Adam Nathan

Full Color

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

Windows® 8.1 Apps

with XAML and C#

UNLEASHED

FREE SAMPLE CHAPTER

Adam Nathan

Windows® 8.1 Apps

with XAML and C#

UNLEASHED

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

Windows® 8.1 Apps with XAML and C# Unleashed

ISBN-13: 978-0-672-33708-6
ISBN-10: 0-672-33708-8

Library of Congress Control Number: 2013951289

Printed in the United States on America

First Printing December 2013

EDITOR-IN-CHIEF
Greg Wiegand

EXECUTIVE EDITOR
Neil Rowe

DEVELOPMENT EDITOR
Mark Renfrow

MANAGING EDITOR
Kristy Hart

SENIOR PROJECT EDITOR
Betsy Gratner

INDEXER
Tim Wright

PROOFREADER
Kathy Ruiz

TECHNICAL EDITOR
Ashish Shetty

PUBLISHING COORDINATOR
Cindy Teeters

COVER DESIGNER
Mark Shirar

COMPOSITION
Nonie Ratcliff

Contents at a Glance

Table of Contents

About the Author

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

Adam's books are considered required reading by many inside Microsoft and throughout the industry. Adam is the author of *Windows 8 Apps with XAML and C# Unleashed* (Sams, 2013), *101 Windows Phone 7 Apps* (Sams, 2011), *Silverlight 1.0 Unleashed* (Sams, 2008), *WPF Unleashed* (Sams, 2006), *WPF 4 Unleashed* (Sams, 2010), *WPF 4.5 Unleashed* (Sams, 2013), and *.NET and COM: The Complete Interoperability Guide* (Sams, 2002); a coauthor of *ASP.NET: Tips, Tutorials, and Code* (Sams, 2001); and a contributor to books including *.NET Framework Standard Library Annotated Reference, Volume 2* (Addison-Wesley, 2005) and *Windows Developer Power Tools* (O'Reilly, 2006). You can find Adam online at www.adamnathan.net or @adamnathan on Twitter.

This page intentionally left blank

Dedication

To Mom and Dad.

Acknowledgments

First, I thank Eileen Chan for the encouragement and patience that enabled me to complete this book. I'd also like to give special thanks to Ashish Shetty, Tim Heuer, Mark Rideout, Jonathan Russ, Joe Duffy, Chris Brumme, Eric Rudder, Neil Rowe, Betsy Gratner, Ginny Munroe, Bill Chiles, and Valery Sarkisov. As always, I thank my parents for having the foresight to introduce me to Basic programming on our IBM PCjr when I was in elementary school.

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

We Want to Hear from You!

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

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

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

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

E-mail: consumer@samspublishing.com

Mail: Sams Publishing
ATTN: Reader Feedback
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

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

Introduction

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

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

The Windows Store is an app store like no other, and it keeps getting better. When you consider the number of people who use Windows 8.1 (and Windows RT) compared to the number of people who use any other operating system on the planet, you realize what a unique and enormous opportunity the Windows Store provides. That's one of the reasons that the Windows Store is the fastest-growing app store in history.

When you write a Windows Store app, you have three main choices for programming language and UI framework pairings:

- JavaScript with an HTML user interface
- C#, Visual Basic, or C++ with a XAML user interface
- C++ with a DirectX user interface

You can also leverage a number of features and componentization techniques to mix and match these languages and UI frameworks within the same app.

C# and XAML has been a very popular choice for writing Windows Store apps. It is the choice for apps such as Netflix, Hulu Plus, Fresh Paint, SkyDrive, Evernote Touch, Reader, Alarms, Movie Moments, Maps, OneNote, Lync, and many, many more. It is also the implementation choice for many core experiences in Windows, such as the PC Settings app, the Search app, and new Contact/Calendar functionality in Windows 8.1. The XAML team has stated that their goal is to be the high fidelity, high performance framework for *any* scenario.

Then why does Microsoft provide so many choices? The idea is to enable you to work with whatever is most comfortable for you, whatever best leverages your existing assets, or whatever most naturally consumes the third-party SDK you must use.

Your choice can have other benefits. HTML tends to be the best choice if you need to support your versions of your app on non-Microsoft platforms or a website. XAML is best at interoperability, as it's easy to mix both HTML and DirectX content in a XAML app. DirectX, the best choice for hardcore games, provides the most potential for getting the highest performance.

Common perceptions of performance differences between the UI frameworks are often wrong, however. It's important to realize that no matter which of the three UI frameworks you use, about 80% of their core implementation is identical, the Windows APIs are the same, and the graphics are hardware accelerated. Although DirectX offers the most *potential* for getting the highest performance, you have to do a lot of work to realize that potential! Often, a C#/XAML implementation can outperform a simple C++/DirectX implementation due to the impressive optimizations that the XAML UI Framework does on your behalf. Not only that, but the XAML UI Framework gives you a number of additional features automatically, such as accessibility and localization.

Although your choice of language is generally dictated by your choice of UI Framework, each language has its strengths. JavaScript benefits from a large community that produces interesting libraries. C# has the best features for writing concise asynchronous code, and doesn't have the same multithreading limitations that plague JavaScript. C++ provides the most potential for getting the highest performance. (Does that line sound familiar?) Of course, you have to earn that performance, and you have to be especially careful with how you mix standard C and C++ code with the C++/CX code that is needed to communicate with Windows.

The key to the multiple language support is the Windows Runtime, or WinRT for short. You can think of it like .NET's Common Language Runtime, except it spans both managed and unmanaged languages. To enable this, WinRT is COM-based. Most of the time, you can't tell when you interact with WinRT. And most of the time, it doesn't matter. This is a modern, friendlier version of COM that is more amenable to *automatic* correct usage from environments such as .NET or JavaScript. (Contrast this to over a decade ago, when I wrote a book about mixing COM with .NET. This topic alone required over 1,600 pages!)

WinRT APIs are automatically *projected* into the programming language you use, so they look natural for that language. Projections are more than just exposing the raw APIs, however. Core WinRT data types such as `String`, collection types, and a few others are mapped to appropriate data types for the target environment. For C# or other .NET languages, this means exposing them as `System.String`, `System.Collections.Generic.IList<T>`, and so on. To match conventions, member names are even morphed to be Camel-cased for JavaScript and Pascal-cased for other languages, which makes the MSDN reference documentation occasionally look goofy.

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

In the set of APIs exposed by Windows:

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

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

I wrote this book with the following goals in mind:

- To provide a solid grounding in the underlying concepts, in a practical and approachable fashion
- To answer the questions most people have when learning how to write Windows Store apps and to show how commonly desired tasks are accomplished
- To be an authoritative source, thanks to input from members of the team who designed, implemented, and tested Windows 8.1 and Visual Studio 2013
- To be clear about where the technology falls short rather than blindly singing its praises
- To optimize for concise, easy-to-understand code rather than enforcing architectural patterns that can be impractical or increase the number of concepts to understand
- To be an easily navigated reference that you can constantly come back to

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

Whether you're new to XAML or a long-time XAML developer, I hope you find this book to exhibit all these attributes.

Who Should Read This Book?

This book is for software developers who are interested in creating apps for the Windows Store, whether they are for tablets, laptops, or desktops. It does not teach you how to program, nor does it teach the basics of the C# language. However, it is designed to be understandable even for folks who are new to .NET, and does not require previous experience with XAML.

If you are already well versed in XAML, I'm confident that this book still has a lot of helpful information for you. And if you are already familiar with writing Windows Store apps for Windows 8 (perhaps thanks to the first edition of this book), you will still benefit from the significant amount of new content that covers new features in Windows 8.1. It also covers features that were already present in Windows 8 in more depth than ever before. At the very least, this book should be an invaluable reference for your bookshelf.

Software Requirements

This book targets Windows 8.1, Windows RT, and the corresponding developer tools. The tools are a free download at the Windows Dev Center: http://dev.windows.com. The download includes the Windows 8.1 SDK, a version of Visual Studio Express specifically for Windows Store apps, and miscellaneous tools. It's worth noting that although this book almost exclusively refers to Windows 8.1, the content applies to Windows RT as well.

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

Code Examples

Source code for examples in this book can be downloaded from www.informit.com/title/9780672337086.

How This Book Is Organized

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

Part I: Getting Started

This part includes the following chapters:

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

Part I provides the foundation for the rest of the book. If you have previously created Windows Phone apps or worked with XAML in the context of other Microsoft technologies, a lot of this should be familiar to you. There are still several unique aspects for Windows 8.1 and the Windows Store, however. Chapter 1 helps you understand all the tools available at your disposal, and even dives into topics such as accessibility and localization, so you can be prepared to get the broadest set of customers possible for your app. This last set of topics is new to this edition of the book.

Part II: Building an App

This part includes the following chapters:

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

Part II equips you with the knowledge of how to place things on the screen, how to make them adjust to the wide variety of screen types, and how to interact with the user. Windows 8.1 introduces a new model for how apps should resize, and this is covered in Chapter 4. In Chapter 6, this edition contains new coverage on supporting pens, including rendering strokes and performing handwriting recognition.

Part III: Working with the App Model

This part includes the following chapters:

- ➔ Chapter 7: "App Lifecycle"
- ➔ Chapter 8: "Threading, Windows, and Pages "
- ➔ Chapter 9: "The Many Ways to Earn Money"

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

expands the coverage on trials and in-app purchases, and covers the new Windows 8.1 in-app purchase features. It also contains new coverage on integrating ads into your apps, the threading model for Windows Store apps, and new support for having multiple windows.

Part IV: Understanding Controls

This part includes the following chapters:

- ➔ Chapter 10: "Content Controls"
- ➔ Chapter 11: "Items Controls"
- ➔ Chapter 12: "Text"
- ➔ Chapter 13: "Images"
- ➔ Chapter 14: "Audio, Video, and Speech"
- ➔ Chapter 15: "Other Controls"

Part IV provides a tour of the controls built into the XAML UI Framework. There are many controls that you expect to have available, plus several that you might not expect. Windows 8.1 adds many new controls and many features to existing controls. Windows 8.1 also introduces speech synthesis features, which are covered in Chapter 14.

Part V: Leveraging the Richness of XAML

This part includes the following chapters:

- ➔ Chapter 16: "Vector Graphics"
- ➔ Chapter 17: "Animation"
- ➔ Chapter 18: "Styles, Templates, and Visual States"
- ➔ Chapter 19: "Data Binding"

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

Part VI: Exploiting Windows 8.1

This part includes the following chapters:

- ➔ Chapter 20: "Working with Data"
- ➔ Chapter 21: "Supporting Charms"
- ➔ Chapter 22: "Leveraging Contracts"

- Chapter 23: "Reading from Sensors"
- Chapter 24: "Controlling Devices"
- Chapter 25: "Thinking Outside the App: Live Tiles, Notifications, and the Lock Screen"

This part of the book could just as easily appear in a book about JavaScript or C++ Windows Store apps, with the exception of its code snippets. It covers unique and powerful Windows features that are not specific to XAML or C#, but they are things that all Windows Store app developers should know. The most notable new support in Windows 8.1 is covered in Chapter 24: supporting custom devices.

Part VII: Advanced Features

This part includes the following chapters:

- Chapter 26: "Integrating DirectX"
- Chapter 27: "Custom Controls and Components"
- Chapter 28: "Layout with Custom Panels"

The advanced features covered in the last part of the book highlight very different scenarios. Integrating DirectX into your XAML app enables you to do things that aren't possible otherwise, whereas the last two chapters are about ways to reuse your code. The coverage of all these features is new to this edition. These features all existed in Windows 8, although the DirectX integration support has been improved for Windows 8.1.

Conventions Used in This Book

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

Typeface	Meaning
Italic	Italic is used for new terms or phrases when they are initially defined and occasionally for emphasis.
`Monospace`	`Monospace` is used for screen messages, code listings, and filenames. In code listings, *`italic monospace type`* is used for placeholder text.
	Code listings are colorized similarly to the way they are colorized in Visual Studio. `Blue monospace type` is used for XML elements and C# keywords, `brown monospace type` is used for XML element names and C# strings, `green monospace type` is used for comments, `red monospace type` is used for XML attributes, and `teal monospace type` is used for type names in C#.
Bold	When appropriate, bold is used for code directly related to the main lesson(s) in a chapter.

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

What is a FAQ sidebar?

A Frequently Asked Question (FAQ) sidebar presents a question you might have about the subject matter—and then provides a concise answer.

Digging Deeper

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

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

This is a warning!

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

Chapter 1

In This Chapter

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

HELLO, *REAL* WORLD!

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

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

Creating, Deploying, and Profiling an App

In Visual Studio, let's create a new Visual C# **Blank App (XAML)** project called `HelloRealWorld`. This gives us a project that's ready to compile and run. Although pressing F5 or clicking the **Start Debugging** button in Visual Studio launches the app locally, you've got three slick options to choose from via the button's dropdown menu, shown in Figure 1.1 under Visual Studio's light theme (used throughout this book).

With the **Remote Machine** option, you can deploy and debug to any other Windows 8.x computer reachable on your network (although not over the Internet). This is extremely handy for testing things on a Surface or other tablets. The target device must have the Remote Tools for Visual Studio installed and running, which you can download from the Windows Dev Center.

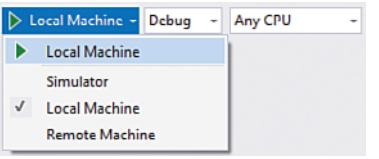


FIGURE 1.1 The three ways to launch your app in Visual Studio

The **Simulator** option is the next best thing to having a real tablet, as it provides mechanisms to simulate touch input, device orientations, network conditions, location services, and more. The simulator is shown in Figure 1.2. In fact, it has one huge advantage over testing on a physical device: It enables you to experience your app in a number of different resolutions and virtual screen sizes, including different aspect ratios. Given the wide variety of shapes and sizes of screens out there that run Windows Store apps, testing your app in this fashion is a must.

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

The simulator is your actual computer!

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

How do I run my app outside of Visual Studio?

Although compiling your app produces an `.exe` file in the `bin\Debug` or `bin\Release` subfolder, you can't simply double-click it from the Windows desktop to run it. If you try, you get an error that explains, "This application can only run in the context of an app container." (An "app container" refers to the sandbox in which all Windows Store apps run.)

Instead, you can launch it from the searchable list of apps underneath the tiles on the Start screen. Visual Studio automatically installs your app the first time you launch it. Like all Windows Store apps in Windows 8.1, however, its tile does not automatically get pinned. Because the Start screen has been enhanced to make it easier to find apps, pinning is now meant to be done selectively by a user, the same as with pinning apps to the desktop taskbar.

When you run the `HelloRealWorld` project without any changes, you'll see why the project type was called "Blank App." The app doesn't actually do anything other than fill the screen with darkness. (If you launch the app in debug mode, you'll also see four numbers on the top edge of the screen. These are frame rate counters described in Chapter 17, "Animation.") It does, however, set up a lot of infrastructure that would be difficult and tedious to create from scratch. The project contains the following items:

- The package manifest, a temporary certificate used to sign it, and some images
- The main page (`MainPage.xaml` and `MainPage.xaml.cs`)
- The application definition: `App.xaml`, `App.xaml.cs`, and `AssemblyInfo.cs`

The next section examines the package manifest and the images used by it. After that, we'll look at the XAML and C# files and make some code changes.

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

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

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

Understanding the App Package

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

- Application
- Visual Assets
- Capabilities
- Declarations
- Content URIs
- Packaging

For our `HelloRealWorld` app, we don't need to change anything in the package manifest. But now is a good time to understand what can be done on each of these tabs.

Application

On the Application tab, you can set the app's name and description, default language, its minimum width, and notification settings (if your app supports them). Notifications are covered in Chapter 25, "Thinking Outside the App: Live Tiles, Notifications, and the Lock Screen." You can even restrict the preferred orientations of your app if you'd rather not have it automatically rotate to all four of them:

- **Landscape** (horizontal)
- **Landscape-flipped** (horizontal but upside down)
- **Portrait** (vertical, with the hardware Start button on the left)
- **Portrait-flipped** (vertical, with the hardware Start button on the right)

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

Visual Assets

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

Customizing the Splash Screen

To ensure that every app's splash screen can be displayed practically instantaneously (before your app even gets loaded), you have little control over it. You specify a 620x300 image (plus two optional larger sizes to support high DPI screens), and a background color for the splash screen. That's it. Visual Studio gives you an appropriately sized placeholder `SplashScreen.scale-100.png` file in an Assets subfolder, intentionally made ugly to practically guarantee you won't forget to change it before submitting your app to the Windows Store.

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

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

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

Customizing Logo Images

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

- **Square 70x70 Logo**—This is used for the **small** version of your app's tile on the Start screen. Although assigning an image here is optional, the small tile size is not. If you don't provide an image, the medium tile image is used (and scaled down) when a user changes your tile size to small.
- **Square 150x150 Logo**—This is used for the **medium** version of your app's tile on the Start screen. The medium tile size is the one required size, so at least a 100% scale image is required.
- **Wide 310x150 Logo**—This is used for the **wide** version of your app's tile on the Start screen, if you choose to support that tile size. If you assign at least a 100% scale image here, your app automatically supports the wide tile size. Otherwise, it doesn't.

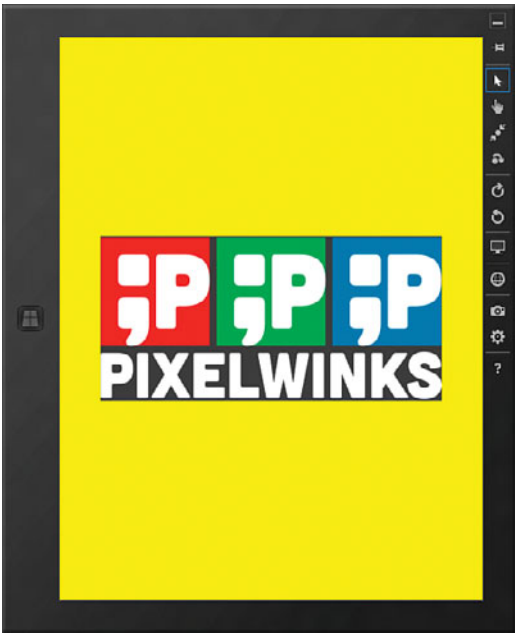


FIGURE 1.4 A live splash screen shown inside the simulator with a garish yellow background to clearly show the bounds of the image

- **Large 310x310 Logo**—This is used for the **large** version of your app's tile on the Start screen, if you choose to support that tile size. If you assign at least a 100% scale image here *and* for the wide logo, your app automatically supports the large tile size. (Your app can only support a large tile if it also supports a wide tile.) Otherwise, it doesn't.
- **Square 30x30 Logo**—This is used throughout Windows, including on the desktop. It is used by the apps list, search results, the Share pane, the file picker, an overlay on live tiles, the Alt+Tab user interface, Task Manager, file icons for associated file types, and so on. At least the 100% scale image is required. Although the image is nominally 30x30 pixels, this logo supports four additional sizes to be used for file icons on the desktop (if your app has associated file types): 16x16, 32x32, 48x48, and 256x256.
- **Store Logo**—A 50x50 image (at 100% scale) used by the Windows Store. At least the 100% scale image is required.

Visual Studio provides placeholder image files for the required logo images only: the square 150x150 logo, the square 30x30 logo, and the store logo.

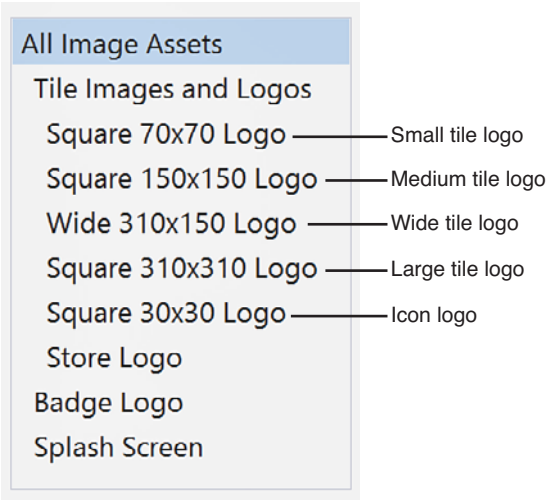


FIGURE 1.5 More understandable names for the different logo images you can provide

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

Furthermore, it's best *not* to support a large tile and/or wide tile unless you're going to make it a live tile (covered in Chapter 25). Otherwise, your pinned app occupies more space without adding any extra value.

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

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

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

- → `Assets\Logo.scale-80.png` (for 80% scale)
- → `Assets\Logo.scale-100.png` (for 100% scale)
- → `Assets\Logo.scale-140.png` (for 140% scale)
- → `Assets\Logo.scale-180.png` (for 180% scale)

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

The store logo and splash screen images don't support the 80% scale size because they are never shown on a tile on the Start screen. The additional four sizes of the square 30x30 logo, assigned to `Assets\SmallLogo.png` by default, use a similar naming scheme:

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

You can use a similar technique for providing different files for high contrast mode, different cultures, and more. This applies not just for the images here, but for images used inside your app as well. See Chapter 13, "Images," for more details.

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

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

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

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

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

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

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

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

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

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

Capabilities

On the Capabilities tab, you select each capability required by your app. A *capability* is a special permission for actions that users might not want certain apps to perform, whether for privacy concerns or concerns about data usage charges. In the Windows Store, prospective users are told what capabilities each app requires before they decide whether to download it. To users, they are described as *permissions*, sometimes with more descriptive names, as shown in Figure 1.6.

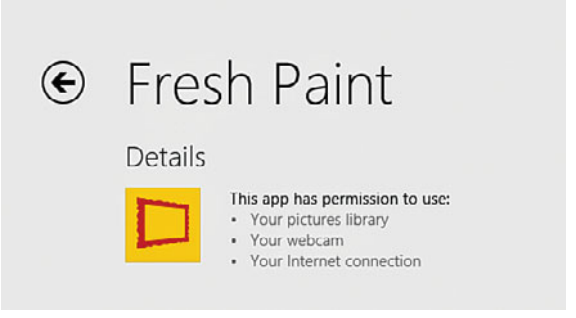
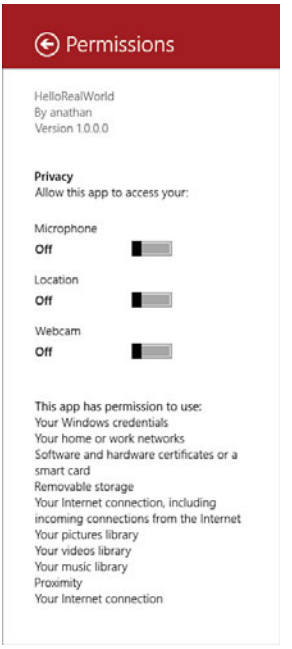


FIGURE 1.6 The Fresh Paint app uses three capabilities: Pictures Library, Webcam, and Internet (Client).

For the most part, user approval of all requested permissions is an implicit part of downloading an app. However, the use of privacy-related capabilities, such as location services, prompts the user the first time an app invokes a relevant API. Furthermore, some capabilities can be disabled or reenabled at any time by a user. When the Settings charm is invoked while a Windows Store app is running, it contains a "Permissions" link that displays an app's capabilities and toggle switches for any that can be turned on and off. Figure 1.7 shows what this looks like while running `HelloRealWorld`, both with the default capability already chosen in our package manifest—Internet (Client)—and after selecting every listed capability in the package manifest.

When the app uses the Internet (Client) capability



When the app uses every listed capability

FIGURE 1.7 The "Permissions" section of the Settings charm lists the current app's capabilities, and enables turning some of them on or off at runtime.

The long list of available capabilities can be grouped into four different categories:

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

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

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

File Capabilities

As you'll read in Chapter 20, "Working with Data," apps can read and write their own private files in an isolated spot, and those files can even participate in automatic roaming between a user's devices. In addition, users can give apps explicit permission to read/write other "normal" files and folders via the Windows file picker. This is all that most apps need, and does not require any capabilities.

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

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

Device Capabilities

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

- **Location**—Reveals the computer's location, either precise coordinates from a GPS sensor (if one exists) or an estimation based on network information.

- **Microphone**—Enables recording audio from a microphone.
- **Webcam**—Enables recording video—or capturing still pictures—from a camera. Note that this doesn't include sound. If you want to record audio and video, you need both Webcam and Microphone capabilities.
- **Proximity**—Enables communication with nearby devices, either via Wi-Fi Direct or near field communication (NFC).

Chapters 14, "Audio, Video, and Speech," and 23, "Reading from Sensors," explain how to write apps that take advantage of these capabilities. Additional device capabilities exist that don't appear on the Capabilities tab. These must be added manually to the package manifest XML. See Chapter 24, "Controlling Devices," for more information.

Network Capabilities

Without any network capabilities, a Windows Store app cannot do any communication over any kind of network except for the automatic roaming of application data described in Chapter 20, the seamless opening/saving of network files enabled by the file picker, or the peer-to-peer connections enabled by the Proximity capability. Four types of network capabilities exist:

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

Identity Capabilities

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

Visual Studio project templates enable the "Internet (Client)" capability by default!

This is done because the Visual Studio team feared that it would be too confusing for developers if simple network-dependent calls failed in their brand new projects. Therefore, be sure to remove the capability if you don't need it. Otherwise, your app's store listing will say that your app "has permission to use your Internet connection."

Declarations

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

Your app can be the source for a contract without doing anything in the package manifest. (It just makes various API calls.) To be the target, however, your app must be activated in a special manner. This is what requires the declaration in the package manifest. Therefore, you can think of the list of available *declarations* as the list of available *contract targets*.

Unlike capabilities, contract target declarations are *not* listed in the Windows Store as potentially unwanted features. In fact, you should go out of your way to mention your supported contract scenarios, because they can be very useful! There's nothing about being a contract target that is inherently dangerous for the user. Supporting certain contracts does require relevant capabilities, but many don't require any. See Chapter 22, "Leveraging Contracts," for specific examples.

Content URIs

This tab, new to Windows 8.1, only applies if you are hosting HTML content inside your XAML app. It simply houses a list of HTTPS URLs whose JavaScript is allowed (or disallowed) to raise events that can be handled by your app. For more information, see the discussion of the `WebView` control in Chapter 15, "Other Controls."

Packaging

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

- The **package name** is a unique identifier. Visual Studio automatically fills it in with a globally-unique identifier known as a GUID. That said, for easier debugging and identification of your app's local data store, it's best to replace the GUID with a human-readable name, such as *CompanyName.AppName*. This name doesn't impact real users of your app, as the Windows Store assigns this value in the package that users download.
- The **package display name** is the name of your app in the store, but this also gets replaced when you follow the procedure to upload an app, so you can leave this item alone.
- The **version**, set to 1.0.0.0 by default, is a four-part value interpreted as *Major.Minor.Build.Revision*. You can set this value however you like. There are only two requirements enforced by the Windows Store:

1. Each new published version has a higher version number than previous published versions (for the same target version of Windows).
2. If your app simultaneously has a package for Windows 8 and a package for Windows 8.1, the Windows 8 package version number must never exceed the version number of your first published Windows 8.1 package.

➔ The bottom of this tab contains publisher information based on the certificate used to authenticate the package. Visual Studio configures this to work with the temporary certificate it generates, and the store upload process reconfigures it to work with your developer account.

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

Updating XAML and C# Code

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

The Main Page User Interface

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

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

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

At a quick glance, this file tells us:

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

Listing 1.2 updates the blank-screen `MainPage.xaml` with a few elements to produce the result in Figure 1.8.

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

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

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

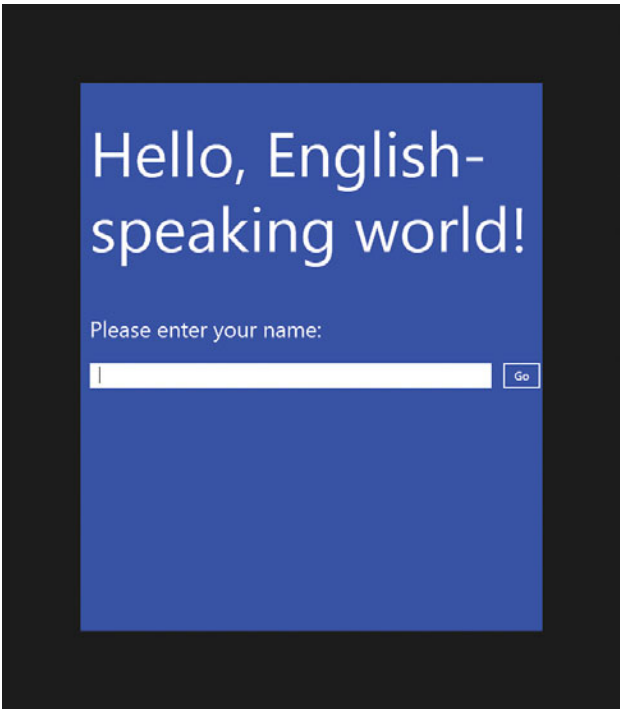


FIGURE 1.8 The `HelloRealWorld` user interface asks the user to type his or her name.

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

The Main Page Logic

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

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

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

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

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

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

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

The named elements in the XAML correspond to fields in this class, so this code updates the `result` `TextBlock` with the

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

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

text from the `nameBox TextBox`. Figure 1.9 shows what this looks like, after the user types "Adam" then clicks the `Button`.

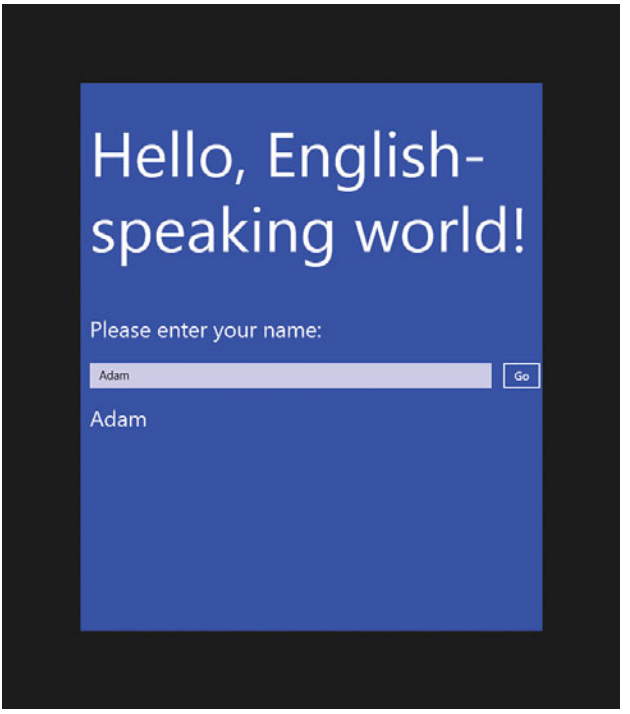


FIGURE 1.9 The `result TextBlock` contains the typed text after the user clicks the `Button`.

The Application Definition

The application definition is contained in `App.xaml` and its code-behind file, `App.xaml.cs`. `App.xaml` is a special XAML file that doesn't define any visuals, but rather defines an `App` class that can handle application-level tasks. Usually the only reason to touch this XAML file is to place new application-wide resources, such as custom styles, inside its `Application.Resources` collection. Chapter 18, "Styles, Templates, and Visual States" contains many examples of this. Listing 1.4 shows the contents of `App.xaml` in our `HelloRealWorld` project.

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

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

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

- A constructor, which is effectively the app's `main` method. The plumbing that makes it the app's entry point is enabled by an "Entry point" setting in the package manifest (on the Application tab). When you create a project, Visual Studio automatically sets it to the namespace-qualified name of the project's `App` class (`HelloRealWorld.App` in this example).
- Logic inside an `OnLaunched` method that enables the frame rate counter overlay in debug mode, navigates to the app's first (and in this case only) page, and calls `Window.Current.Activate` to dismiss the splash screen. If you want to add a new page and make it be the starting point of the app, or if you want to customize the initialization logic, this is where you can do it. See Chapter 7, "App Lifecycle," for more information.
- An `OnSuspending` method that is attached to the base class's `Suspending` event. This gives you an opportunity to save state before your app is suspended, although the generated code does nothing here other than provide a TODO comment. Chapter 7 examines app suspension.

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

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

namespace HelloRealWorld
{
  /// <summary>
  /// Provides application-specific behavior to supplement the base class.
  /// </summary>
  sealed partial class App : Application
```

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

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

      Frame rootFrame = Window.Current.Content as Frame;

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

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

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

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

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

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

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

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

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

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

Making the App World-Ready

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

Globalization refers to making your app act appropriately for different markets without any changes or customizations. An example of this is formatting the display of currency correctly for the current region without writing special-case logic. The `Windows.Globalization` namespace contains a lot of functionality for handling dates and times, geographic regions, number formatting, and more. Plus, built-in XAML controls such as `DatePicker` and `TimePicker`, discussed in Chapter 15, are globalization-ready. For many apps, these features might not apply.

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

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

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

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

```
            <ColumnDefinition/>
            <ColumnDefinition Width="Auto"/>
          </Grid.ColumnDefinitions>
          <TextBox Name="nameBox" Margin="12"/>
          <Button x:Uid="GoButton" Grid.Column="1" Click="Button_Click"/>
        </Grid>
        <TextBlock Name="result" FontSize="28" Margin="12"/>
      </StackPanel>
    </Grid>
</Page>
```

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

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

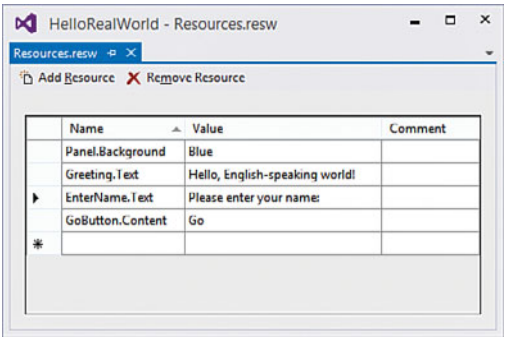


FIGURE 1.10 The `Resources.resw` file in the en folder is populated with English-specific values.

Right-click on the **en** folder and select **Add**, **New Item**, then pick **Resources file** from the **General** tab. The default name of `Resources.resw` is fine. This file is a table for all your language-specific strings. Figure 1.10 shows this file populated for English.

Each value must be given a name of the form *`UniqueId.PropertyName`*. *`UniqueId`* must match the `x:Uid` value for the relevant element, so the `Panel.Background` entry in Figure 1.10 sets `Background` to `Blue` on the `StackPanel` marked with `x:Uid="Panel"` in Listing 1.6. From the listing, it's not obvious that `GoButton`'s relevant property is called `Content`, unlike the `TextBlocks`' property called `Text`, but as you learn about the different elements throughout this book, you'll understand which properties to set.

After filling out the `Resources.resw` file, you can run the `HelloRealWorld` app and the result is identical to what we saw earlier in Figures 1.6 and 1.7. However, the app is now ready to be localized for other languages.

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

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

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

We'll leverage the Pseudo Language in a moment, but first let's add support for a second *real* language: Traditional Chinese. To do this, right-click on your project in Solution Explorer and select **Add translation languages...**. This produces the dialog shown in Figure 1.11.

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

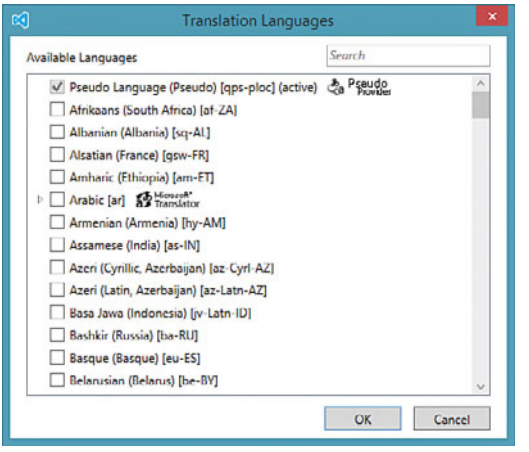


FIGURE 1.11 The Multilingual App Toolkit automates the process for supporting new languages.

What is Pseudo Language?

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

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

What are `.xlf` files?

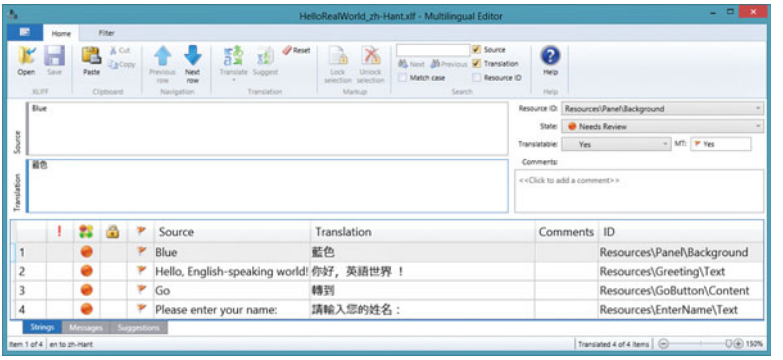
These files, which are generated by the Multilingual App Toolkit, are XLIFF files, an industry-standard XML format for localizable data. In addition to listing source and target strings (with optional comments), these files enable a workflow in which resources can be marked as New, Needs Review, Translated,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

The benefit of using XLIFF files to store translations is that you can send them directly to a professional translation vendor, as they should already have a workflow involving this format. Or, if you leverage friends to do your translations, you can have them install the Multilingual App Toolkit and use its Multilingual Editor in a standalone fashion. No Visual Studio installation is necessary.

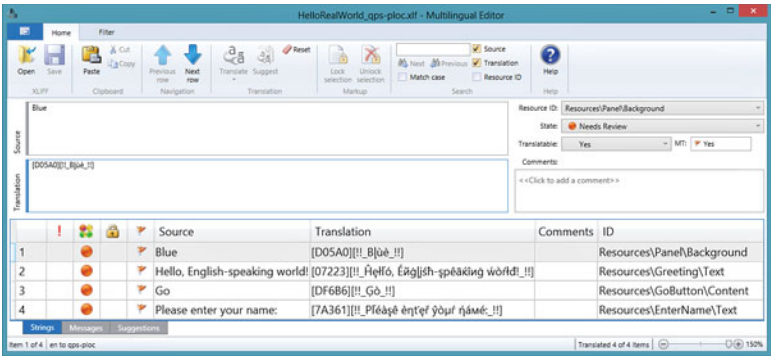
Visual Studio includes functionality for packaging and sending XLIFF files, as well as importing updated files that merge with your local content. These options can be found by right-clicking an `.xlf` file in Solution Explorer.

Now rebuild the `HelloRealWorld` app. This populates each `.xlf` file with a "translation" for each item from the default language `.resw` file. Initially, each translation is just the duplicated English text. However, for some languages, such as the two we've chosen, you can generate machine translations based on the Microsoft Translator service! To do this for the entire file, right-click on each `.xlf` file and select **Generate machine translations**. Voilà! Now we've got initial translations for all of our resources, which you can see by opening each `.xlf` file and examining the list inside the multilingual editor. This is shown in Figure 1.12.

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



Chinese (Traditional)



Pseudo Language

FIGURE 1.12 Each `.xlf` file contains machine-generated initial translations, courtesy of Microsoft Translator.

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

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

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

To add Pseudo Language (and make it the default language), you have to use a hidden trick in Control Panel. After clicking **Add languages**, type **qps-ploc** in the search box for

the entry called **English (qps-ploc)** to appear. You must type *the whole thing* for this to work! This language is hidden in this way because no normal user should ever enable it.

Figure 1.13 shows the result of running `HelloRealWorld` when Windows is set to use each of the two non-English languages. These changes are handled completely by the resource-loading mechanism. Other than the switch to marking elements with `x:Uid`, no code changes were needed. This figure also highlights Pseudo Language's knack for using really long strings that can highlight potential weaknesses in your app's layout.

Chinese (Traditional) Pseudo Language

FIGURE 1.13 `HelloRealWorld` now acts appropriately for Traditional Chinese and for the test-only Pseudo Language.

You can add additional languages to your apps that have already been published in the Windows Store, thanks to *resource pack* support in Windows 8.1. As long as you don't update any code or your version number, your new resources get downloaded only to users with a matching language preference.

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

Making the App Accessible

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

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

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

Improving Screen Reading

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

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

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

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

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

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

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

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

> *"Go button"*

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

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

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

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

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

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

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

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

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

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

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

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

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

Handling High Contrast Themes

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

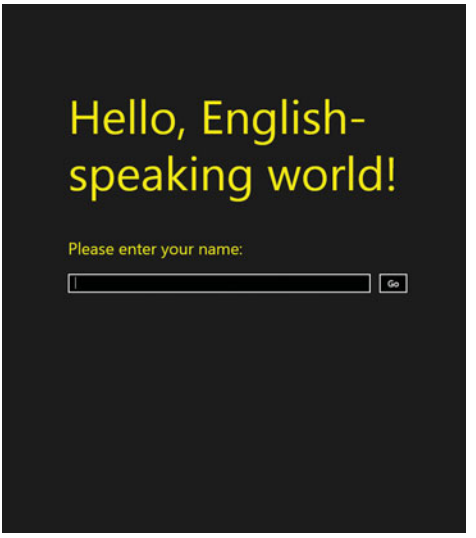
`HelloRealWorld` doesn't use any images, but Chapter 13 explains how you can provide separate versions of your images that can be used for high contrast themes only.

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

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

  public MainPage()
  {
    InitializeComponent();

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

```
    AccessibilitySettings settings = new AccessibilitySettings();

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

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

  void OnHighContrastChanged(AccessibilitySettings sender, object args)
  {
    this.stackPanel.Background =
      sender.HighContrast ? null : this.defaultBackground;
  }

  …
}
```

Because the user could change the theme while our app is running, we need to handle the `HighContrastChanged` event to adjust accordingly. The rest of the app's elements already adjust automatically. Figure 1.14 shows the result of adding this code then running the app under two different high contrast themes. Chapter 18 explains how you can define theme-specific colors without needing to write C# code such as this.



High contrast #1 theme

High contrast white theme

FIGURE 1.14 Removing the explicit `StackPanel` background makes the app look appropriate under any high contrast theme.

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

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

If you do the following:

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

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

Submitting to the Windows Store

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

- ➔ **Set up your developer account** at http://dev.windows.com, get it verified, and fill out your payout and tax information. This can take a couple of days for an individual account, or a couple of weeks for a business account.
- ➔ **Reserve your app name** with the Windows Store, as it requires each app's name to be unique. You can reserve names at any time, and you have up to a year to submit the app before losing each reservation. You can also reserve additional names for other languages.

- **Download, install, and run the Windows App Certification Kit** (WACK) from the Windows Dev Center. This tests your app for violations that cause it to fail the Windows Store certification process, so running it in advance can save you a lot of time.

The Windows Store certification process consists of three parts:

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

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

- If your app requires a network capability, you must write a privacy statement that explains what data you collect, how you store or share it, how users can access the collected data, and so on. Requirement 4.1 in the Windows Dev Center helps you figure out how to write one. Furthermore, a link to the statement must be reachable from the Settings pane for your app, and the same link must be included in your listing in the Windows Store. See Chapter 21, "Supporting Charms," for information about adding content to the Settings pane.
- You must select an appropriate age rating, using guidelines from the Windows Dev Center. For example, most apps that share personal information must be rated at least 12+. Regardless of your app's rating, its *listing* for the Windows Store cannot contain content that is considered too mature for a 12+ rating.
- You must provide descriptions and screenshots for every language you support. If your app is only partially localized for some languages, you must mention this in your listing.

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

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

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

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

Summary

You've now seen the basic structure of a Visual Studio project for a XAML-based Windows Store app and gotten a taste for making an app that is ready to sell across the world. If you've previously done .NET development, much of this should look familiar. If you've previously dabbled in Windows Presentation Foundation (WPF) and/or Silverlight, the role of the XAML files and the C# files should be obvious. And if you've previously done development for Windows Phone, then all of these concepts, including things like capabilities, shouldn't surprise you one bit. If you don't have any such experience, then you should at least be able to appreciate how easy it is to hit the ground running.

Personally, I'm struck by how easy it has become to localize your app and make it accessible. Software development has come a long way over the years, and you'll see evidence of this throughout the book, when it comes to handling heterogeneous screen DPI, making money through the Windows Store, communicating with slick peripherals, and much more. The team behind Windows Store apps has taken the best ideas from .NET, XAML, Windows Phone, the Web, C++, and COM in order to create a compelling platform that's easy for developers to dive into. And now it's time to dive much deeper into the language of XAML.

INDEX

Symbols & Numerics

A

B

C

D

E

F

G

H

I

J-K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y-Z