y * bitmap.PixelWidth + x);
                    pixels[index + 0] = (byte)x;    // Blue
                    pixels[index + 1] = 0;          // Green
                    pixels[index + 2] = (byte)y;    // Red
                    pixels[index + 3] = 255;        // Alpha
                }

            using (Stream pixelStream = bitmap.PixelBuffer.AsStream())
            {
                await pixelStream.WriteAsync(pixels, 0, pixels.Length);
            }
            bitmap.Invalidate();
            image.Source = bitmap;
        }
    }
}
```

The *WriteableBitmap* constructor requires a pixel width and height. The program allocates a *byte* array for the pixels based on those dimensions:

```
byte[] pixels = new byte[4 * bitmap.PixelWidth * bitmap.PixelHeight];
```

The array size for a *WriteableBitmap* is always calculated like this.

The loops for the rows and columns touch every pixel in the bitmap. An index into the *pixels* array

to reference a particular pixel is calculated like this:

```
int index = 4 * (y * bitmap.PixelWidth + x);
```

Each pixel can then be set in the order blue, green, red, alpha.

In this particular example, the two loops are addressing the pixels in the order in which they're stored, so *index* really doesn't have to be recalculated for every pixel. It could be initialized at zero initially and then incremented like so:

```
int index = 0;
for (int y = 0; y < bitmap.PixelHeight; y++)
    for (int x = 0; x < bitmap.PixelWidth; x++)
    {
        pixels[index++] = (byte)x;    // Blue
        pixels[index++] = 0;         // Green
        pixels[index++] = (byte)y;    // Red
        pixels[index++] = 255;       // Alpha
    }
```

This is almost assuredly somewhat faster than the approach I've used, but it's less versatile.

After the *byte* array has been filled, it must be transferred into the *WriteableBitmap*. This process seems puzzling on first inspection. The *PixelBuffer* property defined by *WriteableBitmap* is of type *IBuffer*, which defines only two properties: *Capacity* and *Length*.

Fortunately, an extension method named *AsStream* is defined to convert this *IBuffer* to a .NET *Stream* object:

```
Stream pixelStream = bitmap.PixelBuffer.AsStream();
```

To use this extension method, you must include a *using* directive for the *System.Runtime.InteropServices.WindowsRuntime* namespace.

You can then use the normal *Write* method defined by *Stream* to write the byte array to the *Stream* object, or you can use *WriteAsync* as I've done. Because the call merely transfers an array of bytes across the API, *Write* should be fast enough to justify the asynchronous operation. You can dispose of the *Stream* "manually" or you can let it be disposed of automatically, or you can put the *Stream* logic in a *using* statement as I've done:

```
using (Stream pixelStream = bitmap.PixelBuffer.AsStream())
{
    await pixelStream.WriteAsync(pixels, 0, pixels.Length);
}
```

If you retain this *Stream* object for further manipulation of the bitmap—perhaps the image changes over time—you'll need to call *Seek* on it to set the current position back to the beginning. But notice also that you have the option of writing only part of the *byte* array to the bitmap.

Whenever you change the pixels of a *WriteableBitmap*, it's a good idea to get into the habit of calling *Invalidate* on the bitmap:


```
bitmap.Invalidate();
```

This call requests that the bitmap be redrawn. The call isn't strictly required in this particular context, but it's important in others.

Finally, do not forget to display the final bitmap! This program simply sets it to the *Source* property of the *Image* element in the XAML file:

```
image.Source = bitmap;
```

And here's the result:



Let's try another. This next program I call *CircularGradient*. The gradient is based on the angle a particular pixel makes with the center of the bitmap. (The math is easier than you might think.)

The XAML file defines an *Ellipse* with an *ImageBrush* for the *Stroke* property. An animation rotates the *Ellipse* around its center:

Project: *CircularGradient* | File: *MainPage.xaml* (excerpt)

```
<Page ... >
  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Ellipse Width="576"
      Height="576"
      StrokeThickness="48"
      RenderTransformOrigin="0.5 0.5">
      <Ellipse.Stroke>
        <ImageBrush x:Name="imageBrush" />
      </Ellipse.Stroke>

      <Ellipse.RenderTransform>
        <RotateTransform x:Name="rotate" />
      </Ellipse.RenderTransform>
    </Ellipse>
  </Grid>
</Page>
```

```

</Grid>

<Page.Triggers>
    <EventTrigger>
        <BeginStoryboard>
            <Storyboard>
                <DoubleAnimation Storyboard.TargetName="rotate"
                                Storyboard.TargetProperty="Angle"
                                From="0" To="360" Duration="0:0:3"
                                RepeatBehavior="Forever" />
            </Storyboard>
        </BeginStoryboard>
    </EventTrigger>
</Page.Triggers>
</Page>

```

As the two loops march through the rows and columns of the bitmap, each pixel has a position (x, y) relative to the upper-left corner. The pixel in the center has the coordinate ($bitmap.PixelWidth / 2, bitmap.PixelHeight / 2$). By subtracting that center from an individual pixel and dividing by the bitmap width and height, the pixel coordinate is converted to values between $-1/2$ and $1/2$, which can then be passed to the *Math.Atan2* method to get exactly the angle we need:

Project: CircularGradient | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();
        Loaded += OnMainPageLoaded;
    }

    async void OnMainPageLoaded(object sender, RoutedEventArgs args)
    {
        WriteableBitmap bitmap = new WriteableBitmap(256, 256);
        byte[] pixels = new byte[4 * bitmap.PixelWidth * bitmap.PixelHeight];
        int index = 0;
        int centerX = bitmap.PixelWidth / 2;
        int centerY = bitmap.PixelHeight / 2;

        for (int y = 0; y < bitmap.PixelHeight; y++)
            for (int x = 0; x < bitmap.PixelWidth; x++)
            {
                double angle =
                    Math.Atan2(((double)y - centerY) / bitmap.PixelHeight,
                               ((double)x - centerX) / bitmap.PixelWidth);
                double fraction = angle / (2 * Math.PI);
                pixels[index++] = (byte)(fraction * 255);           // Blue
                pixels[index++] = 0;                               // Green
                pixels[index++] = (byte)(255 * (1 - fraction));     // Red
                pixels[index++] = 255;                              // Alpha
            }

        using (Stream pixelStream = bitmap.PixelBuffer.AsStream())

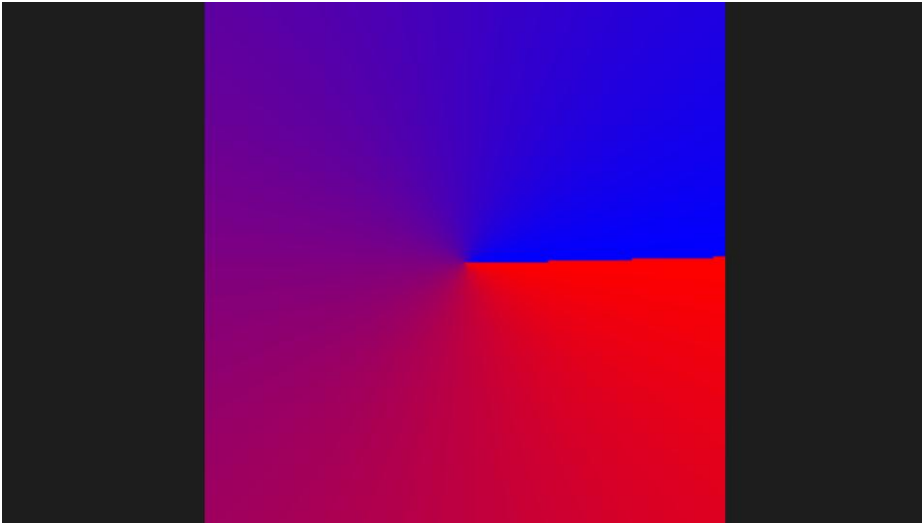
```

```

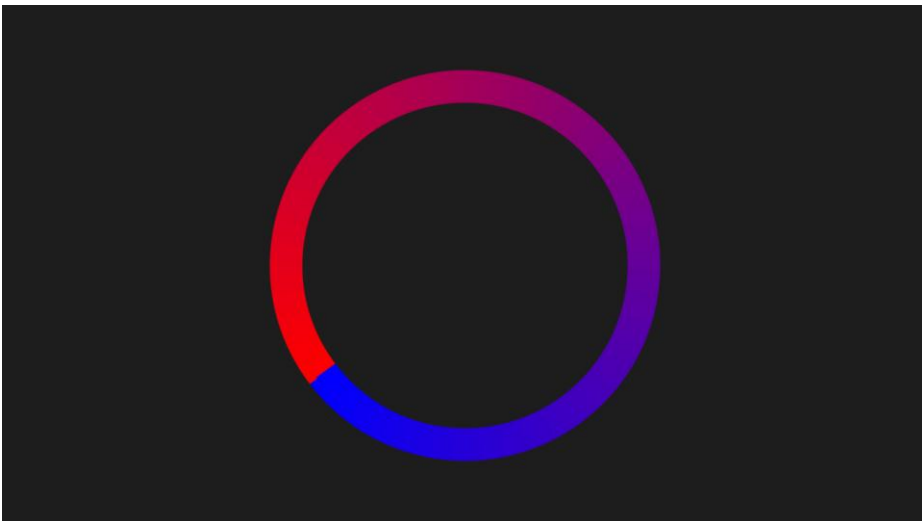
    {
        await pixelStream.WriteAsync(pixels, 0, pixels.Length);
    }
    bitmap.Invalidate();
    imageBrush.ImageSource = bitmap;
}
}

```

That angle is then converted to a fraction between 0 and 1 for calculating the gradient. Here's what the bitmap looks like in its entirety without animation:



However, it looks much more interesting when it's restricted to a circle and made to rotate. It seems as if the gradient itself is rotating:



As you've seen, brushes in the Windows Runtime are generally stretched to the element they're coloring. An *ImageBrush* does that as well, so in one sense the size of the underlying bitmap doesn't matter. But of course it does matter. A bitmap that is too small might not have the desired detail, while one that is too large is just a waste of pixels.

Transparency and Premultiplied Alphas

When a bitmap is rendered on a surface such as the video display, the pixels of the bitmap are generally not simply transferred to the video display surface. If the bitmap supports transparency, a pixel must be combined with the color of existing surface at that point based on the Alpha setting of that pixel. If the Alpha value is 255 (opaque), the bitmap pixel can be simply copied to the surface. If the Alpha value is 0 (transparent), it doesn't need to be copied at all. If the Alpha value is 128, the result is the average of the bitmap pixel and the surface pixel prior to the rendering.

The following formulas show this calculation for a single pixel. In reality the values A, R, G, and B range from 0 to 255, but the following simplified formulas assume they've been normalized to values 0 through 1. The subscripts indicate the "result" of rendering a partially transparent "bitmap" pixel on an existing "surface":

$$R_{result} = (1 - A_{bitmap}) \cdot R_{surface} + A_{bitmap} \cdot R_{bitmap}$$

$$G_{result} = (1 - A_{bitmap}) \cdot G_{surface} + A_{bitmap} \cdot G_{bitmap}$$

$$B_{result} = (1 - A_{bitmap}) \cdot B_{surface} + A_{bitmap} \cdot B_{bitmap}$$

Notice that second multiplication in each line. That's a multiplication that involves only the bitmap pixel itself and not the surface. The entire process can be speeded up if the R, G, and B values of the pixel have already been multiplied by the A value. This is called "premultiplied alpha."

For example, suppose a bitmap contains a pixel with the ARGB value (192, 40, 60, 255). That alpha value of 192 indicates 75% opacity (192 divided by 255). The equivalent pixel with a premultiplied alpha is (192, 30, 45, 192). The red, green, and blue values have been multiplied by 75%.

The pixel data encapsulated in a *WriteableBitmap* is assumed to have premultiplied alphas. For any pixel, none of the R, G, or B values should be greater than the A value. Nothing will "blow up" if that's not the case, but you won't get the colors and level of transparency you want.

Let's look at some examples. Back in Chapter 9, "Transforms," I showed you how to flip over an image and make it fade out so that it looked like a reflection. However, because the Windows Runtime doesn't support an opacity mask, I had to fade out the reflected image by covering it with a partially transparent rectangle.

In the *ReflectedAlphamage* project I take a different approach. The XAML file has two *Image* elements occupying the same top cell of a two-row *Grid*. The second *Image* element has a *RenderTransformOrigin* and *ScaleTransform* to flip it around its bottom edge, but no bitmap has been

specified:

Project: ReflectedAlphaImage | File: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <Grid.RowDefinitions>
        <RowDefinition Height="*" />
        <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <Image Source="http://www.charlespetzold.com/pw6/PetzoldJersey.jpg"
        HorizontalAlignment="Center" />

    <Image Name="reflectedImage"
        RenderTransformOrigin="0 1"
        HorizontalAlignment="Center">
        <Image.RenderTransform>
            <ScaleTransform ScaleY="-1" />
        </Image.RenderTransform>
    </Image>
</Grid>
```

The same bitmap referenced by the first *Image* element must be loaded independently in the code-behind file. (You might wonder if it's possible to obtain a *WriteableBitmap* based on the object that's set to the *Source* property of the first *Image* object. But that's an object of type *BitmapSource*, and you can't create a *WriteableBitmap* from a *BitmapSource*.) If it's not necessary to modify the bitmap loaded from the code-behind file, the code in the constructor would look something like this:

```
Loaded += async (sender, args) =>
{
    Uri uri = new Uri("http://www.charlespetzold.com/pw6/PetzoldJersey.jpg");
    RandomAccessStreamReference streamRef = RandomAccessStreamReference.CreateFromUri(uri);
    IRandomAccessStreamWithContentType fileStream = await streamRef.OpenReadAsync();
    WriteableBitmap bitmap = new WriteableBitmap(1, 1);
    bitmap.SetSource(fileStream);
    reflectedImage.Source = bitmap;
};
```

It's necessary to put this code in the *Loaded* handler because some asynchronous processing is involved. Notice that a *WriteableBitmap* can be created with essentially an "unknown" size if the data is coming from the *SetSource* method. When the *WriteableBitmap* reads that JPEG stream, it can figure out what the actual size is.

However, when that *fileStream* object is passed to the *SetSource* method of *WriteableBitmap* and when that *WriteableBitmap* is set to the *Source* property of the *Image* element, the bitmap has not yet been read into local memory. That loading occurs asynchronously within *WriteableBitmap*. This means that you can't yet start modifying the pixels because the pixels have not yet arrived! It would be nice if *WriteableBitmap* defined an event that indicated when *SetSource* completed loading the bitmap file like *BitmapImage* does, but that's not the case. Nor does the *ImageOpened* event of the *Image* element provide this information for a *WriteableBitmap*.

So, we're left with the job of loading in the bitmap file ourselves and then making the modifications to it. Some of this code can be simplified with other classes covered later in this chapter. Here's the *Loaded* handler:

Project: ReflectedAlphamage | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    public MainPage()
    {
        this.InitializeComponent();
        Loaded += OnMainPageLoaded;
    }

    async void OnMainPageLoaded(object sender, RoutedEventArgs args)
    {
        Uri uri = new Uri("http://www.charlespetzold.com/pw6/PetzoldJersey.jpg");
        RandomAccessStreamReference streamRef = RandomAccessStreamReference.CreateFromUri(uri);

        // Create a buffer for reading the stream
        Windows.Storage.Streams.Buffer buffer = null;

        // Read the entire file
        using (IRandomAccessStreamWithContentType fileStream = await streamRef.OpenReadAsync())
        {
            buffer = new Windows.Storage.Streams.Buffer((uint)fileStream.Size);
            await fileStream.ReadAsync(buffer, (uint)fileStream.Size, InputStreamOptions.None);
        }

        // Create WriteableBitmap with unknown size
        WriteableBitmap bitmap = new WriteableBitmap(1, 1);

        // Create a memory stream for transferring the data
        using (InMemoryRandomAccessStream memoryStream = new InMemoryRandomAccessStream())
        {
            await memoryStream.WriteAsync(buffer);
            memoryStream.Seek(0);

            // Use the memory stream as the Bitmap source
            bitmap.SetSource(memoryStream);
        }

        // Now get the pixels from the bitmap
        byte[] pixels = new byte[4 * bitmap.PixelWidth * bitmap.PixelHeight];
        int index = 0;

        using (Stream pixelStream = bitmap.PixelBuffer.AsStream())
        {
            await pixelStream.ReadAsync(pixels, 0, pixels.Length);

            // Apply opacity to the pixels
            for (int y = 0; y < bitmap.PixelHeight; y++)
            {
                double opacity = (double)y / bitmap.PixelHeight;
```

```

        for (int x = 0; x < bitmap.PixelWidth; x++)
            for (int i = 0; i < 4; i++)
            {
                pixels[index] = (byte)(opacity * pixels[index]);
                index++;
            }

        // Put the pixels back in the bitmap
        pixelStream.Seek(0, SeekOrigin.Begin);
        await pixelStream.WriteAsync(pixels, 0, pixels.Length);
    }
    bitmap.Invalidate();
    reflectedImage.Source = bitmap;
}
}

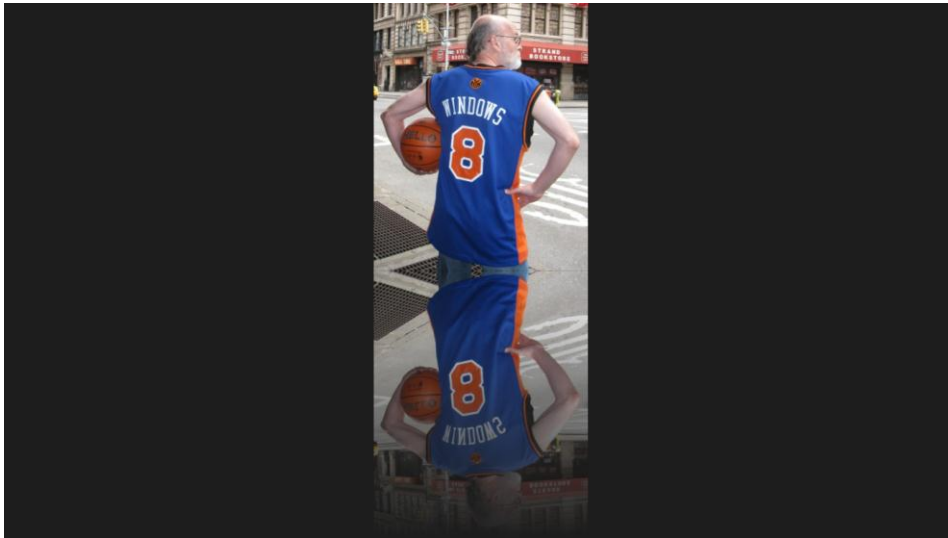
```

One objective here is to pass an object of type *IRandomAccessStream* to the *SetSource* method of the *WriteableBitmap*. However, immediately after this is done, we want to start working with the pixels of the resultant bitmap. This can't happen unless the file has been fully read.

That's the rationale for creating a *Buffer* object for reading the *fileStream* object, and then using that same *Buffer* object to write the contents to an *InMemoryRandomAccessStream*. As its name suggests, the *InMemoryRandomAccessStream* class implements the *IRandomAccessStream* interface so that it can be passed to the *SetSource* method of *WriteableBitmap*. (But notice that the stream position must first be set back to zero.)

It's important to realize that we're working with two very different chunks of data here. The *fileStream* is the PNG file, which is 82,824 bytes of compressed image data. The *InMemoryRandomAccessStream* is that same chunk of data. Once that stream has been passed to the *SetSource* method of *WriteableBitmap*, it is decoded into rows and columns of pixels. The *pixels* array is 512,000 bytes in size, and the *pixelStream* object references those decompressed pixels. The *pixelStream* object is first used to read the pixels into the *pixels* array and then to write them out.

Between those two calls is the actual application of the gradient opacity. If the pixels of a *WriteableBitmap* were not assumed by the Windows Runtime to have a premultiplied alpha format, only the Alpha byte would need to be modified. The premultiplied format requires the color bytes to be multiplied as well. Here's the result:



If you want to see what happens if you adjust only the Alpha byte, substitute the following code for the inner loop:

```
for (int i = 0; i < 4; i++)
{
    if (i == 3)
        pixels[index] = (byte)(opacity * pixels[index]);
    index++;
}
```

You get the transparency you want, but only if the background is white. If the background is black, there's no transparency at all! Look at the formulas and it all becomes clear.

Suppose you wanted to alter the CircularGradient project so that the gradient is from a solid color to complete transparency. Here's the altered code to set the four bytes:

```
pixels[index++] = (byte)(fraction * 255);    // Blue
pixels[index++] = 0;                        // Green
pixels[index++] = 0;                        // Red
pixels[index++] = (byte)(fraction * 255);    // Alpha
```

The Blue component and the Alpha component get the same setting. With a non-premultiplied Alpha format, the Blue component would always be 255. Here's the result:



A Radial Gradient Brush

One of the many mysterious missing pieces of the Windows Runtime is *RadialGradientBrush*, which is generally used to color a circle with a gradient from a point within that circle to the perimeter. One common use of *RadialGradientBrush* is to color a ball so that it looks as if some light is reflecting off an area near the upper-left corner.

I began writing my *RadialGradientBrushSimulator* class with an idea about animating the *GradientOrigin* property of this class in a XAML file. For that reason, I made *RadialGradientBrushSimulator* a *FrameworkElement* derivative even though it doesn't display anything on its own. By making it derive from *FrameworkElement* I could more easily instantiate the class in XAML. And because I was thinking about animations and bindings, I defined all the properties as dependency properties. Here's the part of the class containing little more than the dependency property overhead:

Project: RadialGradientBrushDemo | File: RadialGradientBrushSimulator.cs (excerpt)

```
public class RadialGradientBrushSimulator : FrameworkElement
{
    ...
    static readonly DependencyProperty gradientOriginProperty =
        DependencyProperty.Register("GradientOrigin",
            typeof(Point),
            typeof(RadialGradientBrushSimulator),
            new PropertyMetadata(new Point(0.5, 0.5), OnPropertyChanged));

    static readonly DependencyProperty innerColorProperty =
        DependencyProperty.Register("InnerColor",
            typeof(Color),
```

```

        typeof(RadialGradientBrushSimulator),
        new PropertyMetadata(Colors.White, OnPropertyChanged));

static readonly DependencyProperty outerColorProperty =
    DependencyProperty.Register("OuterColor",
        typeof(Color),
        typeof(RadialGradientBrushSimulator),
        new PropertyMetadata(Colors.Black, OnPropertyChanged));

static readonly DependencyProperty clipToEllipseProperty =
    DependencyProperty.Register("ClipToEllipse",
        typeof(bool),
        typeof(RadialGradientBrushSimulator),
        new PropertyMetadata(false, OnPropertyChanged));

public static DependencyProperty imageSourceProperty =
    DependencyProperty.Register("ImageSource",
        typeof(ImageSource),
        typeof(RadialGradientBrushSimulator),
        new PropertyMetadata(null));

public RadialGradientBrushSimulator()
{
    SizeChanged += OnSizeChanged;
}

public static DependencyProperty GradientOriginProperty
{
    get { return gradientOriginProperty; }
}

public static DependencyProperty InnerColorProperty
{
    get { return innerColorProperty; }
}

public static DependencyProperty OuterColorProperty
{
    get { return outerColorProperty; }
}

public static DependencyProperty ClipToEllipseProperty
{
    get { return clipToEllipseProperty; }
}

public static DependencyProperty ImageSourceProperty
{
    get { return imageSourceProperty; }
}

public Point GradientOrigin
{
    set { SetValue(GradientOriginProperty, value); }
}

```

```

        get { return (Point)GetValue(GradientOriginProperty); }
    }

    public Color InnerColor
    {
        set { SetValue(InnerColorProperty, value); }
        get { return (Color)GetValue(InnerColorProperty); }
    }
    public Color OuterColor
    {
        set { SetValue(OuterColorProperty, value); }
        get { return (Color)GetValue(OuterColorProperty); }
    }
    public bool ClipToEllipse
    {
        set { SetValue(ClipToEllipseProperty, value); }
        get { return (bool)GetValue(ClipToEllipseProperty); }
    }

    public ImageSource ImageSource
    {
        private set { SetValue(ImageSourceProperty, value); }
        get { return (ImageSource)GetValue(ImageSourceProperty); }
    }

    void OnSizeChanged(object sender, SizeChangedEventArgs args)
    {
        this.RefreshBitmap();
    }

    static void OnPropertyChanged(DependencyObject obj, DependencyPropertyChangedEventArgs args)
    {
        (obj as RadialGradientBrushSimulator).RefreshBitmap();
    }
    ...
}

```

In the *RefreshBitmap* method shown below, the class uses the *GradientOrigin*, *InnerColor*, *OuterColor*, and *ClipToEllipse* properties (as well as the *ActualWidth* and *ActualHeight* of the element) to create a *WriteableBitmap* that the class exposes through the *ImageSource* property, allowing another element in the XAML file to reference that through a binding to the *ImageSource* property of an *ImageBrush*.

It was then that I discovered that the algorithm to make an image of a radial gradient brush was not exactly trivial. Conceptually, you're dealing with an ellipse, although you can use the bitmap to color a rectangle or anything else. The color at the boundary of the ellipse is the *OuterColor* property. The *GradientOrigin* property of type *Point* is in relative coordinates. For example, a value of (0.5, 0.5) would set the *GradientOrigin* to the center of the ellipse. The color at the *GradientOrigin* is the property *InnerColor*.

For any point (x, y) within the bitmap, the algorithm needs to find an interpolation factor to calculate a color between *InnerColor* and *OuterColor*. This interpolation factor is based on a straight

line from the *GradientOrigin* through the point (x, y) to the circumference of the ellipse. Where the point (x, y) divides that line determines the value of the interpolation factor.

My strategy involved finding the intersection of the line from the *GradientOrigin* to (x, y) with the circumference of the ellipse. This involved solving a quadratic equation for every point in the bitmap.

Here's the *RefreshBitmap* method:

Project: RadialGradientBrushDemo | File: RadialGradientBrushSimulator.cs (excerpt)

```
public class RadialGradientBrushSimulator : FrameworkElement
{
    WriteableBitmap bitmap;
    byte[] pixels;
    Stream pixelStream;
    ...
    void RefreshBitmap()
    {
        if (this.ActualWidth == 0 || this.ActualHeight == 0)
        {
            this.ImageSource = null;
            bitmap = null;
            pixels = null;
            pixelStream = null;
            return;
        }

        if (bitmap == null || (int)this.ActualWidth != bitmap.PixelWidth ||
            (int)this.ActualHeight != bitmap.PixelHeight)
        {
            bitmap = new WriteableBitmap((int)this.ActualWidth, (int)this.ActualHeight);
            this.ImageSource = bitmap;
            pixels = new byte[4 * bitmap.PixelWidth * bitmap.PixelHeight];
            pixelStream = bitmap.PixelBuffer.AsStream();
        }
        else
        {
            for (int i = 0; i < pixels.Length; i++)
                pixels[i] = 0;
        }

        double xOrigin = 2 * this.GradientOrigin.X - 1;
        double yOrigin = 2 * this.GradientOrigin.Y - 1;

        byte aOutsideCircle = 0;
        byte rOutsideCircle = 0;
        byte gOutsideCircle = 0;
        byte bOutsideCircle = 0;

        if (!this.ClipToEllipse)
        {
            double opacity = this.OuterColor.A / 255.0;
            aOutsideCircle = this.OuterColor.A;
            rOutsideCircle = (byte)(opacity * this.OuterColor.R);
```

```

        gOutsideCircle = (byte)(opacity * this.OuterColor.G);
        gOutsideCircle = (byte)(opacity * this.OuterColor.B);
    }

    int index = 0;

    for (int yPixel = 0; yPixel < bitmap.PixelHeight; yPixel++)
    {
        // Calculate y relative to unit circle
        double y = 2.0 * yPixel / bitmap.PixelHeight - 1;

        for (int xPixel = 0; xPixel < bitmap.PixelWidth; xPixel++)
        {
            // Calculate x relative to unit circle
            double x = 2.0 * xPixel / bitmap.PixelWidth - 1;

            // Check if point is within circle
            if (x * x + y * y <= 1)
            {
                // relative length from gradient origin to point
                double length1 = 0;

                // relative length from point to unit circle
                // (length1 + length2 = 1)
                double length2 = 0;

                if (x == xOrigin && y == yOrigin)
                {
                    length2 = 1;
                }
                else
                {
                    // Remember: xCircle^2 + yCircle^2 = 1
                    double xCircle = 0, yCircle = 0;

                    if (x == xOrigin)
                    {
                        xCircle = x;
                        yCircle = (y < yOrigin ? -1 : 1) * Math.Sqrt(1 - x * x);
                    }
                    else if (y == yOrigin)
                    {
                        xCircle = (x < xOrigin ? -1 : 1) * Math.Sqrt(1 - y * y);
                        yCircle = y;
                    }
                    else
                    {
                        // Express line from origin to point as y = mx + k
                        double m = (yOrigin - y) / (xOrigin - x);
                        double k = y - m * x;

                        // Now substitute (mx + k) for y into x^2 + y^2 = 1
                        // x^2 + (mx + k)^2 = 1
                        // x^2 + (mx)^2 + 2mxk + k^2 - 1 = 0

```

```

        // (1 + m^2)x^2 + (2mk)x + (k^2 - 1) = 0 is quadratic equation
        double a = 1 + m * m;
        double b = 2 * m * k;
        double c = k * k - 1;

        // Now solve for x
        double sqrtTerm = Math.Sqrt(b * b - 4 * a * c);
        double x1 = (-b + sqrtTerm) / (2 * a);
        double x2 = (-b - sqrtTerm) / (2 * a);

        if (x < xOrigin)
            xCircle = Math.Min(x1, x2);
        else
            xCircle = Math.Max(x1, x2);

        yCircle = m * xCircle + k;
    }

    // Length from origin to point
    length1 = Math.Sqrt(Math.Pow(x - xOrigin, 2) + Math.Pow(y - yOrigin, 2));

    // Length from point to circle
    length2 = Math.Sqrt(Math.Pow(x - xCircle, 2) + Math.Pow(y - yCircle, 2));

    // Normalize those lengths
    double total = length1 + length2;
    length1 /= total;
    length2 /= total;
}

// Interpolate color
double alpha = length2 * this.InnerColor.A + length1 * this.OuterColor.A;
double red= alpha * (length2 * this.InnerColor.R +
                    length1 * this.OuterColor.R) / 255;
double green = alpha * (length2 * this.InnerColor.G +
                       length1 * this.OuterColor.G) / 255;
double blue = alpha * (length2 * this.InnerColor.B +
                      length1 * this.OuterColor.B) / 255;

// Store in array
pixels[index++] = (byte)blue;
pixels[index++] = (byte)green;
pixels[index++] = (byte)red;
pixels[index++] = (byte)alpha;
}
else
{
    pixels[index++] = bOutsideCircle;
    pixels[index++] = gOutsideCircle;
    pixels[index++] = rOutsideCircle;
    pixels[index++] = aOutsideCircle;
}
}
}
}

```

```

        pixelStream.Seek(0, SeekOrigin.Begin);
        pixelStream.Write(pixels, 0, pixels.Length);
        bitmap.Invalidate();
    }
}

```

With an eye towards making this animatable, the array of pixels and the *Stream* object used to transfer the pixels into the bitmap are both saved as fields. No allocations from the heap are required in the *RefreshBitmap* method unless the *WriteableBitmap* needs to be recreated because the size of the element has changed.

As it turned out, however, animation performance was very poor, even with rather small dimensions. But if you avoid animating the gradient itself, you can surely animate an object colored with this bitmap. The *MainPage.xaml* file instantiates both a *RadialGradientBrushSimulator* and an *Ellipse* with a binding to the simulator, as well as a couple animations:

Project: RadialGradientBrushDemo | File: MainPage.xaml (excerpt)

```

<Page ... >
    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Canvas SizeChanged="OnCanvasSizeChanged"
            Margin="0 0 96 96">

            <Grid Name="ballContainer"
                Width="96"
                Height="96">

                <Ellipse Name="ellipse">
                    <Ellipse.Fill>
                        <ImageBrush ImageSource="{Binding ElementName=brushSimulator,
                            Path=ImageSource}" />
                    </Ellipse.Fill>
                </Ellipse>

                <local:RadialGradientBrushSimulator x:Name="brushSimulator"
                    InnerColor="White"
                    OuterColor="Red"
                    GradientOrigin="0.3 0.3" />

            </Grid>
        </Canvas>
    </Grid>

    <Page.Triggers>
        <EventTrigger>
            <BeginStoryboard>
                <Storyboard>
                    <DoubleAnimation x:Name="leftAnima"
                        Storyboard.TargetName="ballContainer"
                        Storyboard.TargetProperty="(Canvas.Left)"
                        From="0" Duration="0:0:2.51"
                        AutoReverse="True"
                        RepeatBehavior="Forever" />
                </Storyboard>
            </BeginStoryboard>
        </EventTrigger>
    </Page.Triggers>

```

```

        <DoubleAnimation x:Name="rightAnima"
                        Storyboard.TargetName="ballContainer"
                        Storyboard.TargetProperty="(Canvas.Top)"
                        From="0" Duration="0:0:1.01"
                        AutoReverse="True"
                        RepeatBehavior="Forever" />
    </Storyboard>
</BeginStoryboard>
</EventTrigger>
</Page.Triggers>
</Page>

```

Notice how I've put the *Ellipse* and the *RadialGradientBrushSimulator* in the same 96-pixel square *Grid* so that both elements have the same size and the simulator generates a bitmap of exactly the same size as the *Ellipse* it's used to color. The code-behind file simply adjusts the *To* values on the animations based on the size of the *Canvas*:

Project: RadialGradientBrushDemo | File: MainPage.xaml.cs (excerpt)

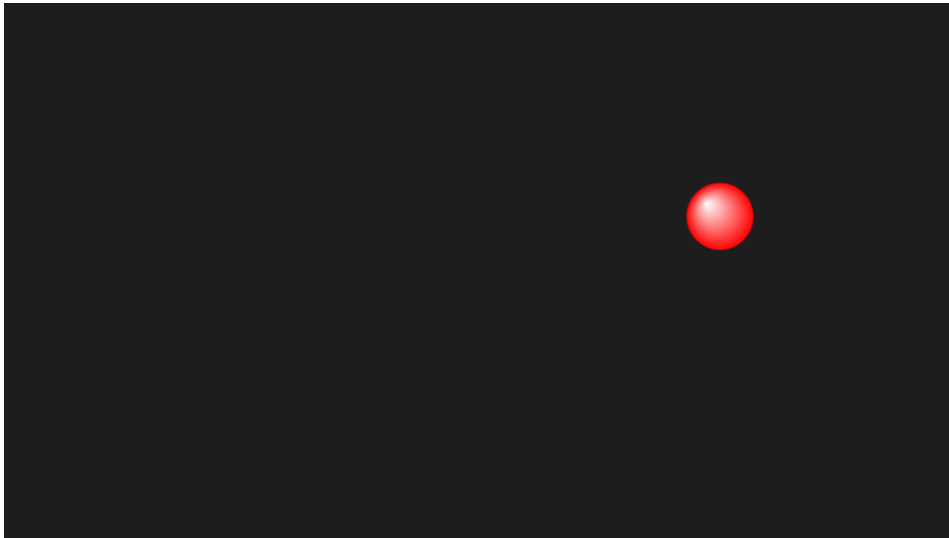
```

void OnCanvasSizeChanged(object sender, SizeChangedEventArgs args)
{
    // Canvas.Left animation
    leftAnima.To = args.NewSize.Width;

    // Canvas.Top animation
    rightAnima.To = args.NewSize.Height;
}

```

The simulated light reflection makes the *Ellipse* come one step closer to looking like something that might be found in the real world:



Loading and Saving Image Files

As you've seen, you can give the *SetSource* method of *WriteableBitmap* a stream referencing a PNG file, and it will graciously decode that compressed file and convert it into an array of rows and columns. You can get closer to this process with classes in the *Windows.Graphics.Imaging* namespace, and you can go the other way: You can save a *WriteableBitmap* that's been created in your program to a file in one of several popular image formats.

Bitmap file formats are generally differentiated by the type of compression they use (including none at all), and of course contain unique headers and techniques for storing the compressed data. Code that knows how to read a specific file format and convert it into an array of pixels is known as a *decoder*. Decoders allow you to load image files into an application. The *BitmapDecoder* class in the *Windows.Graphics.Imaging* namespace supports the following formats:

File Format	MIME Types	Filename Extensions
Windows Bitmap	image/bmp	.bmp .dib .rle
Windows Icon	image/ico image/x-icon	.ico .icon
GIF files	image/gif	.gif
JPEG	image/jpeg image/jpe image/jpg	.jpeg .jpe .jpg .jfif .exif
PNG	image/png	.png
TIFF	image/tiff image/tif	.tiff .tif
WMPhoto	image/vnd.ms-photo	.wdp .jxr

The *BitmapDecoder* class will determine what type of file it's been asked to load and raise an exception if it can't figure it out.

Code that creates a file of a particular format is called an *encoder*, and in the Windows Runtime it's the *BitmapEncoder* class. Using an encoder is a little different from using a decoder. A decoder might be able to determine what type of file you're trying to load, but an encoder can't read your mind and determine the file format for a save. It must be told. The *BitmapEncoder* class supports the same formats as the *BitmapDecoder* except for the Windows Icon file.

Sometimes encoders and decoders are referred to collectively as *codecs*, which could stand for "coder/decoder" or "compressor/decompressor."

The seven file formats shown in the above table are identified by global unique IDs (objects of type *Guid*), but you really don't have to hard-code these IDs in your program, or indeed, include much specific information at all.

The *ImageFileIO* program demonstrates how to use the *FileOpenPicker* and a *BitmapDecoder* to

load a bitmap file into an application and how to use the *FileSavePicker* and a *BitmapEncoder* to select a file format and save a bitmap file from an application. In between, it has a couple application bar buttons to rotate an image by 90 degrees.

The XAML file defines an *Image* element, a *TextBlock* for displaying some information about the loaded bitmap, and an *AppBar*:

Project: ImageFileIO | File: MainPage.xaml (excerpt)

```
<Page ... >
    <Grid Background="Gray">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="*" />
        </Grid.RowDefinitions>

        <TextBlock Name="txtblk"
            Grid.Row="0"
            HorizontalAlignment="Center"
            FontSize="18" />

        <Image Name="image"
            Grid.Row="1" />
    </Grid>

    <Page.BottomAppBar>
        <AppBar IsOpen="True"
            Padding="10 0">
            <Grid>
                <StackPanel Orientation="Horizontal"
                    HorizontalAlignment="Left">

                    <Button Name="rotateLeftButton"
                        IsEnabled="False"
                        Style="{StaticResource AppBarButtonStyle}"
                        Content="⌂"
                        AutomationProperties.Name="Rotate Left"
                        Click="OnRotateLeftAppBarButtonClick" />

                    <Button Name="rotateRightButton"
                        IsEnabled="False"
                        Style="{StaticResource AppBarButtonStyle}"
                        Content="⌂"
                        AutomationProperties.Name="Rotate Right"
                        Click="OnRotateRightAppBarButtonClick" />
                </StackPanel>

                <StackPanel Orientation="Horizontal"
                    HorizontalAlignment="Right">

                    <Button Style="{StaticResource AppBarButtonStyle}"
                        Content="⌂"
                        AutomationProperties.Name="Open"
                        Click="OnOpenAppBarButtonClick" />
                </StackPanel>
            </Grid>
        </AppBar>
    </Page.BottomAppBar>
</Page>
```

```

        <Button Name="saveAsButton"
            IsEnabled="False"
            Style="{StaticResource SaveAppBarButtonStyle}"
            AutomationProperties.Name="Save As"
            Click="OnSaveAsAppBarButtonClick" />
    </StackPanel>
</Grid>
</AppBar>
</Page.BottomAppBar>
</Page>

```

Notice that the *AppBar* has its *IsOpen* property set to *true*. The program can't do anything until a file is loaded and all the other buttons on the *AppBar* are disabled.

To keep the program relatively simple, it doesn't retain a lot of information. Any bitmap the program loads from the disk is saved only as the *Source* property of the *Image* element. The only fields defined in the code-behind file serve only to retain bitmap resolution information, and that's not crucial:

Project: ImageFileIO | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    double dpiX, dpiY;

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

```

When the user clicks the Open button on the *AppBar*, the program creates a *FileOpenPicker* initialized to display the files in the Pictures folder:

Project: ImageFileIO | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    ...
    async void OnOpenAppBarButtonClick(object sender, RoutedEventArgs args)
    {
        // Create FileOpenPicker
        FileOpenPicker picker = new FileOpenPicker();
        picker.SuggestedStartLocation = PickerLocationId.PicturesLibrary;

        // Initialize with filename extensions
        IReadOnlyList<BitmapCodecInformation> codecInfos =
            BitmapDecoder.GetDecoderInformationEnumerator();

        foreach (BitmapCodecInformation codecInfo in codecInfos)
            foreach (string extension in codecInfo.FileExtensions)
                picker.FileTypeFilter.Add(extension);
    }
}

```

```

        // Get the selected file
        StorageFile storageFile = await picker.PickSingleFileAsync();

        if (storageFile == null)
            return;
        ...
    }
    ...
}

```

The static *BitmapDecoder.GetDecoderInformationEnumerator* is of enormous assistance here. It returns a collection of seven *BitmapCodecInformation* objects corresponding to the seven file formats in the table shown a few pages ago. Each of these contains a collection of MIME types and a collection of filename extensions. (This is what I used to obtain information for that table.) Those filename extensions can go right into the *FileOpenPicker* object, and the *FileOpenPicker* then displays all the files with those extensions.

If the *PickSingleFileAsync* call returns a non-null *StorageFile* object, the next step is to create a *BitmapDecoder* from that file:

Project: ImageFileIO | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    ...
    async void OnOpenAppBarButtonClick(object sender, RoutedEventArgs args)
    {
        ...
        // Open the stream and create a decoder
        BitmapDecoder decoder = null;

        using (IRandomAccessStreamWithContentType stream = await storageFile.OpenReadAsync())
        {
            string exception = null;

            try
            {
                decoder = await BitmapDecoder.CreateAsync(stream);
            }
            catch (Exception exc)
            {
                exception = exc.Message;
            }

            if (exception != null)
            {
                MessageDialog msgdlg =
                    new MessageDialog("That particular image file could not be loaded. " +
                                     "The system reports an error of: " + exception);
                await msgdlg.ShowAsync();
                return;
            }
            ...
        }
    }
}

```

```

    }
    ...
}

```

BitmapDecoder.CreateAsync could raise an exception if it is given a nonimage file or something else it can't handle.

As you may know, a GIF file can contain multiple images that in sequence play a rudimentary animation. These individual images are known as *frames*, and they are supported by the Windows Runtime. After you create a *BitmapDecoder* object, the next step is generally to start extracting frames. However, if you don't want to bother with multiframe GIF files—and I don't blame you if you don't!—you can simply extract the first frame and call it a day. This is what I've done in the next section of the code:

Project: ImageFileIO | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    ...
    async void OnOpenAppBarButtonClick(object sender, RoutedEventArgs args)
    {
        ...
        // Get the first frame
        BitmapFrame bitmapFrame = await decoder.GetFrameAsync(0);

        // Set information title
        txtblk.Text = String.Format("{0}: {1} x {2} {3} {4} x {5} DPI",
                                    storageFile.Name,
                                    bitmapFrame.PixelWidth, bitmapFrame.PixelHeight,
                                    bitmapFrame.BitmapPixelFormat,
                                    bitmapFrame.DpiX, bitmapFrame.DpiY);

        // Save the resolution
        dpiX = bitmapFrame.DpiX;
        dpiY = bitmapFrame.DpiY;

        // Get the pixels
        PixelDataProvider dataProvider =
            await bitmapFrame.GetPixelDataAsync(BitmapPixelFormat.Bgra8,
                                                BitmapAlphaMode.Premultiplied,
                                                new BitmapTransform(),
                                                ExifOrientationMode.RespectExifOrientation,
                                                ColorManagementMode.ColorManageToSRgb);

        byte[] pixels = dataProvider.DetachPixelData();
        ...
    }
    ...
}

```

The method displays information about the first frame in the *TextBlock* at the top of the page and saves the resolution settings as fields.

The *BitmapPixelFormat* and *BitmapAlphaMode* properties of the *BitmapFrame* contain important information about the format of the pixels. *BitmapPixelFormat* is an enumeration with the members *Rgba16* (red, green, blue, and alpha 16-bit values), *Rgba8* (red, green, blue, and alpha 8-bit values), or *Bgra8* (blue, green, red, and alpha 8-bit values), the last of which is compatible with the format associated with *WriteableBitmap*. Pixel data from the file is always converted into one of these formats. The *BitmapAlphaMode* property can indicate *Ignore*, *Straight*, or *Premultiplied*.

You can obtain a byte array of pixels in the pixel format of the frame format simply by calling the *GetPixelDataAsync* method with no arguments. However, if you want to use the bitmap data to create a *WriteableBitmap*, you should use the longer version of *GetPixelDataAsync* as shown here to specify the format compatible with *WriteableBitmap*.

After *GetPixelDataAsync* obtains an array of bytes in the same format supported by *WriteableBitmap*, the code to create and display the bitmap is similar to what you've seen before:

Project: ImageFileIO | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    ...
    async void OnOpenAppBarButtonClick(object sender, RoutedEventArgs args)
    {
        ...
        // Create WriteableBitmap and set the pixels
        WriteableBitmap bitmap = new WriteableBitmap((int)bitmapFrame.PixelWidth,
                                                    (int)bitmapFrame.PixelHeight);

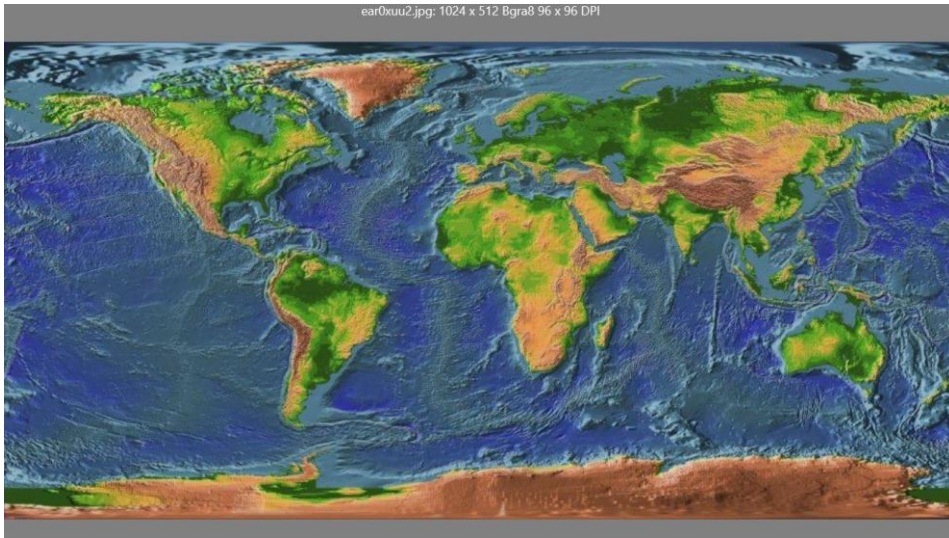
        using (Stream pixelStream = bitmap.PixelBuffer.AsStream())
        {
            pixelStream.Write(pixels, 0, pixels.Length);
        }

        // Invalidate the WriteableBitmap & set as Image source
        bitmap.Invalidate();
        image.Source = bitmap;
    }

    // Enable the other buttons
    saveAsButton.IsEnabled = true;
    rotateLeftButton.IsEnabled = true;
    rotateRightButton.IsEnabled = true;
}
...
}
```

That concludes the processing of the Open button. You've seen how the *FileOpenPicker* returns a *StorageFile* object, how this is opened and a stream passed to *BitmapDecoder.CreateAsync*, how the *BitmapDecoder* object exposes the images as *BitmapFrame* objects, and how to use the *GetPixelDataAsync* method to obtain an array of bytes that can be used to create a *WriteableBitmap*.

Here's the program displaying a bitmap I used in Chapter 13, "Touch, Etc.":



The Save As button on the application bar executes the *OnSaveAsAppBarButtonClick* method, which begins by creating a *FileSavePicker* object. The *BitmapEncoder.GetEncoderInformationEnumerator* provides information about the file formats supported by the *BitmapEncoder* class, but this information is used in a somewhat different way than with the *FileOpenPicker*.

The *FileSavePicker* wants a list of file types accompanied by one or more filename extensions for each type. The *FriendlyName* property of the *BitmapCodecInformation* object is unfortunately a string like "JPEG Encoder", so I use the *Split* method of *String* to extract just the first word (for example "JPEG"), and I combine that with the accumulated filename extensions. I also construct a dictionary of the supported MIME types and the *Guid* objects associated with those types:

Project: ImageFileIO | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    ...
    async void OnSaveAsAppBarButtonClick(object sender, RoutedEventArgs args)
    {
        FileSavePicker picker = new FileSavePicker();
        picker.SuggestedStartLocation = PickerLocationId.PicturesLibrary;

        // Get the encoder information
        Dictionary<string, Guid> imageTypes = new Dictionary<string, Guid>();
        IReadOnlyList<BitmapCodecInformation> codecInfos =
            BitmapEncoder.GetEncoderInformationEnumerator();

        foreach (BitmapCodecInformation codecInfo in codecInfos)
        {
            List<string> extensions = new List<string>();
```

```

        foreach (string extension in codecInfo.FileExtensions)
            extensions.Add(extension);

        string filetype = codecInfo.FriendlyName.Split(' ')[0];
        picker.FileTypeChoices.Add(filetype, extensions);

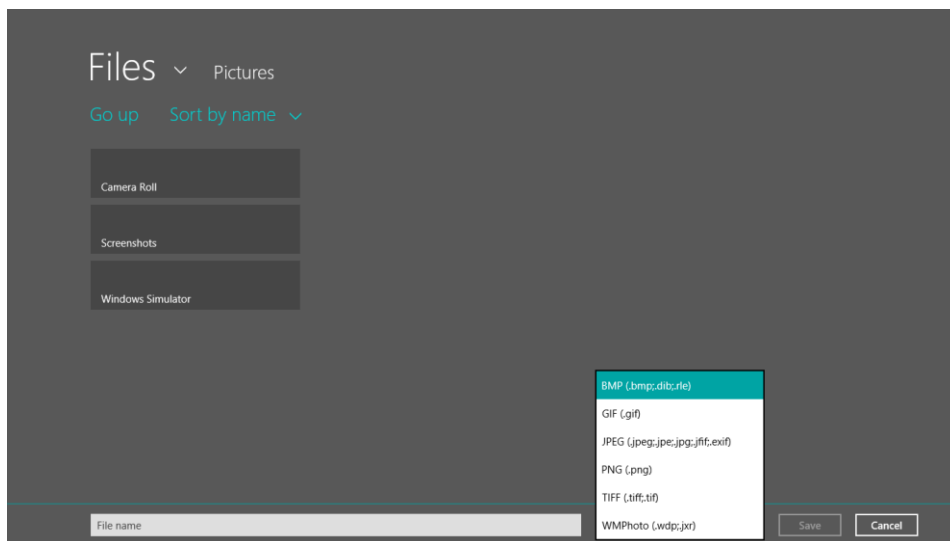
        foreach (string mimeType in codecInfo.MimeTypes)
            imageTypes.Add(mimeType, codecInfo.CodecId);
    }

    // Get a selected StorageFile
    StorageFile storageFile = await picker.PickSaveFileAsync();

    if (storageFile == null)
        return;
    ...
}
...
}

```

When the *FileSavePicker* displays itself, the user can select one of the file types from the popup box:



The *StorageFile* object returned from the *FileSavePicker* has a *ContentType* field, which is a MIME type string that identifies the file type that the user chose from that popup. The program can use this with its own dictionary to obtain the *Guid* object associated with that type:

Project: ImageFileIO | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    ...
    async void OnSaveAsAppBarButtonClick(object sender, RoutedEventArgs args)
    {
        ...
    }
}

```



```

// Open the StorageFile
using (IRandomAccessStream fileStream =
    await storageFile.OpenAsync(FileAccessMode.ReadWrite))
{
    // Create an encoder
    Guid codecId = imageTypes[storageFile.ContentType];
    BitmapEncoder encoder = await BitmapEncoder.CreateAsync(codecId, fileStream);

    // Get the pixels from the existing WriteableBitmap
    WriteableBitmap bitmap = image.Source as WriteableBitmap;
    byte[] pixels = new byte[4 * bitmap.PixelWidth * bitmap.PixelHeight];

    using (Stream pixelStream = bitmap.PixelBuffer.AsStream())
    {
        pixelStream.Read(pixels, 0, pixels.Length);
    }

    // Write those pixels to the first frame
    encoder.SetPixelData(BitmapPixelFormat.Bgra8, BitmapAlphaMode.Premultiplied,
        (uint)bitmap.PixelWidth, (uint)bitmap.PixelHeight,
        dpiX, dpiY, pixels);

    await encoder.FlushAsync();
}
}
...
}

```

With the help of that *Guid*, the static *BitmapEncoder.CreateAsync* method returns a *BitmapEncoder* object. That object has a *SetPixelData* method that can be used to transfer the byte array from the *WriteableBitmap* into the first frame of the new image file. The Save As operation is complete.

The remainder of the program supports rotating the images by 90 degrees. This feature is actually available in the *BitmapEncoder* class. The class defines a *Transform* property that you can use to scale, flip, crop, or rotate the image in 90-degree increments as it's being saved. However, if you want to see transformed image, you'll have to do that yourself.

Here are the three methods involved in rotating the image by 90 degrees:

Project: ImageFileIO | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    ...
    void OnRotateLeftAppBarButtonClick(object sender, RoutedEventArgs args)
    {
        Rotate((BitmapSource bitmap, int x, int y) =>
        {
            return 4 * (bitmap.PixelWidth * x + (bitmap.PixelWidth - y - 1));
        }));
    }

    void OnRotateRightAppBarButtonClick(object sender, RoutedEventArgs args)
    {

```

```

        Rotate((BitmapSource bitmap, int x, int y) =>
        {
            return 4 * (bitmap.PixelWidth * (bitmap.PixelHeight - x - 1) + y);
        });
    }

    async void Rotate(Func<BitmapSource, int, int, int> calculateSourceIndex)
    {
        // Get the source bitmap pixels
        WriteableBitmap srcBitmap = image.Source as WriteableBitmap;
        byte[] srcPixels = new byte[4 * srcBitmap.PixelWidth * srcBitmap.PixelHeight];

        using (Stream pixelStream = srcBitmap.PixelBuffer.AsStream())
        {
            await pixelStream.ReadAsync(srcPixels, 0, srcPixels.Length);
        }

        // Create a destination bitmap and pixels array
        WriteableBitmap dstBitmap =
            new WriteableBitmap(srcBitmap.PixelHeight, srcBitmap.PixelWidth);
        byte[] dstPixels = new byte[4 * dstBitmap.PixelWidth * dstBitmap.PixelHeight];

        // Transfer the pixels
        int dstIndex = 0;
        for (int y = 0; y < dstBitmap.PixelHeight; y++)
            for (int x = 0; x < dstBitmap.PixelWidth; x++)
            {
                int srcIndex = calculateSourceIndex(srcBitmap, x, y);

                for (int i = 0; i < 4; i++)
                    dstPixels[dstIndex++] = srcPixels[srcIndex++];
            }

        // Move the pixels into the destination bitmap
        using (Stream pixelStream = dstBitmap.PixelBuffer.AsStream())
        {
            await pixelStream.WriteAsync(dstPixels, 0, dstPixels.Length);
        }
        dstBitmap.Invalidate();

        // Swap the DPIs
        double dpi = dpiX;
        dpiX = dpiY;
        dpiY = dpi;

        // Display the new bitmap
        image.Source = dstBitmap;
    }
}

```

In Progress

In the final edition of this book, this chapter will continue with a Posterizer example that lets you load bitmaps, posterize them, and save the result. And it will include another FingerPaint program that draws lines on a *WriteableBitmap* by using line-drawing algorithms and that includes a color-selection dialog based on an HSL (hue-saturation-lightness) model similar to that used in Windows Paint.

Completed

The sample code includes PhotoScatter, which displays the contents of your Pictures folder and subfolders, and lets you move, scale, and rotate the images on the screen with your fingers.

Chapter 15

Printing

Sweep your finger into the right edge of the Windows 8 screen (or press Windows+C), and you'll see not only the current date and time pop up but also the column of five icons known as *charms*. The charm in the center navigates straight to the Start screen, but the others—labeled Search, Share, Devices, and Settings—are intended to provide services for your application.

This chapter focuses on the Devices charm, which is mostly to give your programs access to printers.

Basic Printing

Three namespaces play a role in printing:

- The *Windows.UI.Xaml.Printing* namespace has the *PrintDocument* class and support for its events. As the name suggests, a *PrintDocument* represents something that the user of your program wishes to print.
- The *Windows.Graphics.Printing* namespace has *PrintManager*, which is the interface to the panel that Windows 8 provides that lists printers and printer options; and the *PrintTask*, *PrintTaskRequest*, and *PrintTaskOptions* classes. A print "task" is the same thing as a print "job"—a particular use of the printer to print a particular document.
- The *Windows.Graphics.Printing.OptionDetails* namespace contains classes you'll use for customizing printing options.

Much of the printer API involves overhead rather than the process of actually defining text and graphics for the printer page. Indeed, a Windows 8 application prints on a printer page in the same way that it draws on the screen: with a visual tree of instances of classes that derive from *UIElement*. Generally the root element is a *Border* or a *Panel* of some sort with children. This visual tree can be defined in XAML but is probably more often constructed in code.

When defining elements to be displayed on the screen, one useful guideline is to treat the video display as if it has a resolution of 96 pixels per inch. For the printer, you do the same except that the equivalence is exact. Regardless of the actual resolution of the printer, you always treat it as a 96 DPI device.

For any program shown so far in this book, if you invoke the charms and press the Devices icon, you'll get a devices panel that won't mention anything about printers. Your application needs to register with Windows 8 that it is capable of printing. To persuade Windows 8 to list printers when the user taps the Devices charm, the first task is to set an event handler:

```
PrintManager printManager = PrintManager.GetForCurrentView();
printManager.PrintTaskRequested += OnPrintManagerPrintTaskRequested;
```

Those two lines can be combined into one:

```
PrintManager.GetForCurrentView().PrintTaskRequested += OnPrintManagerPrintTaskRequested;
```

The static *GetForCurrentView* method obtains a *PrintManager* instance that is associated with your program's window. By setting a handler for the *PrintTaskRequested* event, your program is announcing its availability for printing. The handler looks like this:

```
void OnPrintManagerPrintTaskRequested(PrintManager sender, PrintTaskRequestedEventArgs args)
{
    ...
}
```

That handler is called when the user clicks the Devices charm (or presses Windows+K), but (as you'll see shortly) it needs to call another method with a callback function in order for Windows to display the list of printers.

This *PrintTaskRequested* handler should be attached only when the application is in a position to actually print something. If the application requires some preliminary information from the user or needs to load a document before it can legitimately print, the handler should not be attached to the *PrintTaskRequested* event. The handler should be detached when the program finds itself again in a position where it doesn't make any sense to print anything:

```
PrintManager.GetForCurrentView().PrintTaskRequested -= OnPrintManagerPrintTaskRequested;
```

In the sample programs in this chapter, I mostly attach and detach this event handler in the *OnNavigatedTo* and *OnNavigatedFrom* overrides.

The handler for the *PrintTaskRequested* event is one of five callback methods and event handlers required of a program that performs simple printing. All five are required. Moreover, before the *PrintTaskRequested* event is even fired, your program needs to have prepared for printing by creating a *PrintDocument* object and attaching three event handlers to it.

So let's look at a complete program that prints a one-page document consisting of a single *TextBlock* announcing "Hello, Printer!" The XAML file in the HelloPrinter project doesn't play a role in the program logic and simply informs a new user how to print something:

Project: HelloPrinter | File: MainPage.xaml (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <TextBlock FontSize="48"
               HorizontalAlignment="Center"
               VerticalAlignment="Center"
               TextAlignment="Center">
        Hello, Printer!
    </TextBlock />
    <Run FontSize="24">
        (Invoke charms, select Devices and a printer)
    </Run>
```

```

    </TextBlock>
</Grid>

```

The code-behind file defines three fields, one of which is the *TextBlock* that the program prints:

Project: HelloPrinter | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    PrintDocument printDocument;
    IPrintDocumentSource printDocumentSource;

    // UIElement to print
    TextBlock txtblk = new TextBlock
    {
        Text = "Hello, Printer!",
        FontFamily = new FontFamily("Times New Roman"),
        FontSize = 48,
        Foreground = new SolidColorBrush(Colors.Black)
    };
    ...
}

```

The *PrintDocument* object represents what your application prints. Generally a program will create just one *PrintDocument* object and use it for every print task. In some cases, it might make sense for a program to maintain multiple *PrintDocument* objects—perhaps one to print the whole document, another to print a document outline, and a third to print thumbnails—but you shouldn’t be creating new *PrintDocument* objects for every print task. (As you’ll see, by the time a print task is requested, it’s actually too late to create the *PrintDocument*!) If it’s convenient, you can derive a class from *PrintDocument* to encapsulate some printing logic, but there’s nothing in *PrintDocument* to override.

For a program that deals with a single type of document, you’ll probably define the *PrintDocument* and *IPrintDocumentSource* as fields as I’ve done and create the *PrintDocument* object during program initialization:

Project: HelloPrinter | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    ...
    public MainPage()
    {
        this.InitializeComponent();

        // Create PrintDocument and attach handlers
        printDocument = new PrintDocument();
        printDocumentSource = printDocument.DocumentSource;
        printDocument.Paginate += OnPrintDocumentPaginate;
        printDocument.GetPreviewPage += OnPrintDocumentGetPreviewPage;
        printDocument.AddPages += OnPrintDocumentAddPages;
    }
    ...
}

```

The second field—the object of type *IPrintDocumentSource*—is obtained from the *PrintDocument* object. In addition, three events defined by *PrintDocument* require handlers. These event handlers are responsible for supplying a page count as well as the actual pages for print preview and actual printing.

The HelloPrinter program attaches a handler for the *PrintTaskRequested* event of the *PrintManager* during *OnNavigatedTo* and detaches it during *OnNavigatedFrom*, using two statements in the first case and one statement in the second just for a little variety.

Project: HelloPrinter | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    ...
    protected override void OnNavigatedTo(NavigationEventArgs args)
    {
        // Attach PrintManager handler
        PrintManager printManager = PrintManager.GetForCurrentView();
        printManager.PrintTaskRequested += OnPrintManagerPrintTaskRequested;

        base.OnNavigatedTo(args);
    }

    protected override void OnNavigatedFrom(NavigationEventArgs e)
    {
        // Detach PrintManager handler
        PrintManager.GetForCurrentView().PrintTaskRequested -= OnPrintManagerPrintTaskRequested;

        base.OnNavigatedFrom(e);
    }
    ...
}
```

In a real-life program, you'll be attaching this handler when your program is capable of printing, and detaching it when it has nothing to print.

When your program has this handler attached and the user sweeps a finger on the right side of the screen and then selects Devices, the *PrintTaskRequested* event handler is called. Here's the standard way to respond to that event:

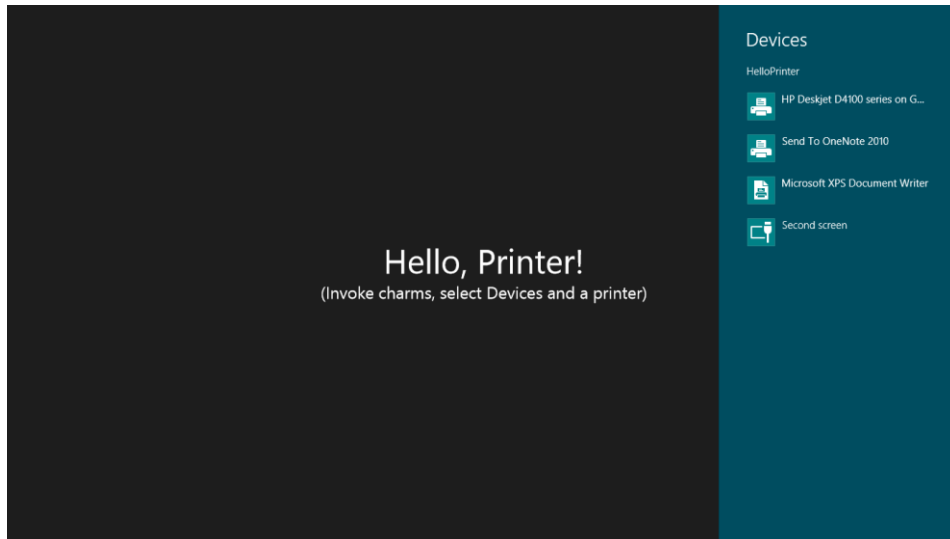
Project: HelloPrinter | File: MainPage.xaml.cs (excerpt)

```
void OnPrintManagerPrintTaskRequested(PrintManager sender, PrintTaskRequestedEventArgs args)
{
    args.Request.CreatePrintTask("Hello Printer", OnPrintTaskSourceRequested);
}
```

The event arguments to the *PrintTaskRequested* event include a property of type *Request*, and the program usually responds by calling the *CreatePrintTask* method of that *Request* object, passing to it the name of the printer task—this could be the name of the application or the name of a document being printed by the application—and a callback function. The *CreatePrintTask* method returns a *PrintTask* object, but it's not usually necessary to retain that object here.

Windows 8 then displays a list of printers. Here's what comes up on my screen (your mileage may

vary):



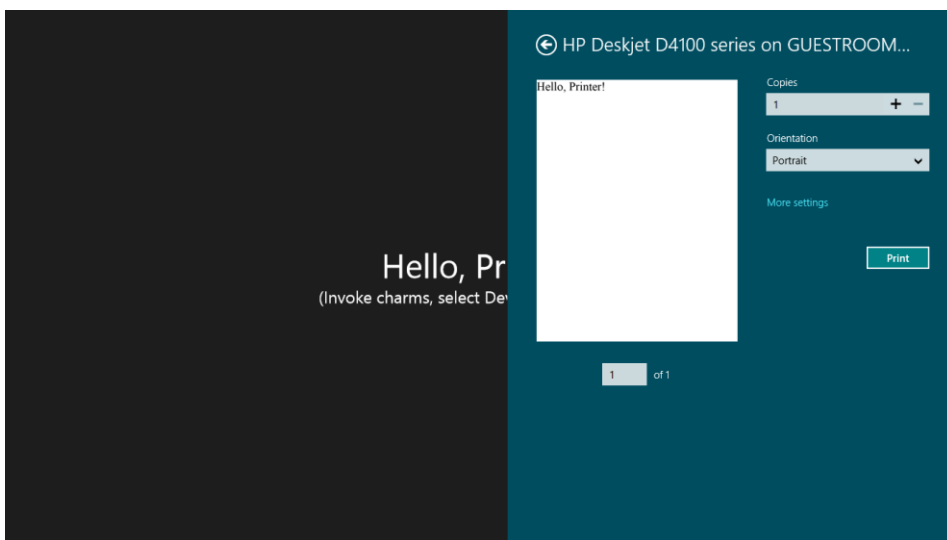
The only real printer here is the first one on the list. The second two items cause printing output to be saved in files. The last item isn't a printer at all and instead configures the second monitor attached to my tablet.

The callback I've named *OnPrintTaskSourceRequested* is called when the user selects one of the printers on the list. In the simplest case, the handler can respond by calling *SetSource* on the event arguments, passing to it the *IPrintDocumentSource* object obtained earlier from the *PrintDocument* object:

Project: HelloPrinter | File: MainPage.xaml.cs (excerpt)

```
void OnPrintTaskSourceRequested(PrintTaskSourceRequestedArgs args)
{
    args.SetSource(printDocumentSource);
}
```

When this method returns control back to Windows, a printer-specific panel is displayed:



The name of the printer at the top of the screen might look a little odd. This particular printer isn't actually attached to the tablet on which I'm working but is instead on a computer in the guest room of the house where I'm writing this chapter, and what you're seeing there is part of that computer's name.

Over at the right is a box to specify the number of copies and a dropdown to select Portrait or Landscape. These are standard settings for printers. (For the "Send To OneNote 2010" and "Microsoft XPS Document Writer" options, only the Orientation option is shown in this area.) Pressing "More settings" makes available an option to select page size as well as some printer-specific options.

At the left of this panel is a preview of the page to be printed. If the document has more than one page, you can select the page to view by using the little box below it. But the page preview is a *FlipView* control, and it's easiest just to sweep it from side to side.

The total number of pages comes from the handler for the *Paginate* event, which is one of the three events defined by *PrintDocument*. Handlers for all three of these events were attached in the *MainPage* constructor. In *HelloPrinter* the *Paginate* handler is implemented simply like this:

Project: *HelloPrinter* | File: *MainPage.xaml.cs* (excerpt)

```
void OnPrintDocumentPaginate(object sender, PaginateEventArgs args)
{
    printDocument.SetPreviewPageCount(1, PreviewPageCountType.Final);
}
```

This *Paginate* handler is the application's opportunity to prepare all the pages for printing and then call a *PrintDocument* method indicating the number of pages available and whether this is a preliminary or final count. (If it's not possible or convenient to do all that work in one shot, things get a little more complex.)

The preview of the print page is supplied by the handler for the *GetPreviewPage* event also defined by *PrintDocument* and also set earlier in the *MainPage* constructor:

Project: HelloPrinter | File: MainPage.xaml.cs (excerpt)

```
void OnPrintDocumentGetPreviewPage(object sender, GetPreviewPageEventArgs args)
{
    printDocument.SetPreviewPage(args.PageNumber, txtblk);
}
```

The *PageNumber* property of the event arguments is one-based and can range from 1 to the number specified in the *SetPreviewPageCount* call. For this particular program, it will always equal 1. The program responds to this event by calling the *SetPreviewPage* method of *PrintDocument*, passing to it the page number and the *TextBlock* that I defined as a field. That's what's displayed in the print preview.

When the Print button is pressed, the final event handler is called:

Project: HelloPrinter | File: MainPage.xaml.cs (excerpt)

```
void OnPrintDocumentAddPages(object sender, AddPagesEventArgs args)
{
    printDocument.AddPage(txtblk);
    printDocument.AddPagesComplete();
}
```

The handler for the *AddPages* event is responsible for calling *AddPage* for every page in the document. In the usual case, these are the same objects passed to the *SetPreviewPage* method, but you do have an opportunity to make them different if you wish. It concludes with a call to *AddPagesComplete*. The printing panel disappears, and (with any luck) you'll hear the familiar sound of a printer kicking into action.

Watch out! The *Paginate* handler can be called more than once, particularly if the user starts playing around with various printer options. If your program is actually performing a lot of pagination work, you'll probably want to avoid repeating it when the actual layout of the page doesn't change. In a real-life program generally you'll assemble all your pages in a *List* object during the *Paginate* handler and then deliver them up in the *GetPreviewPage* and *AddPages* handlers.

The *TextBlock* that HelloPrinter prints is given a *FontSize* of 48. That *TextBlock* might be sized somewhat differently on video displays of different resolutions, but when it's printed, that *FontSize* of 48 is an exact measurement and means 48/96 inch, which is half an inch or 36 points.

You'll notice I specified that the *Foreground* property of that printable *TextBlock* as black. Because this program uses a dark theme, the default *Foreground* property is white, and without an explicit *Foreground* setting, this *TextBlock* gets the default and would be invisible on white paper. This is the type of thing that can have you baffled for days! When experimenting with printer code, it might be helpful to get in the habit of using colors such as Red and Blue so that there's less of a chance of printing white text.

As you look over the code in HelloPrinter, you might think you see a couple ways to simplify it. For example, you might think you don't need to create a *PrintDocument* initially and save it as a field. You could simply create it in the *OnPrintTaskSourceRequested* method, set the three event handlers, and

extract the *IPrintDocumentSource* object. The various *PrintDocument* event handlers can get access to the *PrintDocument* from the *sender* argument.

But this will not work. The *PrintDocument* needs to be created and accessed in the user-interface thread, and the *PrintTaskRequested* handler and the callback I've named *OnPrintTaskSourceRequested* do *not* run in the user-interface thread. If you wait until the *PrintTaskRequested* event is fired to create the *PrintDocument*, it's too late.

Printable and Unprintable Margins

Even with the caution of printing the *TextBlock* in black, it's not printed correctly on my printer, and it's probably not printed correctly on your printer either. The *TextBlock* is aligned smack against the upper-left corner of the page, and most printing mechanisms simply can't reach to the edge of the paper, which means that part of the text is sheared off.

If you try to solve this problem by setting the *HorizontalAlignment* and *VerticalAlignment* properties of the *TextBlock* to *Center*, you'll discover these properties don't work in this case. The alignment values are relative to a parent element, and this *TextBlock* has no parent because it's the top-level element on the printer page. *Margin* won't work either for the same reason. Setting the *Padding* property on the *TextBlock* will work, however, because that's something the *TextBlock* handles itself.

A much better general-purpose solution is for every printer page to be a visual tree that begins with a top-level *Border* object. When printed, this *Border* will occupy the entire size of the paper, but the *Border* can include a nonzero *Padding* property that effectively provides a margin for the entire page.

Both *PaginateEventArgs* and *AddPagesEventArgs* include a property named *PrintTaskOptions* of type *PrintTaskOptions*. Most of the properties of this object correspond to the printer properties that the user can set manually. These properties have names like *Collation*, *NumberOfCopies*, *Orientation*, and *PrintQuality*. A program can access these properties to customize printing, but this is generally not necessary. I'll show you later in this chapter how a program can initialize these properties and add some custom options.

PrintTaskOptions also has a method named *GetPageDescription*. The argument is a zero-based page number under the assumption that each page can be a different size. The *PrintPageDescription* structure returned from this method has *DpiX* and *DpiY* properties, which report the actual resolution of the printer (very often values like 600 or 1200) and a *PageSize* of type *Size* in units of 1/96 inch. For an American standard letter size 8½ × 11" paper in portrait mode, the *PageSize* properties are 816 and 1056.

The *PrintPageDescription* structure also includes an *ImageableRect* property of type *Rect* that indicates the rectangular area of the page where the printer can actually print. For letter size paper on my printer, this rectangle has an upper-left corner at (12.48, 11.35748) and a size of (791.04, 988.1575), all in units of 1/96 inch. Compare this with the *PageSize* of 816 by 1056. Perform a few subtractions

and you'll conclude that the printer can't print on the 12.48 units on the left and right edges, 11.35748 units on the top, and 56.48502 units (over ½ inch) on the bottom. In landscape mode, both *PageSize* and *ImageableRect* reflect the particular orientation of the page

Let's examine how accurate these numbers are. The *PrintPrintableArea* program announces the name of itself in its XAML file:

Project: *PrintPrintableArea* | File: *MainPage.xaml* (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <TextBlock Text="Print Printable Area"
        FontSize="24"
        HorizontalAlignment="Center"
        VerticalAlignment="Center" />
</Grid>
```

The code-behind file is structured very much like *HelloPrinter* except that the element to be printed by the program is rather more extensive, consisting of a *Border* with a red background, a nested *Border* with a white background and a black border, and a centered *TextBlock*.

You'll notice also that instead of a separate callback method passed to the *CreatePrintTask* method, I've defined it as an anonymous lambda function. This is a common practice, but I've become less enamored of it for more complex printing logic:

Project: *PrintPrintableArea* | File: *MainPage.xaml.cs* (excerpt)

```
public sealed partial class MainPage : Page
{
    PrintDocument printDocument;
    IPrintDocumentSource printDocumentSource;

    // Element to print
    Border border = new Border
    {
        Background = new SolidColorBrush(Colors.Red),

        Child = new Border
        {
            Background = new SolidColorBrush(Colors.White),
            BorderBrush = new SolidColorBrush(Colors.Black),
            BorderThickness = new Thickness(1),
            Child = new TextBlock
            {
                Text = "Print Printable Area",
                FontFamily = new FontFamily("Times New Roman"),
                FontSize = 24,
                Foreground = new SolidColorBrush(Colors.Black),
                HorizontalAlignment = HorizontalAlignment.Center,
                VerticalAlignment = VerticalAlignment.Center
            }
        }
    };

    public MainPage()
```

```

{
    this.InitializeComponent();

    // Create PrintDocument and attach handlers
    printDocument = new PrintDocument();
    printDocumentSource = printDocument.DocumentSource;
    printDocument.Paginate += OnPrintDocumentPaginate;
    printDocument.GetPreviewPage += OnPrintDocumentGetPreviewPage;
    printDocument.AddPages += OnPrintDocumentAddPages;
}

protected override void OnNavigatedTo(NavigationEventArgs args)
{
    // Attach PrintManager handler
    PrintManager.GetForCurrentView().PrintTaskRequested += OnPrintManagerPrintTaskRequested;

    base.OnNavigatedTo(args);
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    // Detach PrintManager handler
    PrintManager.GetForCurrentView().PrintTaskRequested -= OnPrintManagerPrintTaskRequested;

    base.OnNavigatedFrom(e);
}

void OnPrintManagerPrintTaskRequested(PrintManager sender, PrintTaskRequestedEventArgs args)
{
    args.Request.CreatePrintTask("Print Printable Area", (requestArgs) =>
    {
        requestArgs.SetSource(printDocumentSource);
    });
}

void OnPrintDocumentPaginate(object sender, PaginateEventArgs args)
{
    PrintPageDescription printPageDescription = args.PrintTaskOptions.GetPageDescription(0);

    // Set Padding on outer Border
    double left = printPageDescription.ImageableRect.Left;
    double top = printPageDescription.ImageableRect.Top;
    double right = printPageDescription.PageSize.Width
        - left - printPageDescription.ImageableRect.Width;
    double bottom = printPageDescription.PageSize.Height
        - top - printPageDescription.ImageableRect.Height;
    border.Padding = new Thickness(left, top, right, bottom);

    printDocument.SetPreviewPageCount(1, PreviewPageCountType.Final);
}

void OnPrintDocumentGetPreviewPage(object sender, GetPreviewPageEventArgs args)
{
    printDocument.SetPreviewPage(args.PageNumber, border);
}

```

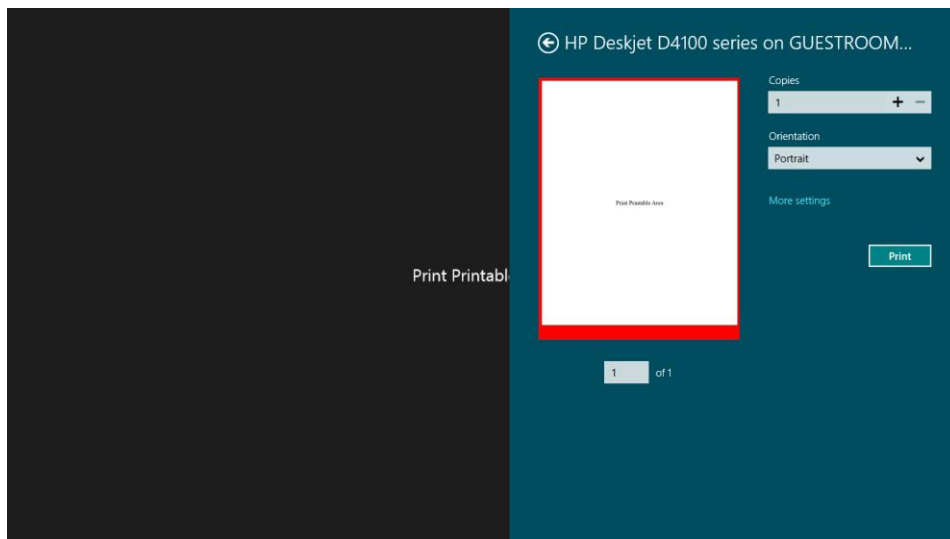
```

    }

    void OnPrintDocumentAddPages(object sender, AddPagesEventArgs args)
    {
        printDocument.AddPage(border);
        printDocument.AddPagesComplete();
    }
}

```

The other big difference is the handler for the *Paginate* event of *PrintDocument*. The handler obtains the *PrintPageDescription* structure and calculates a *Padding* value that it applies to the outer *Border* of the element to be printed. As you can see, the preview displays the red background of the outer border to the edge of the paper:



When the preview is displayed, try switching between Portrait and Landscape. Each change causes another call to the *Paginate* handler and a recalculation of the *Padding* value for the outer *Border*.

The preview obviously doesn't reflect the limitations of the printer in printing to the paper's edge. Otherwise that red area wouldn't be visible at all. I was pleased to discover that the page that actually came out of the printer displayed the black inner *Border* just fine with only a tiny trace of the red background of the outer *Border*, indicating that the *ImageableRect* values for this printer are accurate.

Although a program that prints pictures and other bitmaps might want to print as large as possible on the page, most printing applications prefer to set a larger margin—perhaps an inch, more or less—either of a fixed size or customizable by the user. In these cases it's usually not necessary for the *ImageableRect* property to be accessed at all.

I'll have an example of user-supplied margins coming up.

The Pagination Process

In the general case, a Windows application prints more than a single page, and the number of pages might depend on many factors, for example, the length of a document, font sizes, paper size, page margins, and whether the page is in portrait or landscape mode.

It is the purpose of the *Paginate* event handler not only to prepare these pages for preview and printing, but to report the number of pages in the document. In some cases pagination might require some time, and there are ways to avoid doing it all at once, but it's easiest if you can do it in one shot.

Let's examine a fairly short pagination job by resurrecting the *DependencyObjectClassHierarchy* program from Chapter 4, "Presentation with Panels," and adding a print option to it. As you might recall, *DependencyObjectClassHierarchy* created a *TextBlock* for every class that derived from *DependencyObject* and put them all in a *StackPanel* in a *ScrollViewer*. The XAML file for the *PrintableClassHierarchy* program is the same as in the previous version:

Project: *PrintableClassHierarchy* | File: *MainPage.xaml* (excerpt)

```
<Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
    <ScrollViewer>
        <StackPanel Name="stackPanel" />
    </ScrollViewer>
</Grid>
```

I also decided to use a *StackPanel* containing *TextBlock* children for printing as well, but with a profound difference: For the screen, there's only one *StackPanel* because it's in a *ScrollViewer*. For printing, there must be a *StackPanel* for each page containing only the *TextBlock* elements for that page.

It is tempting to use the same *TextBlock* elements for the screen and the printer. In theory you can print elements already displayed on the screen, but it is my experience that this technique never really works as well as might be hoped. One major restriction is that a particular element cannot have two parents. In this example, a printed *TextBlock* must be a child of a *StackPanel* for the printer page, so it can't also be a child of a *StackPanel* on the screen.

For that reason, the revised version of the class hierarchy program creates a whole separate collection of *TextBlock* elements that it stores in a field named *printerTextBlocks*. This portion of the *MainPage* class is very similar to the code-behind file in *DependencyObjectClassHierarchy* except that the *TextBlock* creation code has been split into a separate method for the convenience of creating two parallel sets of *TextBlock* elements. Notice that the printer *TextBlock* elements are given an explicit *Foreground* of black in the *DisplayAndPrinterPrep* method (renamed from the earlier *Display* method). Most of the printing support is not shown in this excerpt:

Project: *PrintableClassHierarchy* | File: *MainPage.xaml.cs* (excerpt)

```
public sealed partial class MainPage : Page
{
    Type rootType = typeof(DependencyObject);
```

```

TypeInfo rootTypeInfo = typeof(DependencyObject).GetTypeInfo();
List<Type> classes = new List<Type>();
Brush highlightBrush;

// Printing support
List<TextBlock> printerTextBlocks = new List<TextBlock>();
Brush blackBrush = new SolidColorBrush(Colors.Black);
...
public MainPage()
{
    this.InitializeComponent();
    highlightBrush = this.Resources["ApplicationPressedForegroundThemeBrush"] as Brush;

    // Accumulate all the classes that derive from DependencyObject
    AddToClassList(typeof(Windows.UI.Xaml.DependencyObject));

    // 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 TextBlocks added to StackPanel
    DisplayAndPrinterPrep(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 DisplayAndPrinterPrep(ClassAndSubclasses parentClass, int indent)
{
    TypeInfo typeInfo = parentClass.Type.GetTypeInfo();

    // Create TextBlock & add to StackPanel
    TextBlock txtblk = CreateTextBlock(typeInfo, indent);
    stackPanel.Children.Add(txtblk);

    // Create TextBlock & add to printer list
    txtblk = CreateTextBlock(typeInfo, indent);
    txtblk.Foreground = blackBrush;
    printerTextBlocks.Add(txtblk);

    // Call this method recursively for all subclasses
    foreach (ClassAndSubclasses subclass in parentClass.Subclasses)
        DisplayAndPrinterPrep(subclass, indent + 1);
}

TextBlock CreateTextBlock(TypeInfo typeInfo, int indent)
{
    // 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
        });

    return txtblk;
}

```

```

    }
    ...
}

```

The remainder of the printing support is very similar to what you've seen before, except that there's more than one page to print. The *Paginate* method takes on the brunt of the work and stores the formatted pages in the *printerPages* field. Each of these objects is a *Border* with *Padding* set to 96 (one inch) and a child *StackPanel* with a page worth of the *TextBlock* elements created earlier.

Keep in mind that the *Paginate* handler might be called multiple times as the user waffles between Portrait or Landscape mode, and letter or legal page sizes. Because the program is working with a fixed collection of *TextBlock* elements, and because elements are prohibited from having multiple parents, it's essential for the *Paginate* method to begin by ensuring that none of these *TextBlock* elements is still a child of a previously created *StackPanel*.

Project: PrintableClassHierarchy | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    ...
    PrintDocument printDocument;
    IPrintDocumentSource printDocumentSource;
    List<UIElement> printerPages = new List<UIElement>();

    public MainPage()
    {
        ...
        // Create PrintDocument and attach handlers
        printDocument = new PrintDocument();
        printDocumentSource = printDocument.DocumentSource;
        printDocument.Paginate += OnPrintDocumentPaginate;
        printDocument.GetPreviewPage += OnPrintDocumentGetPreviewPage;
        printDocument.AddPages += OnPrintDocumentAddPages;
    }

    protected override void OnNavigatedTo(NavigationEventArgs args)
    {
        // Attach PrintManager handler
        PrintManager.GetForCurrentView().PrintTaskRequested += OnPrintManagerPrintTaskRequested;

        base.OnNavigatedTo(args);
    }

    protected override void OnNavigatedFrom(NavigationEventArgs e)
    {
        // Detach PrintManager handler
        PrintManager.GetForCurrentView().PrintTaskRequested -= OnPrintManagerPrintTaskRequested;

        base.OnNavigatedFrom(e);
    }
    ...
    void OnPrintManagerPrintTaskRequested(PrintManager sender, PrintTaskRequestedEventArgs args)

```

```

{
    args.Request.CreatePrintTask("Dependency Property Class Hierarchy", (requestArgs) =>
    {
        requestArgs.SetSource(printDocumentSource);
    });
}

void OnPrintDocumentPaginate(object sender, PaginateEventArgs args)
{
    // Verbosely set some variables for the page margin
    double leftMargin = 96;
    double topMargin = 96;
    double rightMargin = 96;
    double bottomMargin = 96;

    // Clear out previous printerPage collection
    foreach (UIElement printerPage in printerPages)
        ((printerPage as Border).Child as Panel).Children.Clear();

    printerPages.Clear();

    // Initialize page construction
    Border border = null;
    StackPanel stackPanel = null;
    double maxPageHeight = 0;
    double pageHeight = 0;

    // Look through the list of TextBlocks
    for (int index = 0; index < printerTextBlocks.Count; index++)
    {
        // A null Border object signals a new page
        if (border == null)
        {
            // Calculate the height available for text
            uint pageNumber = (uint)printerPages.Count;
            maxPageHeight =
                args.PrintTaskOptions.GetPageDescription(pageNumber).PageSize.Height;
            maxPageHeight -= topMargin + bottomMargin;
            pageHeight = 0;

            // Create StackPanel and Border
            stackPanel = new StackPanel();
            border = new Border
            {
                Padding = new Thickness(leftMargin, topMargin, rightMargin, bottomMargin),
                Child = stackPanel
            };

            // Add to the list of pages
            printerPages.Add(border);
        }

        // Get the TextBlock and measure it
        TextBlock txtblk = printerTextBlocks[index];
    }
}

```

```

        txtblk.Measure(Size.Empty);

        // Check if OK to add TextBlock to this page
        if (stackPanel.Children.Count == 0 ||
            pageHeight + txtblk.ActualHeight < maxPageHeight)
        {
            stackPanel.Children.Add(txtblk);
            pageHeight += Math.Ceiling(txtblk.ActualHeight);
        }
        // Otherwise, it's the end of the page
        else
        {
            // No longer working with this Border object
            border = null;

            // Reprocess this TextBlock
            index--;
        }
    }

    // Notify about the final page count
    printDocument.SetPreviewPageCount(printerPages.Count, PreviewPageCountType.Final);
}

void OnPrintDocumentGetPreviewPage(object sender, GetPreviewPageEventArgs args)
{
    printDocument.SetPreviewPage(args.PageNumber, printerPages[args.PageNumber - 1]);
}

void OnPrintDocumentAddPages(object sender, AddPagesEventArgs args)
{
    foreach (UIElement printerPage in printerPages)
        printDocument.AddPage(printerPage);

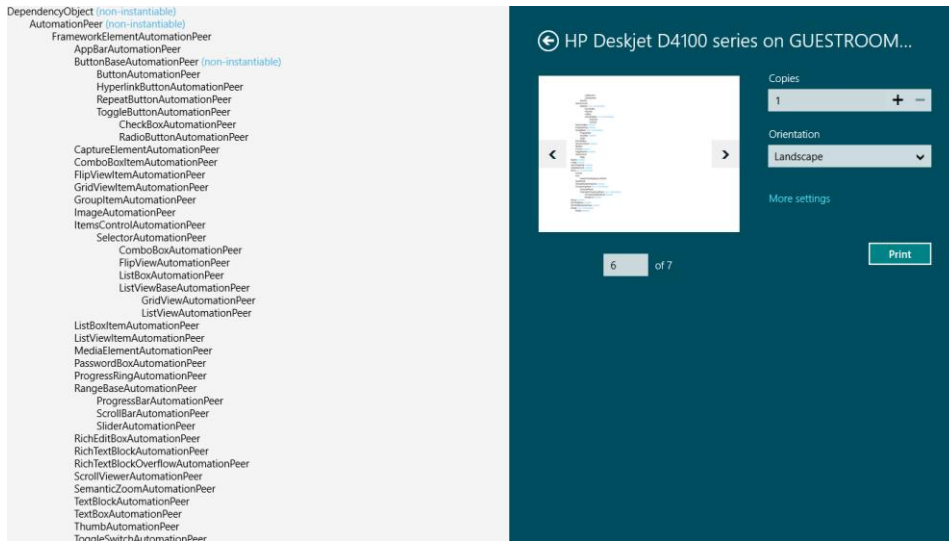
    printDocument.AddPagesComplete();
}
}

```

The strategy for pagination involves calculating a *maxPageHeight* number from the height of the paper page minus one-inch margins on the top and bottom. Another variable named *pageHeight* is increased for every *TextBlock* added to the *StackPanel* for that page. The method calls the *Measure* method on each *TextBlock* to calculate its size, and if the height of the *TextBlock* added to *pageHeight* exceeds *maxPageHeight*, a new page is required.

The *GetPreviewPage* handler uses the one-based *PageNumber* property in the event arguments to access the corresponding element in the *printerPages* list. The *AddPages* handler calls *AddPage* on all the pages.

In the preview, you can examine different pages before printing the whole list:



You may have noticed that the pagination logic increases the *pageHeight* based on each *TextBlock* height with the following code:

```
pageHeight += Math.Ceiling(txtblk.ActualHeight);
```

I originally wasn't using the *Math.Ceiling* call. The default *FontSize* is 11, and *ActualHeight* was reporting 13.2, and with that my program was giving each *StackPanel* 65 lines of text to display in the 9 inches available in portrait mode. However, in the preview, and coming out of the printer, only 62 lines were visible. The line spacing used to stack the text in the *StackPanel* was obviously greater than 13.2, resulting in three *TextBlock* elements per page being clipped because the resultant *StackPanel* was larger than the space allocated for it.

Using *Math.Ceiling* in this case resulted in 61 lines of text per page, which is a little off in the other direction, but at least none of the text disappears.

Still, it's a little odd. On a video display, of course, it makes perfect sense to align text with pixel boundaries for purposes of readability and that's why sizes are rounded up to the nearest pixel. On a printer, however, there are 600 pixels (or so) to the inch, so the rounding doesn't need to be based on a 96 DPI device.

Pagination can be very complex, particularly when text is involved. If you encounter chronic problems with elements not appearing precisely where you want them on the printer page, you might want to switch to using a *Canvas*. The use of *Glyphs* rather than *TextBlock* is popular for sophisticated text layout needs, and if you run into a wall using *Glyphs*, you'll probably want to explore *DirectWrite* to render to a bitmap and then to the printer page.

If you prefer going in the other direction—having the Windows Runtime do more of the work in determining how text is displayed—then exploring *RichTextBlock* can be profitable. This element has an overflow facility that I'll discuss in a later chapter.

Custom Printing Properties

The one-inch margins in `PrintableClassHierarchy` are hard coded. Suppose you want to allow the user to select the margins. While we're at it, let's give the user the option of setting the font size used for printing.

It is possible without too much trouble to customize the printer setup panel, and to have the Windows Runtime do most of the work in creating the appropriate controls and managing input.

The place to perform this customization is in the handler for the *PrintTaskRequested* event of the *PrintManager*. So far, this handler has looked like this:

```
void OnPrintManagerPrintTaskRequested(PrintManager sender, PrintTaskRequestedEventArgs args)
{
    args.Request.CreatePrintTask("My Print Task Title", OnPrintTaskSourceRequested);
}
```

Or, it's used an anonymous lambda function for the callback:

```
void OnPrintManagerPrintTaskRequested(PrintManager sender, PrintTaskRequestedEventArgs args)
{
    args.Request.CreatePrintTask("My Print Task Title", (requestArgs) =>
    {
        requestArgs.SetSource(printDocumentSource);
    });
}
```

Whichever way you do it, the *CreatePrintTask* call actually returns an object of type *PrintTask*, so that can be saved in a local variable:

```
void OnPrintManagerPrintTaskRequested(PrintManager sender, PrintTaskRequestedEventArgs args)
{
    PrintTask printTask = args.Request.CreatePrintTask("My Print Task Title", ... );
}
```

From that *PrintTask* object you can get an object of type *PrintTaskOptionDetails* via a roundabout static call that will probably resist becoming habitual:

```
PrintTaskOptionDetails optionDetails =
    PrintTaskOptionDetails.GetFromPrintTaskOptions(printTask.Options);
```

PrintTaskOptionDetails and related classes are defined in the *Windows.Graphics.Printing.OptionDetails* namespace.

If you want, you can then remove all the options from the first page of the printer setup pane:

```
optionDetails.DisplayedOptions.Clear();
```

Now you won't see the option to change the number of copies or the orientation. You could optionally put these back in, perhaps in reverse order:

```
optionDetails.DisplayedOptions.Add(StandardPrintTaskOptions.Orientation);
optionDetails.DisplayedOptions.Add(StandardPrintTaskOptions.Copies);
```

StandardPrintTaskOptions is a static class, and the properties represent standard printer options identified with string IDs. *StandardPrintTaskOptions.Orientation* is actually the string "PageOrientation" and *StandardPrintTaskOptions.Copies* is the string "JobCopiesAllDocuments". You can initialize these options if that is appropriate for your program:

```
optionDetails.Options[StandardPrintTaskOptions.Orientation].TrySetValue(
    PrintOrientation.Landscape);
```

PrintOrientation is one of eleven similar enumerations in *Windows.Graphics.Printing*.

You can add a less common option if you think it might be appropriate for your application:

```
optionDetails.DisplayedOptions.Add(StandardPrintTaskOptions.Collation);
```

You can also add your own items. You're limited to two types of custom options: a text field, or an expanding list of items like the Orientation option. Let's create a new project named CustomizableClassHierarchy. This program is mostly the same as PrintableClassHierarchy but defines some customizable values as fields initialized with values that the program considers appropriate:

Project: CustomizableClassHierarchy | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    ...
    // Initial values of custom printer settings
    double fontSize = new TextBlock().FontSize;
    double leftMargin = 96;    // 1 inch
    double topMargin = 72;    // 3/4 inch
    double rightMargin = 96;
    double bottomMargin = 72;
    ...
}
```

These fields are accessed in the handler for the *PrintTaskRequested* event of the *PrintManager*. You'll recall that this event is fired when the user taps the Devices charm, probably in the process of selecting a printer:

Project: CustomizableClassHierarchy | File: MainPage.xaml.cs (excerpt)

```
void OnPrintManagerPrintTaskRequested(PrintManager sender, PrintTaskRequestedEventArgs args)
{
    PrintTask printTask = args.Request.CreatePrintTask("Dependency Property Class Hierarchy",
        (requestArgs) =>
    {
        requestArgs.SetSource(printDocumentSource);
    });

    PrintTaskOptionDetails optionDetails =
        PrintTaskOptionDetails.GetFromPrintTaskOptions(printTask.Options);

    // Add item for font size
```

```

optionDetails.CreateTextOption("idFontSize", "Font size (in points)");
optionDetails.DisplayedOptions.Add("idFontSize");
optionDetails.Options["idFontSize"].TrySetValue((72 * fontSize / 96).ToString());

// Add items for page margins
optionDetails.CreateTextOption("idLeftMargin", "Left margin (in inches)");
optionDetails.DisplayedOptions.Add("idLeftMargin");
optionDetails.Options["idLeftMargin"].TrySetValue((leftMargin / 96).ToString());

optionDetails.CreateTextOption("idTopMargin", "Top margin (in inches)");
optionDetails.DisplayedOptions.Add("idTopMargin");
optionDetails.Options["idTopMargin"].TrySetValue((topMargin / 96).ToString());

optionDetails.CreateTextOption("idRightMargin", "Right margin (in inches)");
optionDetails.DisplayedOptions.Add("idRightMargin");
optionDetails.Options["idRightMargin"].TrySetValue((rightMargin / 96).ToString());

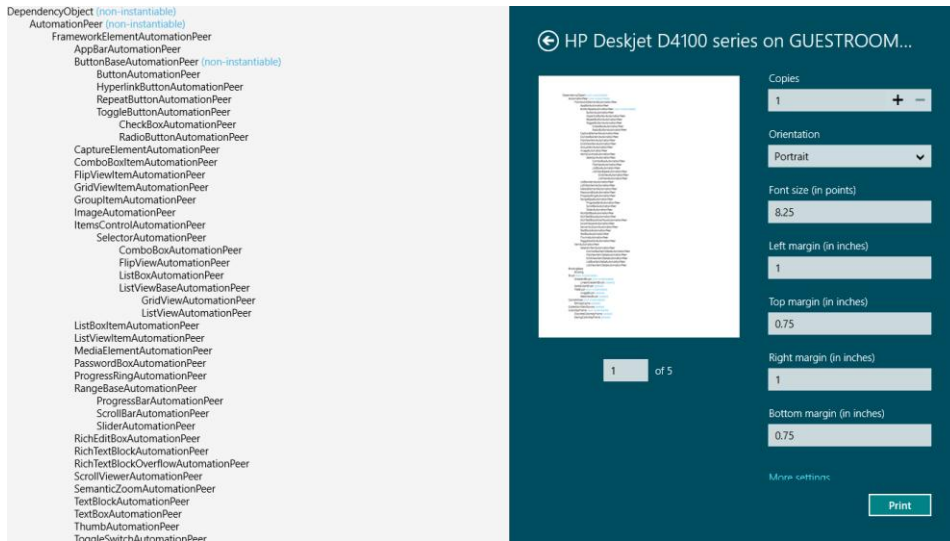
optionDetails.CreateTextOption("idBottomMargin", "Bottom margin (in inches)");
optionDetails.DisplayedOptions.Add("idBottomMargin");
optionDetails.Options["idBottomMargin"].TrySetValue((bottomMargin / 96).ToString());

// Set handler for the option changing
optionDetails.OptionChanged += OnOptionDetailsOptionChanged;
}

```

Each custom option requires at least two steps and possibly three. First the option must be created, in the process giving it an ID string and a label that appears on the printer settings panel. The custom option is then added to the *DisplayedOptions* collection. The third step is optional but sets an initial value. In my code, the fields storing these values are converted from pixels into points (for the font size) and inches (for the margin values).

The last step is setting an event handler for the *OptionChanged* event. This event will be fired for changes to all the printer options, not just the custom options. For text items like these, the event is not fired with every keystroke but only with a press of the Enter key, loss of input focus, or a press of the Print button. Here's what the customized settings panel looks like:



See the five new items? I know it looks like we've gone beyond the limit of the size available for custom options, but the list is scrollable.

Here's the implementation of the *OptionChanged* event handler. This is where validation occurs, and where you signal that the preview needs to be refreshed with new values, which means that your *Paginate* handler will be called again. The *PrintTaskOptionChangedEventArgs* class defines just one property—named *OptionId* of type *object* (but it's really a string) indicating the option that's changed—but *sender* is the important *PrintTaskOptionDetails* object that was the center of the customization process in the *PrintTaskRequested* handler:

Project: CustomizableClassHierarchy | File: MainPage.xaml.cs (excerpt)

```
async void OnOptionDetailsOptionChanged(PrintTaskOptionDetails sender,
                                        PrintTaskOptionChangedEventArgs args)
{
    if (args.OptionId == null)
        return;

    string optionId = args.OptionId.ToString();
    string strValue = sender.Options[optionId].Value.ToString();
    string errorText = String.Empty;
    double value = 0;

    switch (optionId)
    {
        case "idFontSize":
            if (!Double.TryParse(strValue, out value))
                errorText = "Value must be numeric";

            else if (value < 4 || value > 36)
                errorText = "Value must be between 4 and 36";
            break;
    }
}
```

```

        case "idLeftMargin":
        case "idTopMargin":
        case "idRightMargin":
        case "idBottomMargin":
            if (!Double.TryParse(strValue, out value))
                errorText = "Value must be numeric";

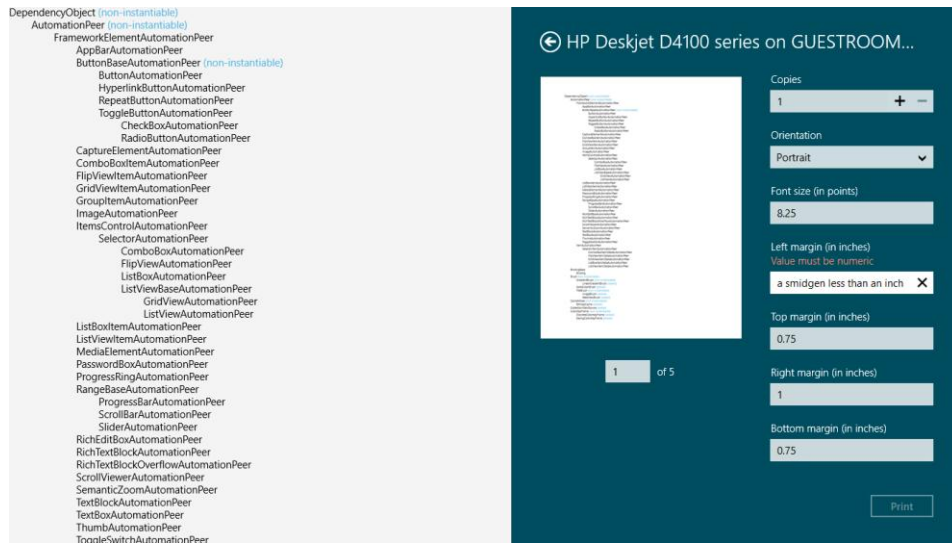
            else if (value < 0 || value > 2)
                errorText = "Value must be between 0 and 2";
            break;
    }

    sender.Options[optionId].ErrorText = errorText;

    // If there's no error, then invalidate the preview
    if (String.IsNullOrEmpty(errorText))
    {
        await this.Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
        {
            printDocument.InvalidatePreview();
        });
    }
}

```

If there's a problem with the input on one of the options, this method needs to set the *ErrorText* property for that option to a short but helpful text string. That string is displayed in red to the user. If the *ErrorText* of any option is set, the Print button also becomes disabled. Here's what it looks like:



Notice how everything below the error message has been shifted down. If the error message you supply is longer than a line, it will be wrapped.

If there's no error, the *InvalidatePreview* method of the *PrintDocument* object should be called. Notice that a *Dispatcher* is required to force that call to occur in the user interface thread. This *OptionChanged* handler is running in a secondary thread.

The *InvalidatePreview* call causes a new *Paginate* event to be fired on the *PrintDocument*. This new version of the *Paginate* handler begins by obtaining all the custom values and converting them into numbers that it can use. The font size is applied to all the *TextBlock* elements stored for printing, and the margin values are used as in the previous version of this method:

Project: CustomizableClassHierarchy | File: MainPage.xaml.cs (excerpt)

```
void OnPrintDocumentPaginate(object sender, PaginateEventArgs args)
{
    // Get values of custom settings
    PrintTaskOptionDetails optionDetails =
        PrintTaskOptionDetails.GetFromPrintTaskOptions(args.PrintTaskOptions);
    fontSize = 96 * Double.Parse(optionDetails.Options["idFontSize"].Value.ToString()) / 72;
    leftMargin = 96 * Double.Parse(optionDetails.Options["idLeftMargin"].Value.ToString());
    topMargin = 96 * Double.Parse(optionDetails.Options["idTopMargin"].Value.ToString());
    rightMargin = 96 * Double.Parse(optionDetails.Options["idRightMargin"].Value.ToString());
    bottomMargin = 96 * Double.Parse(optionDetails.Options["idBottomMargin"].Value.ToString());

    // Set FontSize of stored TextBlocks
    foreach (TextBlock txtblk in printerTextBlocks)
        txtblk.FontSize = fontSize;
    ...
}
```

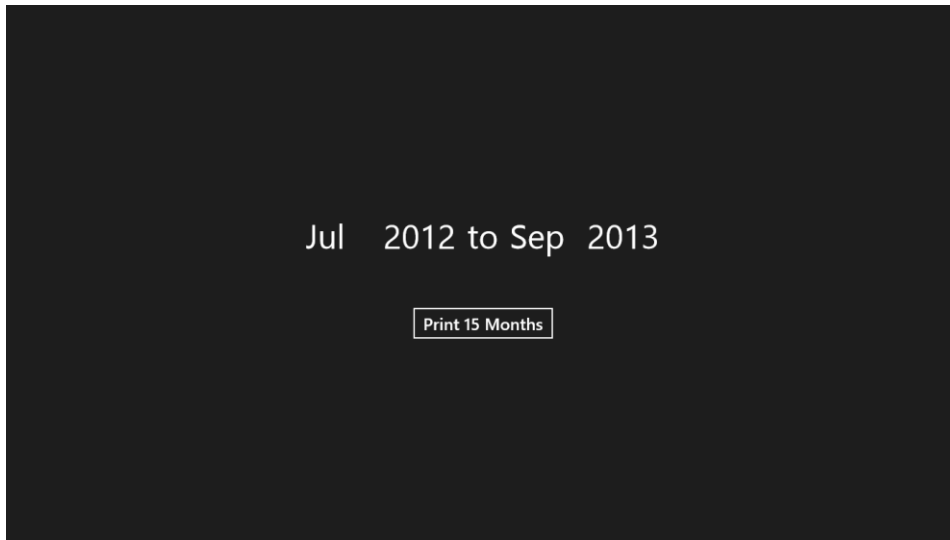
With a little more work, you can check that the margin values entered by the user are high enough to avoid text appearing in the unprintable area of the page. In the *OptionChanged* handler you can easily access the page description from the *PrintTaskOptionDetails* object:

```
Rect imageableRect = sender.GetPageDescription(0).ImageableRect;
```

Printing a Monthly Planner

Sometimes when I'm working on a long project I like to use printed monthly calendars taped up to the wall. These calendars don't need any fancy features—just a lot of white space to write stuff in for each day.

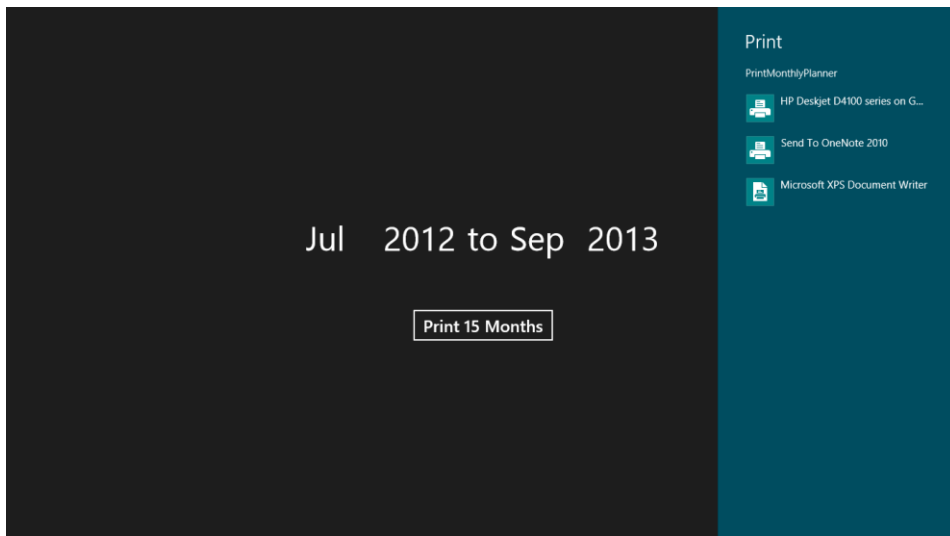
The sole purpose of the *PrintMonthlyPlanner* program is to print a bunch of monthly calendars in a range specified by the user. The main page looks like this:



Each month and year is selectable via a *FlipView* control. The button is enabled only if the start month is less than or equal to the end month. The *Click* handler for the button is implemented with just a single line of code:

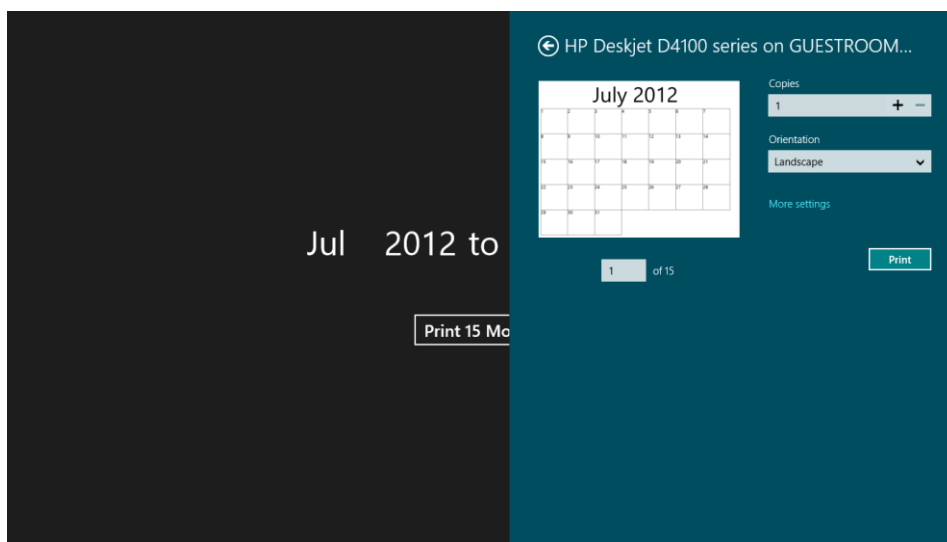
```
await PrintManager.ShowPrintUIAsync();
```

Although the user normally invokes the charms and Devices panel, a program can do so as well. Generally this option is reserved for programs that print only on special occasions, for example, "Print Ticket Confirmation." Interestingly, calling *ShowPrintUIAsync* brings up a slightly different panel than the Devices charm:



Because the *PrintMonthlyPlanner* program is dedicated to printing, the pages that it prints are not

otherwise displayed by the program and are only visible on screen in the printing panel:



Notice that the Orientation is set to Landscape. The program sets that initial value under the assumption that the calendar pages are better like that. Each page is printed to the very edge of the printable margins.

I created a custom control for the user to pick a month and year. This is called *MonthYearSelect*, and the XAML file reveals two templated *FlipView* controls, both with a horizontal *StackPanel* as the *ItemsPanel*:

Project: PrintMonthlyPlanner | File: MonthYearSelect.xaml (excerpt)

```
<UserControl ... >
    <UserControl.Resources>
        <Style TargetType="FlipView">
            <Setter Property="ItemsPanel">
                <Setter.Value>
                    <ItemsPanelTemplate>
                        <StackPanel Orientation="Vertical" />
                    </ItemsPanelTemplate>
                </Setter.Value>
            </Setter>

            <Setter Property="ItemTemplate">
                <Setter.Value>
                    <DataTemplate>
                        <TextBlock Text="{Binding}" VerticalAlignment="Center" />
                    </DataTemplate>
                </Setter.Value>
            </Setter>
        </Style>
    </UserControl.Resources>
```

```

<Grid>
    <StackPanel Orientation="Horizontal">
        <FlipView x:Name="monthFlipView"
            SelectionChanged="OnMonthYearSelectionChanged" />

        <TextBlock Text="&#x00A0;" />

        <FlipView x:Name="yearFlipView"
            SelectionChanged="OnMonthYearSelectionChanged" />
    </StackPanel>
</Grid>
</UserControl>

```

Partially to make use of new features in the Windows Runtime, I decided to make the public interface to this class a *Calendar* object rather than a traditional .NET *DateTime*. I was hoping to make the program generalized for any type of calendar, but the *Calendar* class doesn't seem to be documented sufficiently to go beyond the standard Gregorian. I couldn't even discover a way to determine if the day of the week should begin on a Sunday (the standard in most places) or Monday (used in France, for example).

I also discovered that *Calendar* is a class rather than a structure, and it bothered me to be generating new *Calendar* objects with every spin of the *FlipView*. I decided that the control would create just one *Calendar* object and change the *Month* and *Year* properties of that single object. But in that case it made no sense for *Calendar* to be exposed as a dependency property, which is customary with controls, so that's why it's a plain old property named *MonthYear* with a *MonthYearChanged* event to indicate new values of *Month* or *Year*:

Project: PrintMonthlyPlanner | File: MonthYearSelect.xaml.cs (excerpt)

```

public sealed partial class MonthYearSelect : UserControl
{
    public event EventHandler MonthYearChanged;

    public MonthYearSelect()
    {
        this.InitializeComponent();

        // Create Calendar with current date
        Calendar calendar = new Calendar();
        calendar.SetToNow();

        // Fill the first FlipView with the abbreviated month names
        DateTimeFormatter monthFormatter =
            new DateTimeFormatter(YearFormat.None, MonthFormat.Abbreviated,
                DayFormat.None, DayOfWeekFormat.None);

        for (int month = 1; month <= 12; month++)
        {
            string strMonth = monthFormatter.Format(
                new DateTimeOffset(2000, month, 15, 0, 0, 0, TimeSpan.Zero));
            monthFlipView.Items.Add(strMonth);
        }
    }
}

```

```

    // Fill the second FlipView with years (5 years before current, 25 after)
    for (int year = calendar.Year - 5; year <= calendar.Year + 25; year++)
    {
        yearFlipView.Items.Add(year);
    }

    // Set the FlipViews to the current month and year
    monthFlipView.SelectedIndex = calendar.Month - 1;
    yearFlipView.SelectedItem = calendar.Year;
    this.MonthYear = calendar;
}

public Calendar MonthYear { private set; get; }

void OnMonthYearSelectionChanged(object sender, SelectionChangedEventArgs args)
{
    if (this.MonthYear == null)
        return;

    if (monthFlipView.SelectedIndex != -1)
        this.MonthYear.Month = (int)monthFlipView.SelectedIndex + 1;

    if (yearFlipView.SelectedIndex != -1)
        this.MonthYear.Year = (int)yearFlipView.SelectedItem;

    // Fire the event
    if (MonthYearChanged != null)
        MonthYearChanged(this, EventArgs.Empty);
}
}

```

The MainPage.xaml file instantiates two of these *MonthYearSelect* controls:

Project: PrintMonthlyPlanner | File: MainPage.xaml (excerpt)

```

<Page ...
    FontSize="48">

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Grid HorizontalAlignment="Center"
            VerticalAlignment="Center">
            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="*" />
                <ColumnDefinition Width="Auto" />
            </Grid.ColumnDefinitions>

            <Grid.RowDefinitions>
                <RowDefinition Height="Auto" />
                <RowDefinition Height="Auto" />
            </Grid.RowDefinitions>

            <local:MonthYearSelect x:Name="monthYearSelect1"
                                Grid.Row="0" Grid.Column="0"

```

```

        Height="144"
        VerticalAlignment="Center"
        MonthYearChanged="OnMonthYearChanged" />

<TextBlock Text=" to&#x00A0;"
        Grid.Row="0" Grid.Column="1"
        VerticalAlignment="Center" />

<local:MonthYearSelect x:Name="monthYearSelect2"
        Grid.Row="0" Grid.Column="2"
        Height="144"
        VerticalAlignment="Center"
        MonthYearChanged="OnMonthYearChanged" />

<Button Name="printButton"
        Content="Print 1 Month"
        Grid.Row="1" Grid.Column="0" Grid.ColumnSpan="3"
        FontSize="24"
        HorizontalAlignment="Center"
        Margin="0 24"
        Click="OnPrintButtonClick" />

</Grid>
</Grid>
</Page>

```

This program is a little different from the others in this chapter in that printing is not enabled for the duration of the application. Printing is enabled only if the two *MonthYearSelect* controls are dialed in to an actual range of months. With each change of these two controls, the program needs to generate a new label for the *Button*, determine whether the button should be enabled or disabled, and determine whether to attach or detach the *PrintTaskRequested* event. That logic is much of what's going on in this initial section of the *MainPage* class:

Project: PrintMonthlyPlanner | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    PrintDocument printDocument;
    IPrintDocumentSource printDocumentSource;
    List<UIElement> calendarPages = new List<UIElement>();
    bool printingEnabled;

    public MainPage()
    {
        this.InitializeComponent();

        // Create PrintDocument and attach handlers
        printDocument = new PrintDocument();
        printDocumentSource = printDocument.DocumentSource;
        printDocument.Paginate += OnPrintDocumentPaginate;
        printDocument.GetPreviewPage += OnPrintDocumentGetPreviewPage;
        printDocument.AddPages += OnPrintDocumentAddPages;
    }

    protected override void OnNavigatedTo(NavigationEventArgs args)

```



```

{
    // Attach PrintManager handler
    if (GetPrintableMonthCount() > 0)
    {
        printingEnabled = true;
        PrintManager.GetForCurrentView().PrintTaskRequested +=
            OnPrintManagerPrintTaskRequested;
    }
    base.OnNavigatedTo(args);
}

protected override void OnNavigatedFrom(NavigationEventArgs e)
{
    // Detach PrintManager handler
    if (printingEnabled)
    {
        PrintManager.GetForCurrentView().PrintTaskRequested -=
            OnPrintManagerPrintTaskRequested;
        printingEnabled = false;
    }
    base.OnNavigatedFrom(e);
}

void OnMonthYearChanged(object sender, EventArgs args)
{
    // Calculate number of months and check if it's non-negative
    int printableMonths = GetPrintableMonthCount();
    printButton.Content = String.Format("Print {0} Month{1}", printableMonths,
        printableMonths > 1 ? "s" : "");
    printButton.IsEnabled = printableMonths > 0;

    // Attach or detach PrintManager handler
    if (printingEnabled != printableMonths > 0)
    {
        PrintManager printManager = PrintManager.GetForCurrentView();

        if (printableMonths > 0)
            printManager.PrintTaskRequested += OnPrintManagerPrintTaskRequested;
        else
            printManager.PrintTaskRequested -= OnPrintManagerPrintTaskRequested;

        printingEnabled = printableMonths > 0;
    }
}

int GetPrintableMonthCount()
{
    Calendar cal1 = monthYearSelect1.MonthYear;
    Calendar cal2 = monthYearSelect2.MonthYear;
    return cal2.Month - cal1.Month + 1 + 12 * (cal2.Year - cal1.Year);
}

async void OnPrintButtonClick(object sender, RoutedEventArgs args)
{

```

```

        await PrintManager.ShowPrintUIAsync();
    }

    void OnPrintManagerPrintTaskRequested(PrintManager sender, PrintTaskRequestedEventArgs args)
    {
        // Create PrintTask
        PrintTask printTask = args.Request.CreatePrintTask("Monthly Planner",
                                                            OnPrintTaskSourceRequested);

        // Set orientation to landscape
        PrintTaskOptionDetails optionDetails =
            PrintTaskOptionDetails.GetFromPrintTaskOptions(printTask.Options);

        PrintOrientationOptionDetails orientation =
            optionDetails.Options[StandardPrintTaskOptions.Orientation] as
                PrintOrientationOptionDetails;

        orientation.TrySetValue(PrintOrientation.Landscape);
    }

    void OnPrintTaskSourceRequested(PrintTaskSourceRequestedArgs args)
    {
        args.SetSource(printDocumentSource);
    }
    ...
}

```

Notice also that the *PrintTaskRequested* handler accesses the Orientation option and initializes it to Landscape. This will happen every time the user opens the print panel. It could be that the user really doesn't want to print these calendar months in landscape mode. You might want to keep track of what setting the user ultimately uses by obtaining it, saving it in a field during the *Paginate* handler, and then using that the next time the printer panel comes up. The user's preference could even be saved in user settings for the next time the program is run.

Creating the pages is the responsibility of the *Paginate* handler, which saves them in a field for the *GetPreviewPage* and *AddPages* handlers. These pages are built around a *Grid* with seven columns for the seven days of the week, a number of rows based on the number of weeks in the particular month (which could range from four in February to six in other months), and one more row for the month and year title at the top:

Project: PrintMonthlyPlanner | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    ...
    void OnPrintDocumentPaginate(object sender, PaginateEventArgs args)
    {
        // Prepare to generate pages
        uint pageNumbers = 0;
        calendarPages.Clear();
        Calendar calendar = monthYearSelect1.MonthYear.Clone();
        calendar.Day = 1;
        Brush black = new SolidColorBrush(Colors.Black);
    }
}

```

```

// For each month
do
{
    PrintPageDescription printPageDescription =
        args.PrintTaskOptions.GetPageDescription(pageNumber);

    // Set Padding on outer Border
    double left = printPageDescription.ImageableRect.Left;
    double top = printPageDescription.ImageableRect.Top;
    double right = printPageDescription.PageSize.Width
        - left - printPageDescription.ImageableRect.Width;
    double bottom = printPageDescription.PageSize.Height
        - top - printPageDescription.ImageableRect.Height;
    Border border = new Border { Padding = new Thickness(left, top, right, bottom) };

    // Use Grid for calendar cells
    Grid grid = new Grid();
    border.Child = grid;
    int numberOfWeeks = (6 + (int)calendar.DayOfWeek + calendar.LastDayInThisMonth) / 7;

    for (int row = 0; row < numberOfWeeks + 1; row++)
        grid.RowDefinitions.Add(new RowDefinition
        {
            Height = new GridLength(1, GridUnitType.Star)
        });

    for (int col = 0; col < 7; col++)
        grid.ColumnDefinitions.Add(new ColumnDefinition
        {
            Width = new GridLength(1, GridUnitType.Star)
        });

    // Month and year display at top
    Viewbox viewbox = new Viewbox
    {
        Child = new TextBlock
        {
            Text = calendar.MonthAsSoloString() + " " + calendar.YearAsString(),
            Foreground = black,
            FontSize = 96,
            HorizontalAlignment = HorizontalAlignment.Center
        }
    };
    Grid.SetRow(viewbox, 0);
    Grid.SetColumn(viewbox, 0);
    Grid.SetColumnSpan(viewbox, 7);
    grid.Children.Add(viewbox);

    // Now loop through the days of the month
    for (int day = 1, row = 1, col = (int)calendar.DayOfWeek;
        day <= calendar.LastDayInThisMonth; day++)
    {
        Border dayBorder = new Border

```

```

{
    BorderBrush = black,

    // Avoid double line drawing
    BorderThickness = new Thickness
    {
        Left = day == 1 || col == 0 ? 1 : 0,
        Top = day - 7 < 1 ? 1 : 0,
        Right = 1,
        Bottom = 1
    },

    // Put day of month in upper-left corner
    Child = new TextBlock
    {
        Text = day.ToString(),
        Foreground = black,
        FontSize = 24,
        HorizontalAlignment = HorizontalAlignment.Left,
        VerticalAlignment = VerticalAlignment.Top
    }
};
Grid.SetRow(dayBorder, row);
Grid.SetColumn(dayBorder, col);
grid.Children.Add(dayBorder);

if (0 == (col = (col + 1) % 7))
    row += 1;
}
calendarPages.Add(border);
calendar.AddMonths(1);
pageNumber += 1;
}
while (calendar.Year < monthYearSelect2.MonthYear.Year ||
        calendar.Month <= monthYearSelect2.MonthYear.Month);

printDocument.SetPreviewPageCount(calendarPages.Count, PreviewPageCountType.Final);
}

void OnPrintDocumentGetPreviewPage(object sender, GetPreviewPageEventArgs args)
{
    printDocument.SetPreviewPage(args.PageNumber, calendarPages[args.PageNumber - 1]);
}

void OnPrintDocumentAddPages(object sender, AddPagesEventArgs args)
{
    foreach (UIElement calendarPage in calendarPages)
        printDocument.AddPage(calendarPage);

    printDocument.AddPagesComplete();
}
}

```

Printing a Range of Pages

The final program in this chapter is an experiment that got completely out of control. I wanted to demonstrate how to add an option to the printing panel to allow the user to select a variable range of pages to print. At the same time, I wanted to show how to share *UIElement* instances between the screen and the printer.

For this demonstration I chose to revamp the program from Chapter 4 that presented Beatrix Potter's *The Tale of Tom Kitten*. To allow the pages of this book to be easily printed, I decided that each book page should be a separate *UserControl* derivative. For the on-screen rendition, these separate *UserControl* pages could simply be assembled in a single scrollable *StackPanel*.

There's nothing really wrong with this scheme except that I ended up with 57 *UserControl* derivatives named *TomKitten03* through *TomKitten59*, the number indicating the page from the original book. But it turned out I really couldn't use the same instances of these controls on the screen and the printer unless I wanted the text or image from each page of the book to be printed in the upper-left corner of the printer page, and that was unacceptable.

Elements that are displayed on the screen are subjected to a layout process that defines their relationship to their parent, and in the general case you simply can't lift these elements out of a visual tree and expect that they will render satisfactory on the printer. And you can't mess around with these elements either. You can't set new properties on them just for the printer because those properties will affect how they're displayed on the screen. And you can't put them into another container because that violates the rule that an element can have only one parent.

I finally realized that I could use the same 57 *UserControl* derivatives for the screen and printer, but only if they were separate instances, which means that each of these controls is instantiated twice: one in *MainPage.xaml* for the screen, and again in *MainPage.xaml.cs* for the printer.

So in one sense the experiment was a failure because I couldn't simply reuse the instances, but the experiment also illuminated the awkwardness of a Visual Studio project that contains 57 *UserControl* derivatives! Visual Studio shakes in its boots loading and compiling all these XAML files, and we programmers should be nervous as well. This is not the way to make an e-book!

On the other hand, the program *does* demonstrate how to add a facility to the printer options to select a range of pages to print.

To allow a uniform set of styles to be applied to the *UserControl* derivatives in *MainPage.xaml* and also the *UserControl* derivatives instantiated in *MainPage.xaml.cs*, I moved all the *Style* definitions to *App.xaml*. This makes them available throughout the application.

Project: PrintableTomKitten | File: App.xaml

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

```

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="using:PrintableTomKitten">

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

        <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>
    </ResourceDictionary>
</Application.Resources>
</Application>

```

The MainPage.xaml file does little more than list all the individual pages of the book in a *StackPanel*. The following listing leaves out the middle section:

Project: PrintableTomKitten | File: MainPage.xaml (excerpt)

```

<Page
    x:Class="PrintableTomKitten.MainPage"
    IsTabStop="false"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:PrintableTomKitten">

    <Grid Background="White">
        <ScrollView>
            <StackPanel Name="bookPageStackPanel"
                MaxWidth="640"
                HorizontalAlignment="Center">

                <local:TomKitten03 />
                <local:TomKitten04 />
                <local:TomKitten05 />
            </StackPanel>
        </ScrollView>
    </Grid>

```

```

        <local:TomKitten06 />
        <local:TomKitten07 />
        <local:TomKitten08 />
        <local:TomKitten09 />

        <local:TomKitten10 />
        <local:TomKitten11 />
        <local:TomKitten13 />
        <local:TomKitten12 />

        <local:TomKitten14 />
        <local:TomKitten15 />
        <local:TomKitten17 />
        <local:TomKitten16 />

        ...
        <local:TomKitten50 />
        <local:TomKitten51 />
        <local:TomKitten53 />
        <local:TomKitten52 />

        <local:TomKitten54 />
        <local:TomKitten55 />
        <local:TomKitten56 />
        <local:TomKitten57 />

        <local:TomKitten59 />
        <local:TomKitten58 />
    </StackPanel>
</ScrollView>
</Grid>
</Page>

```

Some of these are seemingly out of sequence. As I discussed in Chapter 4, I found it necessary to swap some of the text and picture pages to provide a more coherent reading experience.

The pages containing only a single image are quite small:

Project: PrintableTomKitten | File: TomKitten20.xaml

```

<UserControl
    x:Class="PrintableTomKitten.TomKitten20"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

    <Image Source="Images/tom20.jpg" Style="{StaticResource imageStyle}" />
</UserControl>

```

Many pages have only one paragraph of text, like the following:

Project: PrintableTomKitten | File: TomKitten21.xaml

```

<UserControl
    x:Class="PrintableTomKitten.TomKitten21"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

```

```

<Grid VerticalAlignment="Center"
      MaxWidth="640">
  <TextBlock Style="{StaticResource paragraphTextStyle}">
    &#x2003;&#x2003;Tom Kitten was very fat, and he had grown;
    several buttons burst off. His mother sewed them on again.
  </TextBlock>
</Grid>
</UserControl>

```

Notice the *VerticalAlignment* and *MaxWidth* settings on the *Grid*. These settings are for the benefit of the printer. The *VerticalAlignment* setting has no effect when the control is displayed on the screen because it's a child of a *StackPanel* with a vertical orientation, and the *StackPanel* itself has a *MaxWidth* setting of 640.

Those pages with more than a paragraph of text require a *StackPanel*:

Project: PrintableTomKitten | File: TomKitten21.xaml

```

<UserControl
  x:Class="PrintableTomKitten.TomKitten22"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <Grid VerticalAlignment="Center"
        MaxWidth="640">
    <StackPanel>
      <TextBlock Style="{StaticResource paragraphTextStyle}">
        &#x2003;&#x2003;When the three kittens were ready, Mrs.
        Tabitha unwisely turned them out into the garden, to be
        out of the way while she made hot buttered toast.
      </TextBlock>

      <TextBlock Style="{StaticResource paragraphTextStyle}">
        &#x2003;&#x2003;"Now keep your frocks clean, children! You
        must walk on your hind legs. Keep away from the dirty
        ash-pit, and from Sally Henny Penny, and from the
        pig-stye and the Puddle-Ducks."
      </TextBlock>
    </StackPanel>
  </Grid>
</UserControl>

```

That's all the XAML you'll see from this project.

As you know, it's very common these days for programs to offer options to print all or part of a document. These options are often labeled something like All, Selection, and Custom Range. Because I have no concept of selection in the PrintableTomKitten project, my options are limited to "Print all pages" and "Print custom range."

It's also become common for this custom range to contain both individual pages and continuous page ranges separated by commas, such as 2-4, 7, 9-11. The constructor of the following *CustomPageRange* class accepts a string with such a custom page range and resolves the information

into a list of consecutive pages. For the string "2-4, 7, 9-11", the *PageMapping* property is set to the list of integers 2, 3, 4, 7, 9, 10, 11. If the string is invalid in some way, then *PageMapping* is null and *IsValid* returns *false*:

Project: PrintableTomKitten | File: CustomPrintRange.cs

```
using System;
using System.Collections.Generic;

namespace PrintableTomKitten
{
    public class CustomPageRange
    {
        // Structure used internally
        struct PageRange
        {
            public PageRange(int from, int to) : this()
            {
                this.From = from;
                this.To = to;
            }

            public int From { private set; get; }
            public int To { private set; get; }
        }

        public CustomPageRange(string str, int maxPageNumber)
        {
            List<PageRange> pageRanges = new List<PageRange>();
            string[] strRanges = str.Split(',');

            foreach (string strRange in strRanges)
            {
                int dashIndex = strRange.IndexOf('-');

                // Just one page number
                if (dashIndex == -1)
                {
                    int page;

                    if (Int32.TryParse(strRange.Trim(), out page) &&
                        page > 0 && page <= maxPageNumber)
                    {
                        pageRanges.Add(new PageRange(page, page));
                    }
                    else
                    {
                        return;
                    }
                }
                // Two page numbers separated by a dash
                else
                {
                    string strFrom = strRange.Substring(0, dashIndex);
```

```

        string strTo = strRange.Substring(dashIndex + 1);
        int from, to;

        if (Int32.TryParse(strFrom.Trim(), out from) &&
            Int32.TryParse(strTo.Trim(), out to) &&
            from > 0 && from <= maxPageNumber &&
            to > 0 && to <= maxPageNumber &&
            from <= to)
        {
            pageRanges.Add(new PageRange(from, to));
        }
        else
        {
            return;
        }
    }
}

// If we made it to this, the input string is valid
this.PageMapping = new List<int>();

// Define a mapping to page numbers
foreach (PageRange pageRange in pageRanges)
    for (int page = pageRange.From; page <= pageRange.To; page++)
        this.PageMapping.Add(page);
}

// Zero-based in, one-based out
public IList<int> PageMapping { private set; get; }

public bool IsValid
{
    get { return this.PageMapping != null; }
}
}
}

```

The PrintableTomKitten program uses this class in two places: when it's validating input that the user has entered in the printer options panel, and later in the *Paginate* event handler. In the second case, the *CustomPageRange* object is stored as a field for use by the *GetPreviewPage* and *AddPages* handlers.

Here's the MainPage.xaml.cs file up through *OnPrintTaskSourceRequested* override. Notice the big array at the top containing additional instances of all the book pages solely for printing:

Project: PrintableTomKitten | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    PrintDocument printDocument;
    IPrintDocumentSource printDocumentSource;
    CustomPageRange customPageRange;
    UIElement[] bookPages =
    {

```

```

        new TomKitten03(), new TomKitten04(), new TomKitten05(), new TomKitten06(),
        new TomKitten07(), new TomKitten08(), new TomKitten09(), new TomKitten10(),
        new TomKitten11(), new TomKitten12(), new TomKitten13(), new TomKitten14(),
        new TomKitten15(), new TomKitten16(), new TomKitten17(), new TomKitten18(),
        new TomKitten19(), new TomKitten20(), new TomKitten21(), new TomKitten22(),
        new TomKitten23(), new TomKitten24(), new TomKitten25(), new TomKitten26(),
        new TomKitten27(), new TomKitten28(), new TomKitten29(), new TomKitten30(),
        new TomKitten31(), new TomKitten32(), new TomKitten33(), new TomKitten34(),
        new TomKitten35(), new TomKitten36(), new TomKitten37(), new TomKitten38(),
        new TomKitten39(), new TomKitten40(), new TomKitten41(), new TomKitten42(),
        new TomKitten43(), new TomKitten44(), new TomKitten45(), new TomKitten46(),
        new TomKitten47(), new TomKitten48(), new TomKitten49(), new TomKitten50(),
        new TomKitten51(), new TomKitten52(), new TomKitten53(), new TomKitten54(),
        new TomKitten55(), new TomKitten56(), new TomKitten57(), new TomKitten58(),
        new TomKitten59()
    };

    public MainPage()
    {
        this.InitializeComponent();

        // Create PrintDocument and attach handlers
        printDocument = new PrintDocument();
        printDocumentSource = printDocument.DocumentSource;
        printDocument.Paginate += OnPrintDocumentPaginate;
        printDocument.GetPreviewPage += OnPrintDocumentGetPreviewPage;
        printDocument.AddPages += OnPrintDocumentAddPages;
    }

    protected override void OnNavigatedTo(NavigationEventArgs args)
    {
        // Attach PrintManager handler
        PrintManager.GetForCurrentView().PrintTaskRequested += OnPrintManagerPrintTaskRequested;

        base.OnNavigatedTo(args);
    }

    protected override void OnNavigatedFrom(NavigationEventArgs e)
    {
        // Detach PrintManager handler
        PrintManager.GetForCurrentView().PrintTaskRequested -= OnPrintManagerPrintTaskRequested;

        base.OnNavigatedFrom(e);
    }

    void OnPrintManagerPrintTaskRequested(PrintManager sender, PrintTaskRequestedEventArgs args)
    {
        PrintTask printTask = args.Request.CreatePrintTask("The Tale of Tom Kitten",
            OnPrintTaskSourceRequested);

        // Get PrintTaskOptionDetails for making changes to options
        PrintTaskOptionDetails optionDetails =
            PrintTaskOptionDetails.GetFromPrintTaskOptions(printTask.Options);
    }

```

```

// Create the custom item
PrintCustomItemListOptionDetails pageRange =
    optionDetails.CreateItemListOption("idPrintRange", "Print range");
pageRange.AddItem("idPrintAll", "Print all pages");
pageRange.AddItem("idPrintCustom", "Print custom range");

// Add it to the options
optionDetails.DisplayedOptions.Add("idPrintRange");

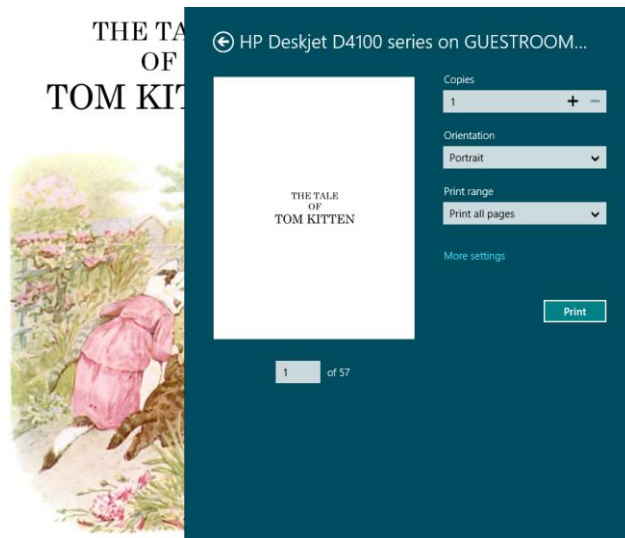
// Create a page-range edit item also, but this only
//     comes into play when user selects "Print custom range"
optionDetails.CreateTextOption("idCustomRangeEdit", "Custom Range");

// Set a handler for the OptionChanged event
optionDetails.OptionChanged += OnOptionDetailsOptionChanged;
}

void OnPrintTaskSourceRequested(PrintTaskSourceRequestedArgs args)
{
    args.SetSource(printDocumentSource);
}
...
}

```

Earlier you saw how to create custom text-entry fields using the *CreateTextOption* method of *PrintTaskOptionDetails*. The only other alternative is *CreateItemListOption* shown here. This is a list of mutually-exclusive options similar to the Orientation option. Give the method a string ID and a label. The method returns an object of type *PrintCustomItemListOptionDetails*. To that you'll need to add the individual items with ID strings and labels and then add those same IDs to the *DisplayedOptions* collection. Here's what it looks like initially:



But notice also that the *PrintTaskRequested* handler also calls *CreateTextOption* to create a text-entry field for the custom page range:

```
optionDetails.CreateTextOption("idCustomRangeEdit", "Custom Range");
```

This is created but it's not added to the *DisplayedOptions* collection yet. You want this item displayed only when the user selects "Print custom range."

This logic occurs in the *OptionChanged* handler. If the option ID string is "idPrintCustom", you then want to add the text-entry field identified by the string "idCustomRangeEdit" to the *DisplayedOptions* collection, and if the ID string is "idPrintAll", it must be removed from *DisplayedOptions*:

Project: PrintableTomKitten | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    ...
    async void OnOptionDetailsOptionChanged(PrintTaskOptionDetails sender,
                                             PrintTaskOptionChangedEventArgs args)
    {
        if (args.OptionId == null)
            return;

        string optionId = args.OptionId.ToString();
        string strValue = sender.Options[optionId].Value.ToString();
        string errorText = String.Empty;

        switch (optionId)
        {
            case "idPrintRange":
                switch (strValue)
                {
                    case "idPrintAll":
                        if (sender.DisplayedOptions.Contains("idCustomRangeEdit"))
                            sender.DisplayedOptions.Remove("idCustomRangeEdit");
                        break;

                    case "idPrintCustom":
                        sender.DisplayedOptions.Add("idCustomRangeEdit");
                        break;
                }
                break;

            case "idCustomRangeEdit":
                // Check to see if CustomPageRange accepts this
                if (!new CustomPageRange(strValue, bookPages.Length).IsValid)
                {
                    errorText = "Use the form 2-4, 7, 9-11";
                }
                break;
        }

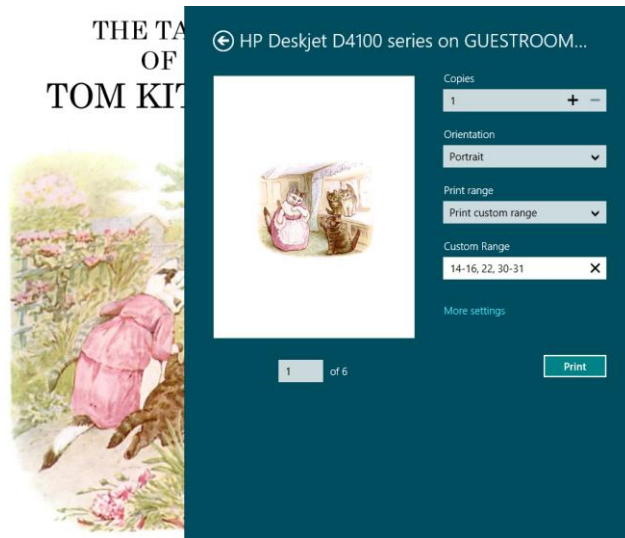
        sender.Options[optionId].ErrorText = errorText;
    }
}
```

```

// If no error, then invalidate the preview
if (String.IsNullOrEmpty(errorText))
{
    await this.Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () =>
    {
        printDocument.InvalidatePreview();
    });
}
...
}

```

If the "idCustomRangeEdit" control is visible, you can also receive notifications from that. To determine whether the range is valid, the *CustomPageRange* constructor is called and a possible error text is set. Here's a page range that is successfully parsed:



Notice that the page numbers underneath the preview indicate the number of pages that should be printed but don't indicate the actual page numbers the user has selected, but I'm not sure that problem can really be fixed unless a page range selection is moved into the standard options, and that's something beyond our control.

Also, notice that the *OptionChanged* handler does not save the *CustomPageRange* object as a field. You don't need to save it in this handler, and you should probably avoid doing so. As the user bounces back and forth among the options, it can be tricky to keep track of what's actually selected and visible and what's not.

Instead, you can obtain the final settings of the options in the three handlers for the *PrintDocument* events. In this example, the *Paginate* handler obtains the settings and saves a *CustomPageRange* object as a field, which is then accessed by the other two methods:

Project: PrintableTomKitten | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    ...
    void OnPrintDocumentPaginate(object sender, PaginateEventArgs args)
    {
        // Obtain the print range option
        PrintTaskOptionDetails optionDetails =
            PrintTaskOptionDetails.GetFromPrintTaskOptions(args.PrintTaskOptions);

        string strValue = optionDetails.Options["idPrintRange"].Value as string;

        if (strValue == "idPrintCustom")
        {
            // Parse the print range for GetPreviewPage and AddPages
            string strPageRange = optionDetails.Options["idCustomRangeEdit"].Value as string;
            customPageRange = new CustomPageRange(strPageRange, bookPages.Length);
        }
        else
        {
            // Make sure field is null if printing all pages
            customPageRange = null;
        }

        int pageCount = bookPages.Length;

        if (customPageRange != null && customPageRange.IsValid)
            pageCount = customPageRange.PageMapping.Count;

        printDocument.SetPreviewPageCount(pageCount, PreviewPageCountType.Final);
    }

    void OnPrintDocumentGetPreviewPage(object sender, GetPreviewPageEventArgs args)
    {
        int oneBasedIndex = args.PageNumber;

        if (customPageRange != null && customPageRange.IsValid)
            oneBasedIndex = customPageRange.PageMapping[args.PageNumber - 1];

        printDocument.SetPreviewPage(args.PageNumber, bookPages[oneBasedIndex - 1]);
    }

    void OnPrintDocumentAddPages(object sender, AddPagesEventArgs args)
    {
        if (customPageRange != null && customPageRange.IsValid)
        {
            foreach (int oneBasedIndex in customPageRange.PageMapping)
                printDocument.AddPage(bookPages[oneBasedIndex - 1]);
        }
        else
        {
            foreach (UIElement bookPage in bookPages)
                printDocument.AddPage(bookPage);
        }
    }
}
```

```

        printDocument.AddPagesComplete();
    }
}

```

I mentioned that I started this printable version of *The Tale of Tom Kitten* to determine whether I could share elements between the screen and printer. If you'd like to see what happens when you print the instances of the *UserControl* derivatives from *MainPage*, simply replace *bookPages* with *bookPageStackPanel.Children* in the *OnPrintDocumentGetPreviewPage* and *OnPrintDocumentAddPages* methods.

Where to Do the Big Jobs?

I hope to have an example of a program that has a nontrivial pagination job to explore the best way of doing such a thing, but a couple clues are already available.

In the callback method that you pass to *CreatePrintTask* (this is the method I've been calling *OnPrintTaskSourceRequested*), after calling *SetSource* on the event arguments, you can use the event arguments to obtain a deferral for performing an asynchronous job:

```

PrintTaskSourceRequestedDeferral deferral = args.GetDeferral();
await BigJobInvolvingPrintingAsync();
deferral.Complete();

```

In this case, the printing panel with the name of the selected printer is displayed, but under that printer name spins a progress ring accompanied by the text "App preparing to print." The user may not enjoy the experience, but it's a valid way for the application to gain a little time.

Also keep in mind that the second argument of the *SetPreviewPageCount* method of *PrintDocument* is a member of the *PreviewPageCountType* enumeration, either *Intermediate* or *Final*. You don't need to restrict calls of this method to the *Paginate* handler. You can call it initially with a preliminary page count and then have a background task continuing with the pagination. A *Dispatcher* to the user interface thread can make additional calls to *SetPreviewPageCount* to keep the count updated.

To assist your application in keeping the user informed of the progress of a long print job, *PrintTask* defines events named *Previewing*, *Submitting*, *Progressing*, and *Completed*.

Chapter 16

Going Native

In the world of Windows 8 programming, it is a sad truth that all languages are not created equal. In theory, any programming language can access any class or function available for new Windows 8 applications, but that's only because the entire API is built on top of the Component Object Model (COM). In the real world of sane programming, the ease of accessing certain areas of the Windows 8 API is dependent on what programming language you're using.

For example, only programmers working with the managed languages of C# and Visual Basic have direct access to the .NET APIs for Windows 8 applications—those namespaces beginning with the word *System*. C++ programmers are expected to use C++ runtime libraries and classes in the *Platform* namespace for these chores.

On the other hand, Windows 8 applications can use a subset of the Win32 and COM API, but these functions and classes are only conveniently available to C++ programmers. To get at these same APIs, C# programmers need to jump through hoops.

This is a chapter showing how to jump through those hoops. I'll discuss two basic techniques. The first is called Platform Invoke (also known as P/Invoke or P/Invoke), which has existed from the very beginning of .NET programming for accessing Win32 functions or functions in other dynamic-link libraries. P/Invoke is particularly suited for accessing a "flat" API—that is, one in which functions are independent and reference handles rather than being consolidated into classes.

The second technique involves writing a "wrapper" DLL in C++ and then accessing that DLL from a C# program. This technique is more suited for object-oriented APIs and, in particular, the big chunk of high-performance graphics and audio classes collectively known as DirectX.

In a Windows 8 application, a DLL written in one language and accessed by another must be in a special format known as a Windows Runtime Component. Visual Studio lets you create a Windows Runtime Component, but there are a bunch of rules and restrictions on what these libraries may do.

Keep in mind that you can't use either of these techniques to enable your program to access functions that aren't allowed in new Windows 8 applications. You can't use these techniques to access arbitrary Win32 functions. You're restricted to those in the subset allowed for Windows 8 applications. Nor can you call functions in DLLs that make calls to Win32 functions not in this subset.

An Introduction to P/Invoke

Suppose you're browsing through the subset of Win32 functions allowed for new Windows 8 applications and you encounter one that you'd like to use. Here's the way it appears in the

documentation:

```
void WINAPI GetNativeSystemInfo(__out LPSYSTEM_INFO lpSystemInfo);
```

If you're completely unfamiliar with the Win32 API, this is likely to look like gibberish. The uppercase identifiers are generally defined using C *#define* or *typedef* statements in the various Windows header files. You're likely to find these header files in subdirectories of the *C:/Program Files (x86)/Windows Kits/8.0* directory. The most basic are *Windows.h*, *WinDef.h*, and *WinBase.h*. The *WINAPI* identifier is the same as *__stdcall*, which is the standard calling convention for Win32 functions. *LPSYSTEM_INFO* is a long pointer—that is, not a 16-bit pointer such as existed back when Windows was a wee child—to a *SYSTEM_INFO* structure, which is defined in the documentation like this:

```
typedef struct _SYSTEM_INFO {
    union {
        DWORD   dwOemId;
        struct {
            WORD  wProcessorArchitecture;
            WORD  wReserved;
        };
    };
    DWORD       dwPageSize;
    LPVOID      lpMinimumApplicationAddress;
    LPVOID      lpMaximumApplicationAddress;
    DWORD_PTR   dwActiveProcessorMask;
    DWORD       dwNumberOfProcessors;
    DWORD       dwProcessorType;
    DWORD       dwAllocationGranularity;
    WORD        wProcessorLevel;
    WORD        wProcessorRevision;
} SYSTEM_INFO;
```

The prefacing of field names with lowercase abbreviations of the data types is a form of simple Hungarian notation, so called because it originated with Hungarian-born Charles Simonyi, and it was popularized by the Windows API and books about the Windows API, but it is no longer widely used in application programming.

In Windows parlance, a *WORD* is a 16-bit unsigned value, which C# programmers know as a *ushort*. The *DWORD* is a double-WORD, or a 32-bit unsigned value or a *uint*. Watch out for references to *long*, which is not a 64-bit C# *long* but instead a C++ *long*, which is the same size as an *int*, or 32 bits.

LPVOID translates as a "long pointer to void" or, in standard C, *void **, and *DWORD_PTR* is either an unsigned 32-bit or 64-bit integer, depending on whether Windows is running on a 32-bit or 64-bit processor. These are equivalent to the C# *IntPtr*.

The reason you need to know how these data types correspond to C# data types is because you need to redefine this structure in C#. Fortunately, the documentation of *SYSTEM_INFO* indicates that the *dwOemId* field is obsolete, which means that you can ignore the *union* and simply create a straight C# structure with public fields, perhaps giving it a more C#-ish name in the process:

```
struct SystemInfo
```

```

{
    public ushort wProcessorArchitecture;
    public ushort wReserved;
    public uint dwPageSize;
    public IntPtr lpMinimumApplicationAddress;
    public IntPtr lpMaximumApplicationAddress;
    public IntPtr dwActiveProcessorMask;
    public uint dwNumberOfProcessors;
    public uint dwProcessorType;
    public uint dwAllocationGranularity;
    public ushort wProcessorLevel;
    public ushort wProcessorRevision;
}

```

Of course, in C# the fields must be defined as *public* if you want to access them from outside the structure. If you want, you can also rename all the fields (for example, *ProcessorArchitecture* and *PageSize*).

You can also specify different data types of the same size—for example, *short* rather than *ushort* and *int* rather than *uint*—if you know that the actual values won't overrun the signed types. To the Windows API, all you're doing is supplying a block of memory. The total structure occupies 36 bytes of memory in 32-bit Windows and 48 bytes in 64-bit Windows.

Very often in P/Invoke code you'll see the structure preceded by the following attribute:

```

[StructLayout(LayoutKind.Sequential)]
struct SystemInfo
{
    ...
}

```

The *StructLayoutAttribute* class and the *LayoutKind* enumeration are defined in the *System.Runtime.InteropServices* namespace, which has lots of other classes related to P/Invoke. This attribute indicates explicitly that these fields should be treated as contiguous and aligned on byte boundaries.

Now that you have a structure to be passed to the *GetNativeSystemInfo* function, you must declare the function itself. In doing so, you make use of the *DllImportAttribute*, also defined in *System.Runtime.InteropServices*, which at the very least indicates the dynamic-link library in which this function can be found. The documentation indicates that *GetNativeSystemInfo* is defined in *kernel32.dll*. Here's the function declaration:

```

[DllImport("kernel32.dll")]
static extern void GetNativeSystemInfo(out SystemInfo systemInfo);

```

This declaration must appear inside a C# class definition at the same level as the other methods. The function must be declared as *static*, which is common in regular C# classes, but also as *extern*, which is not common but means that the actual implementation of this function is external to the class.

If you want the function to be visible outside the class, give it a *public* keyword as well.

With the exception of *extern*, the function declaration otherwise appears to be a C# method. The method returns *void*, and the single argument is a reference to a *SystemInfo* object. Many Windows API calls require or return information in structures, and you'll define those arguments by using either *out* or *ref*. These are functionally identical but with *ref*: the C# compiler checks to see that you've initialized the value type before calling the function.

In some other method of the class, call the function as if it's a normal static method:

```
SystemInfo systemInfo;
GetNativeSystemInfo(out systemInfo);
```

The SystemInfoPInvoke project implements this code and displays all the fields of the structure. The XAML file uses a *Grid* to format the information in a table:

Project: SystemInfoPInvoke | File: MainPage.xaml (excerpt)

```
<Page ...
    FontSize="24">

    <Page.Resources>
        <Style x:Key="rightJustifiedText" TargetType="TextBlock">
            <Setter Property="TextAlignment" Value="Right" />
            <Setter Property="Margin" Value="12 0 0 0" />
        </Style>
    </Page.Resources>

    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Grid HorizontalAlignment="Center"
            VerticalAlignment="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" />
            </Grid.RowDefinitions>

            <Grid.ColumnDefinitions>
                <ColumnDefinition Width="Auto" />
                <ColumnDefinition Width="Auto" />
            </Grid.ColumnDefinitions>

            <TextBlock Text="Processor Architecture: " Grid.Row="0" Grid.Column="0" />
            <TextBlock Name="processorArchitecture" Grid.Row="0" Grid.Column="1"
                Style="{StaticResource rightJustifiedText}" />

            <TextBlock Text="Page Size: " Grid.Row="1" Grid.Column="0" />
            <TextBlock Name="pageSize" Grid.Row="1" Grid.Column="1"
                Style="{StaticResource rightJustifiedText}" />
```

```

<TextBlock Text="Minimum Application Address: " Grid.Row="2" Grid.Column="0" />
<TextBlock Name="minAppAddr" Grid.Row="2" Grid.Column="1"
    Style="{StaticResource rightJustifiedText}" />

<TextBlock Text="Maximum Application Address: " Grid.Row="3" Grid.Column="0" />
<TextBlock Name="maxAppAddr" Grid.Row="3" Grid.Column="1"
    Style="{StaticResource rightJustifiedText}" />

<TextBlock Text="Active Processor Mask: " Grid.Row="4" Grid.Column="0" />
<TextBlock Name="activeProcessorMask" Grid.Row="4" Grid.Column="1"
    Style="{StaticResource rightJustifiedText}" />

<TextBlock Text="Number of Processors: " Grid.Row="5" Grid.Column="0" />
<TextBlock Name="numberOfProcessors" Grid.Row="5" Grid.Column="1"
    Style="{StaticResource rightJustifiedText}" />

<TextBlock Text="Allocation Granularity: " Grid.Row="6" Grid.Column="0" />
<TextBlock Name="allocationGranularity" Grid.Row="6" Grid.Column="1"
    Style="{StaticResource rightJustifiedText}" />

<TextBlock Text="Processor Level: " Grid.Row="7" Grid.Column="0" />
<TextBlock Name="processorLevel" Grid.Row="7" Grid.Column="1"
    Style="{StaticResource rightJustifiedText}" />

<TextBlock Text="Processor Revision: " Grid.Row="8" Grid.Column="0" />
<TextBlock Name="processorRevision" Grid.Row="8" Grid.Column="1"
    Style="{StaticResource rightJustifiedText}" />

</Grid>
</Grid>
</Page>

```

In the code-behind file, both the structure and the function declaration are defined within the *MainPage* class. The function must be within a class definition, but the structure need not be. Here's the complete file:

Project: SystemInfoPInvoke | File: MainPage.xaml.cs

```

using System;
using System.Runtime.InteropServices;
using Windows.UI.Xaml.Controls;

namespace SystemInfoPInvoke
{
    public sealed partial class MainPage : Page
    {
        [StructLayout(LayoutKind.Sequential)]
        struct SystemInfo
        {
            public ushort wProcessorArchitecture;
            public byte wReserved;
            public uint dwPageSize;
            public IntPtr lpMinimumApplicationAddress;
            public IntPtr lpMaximumApplicationAddress;
            public IntPtr dwActiveProcessorMask;
            public uint dwNumberOfProcessors;

```

```

        public uint dwProcessorType;
        public uint dwAllocationGranularity;
        public ushort wProcessorLevel;
        public ushort wProcessorRevision;
    }

    [DllImport("kernel32.dll")]
    static extern void GetNativeSystemInfo(out SystemInfo systemInfo);

    enum ProcessorType
    {
        x86 = 0,
        ia64 = 6,
        x64 = 9,
        Unknown = 65535
    };

    public MainPage()
    {
        this.InitializeComponent();

        SystemInfo systemInfo = new SystemInfo();
        GetNativeSystemInfo(out systemInfo);

        processorArchitecture.Text =
            ((ProcessorType)systemInfo.wProcessorArchitecture).ToString();
        pageSize.Text = systemInfo.dwPageSize.ToString();
        minAppAddr.Text = ((ulong)systemInfo.lpMinimumApplicationAddress).ToString("X");
        maxAppAddr.Text = ((ulong)systemInfo.lpMaximumApplicationAddress).ToString("X");
        activeProcessorMask.Text = ((ulong)systemInfo.dwActiveProcessorMask).ToString("X");
        numberProcessors.Text = systemInfo.dwNumberOfProcessors.ToString("X");
        allocationGranularity.Text = systemInfo.dwAllocationGranularity.ToString();
        processorLevel.Text = systemInfo.wProcessorLevel.ToString();
        processorRevision.Text = systemInfo.wProcessorRevision.ToString("X");
    }
}

```

The documentation indicates that the *wProcessorArchitecture* field can take on values of 0 (for x86 architectures), 6 (for Intel Itanium), 9 (for x64), and 0xFFFF for "unknown." To ease the formatting of this value, I defined a little *enum* called *ProcessorType* and cast the *wProcessorArchitecture* value to that enumeration. For the *IntPtr* fields, I cast to *ulong* and then displayed as hexadecimal. Here's the screen running on the tablet I'm using to write this book:

```
Processor Architecture:      x64
Page Size:                  4096
Minimum Application Address: 10000
Maximum Application Address: FFFFFFFF
Active Processor Mask:      F
Number of Processors:       4
Allocation Granularity:     65536
Processor Level:            6
Processor Revision:         2A07
```

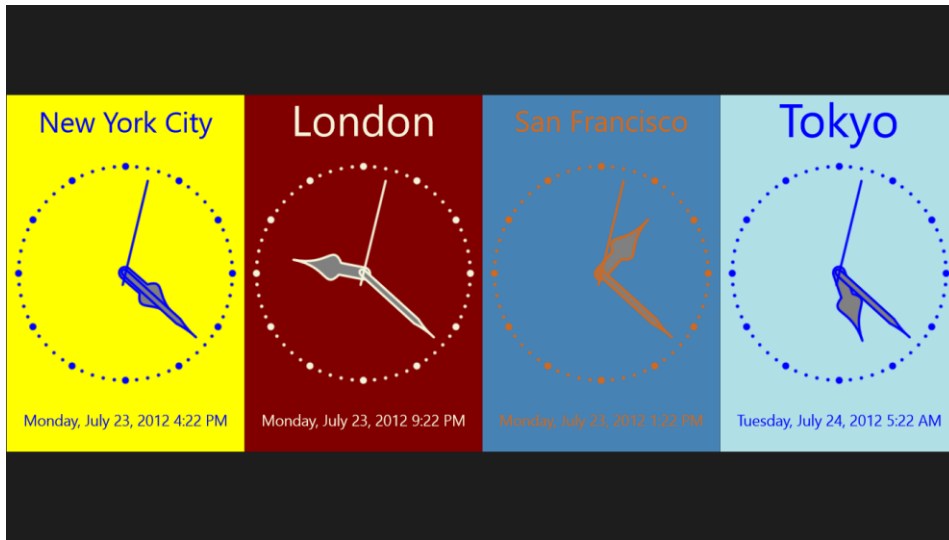
Some Help

When you use P/Invoke to define structures and declare functions, you are taking on all the responsibility for getting it right. You must supply the correct filename of the DLL in which the function is located, and if that's not a system DLL, you must make sure the DLL is referenced by the application. You must also spell the function name correctly and declare all the arguments correctly. There's no IntelliSense for P/Invoke!

These structures and function declarations can often be complex. To help out, there is a wiki website www.pinvoke.net, to which many people have contributed structure definitions and function declarations that you can just copy and paste into your own code. You're even allowed to contribute some of your own!

Time Zone Information

Suppose you'd like to write a Windows 8 style application that displays a bunch of clocks set for various locations around the world, perhaps similar to the ClockRack program I wrote for *PC Magazine* back in the year 2000. Perhaps the Windows 8 version would look something like this:



Such a program would let you add new clocks, set their locations and time zones, and give them unique colors, also retaining this information in application settings.

It would be great to take advantage of the built-in support of Windows for computing the time in various time zones, and particularly for handling the problem of daylight saving time (known in some places as “summer time”).

You might be very enthusiastic upon finding the *TimeZoneInfo* class in the *System* namespace and noting that the static *GetSystemTimeZones* method returns a collection of *TimeZoneInfo* objects for all the time zones around the world. However, when you actually try to use this method, you’ll discover that it’s not available for Windows 8 applications. The only *TimeZoneInfo* object you can obtain in a Windows 8 application is one that’s appropriate for the current system time zone setting or the trivial one for Universal Coordinated Time (UTC), also known less accurately but more commonly as Greenwich Mean Time.

However, a Windows 8 application does have access to several Win32 functions that provide much of the information you’ll need. The *EnumDynamicTimeZoneInformation* function enumerates all the time zones around the world in the form of *DYNAMIC_TIME_ZONE_INFORMATION* structures:

```
typedef struct _TIME_DYNAMIC_ZONE_INFORMATION {
    LONG        Bias;
    WCHAR        StandardName[32];
    SYSTEMTIME StandardDate;
    LONG        StandardBias;
    WCHAR        DaylightName[32];
    SYSTEMTIME DaylightDate;
    LONG        DaylightBias;
    WCHAR        TimeZoneKeyName[128];
    BOOLEAN      DynamicDaylightTimeDisabled;
} DYNAMIC_TIME_ZONE_INFORMATION, *PDYNAMIC_TIME_ZONE_INFORMATION;
```


This is an extended version of the `TIME_ZONE_INFORMATION` structure:

```
typedef struct _TIME_ZONE_INFORMATION {
    LONG        Bias;
    WCHAR        StandardName[32];
    SYSTEMTIME   StandardDate;
    LONG        StandardBias;
    WCHAR        DaylightName[32];
    SYSTEMTIME   DaylightDate;
    LONG        DaylightBias;
} TIME_ZONE_INFORMATION, *PTIME_ZONE_INFORMATION;
```

`WCHAR` is a wide 16-bit Unicode character, and arrays of these characters are essentially zero-terminated strings. The *StandardName* is a string like “Eastern Standard Time”, and *DaylightName* is a string like “Eastern Daylight Time”. The *TimeZoneKeyName* in the `DYNAMIC_TIME_ZONE_INFORMATION` structure is the key used in the Windows registry. In Windows 8, these registry entries can be found at `HKEY_LOCAL_MACHINE/SOFTWARE/Microsoft/Windows NT/CurrentVersion/Time Zones`, and the registry key matches *StandardName*.

The *Bias* field is a number of minutes to subtract from Universal Coordinated Time to get local time. For the eastern US time zone, that’s 300 minutes. The *StandardBias* is always zero, while *DaylightBias* is the number of minutes to subtract from standard time to convert to summer time, usually –60.

The *DaylightDate* and *StandardDate* fields indicate when the switch to daylight saving time and back to standard occurs, and they are of type `SYSTEMTIME`:

```
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

The `SYSTEMTIME` structure is mostly used with the Win32 API *GetLocalTime* and *GetSystemTime* functions to obtain current local time and UTC, respectively. The `SYSTEMTIME` values in the `TIME_ZONE_INFORMATION` structure are specially coded for the purpose of indicating a transition date. The *wHour* and *wMinute* fields indicate the time of the transition, the *wMonth* field indicates the month of the transition (for example, 3 for March), the *wDayOfWeek* field indicates the day of the week of the transition (for example, 1 for Sunday), and the *wDay* field indicates which occurrence of that day of the week, for example, 2 for the second Sunday, or 5 for the last Sunday.

Windows makes a distinction between locales that switch between standard time and daylight saving time on days indicated like that and those locales that dynamically change the date every year. These latter are referred to as using Dynamic DST, but the information by year is not directly available.

A function named *GetTimeZoneInformationForYear* accepts a year argument and a pointer to

DYNAMIC_TIME_ZONE_INFORMATION structure and returns a pointer to a TIME_ZONE_INFORMATION structure with the information appropriate for that year. The *SystemTimeToTzSpecificLocalTime* accepts a pointer to this TIME_ZONE_INFORMATION structure and a pointer to a SYSTEMTIME structure probably obtained from the *GetSystemTime* function and returns a SYSTEMTIME structure indicating local time for that time zone. Thus, it is not necessary for programs to perform their own time conversions.

Take a look at the facility that Windows 8 provides for users to specify a time zone. Use the Settings charm, and tap the “Change PC Settings” label at the bottom. This invokes a program with the title PC Settings and a list. Select General, and a combo box with the time zones is at the top. You’ll discover that time zones are identified by an offset from UTC, sometimes the name of the time zone, and often some sample cities. For example, for Romance Standard Time, the combo box displays

(UTC+01:00) Brussels, Copenhagen, Madrid, Paris

In the Windows registry, you’ll find these labels among the other information identified with a name of “Display,” but this information is not provided by the Win32 functions. You’d need to access the registry to obtain it, and there are no Win32 functions available to Windows 8 applications to access the registry.

Of course, nothing prevents you from writing a little desktop .NET program to access the full *TimeZoneInfo* class and format the resultant strings so that they define a *Dictionary* object. That’s what I’ve done here. The code in the .NET program I used to generate this list is shown in the comment:

Project: ClockRack | File: TimeZoneManager.Display.cs (excerpt)

```
namespace ClockRack
{
    public partial class TimeZoneManager
    {
        // Generated from tiny .NET program:
        // foreach (TimeZoneInfo info in TimeZoneInfo.GetSystemTimeZones())
        //     Console.WriteLine("{0}\", \"{1}\" }", info.StandardName, info.DisplayName);
        static Dictionary<string, string> displayStrings = new Dictionary<string, string>
        {
            { "Dateline Standard Time", "(UTC-12:00) International Date Line West" },
            { "UTC-11", "(UTC-11:00) Coordinated Universal Time-11" },
            { "Hawaiian Standard Time", "(UTC-10:00) Hawaii" },
            { "Alaskan Standard Time", "(UTC-09:00) Alaska" },
            { "Pacific Standard Time (Mexico)", "(UTC-08:00) Baja California" },
            ...
            { "Kamchatka Standard Time", "(UTC+12:00) Petropavlovsk-Kamchatsky - Old" },
            { "Tonga Standard Time", "(UTC+13:00) Nuku'alofa" },
            { "Samoa Standard Time", "(UTC+13:00) Samoa" }
        };
    }
}
```

This dictionary is part of a class called *TimeZoneManager* in the ClockRack project. This is the class I’ve used to consolidate all the P/Invoke logic. No code outside of the *TimeZoneManager* accesses any

Win32 function or structure.

The *TimeZoneManager* class is designed to be instantiated only once and be used for the duration of the application. The class makes time zone data available to the rest of the program as a collection of the following values:

Project: ClockRack | File: TimeZoneDisplayInfo.cs

```
namespace ClockRack
{
    public struct TimeZoneDisplayInfo
    {
        public int Bias { set; get; }
        public string TimeZoneKey { set; get; }
        public string Display { set; get; }
    }
}
```

The *Bias* property is only used for sorting. The *TimeZoneKey* is the same string as the *TimeZoneKeyName* in the DYNAMIC_TIME_ZONE_INFORMATION structure, and the *Display* property is obtained from the *displayStrings* dictionary.

The portion of the *TimeZoneManager* class in the main TimeZoneManager.cs file begins by defining the necessary Win32 structures and declaring three Win32 functions the class requires:

Project: ClockRack | File: TimeZoneManager.cs (excerpt)

```
public partial class TimeZoneManager
{
    [StructLayout(LayoutKind.Sequential)]
    struct SYSTEMTIME
    {
        public ushort wYear;
        public ushort wMonth;
        public ushort wDayOfWeek;
        public ushort wDay;
        public ushort wHour;
        public ushort wMinute;
        public ushort wSecond;
        public ushort wMilliseconds;
    }

    [StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
    struct TIME_ZONE_INFORMATION
    {
        public int Bias;
        [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
        public string StandardName;
        public SYSTEMTIME StandardDate;
        public int StandardBias;
        [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
        public string DaylightName;
        public SYSTEMTIME DaylightDate;
        public int DaylightBias;
    }
}
```

```

}

[StructLayout(LayoutKind.Sequential, CharSet = CharSet.Unicode)]
struct DYNAMIC_TIME_ZONE_INFORMATION
{
    public int Bias;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
    public string StandardName;
    public SYSTEMTIME StandardDate;
    public int StandardBias;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 32)]
    public string DaylightName;
    public SYSTEMTIME DaylightDate;
    public int DaylightBias;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst = 128)]
    public string TimeZoneKeyName;
    public byte DynamicDaylightTimeDisabled;
}

[DllImport("Advapi32.dll")]
static extern uint EnumDynamicTimeZoneInformation(uint index,
                                                ref DYNAMIC_TIME_ZONE_INFORMATION dynamicTzi);

[DllImport("kernel32.dll")]
static extern byte GetTimeZoneInformationForYear(ushort year,
                                                ref DYNAMIC_TIME_ZONE_INFORMATION dtzi,
                                                out TIME_ZONE_INFORMATION tzi);

[DllImport("kernel32.dll")]
static extern byte SystemTimeToTzSpecificLocalTime(ref TIME_ZONE_INFORMATION tzi,
                                                  ref SYSTEMTIME utc, out SYSTEMTIME local);

...
}

```

You'll recall that several of the fields in the `DYNAMIC_TIME_ZONE_INFORMATION` and `TIME_ZONE_INFORMATION` structure are defined as arrays of `WCHAR`. This wouldn't work in C# because a C# array is always a pointer to memory allocated from the heap. Instead, *MarshalAsAttribute* allows indicating that these fields should be treated as C# strings of a particular maximum length.

The constructor of the *TimeZoneManager* class repeatedly calls *EnumDynamicTimeZoneInformation* until it returns a nonzero value, indicating that it's at the end of the list. (With the version of Windows 8 I'm using for this chapter, I get 101 items.) Each item is stored in a private dictionary, and each item is also turned into a *TimeZoneDisplayInfo* value and added to the publicly available collection named *DisplayInformation*:

Project: ClockRack | File: `TimeZoneManager.cs` (excerpt)

```

public partial class TimeZoneManager
{
    ...
    // Internal dictionary for looking up DYNAMIC_TIME_ZONE_INFORMATION values from keys
    Dictionary<string, DYNAMIC_TIME_ZONE_INFORMATION> dynamicTzis =
        new Dictionary<string, DYNAMIC_TIME_ZONE_INFORMATION>();
}

```

```

public TimeZoneManager()
{
    uint index = 0;
    DYNAMIC_TIME_ZONE_INFORMATION tzi = new DYNAMIC_TIME_ZONE_INFORMATION();
    List<TimeZoneDisplayInfo> displayInformation = new List<TimeZoneDisplayInfo>();

    // Enumerate through time zones
    while (0 == EnumDynamicTimeZoneInformation(index, ref tzi))
    {
        dynamicTzis.Add(tzi.TimeZoneKeyName, tzi);

        // Create TimeZoneDisplayInfo for public property
        TimeZoneDisplayInfo displayInfo = new TimeZoneDisplayInfo
        {
            Bias = tzi.Bias,
            TimeZoneKey = tzi.TimeZoneKeyName
        };

        // Look up the display string
        if (displayStrings.ContainsKey(tzi.TimeZoneKeyName))
        {
            displayInfo.Display = displayStrings[tzi.TimeZoneKeyName];
        }
        else if (displayStrings.ContainsKey(tzi.StandardName))
        {
            displayInfo.Display = displayStrings[tzi.StandardName];
        }
        // Or calculate one
        else
        {
            if (tzi.Bias == 0)
                displayInfo.Display = "(UTC) ";
            else
                displayInfo.Display = String.Format("(UTC{0}{1:D2}:{2:D2}) ",
                    tzi.Bias > 0 ? '-' : '+',
                    Math.Abs(tzi.Bias) / 60,
                    Math.Abs(tzi.Bias) % 60);

            displayInfo.Display += tzi.TimeZoneKeyName;
        }

        // Add to collection
        displayInformation.Add(displayInfo);

        // Prepare for next iteration
        index += 1;
        tzi = new DYNAMIC_TIME_ZONE_INFORMATION();
    }

    // Sort the display information items
    displayInformation.Sort((TimeZoneDisplayInfo info1, TimeZoneDisplayInfo info2) =>
    {
        return info2.Bias.CompareTo(info1.Bias);
    });
}

```

```

        // Set to the publicly available property
        this.DisplayInformation = displayInformation;
    }

    // Public interface
    public IList<TimeZoneDisplayInfo> DisplayInformation { protected set; get; }
    ...
}

```

As you'll see shortly, this *DisplayInformation* property is used as an *ItemsSource* for a *ComboBox*.

The only remaining method in *TimeZoneManager* converts a UTC time into local time based on a time zone key value. This is the same string as the *TimeZoneKeyName* field of the DYNAMIC_TIME_ZONE_INFORMATION structure and the *TimeZoneKey* property of the *TimeZoneDisplayInfo* structure:

```

public partial class TimeZoneManager
{
    ...
    public DateTime GetLocalTime(string timeZoneKey, DateTime utc)
    {
        // Convert to Win32 SYSTEMTIME
        SYSTEMTIME utcSysTime = new SYSTEMTIME
        {
            wYear = (ushort)utc.Year,
            wMonth = (ushort)utc.Month,
            wDay = (ushort)utc.Day,
            wHour = (ushort)utc.Hour,
            wMinute = (ushort)utc.Minute,
            wSecond = (ushort)utc.Second,
            wMilliseconds = (ushort)utc.Millisecond
        };

        // Convert to local time
        DYNAMIC_TIME_ZONE_INFORMATION dtzi = dynamicTzis[timeZoneKey];
        TIME_ZONE_INFORMATION tzi = new TIME_ZONE_INFORMATION();
        GetTimeZoneInformationForYear((ushort)utc.Year, ref dtzi, out tzi);

        SYSTEMTIME localSysTime = new SYSTEMTIME();
        SystemTimeToTzSpecificLocalTime(ref tzi, ref utcSysTime, out localSysTime);

        // Convert SYSTEMTIME to DateTime
        return new DateTime(localSysTime.wYear, localSysTime.wMonth, localSysTime.wDay,
            localSysTime.wHour, localSysTime.wMinute,
            localSysTime.wSecond, localSysTime.wMilliseconds);
    }
}

```

The method converts a .NET *DateTime* to a Win32 *SYSTEMTIME*, obtains a *DYNAMIC_TIME_ZONE_INFORMATION* from the private dictionary, and then calls *GetTimeZoneInformationForYear*, which returns information in the form of a *TIME_ZONE_INFORMATION* structure, which is then passed to the *SystemTimeToTzSpecificLocalTime*

function. The resultant SYSTEMTIME is converted back to a .NET *DateTime*.

I'm not entirely happy with this method, and let me tell you why. The ClockRack program displays multiple clocks and uses a *CompositionTarget.Rendering* method to obtain an updated *DateTime.UtcNow* value, which it uses for all the clocks. (I figured this was probably more efficient than for this *GetLocalTime* method to call the Win32 *GetSystemTime* function to obtain a SYSTEMTIME value for UTC for each clock.) What I'm not sure about is repeatedly calling the *GetTimeZoneInformationForYear* method. This function really only needs to be called once for each time zone, and then the TIME_ZONE_INFORMATION can be reused in subsequent calls. However, if the program is running from December 31 to January 1, it needs to be called again for the New Year. I decided not to clutter up the class with logic of this sort.

The year passed to *GetTimeZoneInformationForYear* should be a local year, not a UTC year, and that's something else I'm not quite doing correctly. These two years are potentially only different during a 24-hour period surrounding the UTC New Year, and it really shouldn't matter in a program like this because the transition between standard time and daylight saving time occurs much later in the year.

However, if a particular locale in the southern hemisphere decided to observe daylight saving time in one year but not the next, or vice versa, the time might be calculated incorrectly in the hours around New Year in the transition between those two years.

But let's move on.

You'll recognize much of the actual clock (a *UserControl* derivative called *TimeZoneClock*) from the AnalogClock program from Chapter 9, "Transforms," but I've converted it to use a view model through binding. Also, the analog clock face is now surrounded by two *TextBlock* elements. The top one displays a location, and the bottom one displays the current date and time. Without that text time at the bottom, you might be a little confused about whether the time was before noon or after.

Each of the two *TextBlock* elements has a fixed height from the *RowDefinition* settings on the *Grid*, but they are also each in a *Viewbox* so that if the text gets too long, it is compressed to fit. Moreover, the analog clock is also given a fixed size, but the entire configuration of clock and text is in another *Viewbox*.

Project: ClockRack | File: TimeZoneClock.xaml (excerpt)

```
<UserControl ...
    Name="ctrl">

    <UserControl.Resources>
        <Style TargetType="TextBlock">
            <Setter Property="Margin" Value="12 0" />
            <Setter Property="TextAlignment" Value="Center" />
        </Style>

        <Style TargetType="Path">
            <Setter Property="StrokeThickness" Value="2" />
            <Setter Property="StrokeStartLineCap" Value="Round" />
        </Style>
    </UserControl.Resources>
</UserControl>
```

```

        <Setter Property="StrokeEndLineCap" Value="Round" />
        <Setter Property="StrokelineJoin" Value="Round" />
        <Setter Property="StrokeDashCap" Value="Round" />
        <Setter Property="Fill" Value="Gray" />
    </Style>
</UserControl.Resources>

<UserControl.Foreground>
    <SolidColorBrush Color="{Binding Foreground}" />
</UserControl.Foreground>

<Viewbox>
    <Grid Width="200">
        <Grid.Background>
            <SolidColorBrush Color="{Binding Background}" />
        </Grid.Background>

        <Grid.RowDefinitions>
            <RowDefinition Height="50" />
            <RowDefinition Height="200" />
            <RowDefinition Height="50" />
        </Grid.RowDefinitions>

        <Viewbox Grid.Row="0">
            <TextBlock Text="{Binding Location}" />
        </Viewbox>

        <Grid Grid.Row="1">

            <!-- Transform for entire clock -->
            <Grid.RenderTransform>
                <TranslateTransform X="100" Y="100" />
            </Grid.RenderTransform>

            <!-- Small tick marks -->
            <Path Fill="{x:Null}"
                Stroke="{Binding ElementName=ctrl, Path=Foreground}"
                StrokeThickness="3"
                StrokeDashArray="0 3.14159">
                <Path.Data>
                    <EllipseGeometry RadiusX="90" RadiusY="90" />
                </Path.Data>
            </Path>

            <!-- Large tick marks -->
            <Path Fill="{x:Null}"
                Stroke="{Binding ElementName=ctrl, Path=Foreground}"
                StrokeThickness="6"
                StrokeDashArray="0 7.854">
                <Path.Data>
                    <EllipseGeometry RadiusX="90" RadiusY="90" />
                </Path.Data>
            </Path>
        </Grid>
    </Grid>
</Viewbox>

```



```

<!-- Hour hand pointing straight up -->
<Path Data="M 0 -60 C 0 -30, 20 -30, 5 -20 L 5 0
           C 5 7.5, -5 7.5, -5 0 L -5 -20
           C -20 -30, 0 -30, 0 -60"
      Stroke="{Binding ElementName=ctrl, Path=Foreground}">
  <Path.RenderTransform>
    <RotateTransform Angle="{Binding HourAngle}" />
  </Path.RenderTransform>
</Path>

<!-- Minute hand pointing straight up -->
<Path Data="M 0 -80 C 0 -75, 0 -70, 2.5 -60 L 2.5 0
           C 2.5 5, -2.5 5, -2.5 0 L -2.55 -60
           C 0 -70, 0 -75, 0 -80"
      Stroke="{Binding ElementName=ctrl, Path=Foreground}">
  <Path.RenderTransform>
    <RotateTransform Angle="{Binding MinuteAngle}" />
  </Path.RenderTransform>
</Path>

<!-- Second hand pointing straight up -->
<Path Data="M 0 10 L 0 -80"
      Stroke="{Binding ElementName=ctrl, Path=Foreground}">
  <Path.RenderTransform>
    <RotateTransform Angle="{Binding SecondAngle}" />
  </Path.RenderTransform>
</Path>
</Grid>

<Viewbox Grid.Row="2">
  <TextBlock Text="{Binding FormattedDateTime}" />
</Viewbox>
</Grid>
</Viewbox>
</UserControl>

```

Both *TextBlock* elements and all the *RotateTransform* elements have bindings to properties in a view model. Look closer, and you'll see that this view model also includes properties of type *Color* named *Foreground* and *Background*.

The code-behind file does nothing. Well, that's not entirely true. For reasons, you'll see soon, it provides a public static field that indicates the aspect ratio of the control:

Project: ClockRack | File: TimeZoneClock.xaml.cs

```

using Windows.UI.Xaml.Controls;

namespace ClockRack
{
    public sealed partial class TimeZoneClock : UserControl
    {
        public static double AspectRatio = 1 / 1.5;

        public TimeZoneClock()

```

```

        {
            this.InitializeComponent();
        }
    }
}

```

To keep this program relatively simple, I limited the colors for the background and foreground of each clock to those 140 colors that have names and hence correspond to members of the static *Colors* class. The view model for the *TimeZoneClock* class defines *Foreground* and *Background* properties of type *Color* as you might expect, but it also defines *ForegroundName* and *BackgroundName* properties, and whenever one of these properties is changed, the other changes as well with a little reflection logic:

Project: ClockRack | File: TimeZoneClockViewModel.cs (excerpt)

```

public class TimeZoneClockViewModel : INotifyPropertyChanged
{
    string location = "New York City", timeZoneKey = "Eastern Standard Time";
    Color background = Colors.Yellow, foreground = Colors.Blue;
    string backgroundName = "Yellow", foregroundName = "Blue";
    DateTime dateTime;
    string formattedDateTime;
    double hourAngle, minuteAngle, secondAngle;
    TypeInfo colorsTypeInfo = typeof(Colors).GetTypeInfo();

    public event PropertyChangedEventHandler PropertyChanged;

    public string Location
    {
        set { SetProperty<string>(ref location, value); }
        get { return location; }
    }

    public string TimeZoneKey
    {
        set { SetProperty<string>(ref timeZoneKey, value); }
        get { return timeZoneKey; }
    }

    public string BackgroundName
    {
        set
        {
            if (SetProperty<string>(ref backgroundName, value))
                this.Background = NameToColor(value);
        }
        get { return backgroundName; }
    }

    public Color Background
    {
        set
        {
            if (SetProperty<Color>(ref background, value))

```

```

        this.BackgroundName = ColorToName(value);
    }
    get { return background; }
}

public string ForegroundName
{
    set
    {
        if (SetProperty<string>(ref foregroundName, value))
            this.Foreground = NameToColor(value);
    }
    get { return foregroundName; }
}

public Color Foreground
{
    set
    {
        if (SetProperty<Color>(ref foreground, value))
            this.ForegroundName = ColorToName(value);
    }
    get { return foreground; }
}

public DateTime DateTime
{
    set
    {
        if (SetProperty<DateTime>(ref dateTime, value))
        {
            this.FormattedDateTime = String.Format("{0:D} {1:t}", value, value);
            this.SecondAngle = 6 * (dateTime.Second + dateTime.Millisecond / 1000.0);
            this.MinuteAngle = 6 * dateTime.Minute + this.SecondAngle / 60;
            this.HourAngle = 30 * (dateTime.Hour % 12) + this.MinuteAngle / 12;
        }
    }
    get { return dateTime; }
}

public string FormattedDateTime
{
    set { SetProperty<string>(ref formattedDateTime, value); }
    get { return formattedDateTime; }
}

public double HourAngle
{
    set { SetProperty<double>(ref hourAngle, value); }
    get { return hourAngle; }
}

public double MinuteAngle
{

```

```

        set { SetProperty<double>(ref minuteAngle, value); }
        get { return minuteAngle; }
    }

    public double SecondAngle
    {
        set { SetProperty<double>(ref secondAngle, value); }
        get { return secondAngle; }
    }

    Color NameToColor(string name)
    {
        return (Color)colorsTypeInfo.GetDeclaredProperty(name).GetValue(null);
    }

    string ColorToName(Color color)
    {
        foreach (PropertyInfo property in colorsTypeInfo.DeclaredProperties)
            if (color.Equals((Color)property.GetValue(null)))
                return property.Name;

        return "";
    }

    protected bool SetProperty<T>(ref T storage, T value,
                                   [CallerMemberName] string propertyName = null)
    {
        if (object.Equals(storage, value))
            return false;

        storage = value;
        OnPropertyChanged(propertyName);
        return true;
    }

    protected void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}

```

This view model also includes a *DateTime* property, and whenever that changes, the *HourAngle*, *MinuteAngle*, and *SecondAngle* properties also change, driving the three *RotateTransform* objects in *TimeZoneClock.xaml*.

The *MainPage.xaml* file consists of little more than a *VariableSizedWrapGrid* to hold the clock controls:

Project: ClockRack | File: *MainPage.xaml* (excerpt)

```

<Page ... >
    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <Grid Name="contentGrid"

```

```

        Background="Transparent"
        SizeChanged="OnGridSizeChanged">

        <VariableSizedWrapGrid Name="wrapGrid"
            Orientation="Horizontal"
            HorizontalAlignment="Center"
            VerticalAlignment="Center" />

    </Grid>
</Grid>
</Page>

```

There's no *ScrollView* here. The code-behind file needs to ensure that the *TimeZoneClock* instances are sized appropriately so that they can all fit on one nonscrollable page regardless of how small that page might be.

The constructor of the *MainPage* class is responsible for populating the *VariableSizedWrapGrid* from application settings. The program uses the *ApplicationData* class in the *Windows.Storage* namespace for storing four text items per clock: the location name (selected by the user), time zone key, foreground color name, and background color name. For the first clock, these are stored using keys "0Location", "0TimeZoneKey", "0Foreground", and "0Background", and the second clock has keys that begin with the number 1, and so forth. As each set of settings is retrieved, a *TimeZoneClock* and *TimeZoneClockViewModel* are created and initialized:

Project: ClockRack | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    ...
    IPropertySet appSettings = ApplicationData.Current.LocalSettings.Values;

    public MainPage()
    {
        this.InitializeComponent();

        // Load application settings for clocks
        int index = 0;

        while (appSettings.ContainsKey(index.ToString() + "Location"))
        {
            string preface = index.ToString();

            TimeZoneClock clock = new TimeZoneClock
            {
                DataContext = new TimeZoneClockViewModel
                {
                    Location = appSettings[preface + "Location"] as string,
                    TimeZoneKey = appSettings[preface + "TimeZoneKey"] as string,
                    ForegroundName = appSettings[preface + "Foreground"] as string,
                    BackgroundName = appSettings[preface + "Background"] as string
                },
            };
            wrapGrid.Children.Add(clock);
            index += 1;
        }
    }
}

```

```

    }

    // If there are no settings, make a default Clock
    if (wrapGrid.Children.Count == 0)
    {
        TimeZoneClock clock = new TimeZoneClock
        {
            DataContext = new TimeZoneClockViewModel()
        };
        wrapGrid.Children.Add(clock);
    }

    // Start the Rendering event
    CompositionTarget.Rendering += OnCompositionTargetRendering;
}
...
}

```

The constructor concludes by starting up a *CompositionTarget.Rendering* event. This is responsible for using the *TimeZoneManager* instance to obtain a local time based on the current UTC time with the time zone key for each clock:

Project: ClockRack | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{
    TimeZoneManager timeZoneManager = new TimeZoneManager();
    ...
    void OnCompositionTargetRendering(object sender, object args)
    {
        // Get the time once
        DateTime utc = DateTime.UtcNow;

        foreach (UIElement child in wrapGrid.Children)
        {
            TimeZoneClockViewModel viewModel =
                (child as FrameworkElement).DataContext as TimeZoneClockViewModel;
            string timeZoneKey = viewModel.TimeZoneKey;

            // Set the local time from the TimeZoneManager
            viewModel.DateTime = timeZoneManager.GetLocalTime(timeZoneKey, utc);
        }
    }
    ...
}

```

The tricky job is to make sure all the individual clocks are small enough so that the *VariableSizeWrapGrid* can fit them all on one screen. This is a rather brute-force calculation in a *SizeChanged* handler. The loop tries sizing the clock to fit in one row, then two, and so forth, and determines which number of rows provides the largest size clocks:

Project: ClockRack | File: MainPage.xaml.cs (excerpt)

```

public sealed partial class MainPage : Page
{

```

```

...

void OnGridSizeChanged(object sender, SizeChangedEventArgs args)
{
    RecalculateClockSize();
}

void RecalculateClockSize()
{
    Size containerSize = new Size(contentGrid.ActualWidth, contentGrid.ActualHeight);

    if (containerSize.Width == 0 || containerSize.Height == 0)
        return;

    int numClocks = wrapGrid.Children.Count;
    int bestWidth = 0;
    int bestHeight = 0;

    // Enumerate though possible number of rows
    for (int numRows = 1; numRows <= numClocks; numRows++)
    {
        // Get clock dimensions
        int numCols = (int)Math.Ceiling((double)numClocks / numRows);
        int clockHeight = (int)(containerSize.Height / numRows);
        int clockWidth = (int)(containerSize.Width / numCols);

        // Adjust for aspect ratio
        if (clockWidth > (int)(TimeZoneClock.AspectRatio * clockHeight))
            clockWidth = (int)(TimeZoneClock.AspectRatio * clockHeight);
        else
            clockHeight = (int)(clockWidth / TimeZoneClock.AspectRatio);

        // Check if it's larger than other trials
        if (clockHeight > bestHeight)
        {
            bestHeight = clockHeight;
            bestWidth = clockWidth;
        }
    }
    // Set all the clock dimensions
    foreach (UIElement child in wrapGrid.Children)
    {
        (child as TimeZoneClock).Width = bestWidth;
        (child as TimeZoneClock).Height = bestHeight;
    }
}
...
}

```

I don't think this is the best solution to the problem of sizing and arranging the clocks, and the main problem is that *VariableSizedWrapGrid* simply isn't the right tool for the job. For example, suppose there are six clocks, and this algorithm determines that the clocks will be the maximum size for two rows of five clocks each. *VariableSizedWrapGrid* will then arrange five clocks in the first row, and one

clock in the second row.

This is something that cries out for a custom panel, I think.

A right tap displays the menu:

Project: ClockRack | File: MainPage.xaml.cs (excerpt)

```
public sealed partial class MainPage : Page
{
    ...
    async protected override void OnRightTapped(RightTappedRoutedEventArgs args)
    {
        // Check if the parent of the click element is a TimeZoneClock
        FrameworkElement element = args.OriginalSource as FrameworkElement;

        while (element != null)
        {
            if (element is TimeZoneClock)
                break;

            element = element.Parent as FrameworkElement;
        }

        // Create a PopupMenu
        PopupMenu popupMenu = new PopupMenu();
        popupMenu.Commands.Add(new UICommand("Add...", OnAddMenuItem, element));

        if (element is TimeZoneClock)
        {
            popupMenu.Commands.Add(new UICommand("Edit...", OnEditMenuItem, element));

            if (wrapGrid.Children.Count > 1)
                popupMenu.Commands.Add(new UICommand("Delete", OnDeleteMenuItem, element));
        }

        args.Handled = true;
        base.OnRightTapped(args);

        // Display the menu
        await popupMenu.ShowAsync(args.GetPosition(this));
    }

    void OnAddMenuItem(UICommand command)
    {
        TimeZoneClock timeZoneClock = new TimeZoneClock
        {
            DataContext = new TimeZoneClockViewModel()
        };
        TimeZoneClock insertBeforeClock = command.Id as TimeZoneClock;

        if (insertBeforeClock != null)
        {
            int index = wrapGrid.Children.IndexOf(insertBeforeClock);
            wrapGrid.Children.Insert(index, timeZoneClock);
        }
    }
}
```



```

    }
    else
    {
        wrapGrid.Children.Add(timeZoneClock);
    }

    SaveSettings();
    RecalculateClockSize();
}

void OnEditMenuItem(UICommand command)
{
    TimeZoneClock timeZoneClock = command.Id as TimeZoneClock;
    SettingsDialog settingsDialog = new SettingsDialog(timeZoneManager);
    settingsDialog.DataContext = timeZoneClock.DataContext;

    // Create Popup with SettingsDialog child
    Popup popup = new Popup
    {
        Child = settingsDialog,
        IsLightDismissEnabled = true
    };

    settingsDialog.SizeChanged += (sender, args) =>
    {
        // Get clock center
        Point position = new Point(timeZoneClock.ActualWidth / 2,
                                    timeZoneClock.ActualHeight / 2);

        // Convert to Page coordinates
        position = timeZoneClock.TransformToVisual(this).TransformPoint(position);

        // Position popup so lower-left or lower-right corner
        //     aligns with center of edited clock
        if (position.X > this.ActualWidth / 2)
            position.X -= settingsDialog.ActualWidth;

        position.Y -= settingsDialog.ActualHeight;

        // Adjust for size of page
        if (position.X + settingsDialog.ActualWidth > this.ActualWidth)
            position.X = this.ActualWidth - settingsDialog.ActualWidth;

        if (position.X < 0)
            position.X = 0;

        if (position.Y < 0)
            position.Y = 0;

        // Set the Popup position
        popup.HorizontalOffset = position.X;
        popup.VerticalOffset = position.Y;
    };
}

```

```

        popup.Closed += (sender, args) =>
        {
            SaveSettings();
        };
        popup.IsOpen = true;
    }

    async void OnDeleteMenuItem(UICommand command)
    {
        TimeZoneClock timeZoneClock = command.Id as TimeZoneClock;
        TimeZoneClockViewModel viewModel = timeZoneClock.DataContext as TimeZoneClockViewModel;

        MessageDialog msgdlg = new MessageDialog("Delete clock from collection?",
                                                viewModel.Location);
        msgdlg.Commands.Add(new UICommand("OK"));
        msgdlg.Commands.Add(new UICommand("Cancel"));
        msgdlg.DefaultCommandIndex = 0;
        msgdlg.CancelCommandIndex = 1;

        UICommand msgDlgCommand = await msgdlg.ShowAsync();

        if (msgDlgCommand.Label == "OK")
        {
            wrapGrid.Children.Remove(command.Id as TimeZoneClock);
            SaveSettings();
            RecalculateClockSize();
        }
    }

    void SaveSettings()
    {
        appSettings.Clear();

        for (int index = 0; index < wrapGrid.Children.Count; index++)
        {
            TimeZoneClock timeZoneClock = wrapGrid.Children[index] as TimeZoneClock;
            TimeZoneClockViewModel viewModel =
                timeZoneClock.DataContext as TimeZoneClockViewModel;
            string preface = index.ToString();

            appSettings[preface + "Location"] = viewModel.Location;
            appSettings[preface + "TimeZoneKey"] = viewModel.TimeZoneKey;
            appSettings[preface + "Foreground"] = viewModel.ForegroundName;
            appSettings[preface + "Background"] = viewModel.BackgroundName;
        }
    }
}

```

The settings dialog is defined in this XAML file:

Project: ClockRack | 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="0 16 0 8" />
    </Style>

    <DataTemplate x:Key="colorItemTemplate">
        <!-- Item is SettingsDialog.ColorItem -->
        <StackPanel Orientation="Horizontal">
            <Rectangle Width="96" Height="24" Margin="12 6">
                <Rectangle.Fill>
                    <SolidColorBrush Color="{Binding ColorValue}" />
                </Rectangle.Fill>
            </Rectangle>

            <TextBlock Text="{Binding ColorName}"
                VerticalAlignment="Center" />
        </StackPanel>
    </DataTemplate>
</UserControl.Resources>

<!-- DataContext is TimeZoneClockViewModel -->
<Border Background="{StaticResource ApplicationPageBackgroundThemeBrush}"
    BorderBrush="{StaticResource ApplicationForegroundThemeBrush}"
    BorderThickness="1"
    Padding="7 0 0 0"
    Width="384">
    <StackPanel Margin="24">
        <TextBlock Text="ClockRack settings for"
            Style="{StaticResource SubheaderTextStyle}"
            TextAlignment="Center" />

        <TextBlock Text="{Binding Location}"
            Style="{StaticResource SubheaderTextStyle}"
            TextAlignment="Center"
            Margin="0 0 0 12" />

        <!-- Location -->
        <TextBlock Text="Location"
            Style="{StaticResource DialogCaptionTextStyle}" />

        <TextBox Name="locationTextBox"
            Text="{Binding Location}"
            TextChanged="OnLocationTextBoxTextChanged" />

        <!-- Time Zone -->
        <TextBlock Text="Time Zone"
            Style="{StaticResource DialogCaptionTextStyle}" />

        <ComboBox Name="timeZoneComboBox"
            SelectedValuePath="TimeZoneKey"
            SelectedValue="{Binding TimeZoneKey, Mode=TwoWay}">
            <ComboBox.ItemTemplate>

```

```

        <!-- Data is TimeZoneDisplayInfo -->
        <DataTemplate>
            <TextBlock Text="{Binding Display}" />
        </DataTemplate>
    </ComboBox.ItemTemplate>
</ComboBox>

<!-- Foreground and Background Colors -->
<TextBlock Text="Foreground Color"
            Style="{StaticResource DialogCaptionTextStyle}" />

<ComboBox Name="foregroundComboBox"
            ItemTemplate="{StaticResource colorItemTemplate}"
            SelectedValuePath="ColorName"
            SelectedValue="{Binding ForegroundName, Mode=TwoWay}" />

<TextBlock Text="Background Color"
            Style="{StaticResource DialogCaptionTextStyle}" />

<ComboBox Name="backgroundComboBox"
            ItemTemplate="{StaticResource colorItemTemplate}"
            SelectedValuePath="ColorName"
            SelectedValue="{Binding BackgroundName, Mode=TwoWay}" />
</StackPanel>
</Border>
</UserControl>

```

Here's the code-behind file:

Project: ClockRack | File: SettingsDialog.xaml.cs (excerpt)

```

public sealed partial class SettingsDialog : UserControl
{
    public class ColorItem
    {
        public Color ColorValue { set; get; }
        public string ColorName { set; get; }
    }

    public SettingsDialog(TimeZoneManager timeZoneManager)
    {
        this.InitializeComponent();

        // Set ItemsSource for time zone ComboBox
        timeZoneComboBox.ItemsSource = timeZoneManager.DisplayInformation;

        // Set Items for Foreground and Background ComboBoxes
        foreach (PropertyInfo property in typeof(Colors).GetTypeInfo().DeclaredProperties)
        {
            if (property.Name != "Transparent")
            {
                ColorItem clrItem = new ColorItem
                {
                    ColorValue = (Color)property.GetValue(null),
                    ColorName = property.Name
                }
            }
        }
    }
}

```

```

        };
        foregroundComboBox.Items.Add(cclrItem);
        backgroundComboBox.Items.Add(cclrItem);
    }
}

void OnLocationTextBoxTextChanged(object sender, TextChangedEventArgs args)
{
    (this.DataContext as TimeZoneClockViewModel).Location = (sender as TextBox).Text;
}
}

```

A Windows Runtime Component Wrapper for DirectX

The remainder of this chapter will discuss accessing DirectX from a C# program using a Windows Runtime Component written in C++. A mention will be made that I'm not the only person to do this, and that other solutions are much more extensive and perhaps the reader will want to use something like SharpDX for this job.

The source code for this chapter includes a Windows Runtime Component written in C++ named DirectXWrapper, which is accessed by two C# projects named EnumerateFonts and SpinPaint.

A Windows Runtime Component resembles a DLL except that it must be accessible from all languages supported for Windows 8 programming. For that reason, everything passing across the API must be a primitive type or a Windows Runtime type. Public classes must be sealed or noninstantiable. Structures must have no public members except for fields.

The DirectWrite support of DirectXWrapper is focused on two basic needs: getting a list of fonts installed on the system, and getting font metrics for a particular font. To allow this, I've provided fairly light wrappers on the necessary DirectWrite classes and interfaces. The EnumerateFonts project (which will be improved in the final release of this book) lists all the available fonts and lets you pick one. It then uses font metrics information to display lines indicating the base line and other information of the selected font.

The *SurfaceImageSourceRenderer* class in DirectXWrapper takes a rather different architectural approach. This class is intended mostly to let you draw lines on an object of type *SurfaceImageSource*.

SurfaceImageSource derives from *ImageSource*, and hence it can be set to the *Source* property of Image or the *ImageSource* property of *ImageBrush*. However, you can display something on this object only by using DirectX. The *SurfaceImageSourceRenderer* class performs all the necessary overhead and exposes three public methods: *Clear*, *DrawLine*, and *Update*.

The *SurfaceImageSourceRenderer* is demoed in the SpinPaint project. You can paint on the disk but I've disabled the "spin" function due to inexplicable crashes.



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