



Community Experience Distilled

Mastering Windows 8 C++ App Development

A practical guide to developing Windows Store apps with C++
and XAML

Pavel Yosifovich

[PACKT]
PUBLISHING

www.it-ebooks.info

Mastering Windows 8 C++ App Development

A practical guide to developing Windows Store apps with C++ and XAML

Pavel Yosifovich



BIRMINGHAM - MUMBAI

Mastering Windows 8 C++ App Development

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: April 2013

Production Reference: 1080413

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-502-2

www.packtpub.com

Cover Image by Neha Rajappan (neha.rajappan1@gmail.com)

Credits

Author

Pavel Yosifovich

Project Coordinator

Anurag Banerjee

Reviewers

Daniel Biesiada

Eric van Feggelen

Alon Fliess

James P. McNellis

Yusak Setiawan

Proofreader

Linda Morris

Indexer

Hemangini Bari

Graphics

Aditi Gajjar

Acquisition Editor

Erol Staveley

Production Coordinator

Prachali Bhiwandkar

Lead Technical Editor

Sweny Sukumaran

Cover Work

Prachali Bhiwandkar

Technical Editors

Prasad Dalvi

Worrell Lewis

About the Author

Pavel Yosifovich is the CTO of CodeValue (<http://www.codevalue.net>), a software development, consulting, and training company, based in Israel. He is also the author of *Windows Presentation Foundation 4.5 Cookbook*, Packt Publishing, 2012. He writes, develops, consults, and trains developers on various software development topics, from Windows internals, to .NET enterprise systems, and almost everything in between. He's a Microsoft MVP and a frequent speaker at national events.

In the past, he co-founded the startup company Quiksee that was acquired by Google in September 2010.

Writing a book takes tremendous effort, and would not have been possible without the support and encouragement of my family – my wife Idit and my kids Daniel, Amit, and Yoav. I know it was hard watching me sit at my computer and write for hours at a time. Thank you for your patience!

About the Reviewers

Daniel Biesiada is a software professional with 13 years of experience as a developer, consultant, and most recently as a technology evangelist at Microsoft in Poland. In 2012, he left corporate ranks to pursue individual challenges related to cloud and mobile opportunities as a consultant and architect of the software solutions. He speaks on local events in Poland, as well as internationally and works actively with communities to develop new generations of smart and future-oriented developers. With his startup uShine - IT Consulting he realized several Windows 8 projects for customers in media and education industries including iOS to Windows 8 ports and creating unique intellectual properties for Windows 8 and the modern Web.

He was helping local software leaders at Microsoft for 5 years with executive advisory related to Microsoft software development technologies. In the last two years of work at Microsoft, he helped launch cloud products in local markets (Windows Azure) and to fill Windows Store with high-quality applications targeting Windows 8.

He is the co-author of the book *Windows Azure Platforma Cloud Computing dla programistów, APN Promise* that introduced Windows Azure to Polish developers in the local (Polish) market. He can be reached by e-mail at daniel.biesiada@ushine.pl.

Eric van Feggelen is a passionate and experienced software consultant who delivers high-quality solutions using the latest technology available. He has about 15 years of experience as a developer and has been widely interested in information technology his entire life. In the past few years he worked for major corporations such as Microsoft and Avanade and continues to serve the Microsoft Enterprise space as a private contractor for his own company.

For more information on Eric check out his personal website <http://appbyfex.com/>.

Alon Fliess is the Chief Architect and founder of CodeValue. CodeValue is the home of software experts. CodeValue builds software tools, foundations, and products for the software industry. CodeValue offers mentoring, consulting, and project development services.

Alon got his BSc degree in Electrical and Computer Engineering from The Technion, Israel Institute of Technology. He is an expert in many Microsoft technologies, including Windows client and server programming using C#/C++/.NET, Windows Azure Cloud Computing, ALM with TFS, and Windows internals. Microsoft has recognized his expertise and community activities and granted him two awards: Microsoft Regional Director (MRD) and a VC++ MVP.

He has deep knowledge and understanding of Windows and Windows internals, he is the co-author of *Windows 7 Microsoft Training Program*, Microsoft Press as well as the co-author of *Introducing Windows 7 for Developers*, Microsoft Press.

He delivers courses and lectures in many seminars and conferences around the world such as TechEd Europe, TechEd USA, NDC, and in Israel. He is a senior Software Architect, who deals with vast and complex projects.

Many thanks to Pavel and Anurag Banerjee for giving me the opportunity to take part in the creation of this book.

Yusak Setiawan (@yoesak) works at Tigabelas Technology, the company that he founded 3 years ago. He has 10 years' experience of coding in different languages, especially in C/C++, C#, Objective C, and also JavaScript. His company, and he, now focus on making Windows 8 apps, and also support Microsoft Indonesia by training, and mentoring, young developers and corporates in Indonesia in making good Windows 8 apps. He also worked with some Redmond guys before Visual Studio 2012 was released. You can find his work in Windows 8 Store (AndaTube, Alkitab, and MathBoard).

I would like to thank Sweny Sukumaran and Anurag Banerjee, for giving me the challenge of reviewing this book, also my wife Nidya Chatelya and my newborn baby Edmond Grant; they both are my inspiration.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Introduction to Windows 8 Apps	7
Introducing Windows 8	7
Touch everywhere	8
The Start (Home) screen	8
The AppBar	9
The Charms bar	10
Desktop apps versus Store apps	11
The Windows Runtime	12
Language projections	14
Building the user interface	15
Creating your first Store application	15
Closing an application	19
Application deployment	20
Where did int.ToString come from?	21
Project structure	22
Summary	25
Chapter 2: COM and C++ for Windows 8 Store Apps	27
Welcome to C++11	28
New features in C++11	28
nullptr	28
auto	29
Lambdas	30
Smart pointers	33
Conclusion	36
COM and WinRT	37
The IUnknown interface	41
IInspectable interface	42
Creating a WinRT object	43

WinRT metadata	49
The Windows Runtime Library	52
C++/CX	54
Creating and managing objects	54
Accessing members	56
Methods and properties	57
Delegates	58
Events	60
Defining types and members	61
A WinRT component project	62
Adding properties and methods	63
Adding an event	66
Consuming a WinRT component	67
Building a C++ client	67
Building a C# client	69
The Application Binary Interface	70
Asynchronous operations	71
Using tasks for asynchronous operations	75
Cancelling asynchronous operations	76
Error handling	77
Using existing libraries	77
STL	77
MFC	77
ATL	78
Win32 API	78
CRT	79
DirectX	79
C++ AMP	79
The Windows Runtime class library	80
Strings	80
Collections	81
Exceptions	82
Summary	83
Chapter 3: Building UI with XAML	85
XAML	85
XAML basics	86
Type converters	88
Complex properties	88
Dependency properties and attached properties	89
Content properties	90
Collection properties	92

Markup extensions	93
Naming elements	94
Connecting events to handlers	94
XAML rules summary	95
Introducing the Blend for Visual Studio 2012 tool	96
XAML compilation and execution	97
Connecting XAML, H, and CPP files to the build process	98
Resources	100
Binary resources	100
Logical resources	102
Managing logical resources	104
Duplicate keys	106
Styles	106
Implicit (automatic) styles	108
Style inheritance	109
Store application styles	111
Summary	111
Chapter 4: Layout, Elements, and Controls	113
<hr/>	
Introducing layout	113
Layout panels	115
StackPanel	116
Grid	116
Canvas	118
Adding children to a panel dynamically	122
VariableSizedWrapGrid	122
Panel virtualization	124
Virtualizing panels	125
Working with elements and controls	125
Content controls	126
Buttons	130
ScrollViewer	132
Other content controls to note	132
Collection-based controls	134
ListBox and ComboBox	135
ListView and GridView	136
FlipView	136
Text-based elements	137
Using custom fonts	138
TextBlock	138
TextBox	140
PasswordBox	141
RichTextBlock and RichEditBox	141

Images	142
The SemanticZoom control	144
Summary	145
Chapter 5: Data Binding	147
Understanding data binding	147
Data binding concepts	148
Element-to-element binding	148
Object-to-element binding	150
Binding failures	153
Change notifications	153
Binding to collections	155
Customizing a data view	157
Value converters	157
Other parameters for Convert and ConvertBack	162
Data template selectors	162
Commands	163
Introduction to MVVM	165
MVVM constituents	165
Building an MVVM framework	166
More on MVVM	170
Summary	170
Chapter 6: Components, Templates, and Custom Elements	171
Windows Runtime Components	171
Converting C++ to WinRT	172
Crossing the ABI	177
Consuming Windows Runtime Components	178
Other C++ library projects	181
Custom control templates	182
Building a control template	183
Using the control's properties	185
Handling state changes	186
Customizing using attached properties	189
Custom elements	191
User controls	192
Creating a color picker user control	192
Dependency properties	193
Defining dependency properties	193
Building the UI	196
Adding events	197
Using the ColorPicker	197

Custom controls	198
Creating a ColorPicker custom control	199
Binding in code	201
Custom panels	202
Custom drawn elements	203
Summary	204
Chapter 7: Applications, Tiles, Tasks, and Notifications	205
<hr/>	
Application lifecycle	205
Saving and restoring the state	208
Determining application execution states	210
State store options	210
Helper classes	211
Live tiles	211
Setting application tile defaults	212
Updating the tile's contents	213
Enabling cycle updates	214
Tile expiration	214
Badge updates	215
Creating secondary tiles	215
Activating a secondary tile	217
Using toast notifications	217
Toast options	218
Push notifications	218
Push notification architecture	219
Building a push notification application	220
The application server	220
Registering for push notifications	224
Issuing the push notification	226
Push notifications for secondary tiles	227
Background tasks	228
What is a task?	228
Creating and registering a task	228
Implementing a task	230
Task debugging	232
Task progress and cancellation	232
Playing background audio	234
Playing audio	234
Maintaining background audio	235
Sound-level notifications	238
Lock screen apps	238
Requesting to set a lock screen app	240

Other common operations for lock screen apps	240
Background tasks limits	240
Background transfers	241
Example – downloading a file	241
Summary	244
Chapter 8: Contracts and Extensions	245
Capabilities	245
Contracts	246
Share contract	247
Share source	247
Share target	249
Sharing files	254
Sharing page UI generation	254
FileOpenPicker contract	255
Implementing a FileOpenPicker contract	256
Debugging contracts	261
Extensions	261
Settings extension	262
Other contracts and extensions	264
Summary	264
Chapter 9: Packaging and the Windows Store	265
The application manifest	266
The application view state	266
Implementing view state changes	268
Packaging and validating	271
Using the Windows App Certification Kit	274
Summary	276
Index	277

Preface

Windows 8 is Microsoft's latest client operating system. On the one hand, it continues the trend of Windows 7, establishing a stable, robust, and modern operating system. On the other hand, however, it changes a lot of the assumptions and habits learnt from previous Windows versions.

The ubiquitous Start button is gone from the Taskbar – and the desktop is no longer the first thing to see when a user logs in. A new Start Screen awaits the unsuspecting user, filled with "live tiles" that change their content periodically. The classic Start menu is nowhere to be found; curiously enough, the desktop can be found as one of the tiles in that Start Screen.

The new look and feel of Windows 8 is obviously targeted at Tablet devices – numerous models have sprung up in recent months. The new user interface makes sense on a touch-based device, but the traditional mouse and keyboard setup still works as expected on a laptop or desktop machine.

With this new Windows also comes a new runtime upon which a new kind of applications run – the Windows Runtime. Based on this new runtime, applications can be built and uploaded to the Windows Store – a repository of apps that received a certification, identifying them as safe and useful. In fact, average users can only obtain these new applications – Windows Store apps – through the Windows Store, rather than traditional installation means, such as installers or MSI files.

The classic application, now dubbed Desktop apps, can still be written in the usual way with existing technologies in the native (Win32, COM, ATL, MFC, WTL, and so on) or managed space (WinForms, WPF, WCF, EF, and so on), and these run on Windows 8 much as they do on Windows 7 – perhaps better, because of improvements in the Windows Kernel.

The new Windows Store apps can only run on Windows 8 (and later) OS; they require that Windows Runtime, which is based on the well-established foundation of the **Component Object Model (COM)** technology. These apps look visually different in several respects: they are always full screen (except a special "snap view"), have no chrome, use a new UI design scheme, now called Modern UI, are touch oriented, and have some other not so obvious attributes.

This book is all about those new Windows Store apps. Starting with what they are, we will move through the various facets of the Windows Runtime, focusing on using C++ and the new extensions (C++/CX) to leverage this new runtime to write apps that can then be uploaded to the Store and shared with anyone running Windows 8.

What this book covers

Chapter 1, Introduction to Windows 8 Apps, introduces the Windows 8 operating system from the Windows Store app perspective and discusses some of the concepts around Windows Store apps and the Windows Runtime.

Chapter 2, COM and C++ for Windows 8 Store Apps, introduces important features from C++ 11 and the new language extensions, C++/CX, that allow easier access to the Windows Runtime types. This chapter also discusses other classic technologies and where (if at all) they fit in the Windows Store apps model.

Chapter 3, Building UI with XAML, shows how to build user interface for Windows Store apps by using the declarative XAML language and semantics. The concept of resources as they apply to WinRT are explained in detail.

Chapter 4, Layout, Elements, and Controls, discusses the way controls are laid out to build a flexible user interface. Many elements provided by the Windows Runtime are discussed, paying special attention to groups of controls that share particular characteristics.

Chapter 5, Data Binding, discusses one of the most powerful WinRT features that allow seamless integration between controls and data. The popular **Model-View-ViewModel (MVVM)** pattern is introduced with examples of possible implementations.

Chapter 6, Components, Templates, and Custom Elements, shows how to create reusable WinRT components that can be used by other languages, not just C++. Control templates are discussed, allowing complete change in a control's appearance without affecting its behavior. Finally, the chapter demonstrates how to create custom controls, when some existing behavior is needed but unavailable in the built-in controls.

Chapter 7, Applications, Tiles, Tasks, and Notifications, looks at some of the special features of Windows Store apps, such as Live Tiles and the ways they can be updated locally and from a server. Background tasks are discussed, allowing code to execute even if the app is not in the foreground. The chapter also shows how to leverage the device lock screen, how to make long data transfers, and play background music.

Chapter 8, Contracts and Extensions, shows how Windows Store apps can integrate better with Windows and communicate with other applications by implementing contracts and extensions defined by Windows.

Chapter 9, Packaging and the Windows Store, looks at the procedure of packaging, testing, and deploying an application to the Windows Store, detailing some of the things to watch out for to get successfully certified.

What you need for this book

To work with the examples in the book, you'll need Visual Studio 2012 or later (any version, including the Express edition) running on Windows 8 (any version).

Who this book is for

The book is intended for C++ developers who want to use their existing skills to create Windows Store apps. Knowledge of older technologies such as Win32 or MFC is not required; acquaintance with COM is beneficial, but not required.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The XAML shows a `Page` root element with several attributes and a `Grid` element inside."

A block of code is set as follows:

```
<StackPanel Orientation="Horizontal" Margin="20"
VerticalAlignment="Center">
    <TextBox Width="150" Margin="10" x:Name="_number1" FontSize="30"
Text="0" TextAlignment="Right"/>
    <TextBlock Text="+" Margin="10" FontSize="30"
VerticalAlignment="Center"/>
```

```
<TextBox Width="150" Margin="10" x:Name="_number2" FontSize="30"
Text="0" TextAlignment="Right"/>
<TextBlock Text="=" Margin="10" FontSize="30"
VerticalAlignment="Center"/>
<TextBlock Text="?" Width="150" Margin="10" x:Name="_result"
FontSize="30" VerticalAlignment="Center"/>
<Button Content="Caclulate" Margin="10" FontSize="25" />
</StackPanel>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<Button Style="{StaticResource numericStyle}" Grid.Row="1"
Content="7" Click="OnNumericClick" />
<Button Style="{StaticResource numericStyle}" Grid.Row="1"
Grid.Column="1" Content="8" Click="OnNumericClick"/>
<Button Style="{StaticResource numericStyle}" Grid.Row="1"
Grid.Column="2" Content="9" Click="OnNumericClick"/>
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "The first thing to note about Windows 8 is the new **Start** screen."

 Warnings or important notes appear in a box like this.

 Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Introduction to Windows 8 Apps

Windows 8, Microsoft's latest client operating system, looks quite different than its Windows 7 predecessor. With a new Start (home) screen, it promises to be a redesigned system, not just on the UI front but also in the way applications are written. A new style of applications is available for Windows 8 (and later) that are quite different from the "normal" applications (that are still very much supported).

In this chapter, we'll take a quick tour of the new Windows 8 features, specifically those related to the new application type, known as Windows Store apps (formerly "Metro").

Introducing Windows 8

Windows 8 has been described by Microsoft as "Windows reimagined", which is not a false statement. From a user's perspective Windows 8 looks different; most notably, a new Start screen and the removal of the ubiquitous Start button that existed since Windows 95.

Under the covers, though, Windows is still the operating system we know and love; applications running on Windows 7 should continue to run just as well (and probably better) on Windows 8. Many improvements went into the product, most of them invisible to the typical user; the visible changes are quite evident right from the Start (pun intended).

Touch everywhere

Windows 8 targets touch-enabled devices, such as tablets. Microsoft itself is providing tablet devices under its own brand ("Surface") that are available starting from the Windows 8 **General Availability (GA)** date, October 26, 2012.



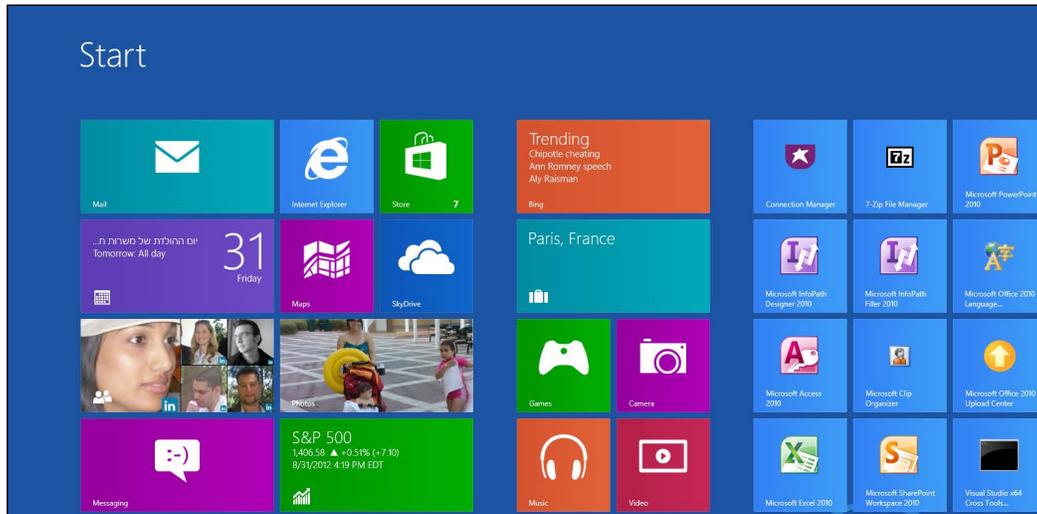
It's worth mentioning that in the same time frame, Microsoft has released Windows Phone 8, the successor of the Windows 7.5 mobile OS, with a similar look and feel to Windows 8. Windows Phone 8 is based on the same kernel that powers Windows 8, and shares portions of the Windows 8 runtime. Going forward, these platforms are likely to merge, or at least come closer together.

Windows 8 is optimized for touch-enabled devices. Swiping the edges of the screen (always towards the visible part of the screen) causes something to happen (the same effect can be achieved with the mouse by moving the cursor to the edges or by using certain keyboard shortcuts). For example, swiping from the right causes the Charms bar to appear (more on the Charms bar in *The Charms bar* section given later in the chapter); the same effect can be achieved by moving the mouse cursor to the right edge of the screen or using the keyboard shortcut Windows key + C.

The Start (Home) screen

The first thing to note about Windows 8 is the new **Start** screen. It's filled with tiles, mostly representing applications installed on the machine. The well-known desktop (from previous Windows versions) appears as a regular tile; clicking it (or tapping it using touch) transfers the user to the familiar desktop environment with largely the same functionality as in previous Windows versions, with shortcut icons, the taskbar, the notifications area, and so on, all except the Start button, which has gone away.

All installed applications are available from the new **Start** screen, whether they are the "normal" desktop applications or the new Store ("Metro") style applications:



The AppBar

Swiping from the bottom in the **Start** screen presents the AppBar. This piece of UI is the replacement for a right-click context menu popular with the mouse. In fact, right-clicking with the mouse anywhere in the **Start** screen shows the AppBar, just as if the screen was swiped from the bottom.

The AppBar provides relevant options depending on the selected object (or no selected object) and is used with the new Store apps just as on the **Start** screen; there is no built-in way to show the classic context menu in a Store application, even if a mouse device is used.



The fact that right-clicking in a Windows Store app (or the **Start** screen) causes the AppBar to appear even though the mouse is used is somewhat annoying, as the user is now forced to move the mouse from the intended object all the way to the bottom (or top for some applications) to select the required option.

The Charms bar

The Charms bar appears when swiping from the right (on a touch device), or by moving the mouse to one of the corners on the right-hand side of the screen. Charms are ways to communicate with other applications from a user's perspective. The standard charms are **Search**, **Share**, **Start**, **Devices**, and **Settings**:

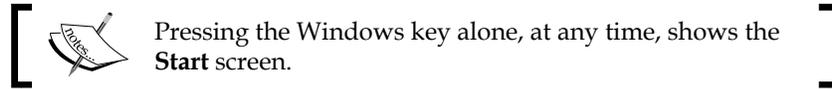


The **Search** charm allows the user to not just search in the operating system's applications (such as Control Panel applications) and the user's personal files (documents, pictures, and so on), but also in any other Store application that indicates it supports the Search contract.

 From the **Start** screen you can start searching just by typing on the keyboard, no need to explicitly activate the **Search** charm.

The **Share** charm allows an app to communicate with other apps without knowing anything about those apps. This is achieved by implementing the Share contract—either as a provider and/or receiver (we'll cover contracts in *Chapter 8, Contracts and Extensions*).

The **Start** charm simply takes the user to the **Start** screen.



The **Devices** charm allows access to device-related activities (if supported by the app), such as printing. And finally, the **Settings** charm allows the user to customize the currently executing app (if supported by the app), or to customize general Windows features.

Desktop apps versus Store apps

All the applications running on Windows systems prior to Windows 8 are called Desktop applications in Windows 8 terminology. These are the regular, normal applications, which can be built with any of the various Microsoft technologies, such as the Win32 API, **Microsoft Foundation Classes (MFC)**, **Active Template Library (ATL)**, .NET technologies (WPF, Silverlight, Windows Forms, and so on), and any logical combination of these. These types of applications are still very much supported in Windows 8, so there's really nothing special here.

The other type of applications supported in Windows 8 is the Store applications. These applications are unsupported in previous Windows versions. Windows Store apps are the focus of this book. We won't be dealing with desktop apps at all.

Store applications are different in many ways from desktop apps. Some of the differences are:

- Store apps are immersive, they are always full screen (except when snapped, see *Chapter 9, Packaging and the Windows Store*); there is no window chrome (that is no caption, no close or minimize buttons, and so on). We'll discuss the user interface aspects of Store apps in *Chapters 3, Building UI with XAML* and *Chapter 4, Layout, Elements, and Controls*.
- The Store apps life cycle is managed by Windows. If another application becomes the foreground app, the previous app is suspended (after a few seconds), consuming no CPU cycles. We'll discuss the application lifecycle in *Chapter 7, Applications, Tiles, Tasks, and Notifications*.
- Only one instance of the app can run at any one time. Clicking on the app tile while the app is running simply switches to the running app. The user should not know, nor care, if the app was actually already in memory or just started up.

- Store apps cannot directly communicate with other running apps, some forms of communication are possible through the idea of contracts. We'll discuss contracts in *Chapter 8, Contracts and Extensions*.
- Store apps run on top of a new runtime, called **Windows Runtime (WinRT)** that is built upon native foundations and the **Component Object Model (COM)** technologies. We'll discuss WinRT and its relation to COM in *Chapter 2, COM and C++ for Windows 8 Store Apps*.
- Store apps are distributed and installed via the Windows 8 Store only (except for special cases for enterprise customers), and not using traditional means of an installer package. We'll discuss the Store in *Chapter 9, Packaging and the Windows Store*.
- Store apps must declare anything they want to use up front through capabilities (such as using the camera that may be present on the device). Anything not declared will cause a failure at runtime. When a user selects the app for downloading, he/she must accept the capabilities the app wants to use; otherwise, the app won't install.

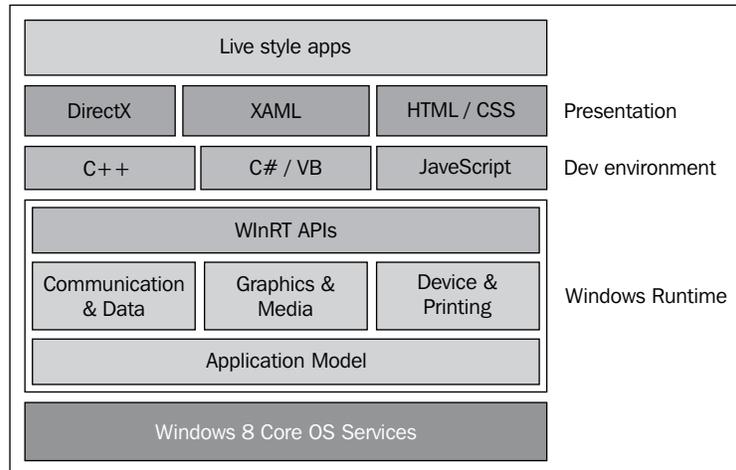
What all this means is that Store apps are different, requiring a different knowledge set, quite unlike the knowledge for writing desktop apps.

 Windows 8 on tablets comes in two main variants, based on the CPU architecture. One is based on Intel/AMD (with 32 bit and 64 bit variants), which is a full Windows 8 that can run desktop apps, as well as Store apps. The second edition is based on the ARM family of processors and is named "Windows RT" (not to be confused with the Windows Runtime). This edition can only run Store apps (at least at the time of writing).

The Windows Runtime

Store applications are built and execute against a new runtime called Windows Runtime (WinRT) that is not present on previous Windows versions. WinRT is built upon the well-established COM technology (with some WinRT-specific enhancements, such as the use of metadata). This means WinRT is entirely native (no .NET CLR anywhere), making C++ a natural and performant choice for targeting this runtime.

WinRT provides a set of services, on which apps can be built. The relationship between the WinRT and applications can be represented by the following diagram:



WinRT APIs have the following characteristics:

- Built as a set of types, implementing interfaces (as mandated by COM). These types are arranged in hierarchical namespaces, logically grouped for easy access and for preventing name clashes.
- Every WinRT object handles its own lifetime by using (mostly) an internal reference count (as done in COM).
- Using the raw WinRT may be pretty verbose, leading to language projections that implement the little details, such as decrementing the reference count automatically when an object is no longer needed by a client.
- All public types are built with metadata, describing the public surface of the API. This is part of the magic that allows various languages to access WinRT relatively easily.
- Many APIs are asynchronous, they start an operation and notify when that operation completes. A general guideline in WinRT is that any operation that may take more than 50 milliseconds should be made asynchronous. This is important so that the UI does not get frozen which makes for a bad user experience.

We'll take a detailed look at WinRT core concepts in *Chapter 2, COM and C++ for Windows 8 Store Apps*.

Language projections

As WinRT uses COM, using it directly is only possible from a language that understands pointers and virtual tables natively, namely C++ (C is technically also possible, but we won't discuss it in this book).

Many developers working with Microsoft technologies work in non-C++ environments, namely .NET (mostly with the C# language, but other languages are used as well, such as Visual Basic and F#) and JavaScript, popular (and necessary) with web development.

Even in C++, using COM is not as easy as we'd like; a lot of details need to be taken care of (such as calling the `IUnknown` interface methods when appropriate), distracting the developer from his/her primary job—building the actual app functionality. This is why Microsoft has created language projections that expose WinRT in selected environments fairly consistently with that particular environment.

Microsoft currently provides three language projections over WinRT:

- C++ has the most lightweight and direct projection. These projections are made possible by a set of language extensions, known as C++/CX (Component Extensions). These make working with WinRT objects much easier than using the raw COM interfaces (we'll discuss this in length in *Chapter 2, COM and C++ for Windows 8 Store Apps*).
- Using managed (.NET) languages such as C# and Visual Basic is possible through projections to the .NET runtime. These projections make it very easy for .NET developers to work with WinRT. **Runtime Callable Wrappers (RCWs)** are created automatically to bridge the managed-native boundary when transitioning to and from WinRT. This mechanism is very similar in principle to the usual way in which .NET code calls COM objects.
- The third supported projection is with the JavaScript language, popular in web development. Clever wrappers over WinRT make using JavaScript relatively easy, including making certain conventions automatic, such as using a lowercase letter for the first word of a method, even though the real WinRT method begins with a capital letter. Using JavaScript also brings in HTML for building the Store app user interface, again potentially leveraging existing knowledge for JavaScript developers.



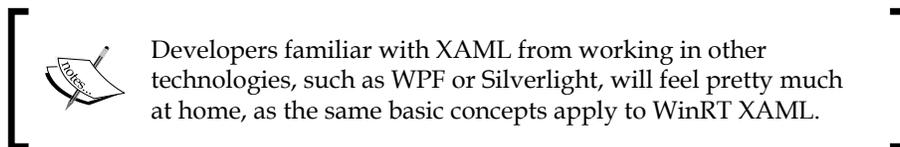
JavaScript is limited to consuming WinRT types. It cannot create new types (.NET and C++ can).

C++ does not require the CLR (.NET runtime), which makes it the most lightweight, both in terms of execution speed and memory consumption. We'll take a detailed look at using C++ throughout this book, starting from the next chapter.

Building the user interface

JavaScript is the only language that has direct access to working with HTML, as the way to create the app's user interface. This is so that JavaScript developers have less to learn, they probably know HTML already. The Windows Library for JavaScript provides access to controls, CSS, and other helpers to bridge the gap to WinRT.

C++ and .NET developers use the XAML language to build user interfaces. XAML is an XML-based declarative language that allows (somewhat simplistically) creating objects and setting their properties. We'll take a closer look at XAML and UI in *Chapter 3, Building UI with XAML*.



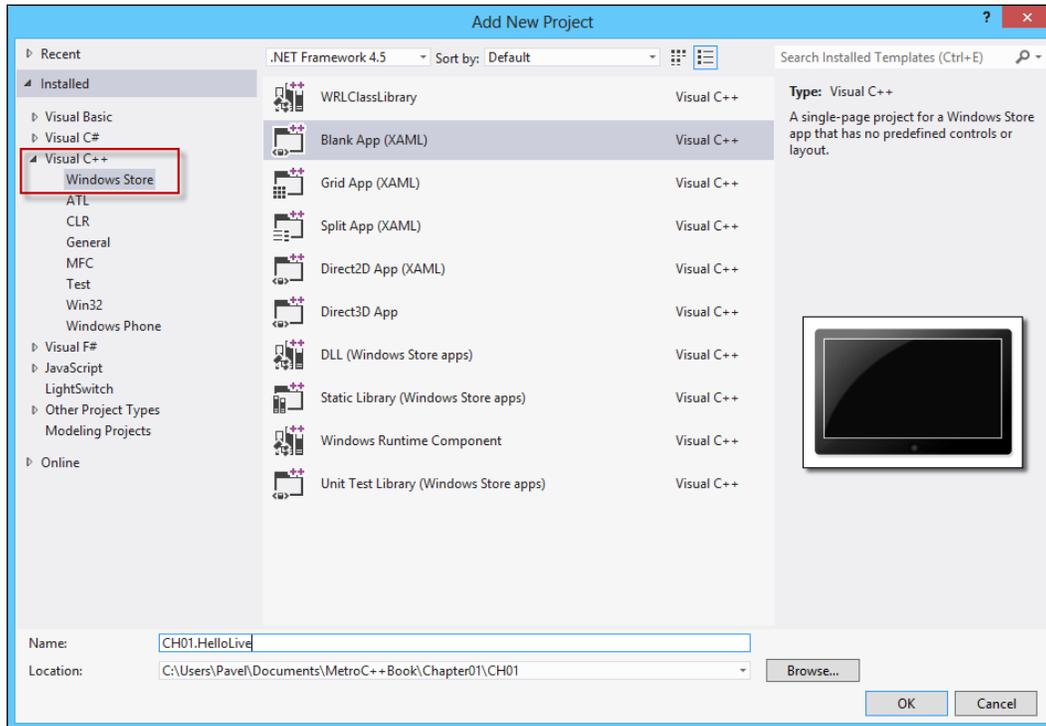
A third option exists, primarily for C++ developers – DirectX. DirectX is the most low-level and powerful graphic API on the Windows platform; thus, it's used mostly for authoring games while utilizing the full potential of the machine by leveraging the power of the **Graphic Processing Unit (GPU)**. As DirectX is, itself, built upon COM, it's naturally accessible from C++. Other languages must go through some wrapper library to gain direct access to the DirectX APIs (no such wrapper is provided by Microsoft at the time of writing, but there are third party libraries such as SharpDX).

Creating your first Store application

Enough talk. It's time to open Visual Studio and create a simple Store app in C++, looking at some of its characteristics. We'll delve deeper into the way a Windows Store app is built in the next chapter.

Store apps must be created using Visual Studio 2012 (or later) running on Windows 8 (or later); although Visual Studio 2012 runs on Windows 7, it cannot be used for developing Store apps on that OS.

Let's open Visual Studio 2012 and create a new Store application project in C++ by selecting the **Windows Store** node under the **Visual C++** node:



Select **Blank App (XAML)** on the right and enter `CH01.HelloLive` in the **Name** textbox and enter some location on your filesystem; then click **OK**.

Visual Studio creates a project with several files. We'll take a look at those files a bit later, but for now open the **MainPage.xaml** file. This is where the UI is located. It has a split view by default, the lower pane showing the XAML markup and the upper pane showing a preview. The XAML shows a `Page` root element with several attributes and a `Grid` element inside. We'll discuss all the details in *Chapter 3, Building UI with XAML*, but for now we'll create a simple addition calculator as our first "Hello World!" application. Add the following markup inside the `Grid` element:

```
<StackPanel Orientation="Horizontal" Margin="20"
VerticalAlignment="Center">
    <TextBox Width="150" Margin="10" x:Name="_number1" FontSize="30"
Text="0" TextAlignment="Right"/>
    <TextBlock Text="+" Margin="10" FontSize="30"
VerticalAlignment="Center"/>
```

```

    <TextBox Width="150" Margin="10" x:Name="_number2" FontSize="30"
    Text="0" TextAlignment="Right"/>
    <TextBlock Text="+" Margin="10" FontSize="30"
    VerticalAlignment="Center"/>
    <TextBox Width="150" Margin="10" x:Name="_number1" FontSize="30"
    Text="0" TextAlignment="Left"/>
    <TextBlock Text="?" Width="150" Margin="10" x:Name="_result"
    FontSize="30" VerticalAlignment="Center"/>
    <Button Content="Caclulate" Margin="10" FontSize="25" />
</StackPanel>

```



Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

The upper preview part should show something like this:



Two `TextBox` controls (named `_number1` and `_number2`) are used for user input and a `TextBlock` element (named `_result`) is used for the output. To make this work, we need to handle the `Button`'s `Click` event. To do that, simply double-click the button in the designer. This will add an event handler in the `MainPage.xaml.cpp` file (as well as the corresponding header file and a `Click` attribute for the button in the XAML). Visual Studio should automatically open `MainPage.xaml.cpp`. The generated event handler will look like the following:

```

void CH01_HelloLive::MainPage::Button_Click_1(
    Platform::Object^ sender,
    Windows::UI::Xaml::RoutedEventArgs^ e)
{
}

```

At the top of the file Visual Studio created some using namespace declarations that we can leverage to simplify the method signature (`CH01_HelloLive`, `Platform`, and `Windows::UI::XAML` namespaces):

```

void MainPage::Button_Click_1(Object^ sender, RoutedEventArgs^ e)
{
}

```

The handler may seem strange at this time, at least due to the "hat" (^) symbol stuck to the `Object` and `RoutedEventArgs` classes. We'll discuss this in the next chapter (this is a C++/CX extension), but the hat basically means a "reference counted pointer" to a WinRT object.

All that's left to do now is implement the handler, so that the calculated result is shown in the result `TextBlock`.

It turns out that naming elements makes those names actual member variables of the class in question (`MainPage`) and, thus, are available for us to use where needed.

First, we need to extract the numbers to add, but the content of the `TextBox` controls is a string. In fact, it's a WinRT string, `Platform::String`. How do we convert that to a number? Do we use some WinRT function for that?

No. We use plain old C++; we just need a way to turn a `Platform::String` into a normal `std::string` or `std::wstring` (`wstring` should be preferred, as all WinRT strings are Unicode). Fortunately, that's easy with the `Data()` member function of `Platform::String` that returns a simple `const wchar_t*` pointing to the string; note that a Unicode pointer is the only one available.

To do the actual conversion, we can use old C-style functions such as `wtoi()`, but for a nicer, modern conversion, we'll use string streams. Add an `#include` near the top of the file (after the existing includes) for `<sstream>`:

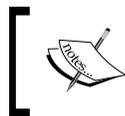
```
#include <sstream>
```

Next, inside the event handler, we'll create two `wstringstream` objects to handle the conversion based on the content of the `TextBox` controls:

```
std::wstringstream ss1(_number1->Text->Data()),  
    ss2(_number2->Text->Data());
```

Notice the arrow (`->`) operator at work. The "hat" references are dereferenced using the arrow dereferencing operator, but they are not pointers (Chapter 2, *COM and C++ for Windows 8 Store Apps* will explain further). Let's continue with the conversion to integers:

```
int number1, number2;  
ss1 >> number1;  
ss2 >> number2;
```



We can actually do the conversion faster with a new C++ 11 function, `std::stoi` that converts a `std::string` (or `std::wstring`) to an integer.

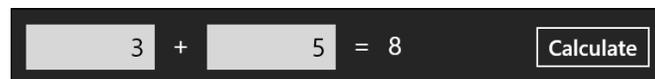
Finally, we need to place the result of adding the numbers to the `TextBlock` named `_result`:

```
_result->Text = (number1 + number2).ToString();
```

The `ToString()` call operating on an integer, provides the conversion to a `Platform::String`, which is very convenient in this case. How is it possible to have a member function on an `int`? It's possible, because it's WinRT's `int`, and all WinRT types derive from an ultimate base class named `Platform::Object` (this is not strictly true, as this is achieved by compiler trickery. A more detailed explanation is provided in the next chapter), which exposes a `ToString()` virtual method to be overridden by derived types. Still, `int` is a primitive type in C++, and should not derive from anything, so how could it have a `ToString()` method? We'll return to that in a moment.

For now, let's test the application. Build the project and run it with the debugger by selecting **Debug | Start Debugging** from the menu, click on the relevant toolbar button (with a green arrow and labeled **Local Machine** by default) or simply press *F5*.

A splash screen of a crossed rectangle should appear for a few seconds and then the application's UI should come up. Type two numbers in the text boxes and click on the button to observe the result:



Not too sophisticated, but a Store app nonetheless! Note the application is full screen, with no title bar, caption buttons, or even a close button for that matter. This is the way Store apps look.

Closing an application

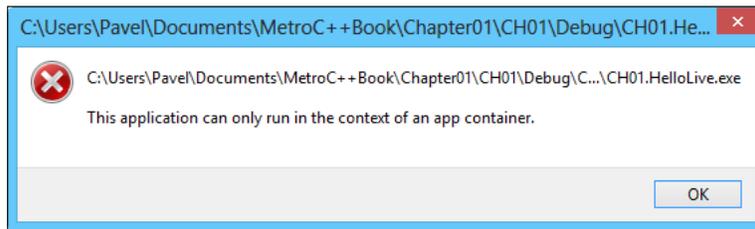
How do we close the application? One – not so convenient with a mouse – way is to grab the window at the top (where a title bar would have been) and drag it all the way to the bottom. This was mainly done because Store apps are not meant to be explicitly closed. An app will become suspended (consume no CPU) if not used and can be terminated if the memory pressure is high; this means a typical user should not care about closing an app.

Luckily, we're not typical users. A simpler way to close the app is to *Alt + Tab* back to Visual Studio and select **Debug | Stop Debugging** from the menu (or *Shift + F5*). That's why it's better to test Store apps from Visual Studio with the debugger attached.

[ Pressing *Alt + F4* also works as a way to close an application.]

Application deployment

Can we run the application without Visual Studio? We can navigate to the folder where the source code is built and locate the resulting `.exe` file. Double-clicking on that from Windows Explorer fails with the following message box:



The error message is basically saying that we cannot simply run the Store app just like a desktop app, there are several steps involved in starting a Store app, for which a simple double-click is not merely enough. So, how can we run the app without Visual Studio? The same way "normal" users do, through the **Start** screen.

If we open the **Start** screen and navigate all the way to the right, we'll find something like this:



The application was deployed automatically by Visual Studio, as if it was downloaded from the Windows Store. It's actually possible to do deployment only without running by selecting **Build | Deploy Solution** from Visual Studio's menu. To remove the application, right-click it in the **Start** screen (or swipe from the bottom) and select **Uninstall**.

Where did `int.ToString` come from?

To find this out, we'll set a breakpoint on the last line of the click event handler we implemented, and run the application until we reach the breakpoint. When the breakpoint hits, right-click at the breakpoint line in the editor and select **Go To Disassembly**. These are the first few lines of assembly code at that point:

```

    _result->Text = (number1 + number2).ToString();
00196441 mov     eax,dword ptr [number1]
00196447 add     eax,dword ptr [number2]
0019644D mov     dword ptr [ebp-2A8h],eax
00196453 lea    ecx,[ebp-2A8h]
00196459 push   ecx
0019645A call   default::int32::ToString (011FBEDh)

```

The last line is the interesting one, calling some static function named `default::int32::ToString`. We can Step Over (*F10*) to that line and then Step Into (*F11*). After a few more Step Into, we finally reach the actual function. Right-clicking the window and selecting **Go To Source Code** leaves out the detailed assembly and shows code from a file called `basetypes.cpp` with the function implemented like the following:

```

VCCORLIB_API Platform::String^ int32::ToString()
{
    wchar_t buf[32];
    swprintf_s(buf,L"%I32d", _value);
    return ref new Platform::String(buf);
}

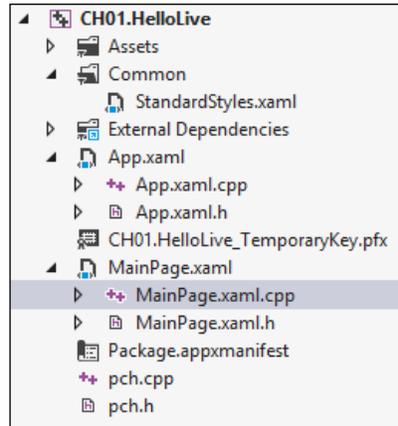
```

All this is in a namespace called `default`. The implementation is trivial, using a "safe" variant on the classic `swprintf` C function before turning that back into a WinRT string, namely `Platform::String`. The strange `ref new` will be discussed in the next chapter, but it essentially means "create an instance of a WinRT type".

Similar helpers exist throughout the C++/CX library to make using WinRT from C++ easier. We'll see a lot more on that in the next chapter.

Project structure

Let's take a closer look at some of the files created as part of the project we created:



Most of the files are new from a C++ developer's perspective, except the files `pch.h` and `pch.cpp`. These files constitute the precompiled header, which means a header that contains seldom changed headers, so it can be compiled just once, saving recompilations later. In other project types, such as a regular Win32 application, MFC, ATL, and so on, these files were named `StdAfx.h/StdAfx.cpp` (which have no real meaning) so their names changed for the better. Their use is exactly the same, placement of seldom changing header files to speed up compilation times.

 It's important to keep the precompiled header file name `pch.h`; this is because some of the code generated by the build processes uses this hardcoded filename.

`MainPage.xaml` holds the XAML markup for the `MainPage` class. The other two files that complete it are the H and CPP files. Note, that the CPP file has an `#include` to `MainPage.xaml.h` and that file has an `#include` to `MainPage.g.h`, which is generated (that's what the "g" stands for) by the XAML compiler (in fact, it changes as needed by editing `MainPage.xaml`, without any actual compilation). There we can find the declarations for the three named elements we used, without declaring them ourselves:

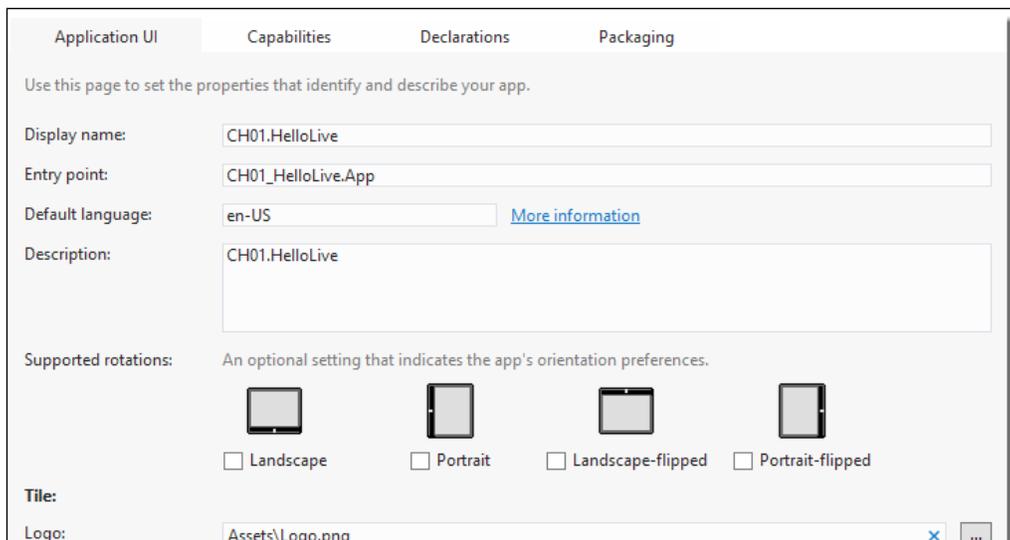
```
private: ::Windows::UI::Xaml::Controls::TextBox^ _number1;
private: ::Windows::UI::Xaml::Controls::TextBox^ _number2;
private: ::Windows::UI::Xaml::Controls::TextBlock^ _result;
```

MainPage.xaml itself indicates which class it's related to with the `x:Class` attribute on its root element:

```
<Page x:Class="CH01_HelloLive.MainPage">
```

App.xaml, App.xaml.h, and App.xaml.cpp have the same kind of connection among themselves as do the MainPage.* files, but their meaning is a bit different. App.xaml.h declares the single application class that provides an entry point for the app, as well as other services that will be discussed in later chapters. It may be curious as to why it has a XAML file. Could the application object have a UI? Not really. The XAML is there mostly to host resources, as we'll see in *Chapter 3, Building UI with XAML*.

The Package.appxmanifest file is where all the application's metadata is stored. Internally it's an XML file, but Visual Studio wraps it in a nice UI that is easier to use most of the time. Double-clicking the file opens Visual Studio's view of the manifest:



Here, we can set the name of the app, description, supported orientations, various images (such as the splash screen image), and many other (more important) settings, such as the capabilities required by the application. We'll discuss the various options in the relevant chapters.

If a raw view of the file as XML is needed, we can right-click the file in the Solution Explorer, select **Open With**, and then select **XML Editor**. Here's what the XML looks like for our calculator application:

```
<Package xmlns="http://schemas.microsoft.com/appx/2010/manifest">
  <Identity Name="a984fde4-222a-4c90-b9c1-44ad95e01400"
    Publisher="CN=Pavel"
    Version="1.0.0.0" />

  <Properties>
    <DisplayName>CH01.HelloLive</DisplayName>
    <PublisherDisplayName>Pavel</PublisherDisplayName>
    <Logo>Assets\StoreLogo.png</Logo>
  </Properties>

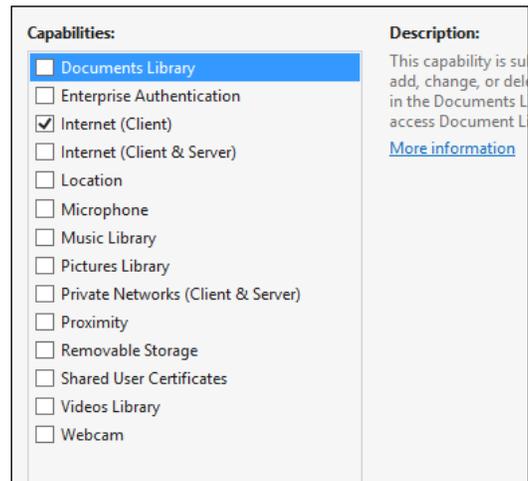
  <Prerequisites>
    <OSMinVersion>6.2.1</OSMinVersion>
    <OSMaxVersionTested>6.2.1</OSMaxVersionTested>
  </Prerequisites>

  <Resources>
    <Resource Language="x-generate"/>
  </Resources>

  <Applications>
    <Application Id="App"
      Executable="$targetnametoken$.exe"
      EntryPoint="CH01_HelloLive.App">
      <VisualElements
        DisplayName="CH01.HelloLive"
        Logo="Assets\Logo.png"
        SmallLogo="Assets\SmallLogo.png"
        Description="CH01.HelloLive"
        ForegroundText="light"
        BackgroundColor="#464646">
        <DefaultTile ShowName="allLogos" />
        <SplashScreen Image="Assets\SplashScreen.png" />
      </VisualElements>
    </Application>
  </Applications>

  <Capabilities>
    <Capability Name="internetClient" />
  </Capabilities>
</Package>
```

The root element is `Package`. Everything else are the settings that differ from the defaults. The `Capabilities` element, for example, shows the required capabilities needed by the app to function correctly. The only element inside is `internetClient`. Clicking on the **Capabilities** tab in the Visual Studio manifest UI shows this:



The **Internet (Client)** option is checked (the only capability that is requested by default), meaning the app can make outbound calls to the network.

Changing the XML affects the Visual Studio UI and vice versa. Sometimes, it may be more convenient to edit changes in the XML mode.

Summary

Windows 8 Store applications are different in many ways from desktop apps. From the way they look, to the way they execute and, of course, the runtime they depend upon. The Windows Runtime provides a rich environment for creating apps that run on desktop and tablet platforms alike, but it is new and, thus, requires getting acquainted with the library and the platform as a whole.

The Windows Runtime is based on the COM programming model, making it possible to create projections for various languages and runtimes. Currently, C++, .NET, and JavaScript are supported, but more may be created in the future by Microsoft and/or other vendors.

C++ developers have the most fine-grained, direct access to WinRT. The C++/CX extensions that we'll see in more detail in the next chapter make developing with C++ almost as easy as with higher-level environments, while leveraging the powerful capabilities of existing C++ libraries and the C++ language.

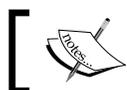
2

COM and C++ for Windows 8 Store Apps

C++ was first released to the public in 1985 by Bjarne Stroustrup, its creator and original implementer. It was first named "C with Classes", extending the C language to include true object-oriented features. It was published in 1998 in an "official" form and started to gain real traction. In 1998, the ISO standard for the language appeared, later to be revised slightly in 2003.

Years went by without a new standard until 2011, in which finally a new C++ standard was finalized (this process was going on for several years) by the standards committee. Going from 1998 to 2011 with nothing official in between made C++ less popular than it used to be, mostly because of new languages and platforms that appeared, mainly Java (over the various Java platforms) and C# (over the .NET platform). Data-driven applications, which traditionally were written in C++, were (and still are) written in Java (in the non-Microsoft world) and C# (and, to a small extent, other .NET-based languages such as Visual Basic, in the Microsoft world). C++ remained a language with many followers, but the lack of advancement has shown cracks in the C++ ecosystem.

This 13-year gap was not without progress in C++, but it was in the library domain, rather than the language domain. The most notable contribution was the boost library (<http://www.boost.org>) that contributed a lot of high-quality libraries that extended the standard C++ libraries; and although boost was not an official standard, it has become a de-facto standard among the C++ community. In fact, parts of boost have made it to the new C++11 standard.



The C++ committee has decided to expedite matters for future standards and is planning a new standard in 2014 (and later in 2017); time will tell.

Welcome to C++11

The C++11 standard was developed for several years, first called "C++0x", where "x" was hoped to be a single digit, making the standard final in 2009 at the latest, but things didn't turn out that way and the standard was finalized in September 2011.

 It's possible to replace "x" with "b", the hexadecimal equivalent of 11 and still maintain a single digit if so desired.

C++11 has many new features, some part of the core language and some part of the new standard libraries. 13 years is practically an eternity in computer years, and that's why there are so many additions to the language; in fact, at the time of writing, no compiler (Microsoft and non-Microsoft alike) exists that implements the entire C++11 standard. Visual Studio 2010 has implemented several features and Visual Studio 2012 implements some more features (and enhances existing features); it will probably take a while until all C++11 features are implemented by compilers.

 For a comprehensive list of supported C++11 features in VS 2012 and VS 2010, refer to this blog post by the Visual C++ team: <http://blogs.msdn.com/b/vcblog/archive/2011/09/12/10209291.aspx>. More features are expected to be available in relatively frequent updates.

New features in C++11

We'll take a look at some of the new C++11 language and library features, that make it easier to develop using C++, not necessarily related to Windows 8 Store apps.

nullptr

`nullptr` is a new keyword that replaces the famous value `NULL` as a pointer that points nowhere. It doesn't seem to be a major feature, but this makes any `#define` for `NULL` unnecessary and also resolves some inconsistencies. Consider the following overloaded functions:

```
void f1(int x) {
    cout << "f1(int)" << endl;
}

void f1(const char* s) {
    cout << "f1(const char*)" << endl;
}
```

Which function would be invoked by calling `f1(NULL)`? The (maybe surprising) answer is `f1(int)`. The reason is that `NULL` is defined as a simple zero by the Microsoft compiler, which is interpreted as an integer by the compiler and not a pointer; this means the compiler's overload resolution selects `f1(int)`, rather than `f1(const char*)`. `nullptr` solves that; calling `f1(nullptr)` invokes the correct function (that accepts an actual pointer). From a purist standpoint, it's hard to imagine a language where pointers are of prime importance, not having a dedicated keyword to indicate a pointer to nothing. This was mainly for compatibility reasons with the C language; now it's finally resolved.

In C++/CX (which we'll discuss later in this chapter), `nullptr` is used to indicate a reference to nothing.

auto

The `auto` keyword existed since the C language, being a redundant keyword that meant "automatic variable", meaning a stack-based variable. The following C declaration is legal, but adds nothing of real value:

```
auto int x = 10;
```

In C++11, `auto` is used to tell the compiler to automatically infer a variable's type based on its initialization expression. Here are a few declarations:

```
int x = 5;
string name = "Pavel";
vector<int> v;
v.push_back(10);
v.push_back(20);
for(vector<int>::const_iterator it = v.begin(); it != v.end();
    ++it)
    cout << *it << endl;
```

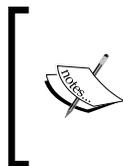
This looks ordinary enough. Let's replace this with the `auto` keyword:

```
auto x = 5;
auto name = "Pavel";
vector<int> v;
v.push_back(10);
v.push_back(20);
for(auto it = v.begin(); it != v.end(); ++it)
    cout << *it << endl;
```

Running these pieces of code produces the same result (displaying 10 and 20 when iterating through the `vector`).

Initializing `x` to 5 using `auto` isn't much better than specifying the actual type (`int`); in fact, it's less clear (by the way, 5 is an `int`, whereas 5.0 is `double`, and so on). The real power of `auto` is with complex types, such as the preceding iterator example. The compiler infers the correct type based on the initialization expression. There is no runtime benefit here, it's just compile time inference. But, it makes the code (usually) more readable and less error prone. The variable type is not some kind of void pointer, it's exactly as if the type was specified explicitly. If `x` is an `int`, it will continue to be an `int` forever. The programmer does not have to think too hard about the correct type, we know it's an iterator (in the preceding example), why should we care about the exact type? Even if we do care, why should we write the complete type (which may contain template arguments which further enlarge the type expression), as the compiler knows the exact type already? `auto` can simplify things, as we'll see later on, when dealing with nontrivial WinRT types.

What about the string initialization? In the non-`auto` case, we used `std::string` explicitly. What about the `auto` case? It turns out the type of `name` is `const char*` and not `std::string`. The point here is that sometimes we need to be careful, we may have to specify the exact type to prevent unwanted compiler inference.



Naturally, specifying something like `auto x;` does not compile, as `x` can be of any type – there must be an initialization expression. Similarly, specifying something like `auto x = nullptr;` fails to compile as well; again, because `x` can be any pointer type (and even non-pointer type with appropriate converting constructor).

Lambdas

Lambda functions, or simply lambdas, are a way to create anonymous functions, specified inline where needed. Let's look at an example. Suppose we want to use the `transform` algorithm to take some items in a container and generate new items based on some transformation function. Here's one prototype of `transform`:

```
template<class InIt, class OutIt, class Fn1>
OutIt transform(InIt first, InIt last, OutIt dest, Fn1 func);
```

The `transform` template function accepts, as its last argument, a transformation function to be invoked on each and every item specified with the start and end iterators.

One way to specify that function is by setting up a global (or a class static) function, as shown in the following code snippet:

```
double f1(int n) {
    return ::sqrt(n);
}
```

```

}

void LambdaDemo() {
    vector<int> v;
    for(int i = 0; i < 5; i++)
        v.push_back(i + 1);
    for each (int i in v)
        cout << i << endl;

    vector<double> v2(5);

    ::transform(begin(v), end(v), begin(v2), f1);

    cout << endl;
    for each (double d in v2)
        cout << d << endl;
}

```

`f1` is passed as the last argument to `transform`, making `v2` contain the square roots of the corresponding numbers in `v`.

This is the "C way" of providing functions – through function pointers. One of its drawbacks is that a function cannot maintain state. In C++, we can use a function object, known as "functor" – an object masquerading as a function:

```

class SqrtFunctor {
public:
    double operator()(int n) {
        return ::sqrt(n);
    }
};

```

And the code to use it:

```

::transform(begin(v), end(v), begin(v2), SqrtFunctor());

```

No state is maintained in this simple case, but it works because of the overloading of the function call operator; `transform` doesn't care, as long as it's something it can invoke.

This still isn't ideal – in both cases we lose code locality – as the invoked function is somewhere else. Lambdas solve this by embedding the code exactly where it's needed:

```

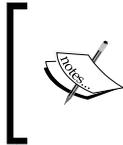
::transform(begin(v), end(v), begin(v2), [](int n) {
    return ::sqrt(n);
});

```

Lambda function syntax may seem strange at first. The syntax has the following ingredients:

- A variable capture list in square brackets (empty in the example)
- The arguments to the function (as expected by its usage), in the previous example is an `int` as that's what `transform` expects, given input iterators that point to a collection of `int` types
- The actual function body
- An optional (and sometimes not so much) return type specifier using some new C++11 syntax:

```
std::transform(begin(v), end(v), begin(v2), [](int n) -> double {  
    return ::sqrt(n);  
});
```



The `std::begin` and `std::end` functions are new to C++11, providing a somewhat more convenient equivalent to the `begin` and `end` member functions of containers. These also work with WinRT collections, as we'll see later.

There are two things to gain when using lambdas:

- Code locality is maintained.
- Possibility to use outer scope variables by "capturing" them inside the lambda. If it were a separate function, it would be difficult to "transfer" the values for the outer scope variables.

Capturing variables can be done by value or by reference. Here are a few examples:

- `[=]` captures all outer scoped variables by value
- `[x, y]` captures `x` and `y` by value
- `[&]` captures all outer scope variables by reference
- `[x, &y]` captures `x` by value, `y` by reference
- Without capturing, the lambda function body can only use the arguments provided and its own declared variables

We'll use lambdas extensively, especially when dealing with asynchronous operations, as we'll see later in this chapter.

Smart pointers

Smart pointers are not a language feature, but rather part of the new standard library. First introduced by boost, they provide automatic management of dynamically allocated objects. Consider this simple object allocation:

```
Car* pCar = new Car;
```

This is a very typical dynamic allocation. The problem with that is, well, it must be de-allocated at some point. This may seem easy enough, given the following statements:

```
pCar->Drive(); // use the car
delete pCar;
```

Even this seemingly simple code is problematic; what if the call to `Car::Drive` throws an exception? In that case, the call to `delete` is skipped and we have ourselves a memory leak.

The solution? Wrapping the pointer by an automatically allocated object, for which the constructor and destructor do the right thing:

```
class CarPtr {
public:
    CarPtr(Car* pCar) : _pCar(pCar) { }
    Car* operator->() const { return _pCar; }
    ~CarPtr() { delete _pCar; }

private:
    Car* _pCar;
};
```

This is known as **Resource Acquisition Is Initialization (RAII)**. The `operator->` ensures accessing the `Car` instance is transparent, making the smart pointer smart enough not to interfere with the normal operations of the car:

```
CarPtr spCar(pCar);
spCar->Drive();
```

The destructor takes care of destroying the object. If an exception is thrown, the destructor is called no matter what (except catastrophic power failures and the like), ensuring the `Car` instance is destroyed before a `catch` handler is searched.

The `CarPtr` class is a very simple smart pointer, but it still may be useful at times. C++11 has a generic implementation of this idea in the form of the `std::unique_ptr<T>` class, where `T` is the type whose pointer is to be managed. In our case, we could have written the `Car` client code like so (we need to `#include <memory>` for this):

```
unique_ptr<Car> spCar2(new Car);
spCar2->Drive();
```



The actual definition of `unique_ptr<>` is more complex than the simple `CarPtr` shown here. For example, what about the assignment operator with a different `Car` object pointer? What about assignment to `nullptr`? Why should the destructor call `delete`—maybe the object was allocated in some other way? These and other details are handled by `unique_ptr<>` correctly.

`unique_ptr<>` is simple enough, but what about objects that need to be passed around? Destroying the object in the destructor of the `unique_ptr` would be premature. For that, we need reference counting, so that when an object is passed to some function (or more interestingly, another thread), a counter should increment. When the smart pointer's destructor is called, it should decrement the counter, and only if it reaches zero, should it actually destroy the object. That's exactly the role of another new smart pointer class, `shared_ptr<T>`. Here's an example with a `Car` object:

```
void UseCar(shared_ptr<Car> car) {
    // ref count: 2
    car->Drive();
}

void CreateCar() {
    shared_ptr<Car> spCar3(new Car); // ref count: 1
    UseCar(spCar3);                // ref count: 2
    // back to ref count of 1
    spCar3->Drive();
}
```

The nice thing about `shared_ptr<>` is that it works on any type, the type does not need to have any special properties. `shared_ptr<>` allocates an extra reference count that is associated with the provided object.

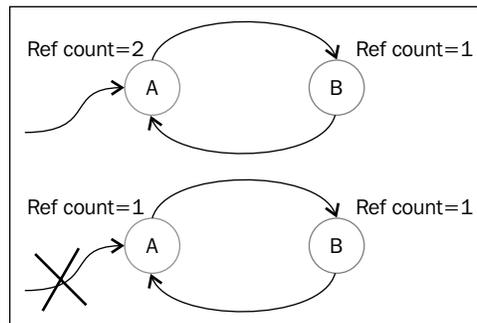
The preferred way to initialize `shared_ptr<>` is by using the `std::make_shared<>` template function, that accepts parameters passed to the actual type's constructor. It creates the object instance (for example `Car`) along with a reference counter, all in one block, hence the extra efficiency. In the `Car` example, this looks as follows:

```
shared_ptr<Car> spCar3 = make_shared<Car>();
spCar3->Drive();
```

[


]
 It's important not to mix smart pointers (like `shared_ptr<>`) with ordinary pointers, else the object may be destroyed while regular pointers to it are still used by other code fragments.

One of the caveats of reference counting is the problem of cyclic references. For example, if some code creates object A, that creates an object B in its constructor and passes itself to B, where B holds a smart pointer back to A, and at some point the client lets go of its A smart pointer, the cycle will leave A and B alive forever:



The original client doesn't even know that a memory leak has occurred. This is something to be aware of, and as we'll see later, the same problem exists with WinRT objects, which are reference counted as well.

One way to avoid the issue is to use another smart pointer class from C++11, `std::weak_ptr<>`. As the name suggests, it holds a weak reference to the object, meaning it won't prevent the object from self-destruction. This would be the way B would hold a reference to A in the previous diagram. If that's the case, how can we access the actual object in case we need it? Or, more precisely, how do we know it actually still exists? Here's the way to do it:

```

shared_ptr<Car> spCar3 = make_shared<Car>();
spCar3->Drive();

weak_ptr<Car> spCar4(spCar3);

auto car = spCar4.lock();
if(car)
    car->Drive();
else
    cout << "Car gone" << endl;
  
```

The `weak_ptr<>::lock` function returns a `shared_ptr<>` to the object in question. If there is no object, the internal pointer would be null. If there is an object, then its ref count is incremented, protecting it from premature destruction even if the original `shared_ptr<>` is released. Adding the following line after the initialization of `spCar4` would show **Car gone** on the display:

```
spCar3 = nullptr;
```



There is another way to break the cycle. A can implement a specific method (for example `Dispose`), that must be called explicitly by the client. In that method, A will release its pointer to B, thus breaking the cycle. The problem here is somewhat similar to manually using `new/delete` – the function needs to be called at the correct time. If called too early it will make the object unusable; `weak_ptr<>` is usually preferred.

In today's C++, the recommended approach is to always use smart pointers and never use raw pointers. Using the `new` and `delete` operators is considered a maintenance headache, that could lead to memory leaks or corruptions because of premature object destruction. Smart pointers are cheap to pass around and guarantee correct behavior even in the face of exceptions.

Conclusion

C++11 has a lot of new features, both in the language and in the standard libraries. We've seen some of the useful ones, but there are certainly others, such as `rvalue` references that provide a way to avoid costly copying operations (and, in fact, are used in the container classes such as `std::vector<>`), a new `enum class` declaration that solves the outer scope problem of classic enums, and many others.



For a complete treatment of the new C++11 features, use web resources such as Microsoft's Channel 9 (<http://channel9.msdn.com>), the Visual C++ team's blog (<http://blogs.msdn.com/b/vcblog/>), and Wikipedia (<http://en.wikipedia.org/wiki/C%2B%2B11>). Also, the almost-final draft of the C++11 standard can be found at <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf>.

COM and WinRT

The COM technology was created by Microsoft at around 1993. It was first named OLE 2.0, because it was used to implement the **Object Linking and Embedding (OLE)** feature in the Microsoft Office suite. This feature allowed, for example, embedding (or linking) an Excel table inside a Word document. The first version of OLE (known as OLE 1.0) was implemented by something called **Dynamic Data Exchange (DDE)**, which is a long time Windows feature that is based on messaging. Microsoft realized that OLE is just one possible use of a more general technology, and so renamed OLE 2.0 to COM.

COM contains many details, but is based on very few principles:

- Client programs against interfaces, never against concrete objects
- Components are loaded dynamically rather than statically

What is an interface? In object-oriented parlance, an interface is an abstraction that groups a set of related operations. This abstraction has no implementation, but various types may implement the interface in some appropriate way. A client can use different implementations because it relies on the interface (being a contract) alone and not on any particular implementation that may be provided indirectly, for example, by some factory component.

A COM interface is something more precise. It specifies a particular binary layout that is provided by the implementer of the interface and consumed by the client. Since the layout is known in advance, the contract presented is not just logical, but also a binary one. This leads to the possibility of mixing languages or technologies when using COM. A COM class can be authored in (say) C++, but consumed by Visual Basic or C#, assuming those languages (in this case the .NET platform) know the binary layout of the interfaces in question.

The layout of a COM interface is a virtual table (also known as V-table), which is the most common mechanism of implementing virtual functions in C++, making C++ a natural choice for developing COM components and COM clients. Here's a definition of a simple interface in C++, as a pure abstract class:

```
class ICar {
public:
    virtual void Drive() = 0;
    virtual void Start() = 0;
    virtual void Refuel(double amount) = 0;
};
```

 By convention, interface names in COM start with a capital "I" and then a name in Pascal casing (for example IAnimal, ILibrary, IObjectBuilder).

Here's a simple (inline) implementation of this interface:

```
class Porche : public ICar {
public:
    Porche() : _running(false), _fuel(50), _speed(0) { }

    void Start() {
        if(_running)
            throw new std::exception(
                "car is already running");
        _running = true;
    }

    void Drive() {
        if(!_running)
            throw new std::exception("car is not running");
        _speed += 10;
    }

    void Refuel(double amount) {
        if((_fuel += amount) > 60)
            _fuel = 60;
    }

private:
    bool _running;
    double _fuel;
    double _speed;
};
```

We can use any ICar interface pointer without knowing anything about the actual implementation:

```
void UseCar(ICar* pCar) {
    pCar->Start();
    pCar->Drive();
    pCar->Drive();
    pCar->Refuel(30);
}
```



```

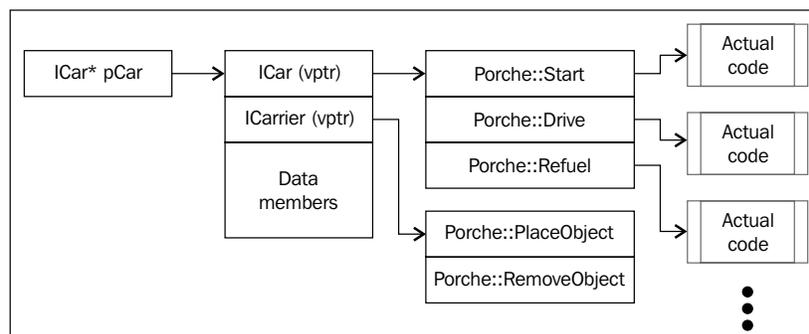
void RemoveObject(int objectID) {
    auto obj = _objects.find(objectID);
    if(obj == _objects.end())
        throw new std::exception("object not found");
    _totalWeight -= obj->second;
    _objects.erase(obj);
}

```

The exact implementation is not important in itself at this time, just the layout of the object in memory:

car	{_totalWeight=0.0000000000000000 _objects={ size=0 }_running=false ...}	Porche
ICar	{...}	ICar
_vfptr	0x003d0bf0 {InterfacesDemo.exe!const Porche::vftable{for 'ICar'}} {0x003c1384 {Interf: void **	
[0]	0x003c1384 {InterfacesDemo.exe!Porche::Drive(void)}	void *
[1]	0x003c114a {InterfacesDemo.exe!Porche::Start(void)}	void *
[2]	0x003c135c {InterfacesDemo.exe!Porche::Refuel(double)}	void *
ICarrier	{...}	ICarrier
_vfptr	0x003d0c04 {InterfacesDemo.exe!const Porche::vftable{for 'ICarrier'}} {0x003c1037 {Int void **	
[0]	0x003c1037 {InterfacesDemo.exe!Porche::PlaceObject(double)}	void *
[1]	0x003c117c {InterfacesDemo.exe!Porche::RemoveObject(int)}	void *
_totalWeight	0.0000000000000000	double
_objects	{ size=0 }	std::map<int,doub
_running	false	bool
_fuel	50.0000000000000000	double
_speed	0.0000000000000000	double

The first two members of the `Porche` instance are the v-table pointers to `ICar` and `ICarrier` (in that order), each of which points to a virtual table of function pointers. Only then the instance member variables are placed. Here's a diagram to show this perhaps more clearly:

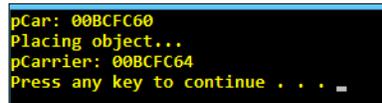


Now, suppose a client holds an `ICar*` interface and wants to see if `ICarrier` is implemented by the object. Doing a C-style case (or `reinterpret_cast<>`) is going to simply make the same pointer value think it's pointing to another v-table, but it's not. In this case invoking `ICarrier::PlaceObject` would actually invoke `ICar::Start` because that's the first function in that v-table; and functions are invoked by offset.

What we need is to query dynamically whether another interface is supported using the `dynamic_cast<>` operator:

```
cout << "pCar: " << pCar << endl;
ICarrier* pCarrier = dynamic_cast<ICarrier*>(pCar);
if(pCarrier) {
    // supported
    cout << "Placing object..." << endl;
    int id = pCarrier->PlaceObject(20);
    cout << "pCarrier: " << pCarrier << endl;
}
```

If successful, `dynamic_cast` adjusts the pointer to the correct v-table. In the `Porche` case, the value of `pCarrier` should be greater than `pCar` by pointer size (4 bytes in a 32-bit process, 8 bytes in a 64-bit process). We can verify that by printing their values:



```
pCar: 00BCFC60
Placing object...
pCarrier: 00BCFC64
Press any key to continue . . . _
```

The offset is 4, since this code was compiled as 32 bit.

The problem with `dynamic_cast<>` is that it's C++ specific. What would other languages do to get another interface on an object? The solution is to factor that functionality into every interface. Coupled with reference counting, this leads to the most fundamental interface in the world of COM, `IUnknown`.

The IUnknown interface

The `IUnknown` interface is the base interface of every COM interface. It encapsulates two pieces of functionality: querying for other interfaces that may be supported and managing the object's reference count. Here's its definition:

```
class IUnknown {
public:
    virtual HRESULT __stdcall QueryInterface(const IID& iid,
        void **ppvObject) = 0;
    virtual ULONG __stdcall AddRef() = 0;
    virtual ULONG __stdcall Release() = 0;
};
```

`QueryInterface` allows getting another supported interface, based on the interface ID, which is a **Global Unique Identifier (GUID)** – a 128-bit number that's generated by an algorithm that statistically guarantees uniqueness. The returned value, an `HRESULT` is the standard return type in COM (and WinRT). For `QueryInterface`, `S_OK` (0) means all is well and the requested interface exists (and is returned indirectly via the `ppvObject` argument) or `E_NOINTERFACE`, meaning no such interface is supported.



All COM/WinRT interface methods are using the standard calling convention (`__stdcall`), which says the callee is responsible for cleaning the parameters on the call stack (rather than the caller). This matters in the 32-bit world, which has several calling conventions. Since COM is intended for cross technology access, this is part of the contract (in x64 only one calling convention exists, so this is not that important).

`AddRef` increments the internal reference count of the object and `Release` decrements it, destroying the object if the count reaches zero.



Keep in mind that this is just an interface, and other implementations are possible. For example, `AddRef` and `Release` may do nothing for an object that always wants to remain in memory, such as a singleton. Most objects, however, are implemented as described.

Any COM interface must derive from `IUnknown`; this means that every v-table has at least three entries corresponding to `QueryInterface`, `AddRef`, and `Release` (in that order).

Inspectable interface

WinRT can be viewed as a better COM. One of the shortcomings of the `IUnknown` interface is that there is no standard way to ask the object to give back a list of interfaces it supports. WinRT defines a new standard interface, `IInspectable` (that of course derives from `IUnknown`) that provides this capability:

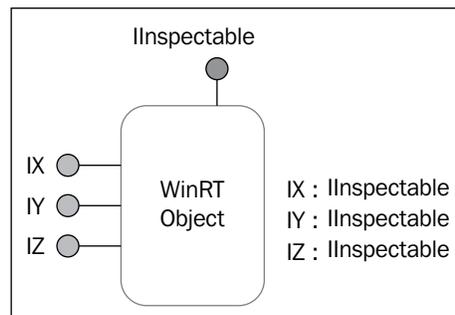
```
class IInspectable : public IUnknown {
public:
    virtual HRESULT __stdcall GetIids(ULONG *iidCount,
        IID **iids) = 0;
    virtual HRESULT __stdcall GetRuntimeClassName(
        HSTRING *className) = 0;
    virtual HRESULT __stdcall GetTrustLevel(
        TrustLevel *trustLevel) = 0;
};
```

The most interesting method is `GetIids`, returning all interfaces supported by the object. This is used by the JavaScript engine running on top of WinRT because of the lack of static typing in JavaScript, but it's not generally useful for C++ clients.

The net result of all this is the following:

- Every WinRT interface must inherit from `IInspectable`. This means every v-table always has at least six entries corresponding to the methods `QueryInterface`, `AddRef`, `Release`, `GetIids`, `GetRuntimeClassName`, and `GetTrustLevel` (in that order).
- A WinRT type implements at least `IInspectable`, but will almost always implement at least another interface; otherwise, this object would be very dull.

The following classic diagram depicts a WinRT object:



Creating a WinRT object

As we've seen, COM/WinRT clients use interfaces to call operations on objects. However, one thing was eluded thus far, how did that object come to be? The process of creation must be generic enough (and not C++ specific), so that other technologies/languages would be able to utilize it.

We'll build a simple example that creates an instance of the WinRT `Calendar` class residing in the `Windows::Globalization` namespace, and call some of its methods. To remove all possible noise, we'll do that in a simple Win32 console application (not a Windows 8 Store app), so that we can focus on the details.



The last sentence also means that WinRT types (most of them, anyway) are accessible and usable from a desktop app, as well as a Store app. This opens up interesting possibilities.

We need to use some new APIs that are part of the Windows Runtime infrastructure. These APIs start with the letters "Ro" (short for Runtime Object). To that end, we'll need a specific `#include` and to link with the appropriate library:

```
#include <roapi.h>

#pragma comment(lib, "runtimeobject.lib")
```

Now, we can start implementing our main function. The first thing to do is to initialize WinRT on the current thread. This is accomplished using the `RoInitialize` function:

```
::RoInitialize(RO_INIT_MULTITHREADED);
```

`RoInitialize` requires specifying the apartment model for the thread. This can be a **Single Threaded Apartment (STA)** denoted by `RO_INIT_SINGLETHREADED` or the **Multithreaded Apartment (MTA)** denoted by `RO_INIT_MULTITHREADED`. The concept of apartments will be discussed a bit later, and is unimportant for the current discussion.

 `RoInitialize` is similar in concept to the classic COM `CoInitialize` (Ex) functions. WinRT apartments are pretty much the same as the classic COM apartments. In fact, since WinRT is built upon COM foundations, most things work very similarly. The object creation mechanism is very similar, with some changes to the details as we'll soon see.

To create an actual object and get back an interface pointer, a call must be made to the `RoActivateInstance` API function. This function is prototyped as follows:

```
HRESULT WINAPI RoActivateInstance(
    _In_   HSTRING activatableClassId,
    _Out_ IInspectable **instance
);
```

The first argument that's needed is the full name of the class, represented as an `HSTRING`. `HSTRING` is the standard WinRT string type, and represents an immutable array of Unicode (UTF-16) characters. Several WinRT APIs exist for creating and manipulating `HSTRING`. As we'll see a bit later, C++/CX provides the `Platform::String` class to wrap an `HSTRING` for ease of use.

The second argument to `RoActivateInstance` is the resulting instance represented through an `IInspectable` interface pointer (recall that all WinRT objects must support this interface).



Interested readers may be wondering why create a new string type. Surely, there are already plenty of those in the Microsoft space: `std::string/wstring` (C++), `CString` (ATL/MFC), and `BSTR` (COM). `BSTR` seems the most likely candidate as it's not C++ specific. The new `HSTRING` is immutable, meaning it cannot change once created. Any apparent modification creates a new `HSTRING`. This attribute makes the `HSTRING` thread safe and more easily projectable to other platforms, such as .NET, where the `System.String` class is immutable as well, so there is no mismatch.

To use the `HSTRING`-related APIs, we'll add `#include` to `<winstring.h>`. Now we can go ahead and create an `HSTRING` for the `Calendar` class:

```
HSTRING hClassName;
wstring className(L"Windows.Globalization.Calendar");
HRESULT hr = ::WindowsCreateString(className.c_str(),
    className.size(), &hClassName);
```

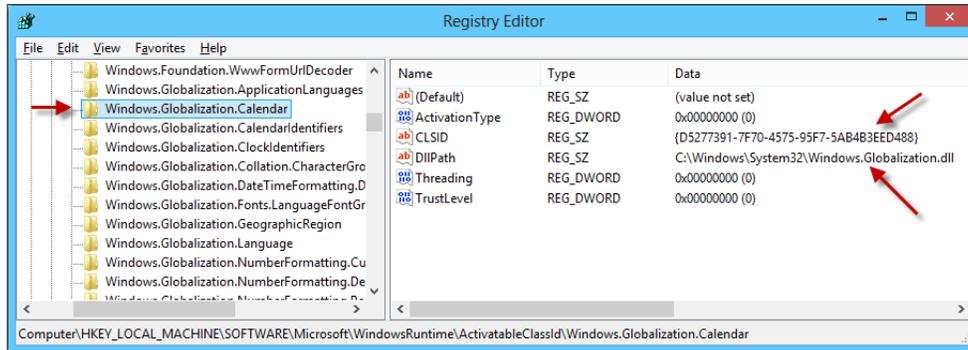
`HSTRING` is created with the `WindowsCreateString` WinRT API, passing the string literal and its length (here acquired with the help of `std::wstring`). Note, that the class name includes its full namespace where a dot (.) is the separator (and not the C++ scope resolution operator `::`).

Now, we can call `RoActivateInstance` (I've omitted any error checking in these code fragments so we can concentrate on the essentials) and get back an interface for the `Calendar`. Since this is `IInspectable`, it's not very interesting. We need a more specific interface for the `Calendar`, that is, we need to call `QueryInterface` to get a more interesting interface to work with.

What does `RoActivateInstance` do? How is that instance actually created? Where is it implemented?

The procedure is very similar to the classic COM creation mechanism.

`RoActivateInstance` consults the Registry at `HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsRuntime\ActivatableClassId` and looks for a key with the name **Windows.Globalization.Calendar**. Here's a screenshot from `RegEdit.exe`:



The screenshot shows the 64-bit key. For 32-bit processes, the key is under `HKLM\Software\Wow6432Node\Windows\WindowsRuntime\ActivatableClassId`. This is transparent to the process, as the Registry APIs, by default, go to the correct location based on the process "bitness".

Several values exist in the class **Name** key. The most interesting are:

- **DllPath** - indicates where the implementing DLL resides. This DLL is loaded into the calling process address space, as we'll see in a moment.
- **CLSID** - the corresponding GUID of the class name. It's not as important as in classic COM, because WinRT implementations are identified by the full class name and not the CLSID, as is evident by the first argument to `RoActivateInstance`.
- **ActivationType** - indicates whether this class is activated in-process (DLL, value of 0) or out-of-process (EXE, value of 1).

For the rest of this discussion, we'll assume an in-process class. `RoActivateInstance` calls another function, `RoGetActivationFactory`, that does the actual work of locating the Registry key and loading the DLL into the process address space. Then, it calls an exported global function from the DLL named `DllGetActivationFactory` (the DLL must export such a function, otherwise the creation process fails), passing in the full class name, the requested factory interface ID, and an output interface pointer as the result:

```
HRESULT RoGetActivationFactory(
    _In_ HSTRING activatableClassId,
```

```

    _In_   REFIID iid,
    _Out_  void **factory
);

```

That global function inside the DLL is responsible for returning a class factory, capable of creating actual instances. The class factory typically implements the `IActivationFactory` interface, with a single method (beyond `IInspectable`):

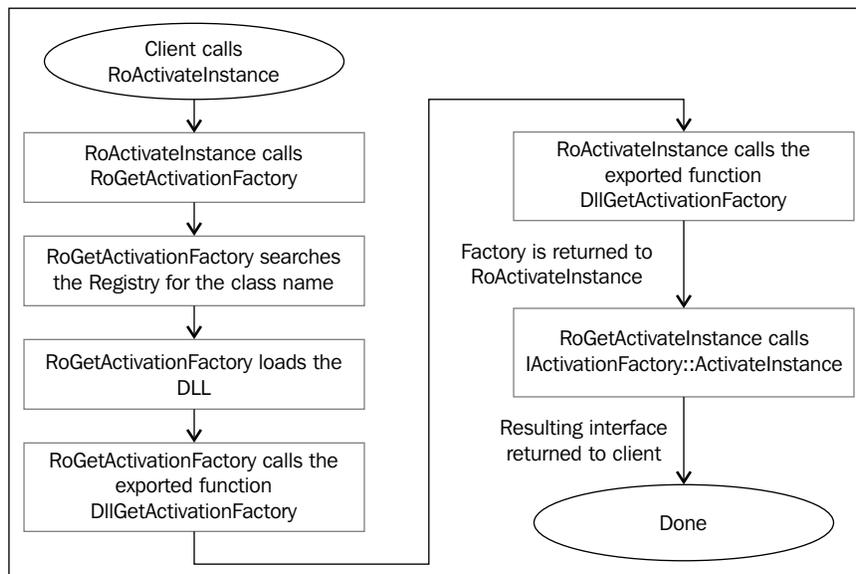
```
HRESULT ActivateInstance(IInspectable **instance);
```

Returning the class factory is the job of `RoGetActivationFactory`. Then `RoActivateInstance` takes over, and calls `IActivationFactory::ActivateInstance` to create the actual instance, which is the result of `RoActivateInstance`.



Readers familiar with classic COM may recognize the similarities: `RoActivateInstance` replaces the classic `CoCreateInstance`; `RoGetActivationFactory` replaces `CoGetClassObject`; `DllGetActivationFactory` replaces `DllGetClassObject`; and finally, `IActivationFactory` replaces `IClassFactory`. Overall, though, the steps are practically the same.

The following diagram sums up the creation process of a WinRT type:





The Registry keys used in this sequence are relevant to desktop apps creating WinRT objects. The keys that a Store app activation uses are different, but the general sequence is the same.

If all goes well, we have an `IInspectable` interface pointer to a `Calendar` instance. But we're interested in a more specific interface that would provide the true capabilities of the `Calendar`.

As it turns out, the definition of the relevant interface is in a header file, named after the namespace where `Calendar` is placed:

```
#include <windows.globalization.h>
```

The interface in question is named `ICalendar` in the `ABI::Windows::Globalization` namespace. We'll add a `using` namespace for easier access:

```
using namespace ABI::Windows::Globalization;
```

Application Binary Interface (ABI) is a root namespace that we'll discuss in a later section.

Since we need an `ICalendar`, we have to `QueryInterface` for it:

```
ICalendar* pCalendar;  
hr = pInst->QueryInterface(__uuidof(ICalendar),  
    (void**) &pCalendar);
```

`pInst` is assumed to be some interface on an object (such as `IInspectable`). If that interface is indeed supported, we'll get back a successful `HRESULT (S_OK)` and an interface pointer we can use. The `__uuidof` operator returns the **interface ID (IID)** of the interface in question; this is possible because of a `__declspec(uuid)` attribute attached to the declared interface.

Now, we can use the interface in any way we see fit. Here are some lines that get the current time and display it to the console:

```
pCalendar->SetToNow();  
INT32 hour, minute, second;  
pCalendar->get_Hour(&hour);  
pCalendar->get_Minute(&minute);  
pCalendar->get_Second(&second);  
  
cout << "Time: " << setfill('0') << setw(2) << hour << ":" <<  
    setw(2) << minute << ":" << setw(2) << second << endl;
```

At this time, the reference count on the `Calendar` instance should be 2. To clean up properly, we need to call `IUnknown::Release` on any obtained interface pointer (when created the ref count was 1, and after `QueryInterface` it became 2); also, since we created an `HSTRING`, it's a good idea to destroy it; finally we'll uninitialized WinRT on the current thread for good measure:

```
pCalendar->Release();
pInst->Release();
::WindowsDeleteString(hClassName);
```

The complete code can be found in the `WinRTAccess1` project, part of the downloadable code for this chapter.

WinRT metadata

The preceding example used the `<windows.globalization.h>` header file to discover the declaration of the `ICalendar` interface, including its IID. However, since COM/WinRT is supposed to provide interoperability between languages/platforms, how would a non-C++ language be able to use that header file?

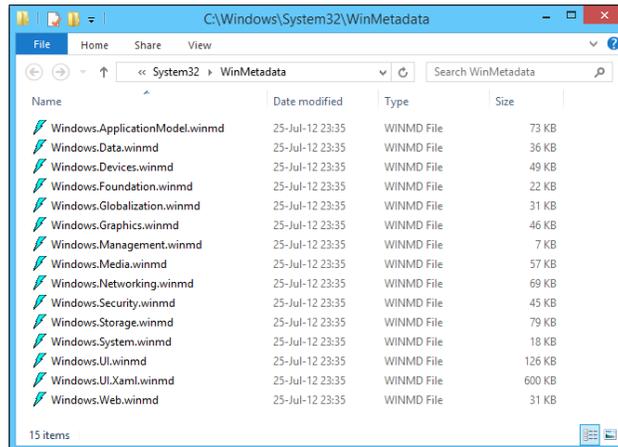
The answer is that other languages can't use that header file; it's specific to C/C++. What we need is a kind of "universal header file", based on a well-defined structure and, thus, usable by any platform. This is the role of metadata files.

The format of metadata files (with extension `.winmd`) is based on the metadata format created for `.NET`. This was simply convenient, as that format is rich, providing all the necessary ingredients for WinRT metadata as well.

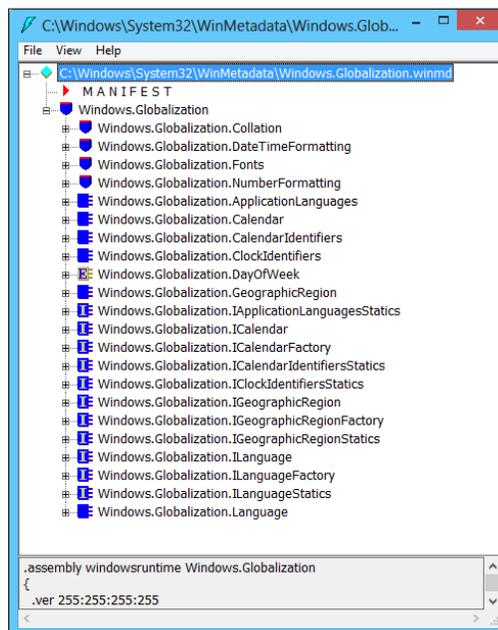


In classic COM this metadata was stored in a type library. Type library format is not as rich as the `.NET` metadata format, and so wasn't used for WinRT.

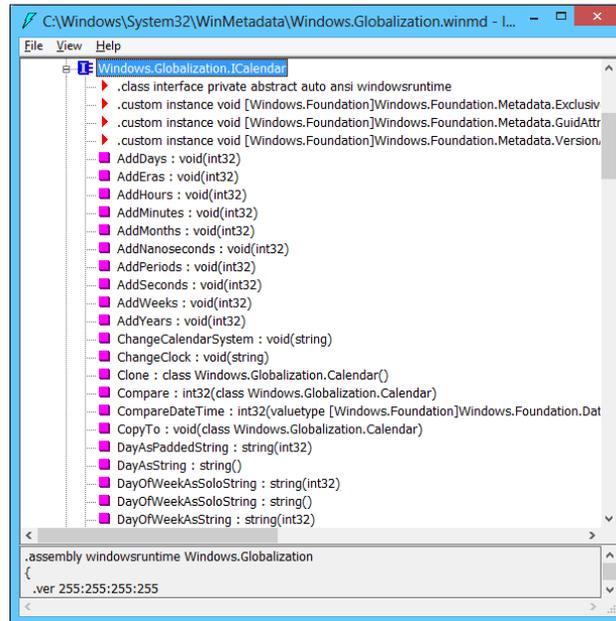
The WinRT metadata files reside in the %System32%\WinMetadata folder, and they are conveniently arranged based on namespaces (in fact, that's a requirement). Here are the files on my machine:



To view a metadata file, we can use any (relatively new) tool that is capable of showing .NET metadata, such as IL Disassembler (ILDasm.exe) from the Visual Studio 2012 tools, or Reflector (<http://www.reflector.net/>). Opening `Windows.Globalization.winmd` in ILDasm.exe shows this:

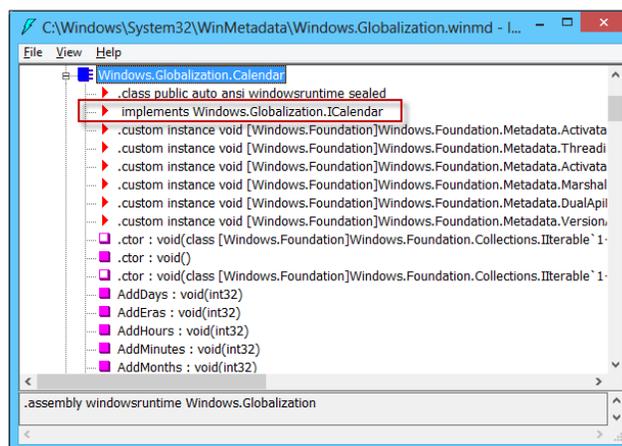


We can see all the classes and interfaces defined in the metadata file. Expanding the `ICalendar` interface node shows its members:



Double-clicking a method does not show its implementation, as it's not really .NET; there's no code there, it's just its metadata format.

What about the `Calendar` class? Expanding its node shows that it implements `ICalendar`. This gives anyone using the metadata (humans, as well as tools) the confidence to `QueryInterface` for this interface with a successful result:



These metadata files are the result of building WinRT components. This way, any platform that understands the metadata format, can consume the classes/interfaces exposed by that component. We'll see an example later in this chapter.

The Windows Runtime Library

The Calendar usage example worked, but the code required was pretty verbose. The **Windows Runtime Library (WRL)** is a set of helper classes and functions that make it easier to work with WinRT types, both as a client and as a server (creator of components). WRL uses standard C++ (no non-standard extensions), keeping things pretty close to the metal. Let's see how we can simplify the Calendar example by using WRL.

First, we need to include the WRL headers; there is a main header and a helper with some convenient wrappers:

```
#include <wrl.h>
#include <wrl/wrappers/corewrappers.h>
```

Next, we'll add some using statements to shorten the code:

```
using namespace Windows::Foundation;
using namespace Microsoft::WRL;
using namespace Microsoft::WRL::Wrappers;
```

In `main()`, we first need to initialize WinRT. A simple wrapper calls `RoInitialize` in its constructor and `RoUninitialize` in its destructor:

```
RoInitializeWrapper init(RO_INIT_MULTITHREADED);
```

To create and manage an `HSTRING`, we can use a helper class, `HString`:

```
HString hClassName;
hClassName.Set(RuntimeClass_Windows_Globalization_Calendar);
```

The long identifier is the full Calendar class name defined in `<windows.globalization.h>`, so we don't have to provide the actual string. `HString` has a `Get()` member function that returns the underlying `HSTRING`; its destructor destroys the `HSTRING`.



The preceding code could actually be simplified (and sped up) by using an HSTRING that references an existing string, thus preventing actual string allocation and copying. This is done with the `HString::MakeReference` static function that internally calls `WindowsCreateStringReference`. It effectively removes the need to destroy the HSTRING, because there was nothing allocated in the first place. This string reference is also known as "fast pass".

Creating the Calendar instance can be simplified by calling the `Windows::Foundation::ActivateInstance` template function that internally calls `RoActivateInstance` and queries for the requested interface:

```
ComPtr<ICalendar> spCalendar;
HRESULT hr = ActivateInstance(hClassName.Get(), &spCalendar);
```

`ComPtr<T>` is WRL's smart pointer for WinRT interfaces. It calls `Release` correctly in its destructor, and provides the necessary operators (such as `->`) so that it's invisible enough when accessing the underlying interface pointer. The rest of the code is pretty much the same, although no cleanup is necessary, as destructors do the right thing:

```
spCalendar->SetToNow();
INT32 hour, minute, second;
spCalendar->get_Hour(&hour);
spCalendar->get_Minute(&minute);
spCalendar->get_Second(&second);

cout << "Time: " << setfill('0') << setw(2) << hour << ":" <<
      setw(2) << minute << ":" << setw(2) << second << endl;
```

`ComPtr<T>` is similar in spirit to ATL's smart pointers, `CComPtr<T>` and `CComQIPtr<T>`. `ComPtr<T>` can do `QueryInterface` by calling the `As` member function. Technically, `CComPtr<T>` can be used with WinRT as well.

WRL also provides classes that help implement WinRT components by implementing boilerplate code, such as `IInspectable`, activation factories, and so on. We will generally use C++/CX to create components, but WRL can be used if low-level control is required or language extensions are undesirable.



There is no project template that is installed by default with Visual Studio 2012 to create WinRT components with WRL; however, such a template was created by Microsoft, and is available by searching online when invoking the **Tools | Extensions and Updates** menu item. This gives a decent starting point for creating a WinRT DLL component. The steps involved are somewhat similar to the steps used to create classic COM components with ATL defining interfaces and members in an **Interface Definition Language (IDL)** file, and implementing the required functionality.

C++/CX

WRL simplifies using and accessing WinRT objects, but it's still a way to go from the normal C++ experience when creating and using objects. Calling the `new` operator is far easier than using `Windows::Foundation::ActivateInstance` and working with a `ComPtr<T>` smart pointer.

To this end, Microsoft has created a set of extensions to the C++ language, called C++/CX that help to bridge the gap, so that working with WinRT objects is almost as simple as working with non-WinRT objects.

The following sections discuss some of the more common extensions. We'll discuss more extensions throughout the book. First, we'll look at creating objects, then we'll examine various members and how to access them, and finally, we'll consider the basics of creating new WinRT types with C++/CX.

Creating and managing objects

WinRT objects are instantiated in C++/CX by the keyword `ref new`. This creates a reference counted object (a WinRT object) and returns a handle to the object using the `^` (hat) notation. Here's an example of creating a `Calendar` object:

```
using namespace Windows::Globalization;
Calendar^ cal = ref new Calendar;
```

The returned value in `cal` is a WinRT object. One thing that may be puzzling is that we're getting back a `Calendar` object and not an interface; but COM/WinRT clients can work with interfaces only; and where is `ICalendar` we were using before?

C++/CX provides a layer of convenience that allows using an object reference rather than an interface reference. However, the interface `ICalendar` is still there, and in fact that's defined as the default interface for the `Calendar` class (the compiler is aware of that), but using the class directly seems more natural. We can verify that by adding a method call and looking at the generated code after adding a specific cast to `ICalendar` and comparing it to the original call:

```
Calendar^ cal = ref new Calendar;
cal->SetToNow();

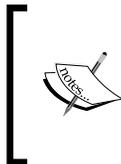
ICalendar^ ical = cal;
ical->SetToNow();
```

Here is the generated code for these calls:

```
cal->SetToNow();
00A420D0 mov     eax,dword ptr [cal]
00A420D3 push    eax
00A420D4 call    Windows::Globalization::ICalendar::SetToNow
(0A37266h)
00A420D9 add     esp,4

ICalendar^ ical = cal;
00A420DC mov     eax,dword ptr [cal]
00A420DF push    eax
00A420E0 call    __abi_winrt_ptr_ctor (0A33094h)
00A420E5 add     esp,4
00A420E8 mov     dword ptr [ical],eax
00A420EB mov     byte ptr [ebp-4],0Ah
    ical->SetToNow();
00A420EF mov     eax,dword ptr [ical]
00A420F2 push    eax
00A420F3 call    Windows::Globalization::ICalendar::SetToNow
(0A37266h)
```

The highlighted portions are the same, proving that the actual call goes through an interface.



Readers familiar with C++/CLI, the C++ extensions for .NET, may recognize the "hat" (^) and some other similar keywords. This is simply syntax borrowed from C++/CLI, but it has nothing to do with .NET. All WinRT stuff is pure native code, whether it's accessed using C++/CX or not.

When the hat variable goes out of scope, `IUnknown::Release` is called automatically as expected. It's also possible to use stack semantics with WinRT types, like so:

```
Calendar c1;  
c1.SetToNow();
```

The object is still allocated dynamically in the usual way. But it's guaranteed to be cleaned up when the variable goes out of scope. This means that it cannot be passed to other methods.

Accessing members

After obtaining a reference to a WinRT object (or interface), members can be accessed with the arrow (`->`) operator, just like a regular pointer. Note, however, that the hat is not a pointer in the normal sense; for example, no pointer arithmetic is ever possible. The hat variable should be thought of as an opaque reference to a WinRT object.

Accessing members through the reference is not exactly the same as accessing the object through a direct (or WRL-like) interface pointer. The main difference is error handling. All interface members must return an `HRESULT`; calling through a hat reference hides the `HRESULT` and instead throws an exception (something deriving from `Platform::Exception`) in case of failure. This is usually what we want, so that we can use the standard language facilities `try/catch` to handle errors and not have to check `HRESULT` for every call.

Another difference appears in case a method has a return value. The actual interface method must return an `HRESULT`, and as such adds an output argument (which must be a pointer) where the result would be stored on success. Since hat references hide the `HRESULT`, they make the return type the actual return value of the method call, which is very convenient. Here's an example that uses the `ICalendar::Compare` method to compare this calendar's date/time to another's. Using WRL to create a second calendar and compare looks as follows:

```
ComPtr<ICalendar> spCal2;  
ActivateInstance(hClassName.Get(), &spCal2);  
spCal2->SetToNow();  
spCal2->AddMinutes(5);  
  
int result;  
hr = spCalendar->Compare(spCal2.Get(), &result);
```

The result is obtained by passing the target variable as the last argument to the Compare call. Here's the equivalent C++/CX version:

```
auto cal2 = ref new Calendar;
cal2->SetToNow();
cal2->AddMinutes(5);
int result = cal->Compare(cal2);
```

The HRESULT is nowhere to be found, and the actual result is returned directly from the method call. If an error had occurred, a Platform::Exception (or a derivative) would have been thrown.



What about static methods or properties? These exist, and can be accessed by the familiar ClassName::MemberName syntax. Curious readers may wonder how these are implemented, as COM does not have a notion of static members, everything must be accessed through an interface pointer, implying an instance must exist. The solution selected is to implement the static members on the activation factory (class factory), as it's typically a singleton, effectively giving out the same net result.

Methods and properties

WinRT is striving for object orientation, at least where members are concerned. Methods are member functions, invoked as expected in C++. This was the case with ICalendar::SetToNow(), ICalendar::AddMinutes(), and ICalendar::Compare(), shown previously.

WinRT also defines the notion of properties, which are really methods in disguise. A property can have a getter and/or a setter. Since C++ doesn't have the concept of properties, these are modeled as methods starting with get_ or put_, while C++/CX provides field-like access to the properties for convenience.

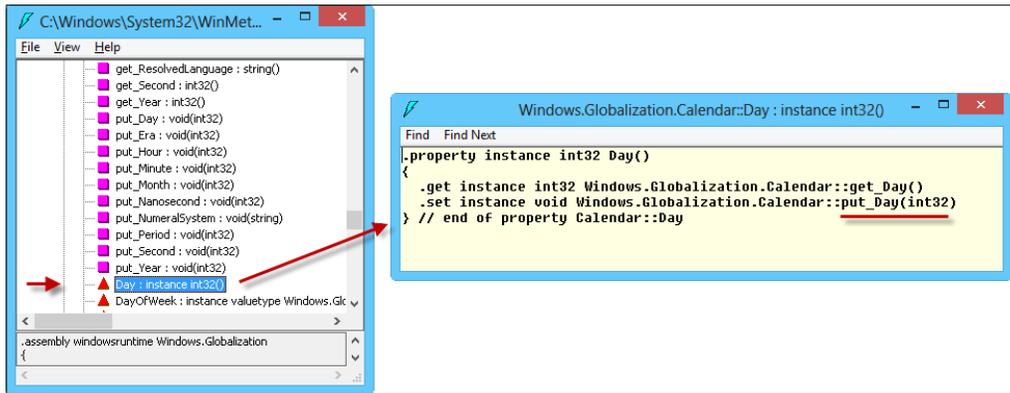
Here's an example that uses the Hour property defined on ICalendar. First, the WRL version:

```
// read current hour
INT32 hour;
spCalendar->get_Hour(&hour);
// set a new hour
spCalendar->put_Hour(23);
```

Next, the C++/CX version:

```
int hour = cal->Hour; // get
cal->Hour = 23;     // set
```

The existence of properties can be seen in the metadata file, in this case `Windows.Globalization.winmd`. Looking at the `Calendar` class (or the `ICalendar` interface), the red triangles indicate properties. Double-clicking any of them shows the following:



It is possible (with C++/CX) to access the actual method or property without the abstraction layer of mapping failed `HRESULT` to exception, if the need arises (this is faster, and is equivalent to WRL generated code). This is done by calling the member prefixed by `__abi_`, as the following code snippet demonstrates:

```
cal->__abi_SetToNow();
int r;
cal->__abi_Compare(cal2, &r);
cal->__abi_set_Hour(22);
```

All these members return `HRESULT` as these are the actual calls through the interface pointer. Curiously enough, property setters must be prefixed by `set_` and not `put_`. This scheme also provides a way to call `IInspectable` methods, such as `GetIids`, which are otherwise inaccessible through a hat reference.

[ Currently, there is no Intellisense for these calls, so red squiggles will show in the editor. The code compiles and runs as expected, though.]

Delegates

Delegates are the WinRT equivalent of function pointers. A delegate is a kind of field that can point to a method. Contrary to function pointers, a delegate can point to a static method or an instance method, as required. Delegates have built-in constructors that accept a method or a lambda function.



The term "delegate" is used because of its similarity to the same concept from the .NET world, where delegates serve much the same purpose as they do in WinRT.

Here's an example with the `IAsyncOperation<T>` interface, which we'll discuss shortly when we look at asynchronous operations. Given an `IAsyncOperation<T>` (`T` is the return type expected from the operation), its `Completed` property is of type `AsyncOperationCompletedHandler<T>`, which is a delegate type. We can hook up the `Completed` property to a member function of the current instance like so:

```
IAsyncOperation<String^>^ operation = ...;
operation->Completed = ref new
    AsyncOperationCompletedHandler<String^>(this, &App::MyHandler);
```

Where `App::MyHandler` is prototyped like so:

```
void MyHandler(IAsyncOperation<String^>^ operation,
    AsyncStatus status);
```

Why this prototype? This is exactly the thing that a delegate defines: a certain prototype that must be followed, else the compiler complains.

As an alternative to a named method, we can bind the delegate to a lambda function, which is more convenient in many cases. Here's the equivalent lambda to the previous code:

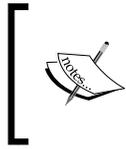
```
operation->Completed = ref new AsyncOperationCompletedHandler<String^>(
    [] (IAsyncOperation<String^>^ operation, AsyncStatus status) {
        // do something...
    });
```

The example captures no variables. The key point here is that the arguments to the lambda are exactly the same as the case with a named method.

What is a delegate really? It's a WinRT class like any other, with a special constructor that allows binding to a method (named or lambda) and an `Invoke` method that actually executes the delegate. In C++/CX, the invocation can be performed by a function call operator `()`, just like any function. Assuming the previous declarations, we can invoke the `Completed` delegate in one of the following ways:

```
operation->Completed->Invoke(operation, AsyncStatus::Completed);
operation->Completed(operation, AsyncStatus::Completed);
```

The two lines are equivalent.



Technically, the preceding code is syntactically correct, but we would never invoke an asynchronous operation completion ourselves. The owner of the operation will do the invocation (we'll look at asynchronous operations later in this chapter).

Events

Delegates are not usually declared as properties as in the `IAsyncOperation<T>::Completed` property. There are two reasons for that:

- Anyone can place `nullptr` in that property (or some other delegate), throwing away any previous delegate instance that might have been set up
- Anyone can invoke the delegate, which is weird, as only the declaring class knows when the delegate should be invoked

What we want is a way to use delegates to connect to interested methods, but do so in a safe way that does not allow arbitrary code to change the delegate directly or to invoke it.

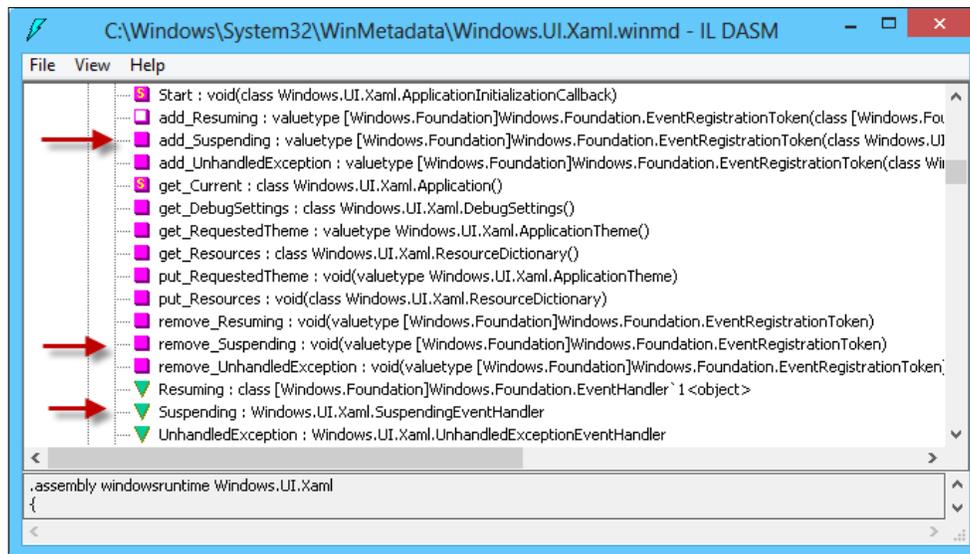
This is where events come in. An event looks like a delegate, but in fact, it has two methods, one for registering a handler for the event and one for revoking the handler. In C++/CX, the `+=` and `-=` operators work on events, so that clients can register for notifications, but can never use the assignment operator to nullify or replace the delegate's value, because it's not exposed in that way.

Here's an example using the `Application::Suspending` event that indicates to the interested parties that the application is about to be suspended, a good time to save a state (we'll discuss the application lifecycle in *Chapter 7, Applications, Tiles, Tasks and Notifications*):

```
this->Suspending += ref new SuspendingEventHandler(  
    this, &App::OnSuspending);
```

Notice that `SuspendingEventHandler` is a delegate type, meaning the method `OnSuspending` must be prototyped in a certain way, as defined by that delegate.

Behind the scenes, an event is just a pair of methods, implemented appropriately (Visual Studio intellisense shows events with a lightning-like icon). Here's a look at the `Application::Suspending` event (other events are shown as well) as described via metadata, shown in `ILDasm.exe`:



The inverted green triangle indicates the event member itself, while the `add_Suspending` and `remove_Suspending` are the actual methods called when the `+=` and `-=` C++/CX operators are used.

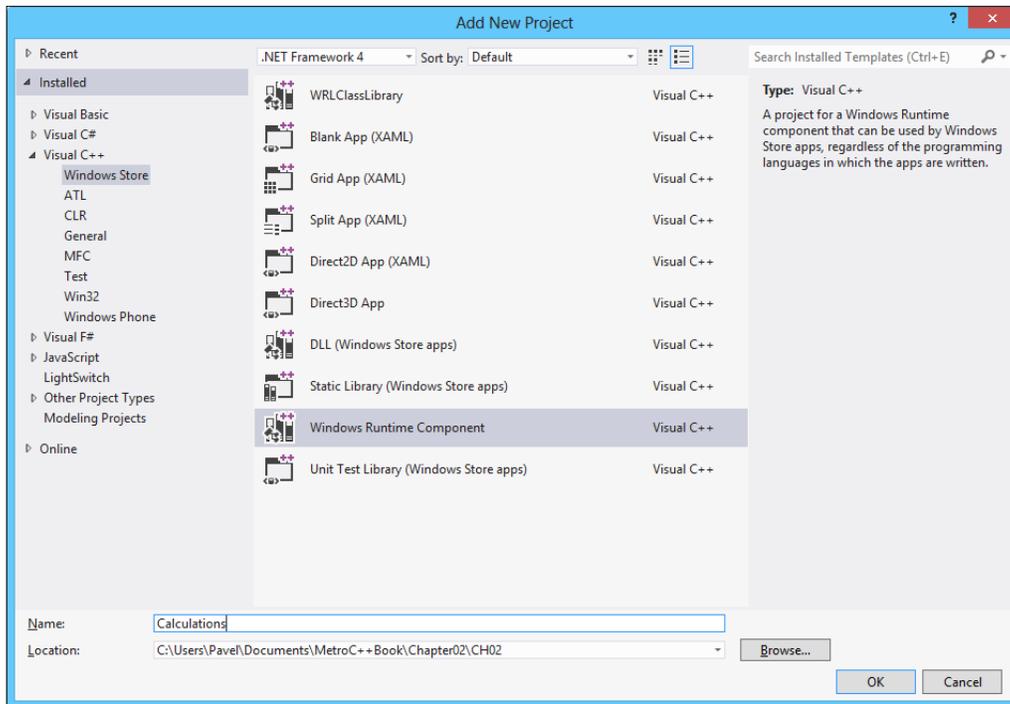
Defining types and members

Defining WinRT types can be done with WRL (by defining interfaces in an IDL file, implementing all boilerplate code, such as `IUnknown` and `IInspectable` implementations, activation factories, DLL global functions, and so on). This provides a very fine-grained way to create components, and is similar in spirit to the way COM components were authored with the **Active Template Library (ATL)**.

With C++/CX, authoring reusable WinRT components is much easier than with WRL. In this section we'll build a simple component, and use it with C++ and C# clients (a JavaScript client would work just as well and is left as an exercise for the interested reader).

A WinRT component project

Visual Studio 2012 includes a project template for creating a WinRT component that can then be used by any WinRT-compliant platform (or another WinRT component). We'll create a new project of type **Windows Runtime Component**, named **Calculations**:



The wizard adds a `Class1` class. We can delete that and add a new C++ class or do the renaming of files and class name. We'll create a WinRT class named `Calculator`, defined with the following code in the header file:

```
namespace Calculations {  
    public ref class Calculator sealed {  
    public:  
        Calculator(void);  
    };  
}
```

A WinRT class must be defined inside a namespace by the `ref class` keyword. It must also be declared `public` so that it would be usable outside the component DLL. The class must also be marked `sealed`, meaning it cannot be inherited from; or, it can inherit from non-sealed classes, which currently are classes provided by the WinRT library residing in the `Windows::UI::Xaml` namespace. A detailed discussion of WinRT inheritance is outside the scope of this section.

Now, it's time to give the class some useful content.

Adding properties and methods

The idea for the `Calculator` class is to be an accumulating calculator. It should hold a current result (starting at zero, by default), and modify the result when new mathematical operations are performed. At any time, its current result can be obtained.

Methods are added as regular member functions, including constructors. Let's add a constructor and a few operations (within the `public` section of the class):

```
// ctor
Calculator(double initial);
Calculator();

// operations
void Add(double value);
void Subtract(double value);
void Multiply(double value);
void Divide(double value);

void Reset(double value);
void Reset();
```

We need a read-only property to convey the current result. Here's how to define it:

```
property double Result {
    double get();
}
```

The `property` keyword is a C++/CX extension that defines a property, followed by its type and its name. Inside the curly braces, `get()` and `set()` methods can be declared (`set` must accept the value with the correct type). The missing `set()` method indicates this is a read-only property—a `get_Result` method would be created, but a `put_Result` method would not.



It's possible to add a simple read/write property backed by a private field by placing a semicolon after the property name (with no curly braces at all).

Next, we add whatever `private` members we need to maintain a proper state; in this simple case, it's just the current result:

```
private:
    double _result;
```

In the CPP file, we need to implement all these members, to get away from **unresolved external linker** errors:

```
#include "Calculator.h"

using namespace Calculations;

Calculator::Calculator(double initial) : _result(initial) {
}

Calculator::Calculator() : _result(0) {
}

void Calculator::Add(double value) {
    _result += value;
}

void Calculator::Subtract(double value) {
    _result -= value;
}

void Calculator::Multiply(double value) {
    _result *= value;
}

void Calculator::Divide(double value) {
    _result /= value;
}

void Calculator::Reset() {
    _result = 0.0;
}
```

```

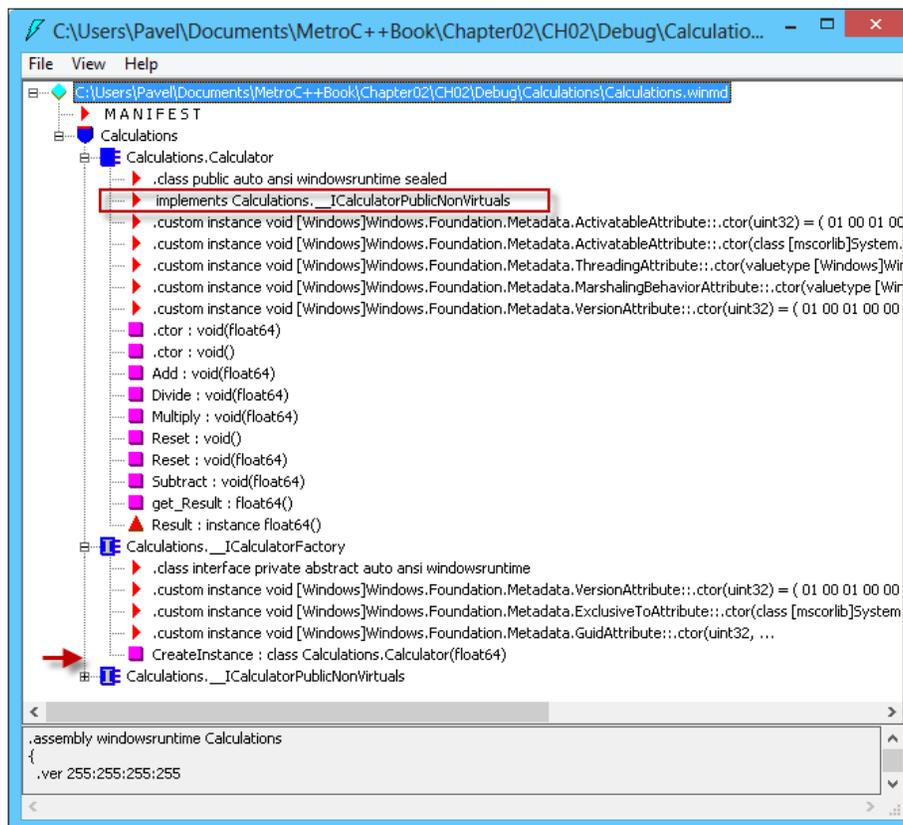
void Calculator::Reset(double value) {
    _result = value;
}

double Calculator::Result::get() {
    return _result;
}

```

There is nothing special in that code, except perhaps the syntax used to implement the `Result` property.

Since this is a WinRT component, a metadata (`.winmd`) file is created as part of the build process; this is the file that will be used to consume the component. Opening it with `ILDasm.exe` shows the result of the code just written:



There are a few interesting points here. Since we've written a WinRT class, it must implement an interface, as WinRT/COM clients can only work with interfaces. In the Calendar case, the interface was named `ICalendar` (which was its default interface), but here we haven't specified any such interface. The compiler created such an interface automatically, and it's named `__ICalculatorPublicNonVirtuals`. This is the actual interface that defines all the methods and properties. The peculiar name hints that these methods are normally only callable from a reference to a `Calculator` object; in any case, the interface name is unimportant.

 Clearly, the `Calendar` class was not created with C++/CX, as its default interface is named `ICalendar`. In fact, it was created with WRL, which allows complete control of every aspect of component authoring, including interface names; WRL was used to build all Microsoft-provided WinRT types.

Another interesting point concerns the overloaded constructors. Since a non-default constructor was provided, the default creation interface, `IActivationFactory` is insufficient, and so the compiler created a second interface, `ICalculatorFactory`, with a `CreateInstance` method accepting a double value. This is another feature that makes C++/CX easy to use – as the burden is on the compiler.

Adding an event

To make it more interesting, let's add an event that fires in case an attempt is made to divide by zero. First, we need to declare a delegate that is appropriate for the event, or use one of the already defined delegates in WinRT.

For demonstration purposes, we'll define a delegate of our own to show how it's done with C++/CX. We add the following declarations just above the `Calculator` definition inside the `Calculations` namespace declaration:

```
ref class Calculator;  
  
public delegate void DivideByZeroHandler(Calculator^ sender);
```

The forward declaration is necessary, as the compiler has not yet stumbled upon the `Calculator` class.

The delegate indicates it can bind to any method that accepts a `Calculator` instance. What should we do with this delegate declaration? We'll add an event that clients can register for. The following declaration is added inside the `public` section of the class:

```
event DivideByZeroHandler^ DivideByZero;
```

This declares the event in the simplest possible way – the compiler implements the `add_DivideByZero` and `remove_DivideByZero` methods appropriately.

Now, we need to update the implementation of the `Divide` method, so that the event fires in case the passed in value is zero:

```
void Calculator::Divide(double value) {
    if(value == 0.0)
        DivideByZero(this);
    else
        _result /= value;
}
```

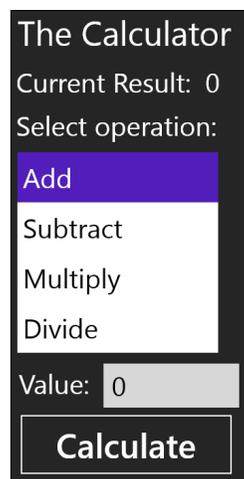
Invoking the event invokes all registered observers (clients) for this event, passing itself as an argument (that may or may not be useful for clients).

Consuming a WinRT component

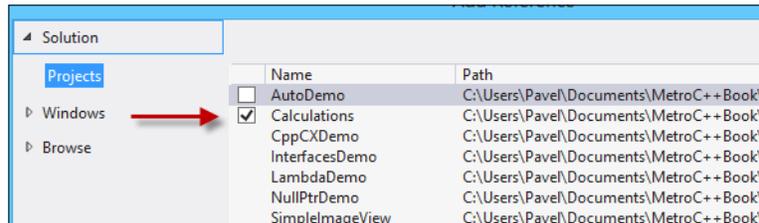
It's time to consume the `Calculator` class we just created. We'll build two clients, a C++ and a C# client, just to show the differences.

Building a C++ client

We'll create a blank C++ Store application project in the same solution and build a simple user interface in XAML to test the functionality of the calculator. The user interface details are unimportant for this discussion; the complete code can be found in the `CalcClient1` project, available in the downloadable code for this chapter. The UI looks like this:



To get the definition of our Calculator, we need to add a reference to the metadata file. This is available by right-clicking on the project node and selecting **References...** In the shown dialog box, we select the **Calculations** project:



Now that the definitions are available, we can use them. In `MainPage.xaml.h`, we add a reference to a Calculator object, so that it exists for the lifetime of the page:

```
private:
    Calculations::Calculator^ _calculator;
```

In the `MainPage` constructor, we need to actually create the instance and optionally connect to the `DivideByZero` event (which we do):

```
_calculator = ref new Calculator;
_calculator->DivideByZero += ref new DivideByZeroHandler([this]
(Calculator^ sender) {
    _error->Text = "Cannot divide by zero";
});
```

`_error` is a `TextBlock` element within the UI that shows the last error (if any). A using namespace for `Calculations` was also added so that the preceding code can compile.

When the **Calculate** button is clicked we need to perform the actual operation based on the currently selected index in the listbox that hosts the available operations:

```
_error->Text = "";
wstringstream ss(_value->Text->Data());
double value;
ss >> value;
switch(_operationList->SelectedIndex) {
case 0:
    _calculator->Add(value); break;
case 1:
    _calculator->Subtract(value); break;
case 2:
    _calculator->Multiply(value); break;
case 3:
```

```

        _calculator->Divide(value); break;
    }
    // update result
    _result->Text = _calculator->Result.ToString();

```

For this code to compile, a using namespace statement was added for `std` and an `#include` was added for `<sstream>`.

That's it. We have consumed a WinRT component. Technically, there is no easy way to know in what language that was written. The only thing that matters is that it's a WinRT component.

Building a C# client

Let's see how this works with another client—a Store app written with C#. First, we'll create a blank C# Store application (named `CalcClient2`) and copy the XAML as is to the C# project from the C++ client project.

Next, we need to add a reference to the `winmd` file. Right-click on the project node and select **Add Reference...** or right-click on the **References** node and select **Add Reference...** A similar dialog appears, allowing the selection of the `Calculations` project (or browsing the filesystem for the file if it's a different solution).

The actual code needed to use `Calculator` is similar to the C++ case, with the syntax and semantics of C# (and .NET). In `MainPage.xaml.cs`, we create a `Calculator` object and register for the `DivideByZero` event (using a C# lambda expression):

```

Calculator _calculator;

public MainPage() {
    this.InitializeComponent();
    _calculator = new Calculator();
    _calculator.DivideByZero += calc => {
        _error.Text = "Cannot divide by zero";
    };
}

```



In C#, a lambda expression can be written without specifying the exact types (as shown in the preceding code snippet); the compiler infers the types on its own (because the delegate type is known). It's possible (and legal) to write the type explicitly like `_calculator.DivideByZero += (Calculator calc) => { ... };`.

A `using Calculations` statement was added at the top of the file. The button's click event handler is pretty self-explanatory:

```
_error.Text = String.Empty;
double value = double.Parse(_value.Text);
switch (_operationList.SelectedIndex) {
    case 0:
        _calculator.Add(value); break;
    case 1:
        _calculator.Subtract(value); break;
    case 2:
        _calculator.Multiply(value); break;
    case 3:
        _calculator.Divide(value); break;
}
// update result
_result.Text = _calculator.Result.ToString();
```

Notice how similar the C# code that accesses the calculator is to the C++ version.

The Application Binary Interface

The Calculator WinRT class created in the previous section leaves some questions. Suppose the following method was added to the public section of the class:

```
std::wstring GetResultAsString();
```

The compiler would refuse to compile this method. The reason has to do with the use of `std::wstring`. It's a C++ type—how would that project into C# or JavaScript? It can't. Public members must use WinRT types only. There is a boundary between the internal C++ implementation and the public-facing types. The correct way to define the method in question is this:

```
Platform::String^ GetResultAsString();
```

`Platform::String` is the C++/CX wrapper over a `HSTRING` WinRT, which is projected as `System.String` to C# and to a JavaScript string in that world.

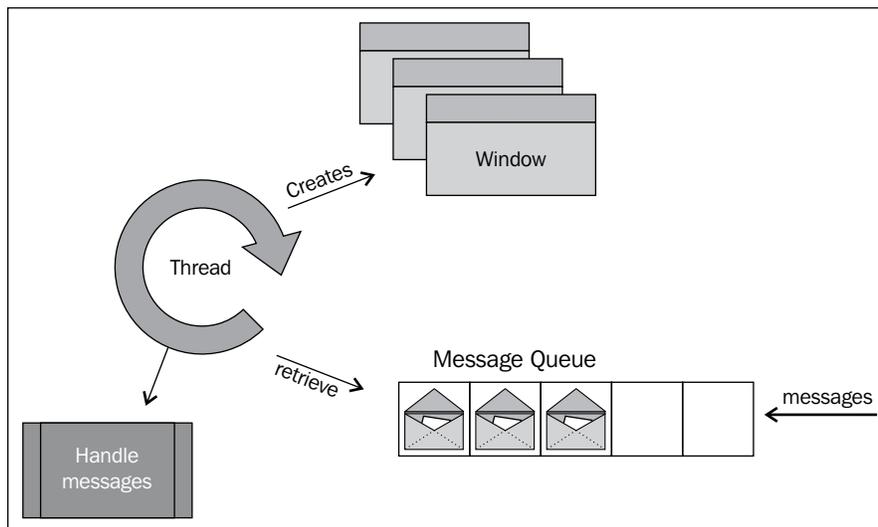
Private members in a WinRT class can be anything, and more often than not these are native C++ types (such as `wstring`, `vector<>`, and anything else that may have been migrated from older code).

Simple types, such as `int` and `double` are automatically mapped between C++ and WinRT. The **Application Binary Interface (ABI)** is the boundary between WinRT types (that are consumable outside the component) and the native types that are specific to the language/technology (true not just for C++, but for C# as well).

Asynchronous operations

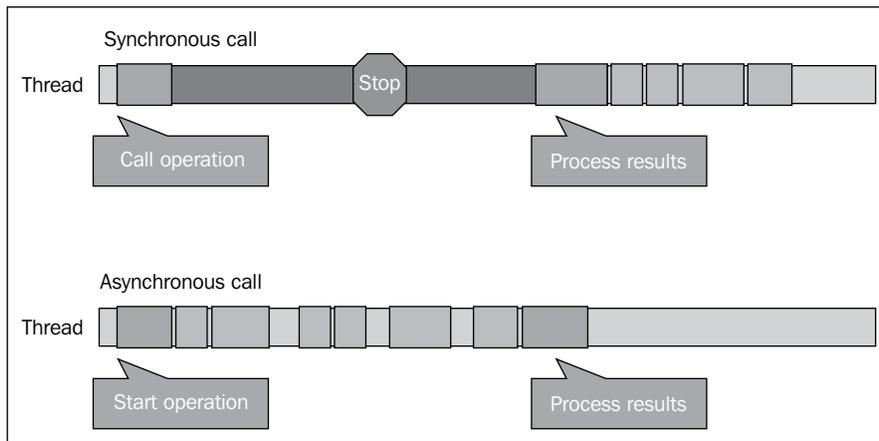
Windows 8 Store applications promise to be "fast and fluid". This expression has several meanings, some of which are related to user experience and user interface design (which won't concern us here), and some related to application responsiveness.

Ever since the first version of the Windows OS, user interface was handled by a single thread in an application. Technically, a thread may create any number of windows, and that thread becomes the owner of those windows, and is the only thread that can handle messages targeting those windows (through a message queue). If that thread becomes very busy and does not handle messages quickly enough, the UI becomes less responsive; in extreme cases, if the thread is stuck for several seconds or longer (for whatever reason), the UI becomes completely unresponsive. This situation is pretty familiar and highly undesirable. The following diagram illustrates the entities involved in UI processing:



The key to responsiveness is to free the UI thread as soon as possible and never block it for more than a few milliseconds. In the world of desktop apps, there's nothing stopping the developer from calling some long running operation (or some long I/O operation), thus preventing the thread from returning to its pumping messages activity, freezing the user interface.

In WinRT, Microsoft has made a conscious decision that if an operation may take longer than 50 milliseconds, then it should be made asynchronous rather than synchronous. The net result is that many methods are executed asynchronously, something that can potentially complicate code. Asynchronous means that the operation starts, but the call returns almost immediately. When the operation is complete, some callback is invoked, so that the application can take further steps. In between, though, the UI thread is doing nothing special and, thus, can pump messages as usual, keeping the UI responsive. The difference between synchronous and asynchronous calls can be illustrated with the following diagram:



Asynchronous operations, although desirable, are more complicated by definition. The code is not sequential anymore. WinRT defines some interfaces that represent on-going operations. These interfaces are returned from various asynchronous methods that start an operation and allow the client to register for the time the operation completes.

Let's see an example of asynchronous operations and how we can handle them. We'll create a simple image viewer application that allows a user to browse for an image and show it (the complete source is in the `SimpleImageView` project available with this chapter's downloads). The user interface is not important at the moment, consisting of a button that initiates the user's selection process and an `Image` element that can show images. When the button is clicked we want to provide the user with a way to select image files and then convert the file to something an `Image` element can show.

The WinRT class to use for selecting files is `Windows::Storage::Pickers::FileOpenPicker`. We'll create an instance and set some properties:

```
auto picker = ref new FileOpenPicker;
picker->FileTypeFilter->Append(".jpg");
picker->FileTypeFilter->Append(".png");
picker->ViewMode = PickerViewMode::Thumbnail;
```



Readers familiar with the desktop app world may wonder where the common open file dialog is, which is available through the Win32 API or other wrappers. That dialog cannot be used in a Store app for several reasons. The first is aesthetic; the dialog is ugly, compared to the modern UI that Windows 8 Store apps try to convey. Second, the dialog has a title bar and other such chrome, and as such is not suitable for the new world. And third (most importantly), `FileOpenPicker` is not just about selecting files from the filesystem. It actually works with the File Open Picker contract, implemented (for example) by the camera (if one is attached), so we can actually take a picture and then select it; the same is true for other sources, such as SkyDrive, Facebook, and so on. The common open file dialog has no such functionality.

Now, it's time to show that picker and allow the user to select something. Looking at the `FileOpenPicker` API, we find the `PickSingleFileAsync` method. The `Async` suffix is the convention used in the WinRT API to indicate a method that starts an asynchronous operation. The result of picking a file should be an instance of `Windows::Storage::StorageFile`, but instead it returns an `IAsyncOperation<StorageFile^>`, which is an object representing the long running operation.

One way to work with this is to set the `Completed` property (a delegate) to a handler method that will be invoked when the operation completes (this can be a lambda function). When that function is called, we can call `IAsyncOperation<T>::GetResults()` to get the actual `StorageFile` object:

```
auto fileOperation = picker->PickSingleFileAsync();
fileOperation->Completed = ref new
    AsyncOperationCompletedHandler<StorageFile^>(
    [this](IAsyncOperation<StorageFile^>^ op, AsyncStatus status) {
        auto file = op->GetResults();
    });
```

Unfortunately, that's not the end of it. Once the file is available, we need to open it, convert its data into a WinRT stream interface and then feed it to a `BitmapImage` object that can be rendered into an `Image` element.

It turns out that opening a `StorageFile` is an asynchronous operation, too (remember, that file can be from anywhere, such as SkyDrive or a network share). We repeat the same sequence after the file is obtained:

```
using namespace Windows::UI::Core;
auto openOperation = file->OpenReadAsync();
openOperation->Completed = ref new AsyncOperationCompletedHandler<IRandomAccessStreamWithContentType^>(
    [this] (IAsyncOperation<IRandomAccessStreamWithContentType^>^
        op, AsyncStatus status) {
        auto bmp = ref new BitmapImage;
        bmp->SetSource(op->GetResults());
        _image->Source = bmp;
    });
```

`_image` is the `Image` element that should display the resulting image using its `Source` property.

This almost works. The "almost" part is a bit subtle. The previous lambda is called by a different thread than the thread that initiated the call. The UI thread started it, but it returned on a background thread. Accessing UI elements (such as the `Image` element) from a background thread causes an exception to be thrown. How can we fix that?

We can use the `Dispatcher` object that is bound to a particular thread in the case of the UI and ask it to execute some piece of code (specified typically as a lambda) on the UI thread:

```
Dispatcher->RunAsync(CoreDispatcherPriority::Normal,
    ref new DispatchedHandler([op, this]() {
        auto bmp = ref new BitmapImage;
        bmp->SetSource(op->GetResults());
        _image->Source = bmp;
    }));
```

`Dispatcher` is a property of `this` (or any UI element for that matter), that posts an operation to be executed by the UI thread when possible (typically almost immediately, assuming the UI thread is not blocked, which we work very hard to avoid).

This whole sequence was not easy, and adding the `Dispatcher` to the mix complicates things further. Fortunately, there is an easier way to work with asynchronous operations – using the `task<T>` class.

Using tasks for asynchronous operations

The `task<T>` class resides in the concurrency namespace and requires `#include` to `<ppltasks.h>`. This class is new to C++11, and is generally related to parallel programming, but here it serves a special purpose for invoking asynchronous operations.

The `task<T>` class represents an operation whose result is of type `T`. It handles the gory details of the `Completed` property registration, calling `GetResults`, and using the `Dispatcher` automatically to maintain thread affinity in case the operation was invoked from the UI thread (technically, a call from a `Single Threaded Apartment`). And all this with nice composition in case we need to invoke several asynchronous operations in sequence (which is true for the case in hand). Here's the complete code:

```
auto fileTask = create_task(picker->PickSingleFileAsync());
fileTask.then([](StorageFile^ file) {
    return create_task(file->OpenReadAsync());
}).then([this](IRandomAccessStreamWithContentType^ stm) {
    auto bmp = ref new BitmapImage;
    bmp->SetSource(stm);
    _image->Source = bmp;
});
```

The `create_task<T>` function is a convenience that creates a `task<T>` with the correct `T`; `create_task<T>` allows using the `auto` keyword. An equivalent alternative would be this:

```
task<StorageFile^> fileTask(picker->PickSingleFileAsync());
```

The `then` instance method expects a function (sometimes called continuation, typically a lambda) that should execute upon completion of the asynchronous operation. It provides the result without any need to call `IAsyncOperation<T>::GetResults()`.

Notice the composition. After the `StorageFile` is available, another task is created and returned from the lambda. This initiates yet another asynchronous operation, to be resolved by the next `then` call.

Finally, the continuations run on the same thread as the operation initiator, if that initiator is running in an STA (which is the case for the UI thread).

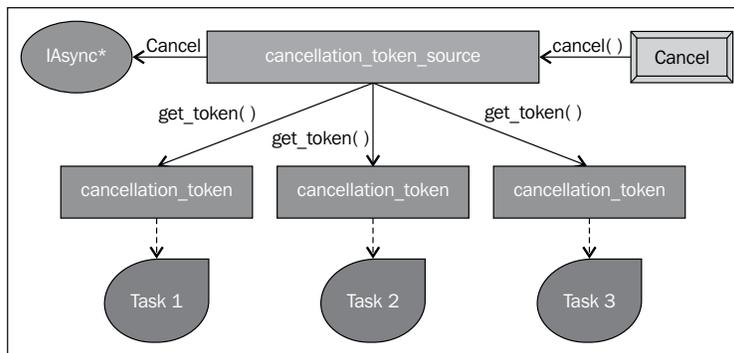


This apartment awareness only works for operations returning `IAsyncAction<T>` or `IAsyncOperation<T>` (and their derivatives).

Cancelling asynchronous operations

An asynchronous operation, by definition, may be long running, so it's a good idea to expose an ability to cancel the operation (if possible). The `IAsync*` family of interfaces have a `Cancel` method that we can call (for example, from some **Cancel** button's click event handler), but it's difficult to expose the `IAsync*` object to outside code.

Fortunately, the `task<>` class provides an elegant solution. A second parameter to the task constructor (or the `create_task` auxiliary function) is a `cancellation_token` object. This token is obtained from a `cancellation_token_source` object using its `get_token()` instance method. `cancellation_token_source` represents an operation that is cancellable. An outside caller can use its `cancel()` method to "signal" all `cancellation_token` objects (typically just one) that were handed out by the `cancellation_token_source`, causing the task(s) to call the `IAsync*::Cancel` method. The following diagram illustrates the process:



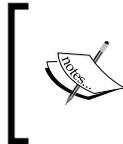
The net result is that if an operation is cancelled, a `task_canceled` exception is thrown. It's propagated (if unhandled) down the `then` chain, so that it can be caught conveniently on the last `then` — in fact, it's better to add a last `then` that does cancellation (and error) handling only:

```

then([])(task<void> t) {
    try {
        t.get();
    }
    catch(task_canceled) {
        // task cancelled
    }
    catch(Exception^ ex) {
        // some error occurred
    }
};

```

The `task<>::get()` method is the one throwing the exceptions. Note that `task_canceled` does not derive from `Platform::Exception`, so it needs a separate `catch` clause to be caught.



Some operations simply return a `nullptr` object to indicate cancellation. This is the case with the `FileOpenPicker` example. If the `StorageFile` object returned is `nullptr`, this means the user selected the **Cancel** button when selecting a file.

Error handling

In asynchronous operations, exceptions may be thrown. One way to handle those is by adding `try/catch` blocks to the appropriate continuations. A more convenient approach is to handle all errors in the last `then` continuation, in much the same way as cancellation.

Using existing libraries

WinRT is a new library we want to use to get access to Windows 8 features in this new Store apps model. What about existing C++ libraries, such as the **Standard Template Library (STL)**, **Active Template Library (ATL)**, **Microsoft Foundation Classes (MFC)**, or some other custom libraries? What about the raw Win32 API? In the following sections, we'll address the common Microsoft libraries and their use in Windows 8 Store apps.

STL

STL is part of the standard C++ libraries (and sometimes considered a synonym for it), and is fully supported in Windows 8 Store apps. In fact, some of the WinRT type wrappers know about STL, making it easier to interoperate.

MFC

MFC library was created more than 20 years ago, to provide a C++ layer over the Windows API (Win16 at the time of creation), mostly for easier creation of user interface.

Windows 8 Store apps provide their own user interface that is very far from the Windows `User32.dll` APIs (which MFC wraps), making MFC obsolete and unusable in the new world. Existing code must be migrated to using XAML, user controls, control templates, or whatever is appropriate for the application in question.

ATL

ATL was created to assist in building COM servers and clients, easing the burden of implementing common functionality such as `IUnknown`, class factories, component registration, and the like. It can technically be used in Windows Store apps, but there's really no point. Anything on that level is covered by the WRL that was discussed earlier in this chapter.

Win32 API

Win32 API (or the Windows API) is a huge set of mostly C-style functions and some COM component that has been, and still is, the low-level API to the Windows OS in user mode.

Every documented function now includes an "applied to" clause, stating whether that API is usable in desktop apps, Store apps, or both. Why would some functions be unavailable in a Windows Store app? A few reasons:

- Some functions are related to a user interface that is inappropriate for Windows Store. For example, `MessageBox` and `CreateWindowEx`.
- Some functions have equivalents in the WinRT API (which is usually superior). For example `CreateFile` (although a new `CreateFile2` API exists that works with Store apps as well), `CreateThread`, and `QueueUserWorkItem`.
- Some functions are inappropriate in some other way, such as violating security constraints. For example `CreateProcess` and `EnumWindows`.

Using a forbidden API fails to compile; that's because the Windows API headers have been changed to conditionally compile based on two constants, `WINAPI_PARTITION_APP` (for Store apps) and `WINAPI_PARTITION_DESKTOP` (for desktop apps).

Theoretically, it's possible to redefine a forbidden function and call it. Here's an example that would work for the `MessageBox` function:

```
extern "C" BOOL WINAPI MessageBoxW(HWND hParent, LPCTSTR msg,
    LPCTSTR title, DWORD flags);

#pragma comment(lib, "user32.lib")
```

Linking to the appropriate library is required in this case, as `user32.dll` is not linked in, by default.

Although this works, and a message box would appear if this function is called, don't do it. The reason is simple: the Windows 8 Store certification process will fail any application that uses a forbidden API.



More information on the allowed Windows API functions can be found at <http://msdn.microsoft.com/en-us/library/windows/apps/br205757>.

CRT

The **C Runtime (CRT)** library contains a slew of functions, originally created as the support library for the C language. Many of those functions are unavailable in Store apps; usually there is a Win32 or WinRT equivalent. For a comprehensive list of unsupported functions, refer to <http://msdn.microsoft.com/EN-US/library/jj606124.aspx>.

DirectX

DirectX is a set of low-level, COM-based APIs, originally created more than 20 years ago, for the purpose of accessing the multimedia capabilities of the PC (graphics, audio, input, and so on) while leveraging the hardware capabilities (such as the graphic card). DirectX has been used for years mostly in the gaming industry.

Windows 8 comes with DirectX 11.1 installed, providing a base for creating high performance games and applications. It's fully supported with Store apps, and can even coexist with XAML-based UI.

C++ AMP

The C++ **Accelerated Massive Parallelism (AMP)** is a relatively new library that has a lofty goal: the ability to use a mainstream programming language (C++) to execute code on CPU and non-CPU devices. Currently, the only other supported device is the **Graphics Processing Unit (GPU)**.

Modern GPUs are capable of much parallelism, but originally they have their own languages for programming arbitrary algorithms that may be unrelated to graphics per se. C++ AMP is an attempt to work with C++, but still be able to run the GPU (and other devices in the future).

C++ AMP is fully supported with Windows 8 Store apps (and requires a DirectX 11 capable card).

The Windows Runtime class library

WinRT provides a comprehensive class library, arranged in hierarchical namespaces; from strings and collections, to controls, to devices, to networking, to graphics; the API covers a lot of ground. Part of the journey into Windows Store apps is learning the various APIs and capabilities that are supported. This kind of knowledge evolves with time. During the course of this book, we'll discuss a fair amount of WinRT APIs, but certainly not all of it.

In the following sections, we'll discuss some of the core types that are used frequently with Store apps and how they are mapped specifically (if at all) to C++.

Strings

WinRT defines its own string type, `HSTRING`. We have already met it a few times. Since `HSTRING` is just an opaque handle to an immutable string, Windows provides some functions for managing `HSTRING`, such as `WindowsCreateString`, `WindowsConcatString`, `WindowsSubString`, `WindowsGetStringLen`, `WindowsReplaceString`, and others. Working with these APIs is not difficult, but very tedious.

Fortunately, an `HSTRING` is wrapped by a reference counted class, `Platform::String`, which provides the necessary calls behind the scenes to the appropriate APIs. It can be constructed given a raw Unicode character pointer (`wchar_t*`) and has a `Data()` method that returns a raw pointer back. This means that interoperating `Platform::String` with `std::wstring` is fairly easy. Here are a few example of using strings:

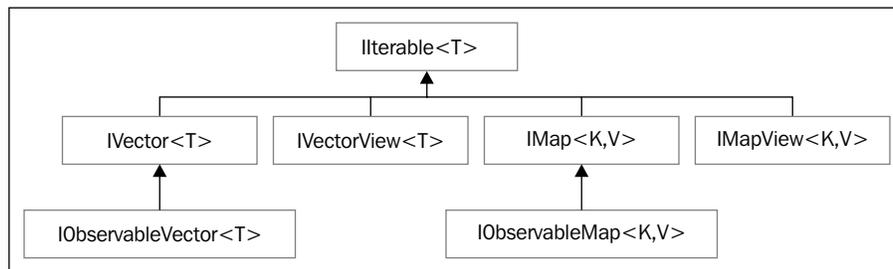
```
auto s1 = ref new String(L"Hello");
std::wstring s2(L"Second");
auto s3 = ref new String(s2.c_str());
int compare = wcscmp(L"xyz", s1->Data());
for(auto i = begin(s3); i != end(s3); ++i)
    DoSomethingWithChar(*i);
```

Notice the iterator-like behavior achieved with `Platform::begin` and `Platform::end`. As a general guideline, when authoring components, it's better to work with a `std::wstring` for all string operations, as `wstring` has a rich function set. Use `Platform::String` only at the ABI boundary; `Platform::String` has very little functionality built in.

Collections

Standard C++ defines several container types, such as `std::vector<T>`, `std::list<T>`, `std::map<K, V>`, and others. These types, however, cannot cross the ABI boundary – they are C++ specific.

WinRT defines its own collection interfaces that must be used across the ABI boundary. Here's a class diagram with those interfaces:



`IIterable<T>` has just one method: `First`, which returns an `IIterator<T>`, which is the WinRT iterator interface. It defines the methods `MoveNext` and `GetMany` and two properties: `Current` returns the current object the iterator points to and `HasCurrent` indicates whether there are any more items to iterate over.

`IVector<T>` represents a sequence of items that are accessible by index. It's a common type to use across the ABI. The C++ support library provides a stock implementation for `IVector<T>` named `Platform::Collections::Vector<T>`. This could be used as the underlying private type within a WinRT class, because it's convertible to `IVector<T>` when needed. Note, however, that for heavy duty operations the STL `std::vector<T>` is more efficient. If `vector<T>` is needed at some point, it has many constructors, some of which accept `std::vector<T>`.

`IVectorView<T>` represents a read only view into a vector. It can be obtained from an `IVector<T>` by calling the `GetView` method. `VectorView<T>` is a C++ private implementation that may be used if needed by custom implementations of `IVector<T>`.

`IObservableVector<T>` inherits from `IVector<T>` and adds a single event, `VectorChanged`. This may be useful for clients that want notifications when items are added, removed, or replaced in the `IObservableVector<T>`.

The `IMap*` series of interfaces manage key/value pairs, and are transferrable across the ABI boundary. `Platform::Collections::Map<K,V>` provides an implementation convertible to this interface, as a balanced binary tree, similar to `std::map<K,V>` (including the ability to change the ordering algorithm via a third template argument). `IMapView<K,V>` is a read-only view of a `IMap<K,V>`.

 The most useful collection type for the ABI is `IVector<T>`. If you can live with `Vector<T>` as the underlying implementation, do so. Otherwise, maintain a `std::vector<T>` and convert to `IVector<T>` only when crossing the ABI boundary.

Exceptions

COM/WinRT does not work with exceptions. The reason may be obvious, exceptions are language or platform specific. They cannot be part of a binary standard that various platforms adhere to. Instead, COM uses `HRESULT`, which are just 32-bit numbers to indicate the success or failure of method calls.

C++, however (and most other modern languages, such as C#) support the notion of exceptions. Handling errors by catching exceptions is far easier and maintainable than checking `HRESULT` after each call (C-style of programming). That's why the calls made through C++/CX reference counted object (hat) translates a failed `HRESULT` into an exception object, derived from `Platform::Exception` that can be caught in the usual way.

This is also true the other way around; when implementing a component in C++/CX, the code can throw an exception derived from `Platform::Exception`; this exception cannot cross the ABI; instead, it's translated to an equivalent `HRESULT`, which is the thing that can cross the ABI. On the other side, it may be turned into an exception object again for that client platform, such as a C++ exception or a .NET exception.

The list of exception types deriving from `Platform::Exception` is predefined and cannot be extended, because each type maps directly to an `HRESULT`. This means it's not possible to add new exception types, because C++/CX can't know to which `HRESULT` to translate the exception when crossing the ABI. For custom exceptions, `Platform::COMException` can be used with some custom `HRESULT`. The complete table of exception types and their `HRESULT` equivalent is shown, as follows:

HRESULT	Exception (Platform namespace)
E_OUTOFMEMORY	OutOfMemoryException
E_INVALIDARG	InvalidArgumentException
E_NOINTERFACE	InvalidCastException
E_POINTER	NullReferenceException
E_NOTIMPL	NotImplementedException
E_ACCESSDENIED	AccessDeniedException
E_FAIL	FailureException
E_BOUNDS	OutOfBoundsException
E_CHANGED_STATE	ChangedStateException
REGDB_E_CLASSNOTREG	ClassNotRegisteredException
E_DISCONNECTED	DisconnectedException
E_ABORT	OperationCanceledException
RPC_E_WRONG_THREAD	WrongThreadException
(user-defined)	COMException
RO_E_CLOSED	ObjectDisposedException

Most of the exception types in the table are self-explanatory. We'll discuss some of these exceptions later in the book.



Throwing something that does not inherit from `Platform::Exception` will be translated to an `E_FAIL` HRESULT.

All exception types have an `HResult` property with the underlying `HRESULT` value and a `Message` property, which is a textual description of the exception (supplied by WinRT and cannot be changed).

Summary

This chapter started with some of the new C++11 features that may be useful for WinRT development. We discussed COM, its concepts and ideas and how they are translated into WinRT. WRL provides helpers for accessing WinRT objects without language extensions. C++/CX provides language extensions that make it far easier to work with WinRT and to author WinRT components.

WinRT has some patterns and idioms we need to learn and get used to, such as ways to work with asynchronous operations, strings, collections, and error handling.

The coverage in this chapter was not exhaustive, but it should give us enough power and understanding to start writing real applications. We'll take a look at some other C++/CX capabilities and other WinRT-related features in later parts of the book.

In the next chapter, we'll dive into building applications, starting with XAML and the way user interfaces are commonly built in WinRT.

3

Building UI with XAML

User interface and user experience play an important role with Windows 8 Store apps. A new design has been created for Store apps, now called modern design style (formerly known as Metro), with keywords such as "fast and fluid", "content first", and "touch centric". The app UI takes up the entire screen (except when in snap view), which makes the UI all the more important. In this chapter (and the next one), we'll discuss the way in which UI for Store apps is built, more on the technical level than on the actual design. Microsoft makes a lot of resources available online for the design part of the UI.

XAML

C++ Store applications typically use **eXtensible Application Markup Language (XAML)** as the main language for creating the user interface. The first question that comes to mind when XAML is first mentioned, is why? What's wrong with C++, or any other existing programming language?

XAML is an XML-based language that describes the what, not the how; it's declarative and neutral. Technically, a complete app can be written without any XAML; there's nothing XAML can do that C++ can't. Here are some reasons why XAML makes sense (or at least may make sense in a little bit):

- C++ is very verbose as opposed to XAML. XAML is usually shorter than the equivalent C++ code.
- Since XAML is neutral, design-oriented tools can read and manipulate it. Microsoft provides the Expression Blend tool just for this purpose.
- The declarative nature of XAML makes it easier (most of the time, after users get used to it) to build user interfaces, as these have a tree-like structure, just like XML.

XAML itself has nothing to do with the user interface in itself. XAML is a way to create objects (usually an object tree) and set their properties. This works for any type that is "XAML friendly", meaning it should have the following:

- A default public constructor
- Settable public properties

The second point is not a strict requirement, but without properties the object is pretty dull.

 XAML was originally created for **Windows Presentation Foundation (WPF)**, the main rich client technology in .NET. It's now leveraged in other technologies, mostly in the .NET space, such as Silverlight and **Windows Workflow Foundation (WF)**.
The XAML level currently implemented in WinRT is roughly equivalent to Silverlight 3 XAML. In particular, it's not as powerful as WPF's XAML.

XAML basics

XAML has a few rules. Once we understand those rules, we can read and write any XAML. The most fundamental XAML rules are as follows:

- An XML element means object creation
- An XML attribute means setting a property (or an event handler)

With these two rules, the following markup means creating a `Button` object and setting its `Content` property to the string `Click me`:

```
<Button Content="Click me!" />
```

The equivalent C++ code would be as follows:

```
auto b = ref new Button;  
b->Content = "Click me";
```

When creating a new Blank App project, a `MainPage.xaml` file is created along with the header and implementation files. Here's how that XAML file looks:

```
<Page  
  x:Class="BasicXaml.MainPage"  
  xmlns="http://schemas.microsoft.com/winfx/  
  2006/xaml/presentation"  
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
  xmlns:local="using:BasicXaml"
```

```
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/
markup-compatibility/2006"
mc:Ignorable="d">

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

It's worth going over these lines in detail. In this example, the project name is `BasicXaml`. The root element is `Page` and an `x:Class` attribute is set, indicating the class that inherits from `Page`, here named `BasicXaml::MainPage`. Note that the class name is the full name including namespace, where the separator must be a period (not the C++ scope resolution operator `::`). `x:Class` can only be placed on the root element.

What follows that root element is a bunch of XML namespace declarations. These give context to the elements used in the entire XAML of this page. The default XML namespace (without a name) indicates to the XAML parser that types such as `Page`, `Button`, and `Grid` can be written as they are, without any special prefix. This is the most common scenario, because most of the XAML in a page constitutes user interface elements.

The next XML namespace prefix is `x` and it points to special instructions for the XAML parser. We have just seen `x:Class` in action. We'll meet other such attributes later in this chapter.

Next up is a prefix named `local`, which points to the types declared in the `BasicXaml` namespace. This allows creating our own objects in XAML; the prefix of such types must be `local` so that the XAML parser understands where to look for such a type (of course, we can change that to anything we like). For example, suppose we create a user control derived type named `MyControl`. To create a `MyControl` instance in XAML, we could use the following markup:

```
<local:MyControl />
```

The `d` prefix is used for designer-related attributes, mostly useful with Expression Blend. The `mc:Ignorable` attribute states that the `d` prefix should be ignored by the XAML parser (because it's related to the way Blend works with the XAML).

The `Grid` element is hosted inside the `Page`, where "hosted" will become clear in a moment. Its `Background` property is set to `{StaticResource ApplicationPageBackgroundThemeBrush}`. This is a markup extension, discussed in a later section in this chapter.

 XAML is unable to invoke methods directly; it can just set properties. This is understandable, as XAML needs to remain declarative in nature; it's not meant as a replacement for C++ or any other programming language.

Type converters

XML deals with strings. However, it's clear that many properties are not strings. Many can still be specified as strings, and still work correctly thanks to the type converters employed by the XAML parser. Here's an example of a `Rectangle` element:

```
<Rectangle Fill="Red" />
```

Presumably, the `Fill` property is not of a string type. In fact, it's a `Brush`. `Red` here really means `ref new SolidColorBrush(Colors.Red)`. The XAML parser knows how to translate a string, such as `Red` (and many others) to a `Brush` type (in this case the more specific `SolidColorBrush`).

Type converters are just one aspect of XAML that make it more succinct than the equivalent C++ code.

Complex properties

As we've seen, setting properties is done via XML attributes. What about complex properties that cannot be expressed as a string and don't have type converters? In this case, an extended syntax (property element syntax) is used to set the property. Here's an example:

```
<Rectangle Fill="Red">
  <Rectangle.RenderTransform>
    <RotateTransform Angle="45" />
  </Rectangle.RenderTransform>
</Rectangle>
```

Setting the `RenderTransform` property cannot be done with a simple string; it must be an object that is derived from the `Transform` class (`RotateTransform` in this case).

 The exact meaning of the various example properties (`Fill`, `RenderTransform`, and others) will be discussed in *Chapter 4, Layout, Elements, and Controls*.

The preceding markup is equivalent to the following C++ code:

```
auto r = ref new Rectangle;
r->Fill = ref new SolidColorBrush(Colors::Red);
auto rotate = ref new RotateTransform();
rotate->Angle = 45;
r->RenderTransform = rotate;
```

Dependency properties and attached properties

Most properties on various elements and controls are not normal, in the sense that they are not simple wrappers around private fields. The significance of dependency properties will be discussed in *Chapter 5, Data Binding*. For now, it's important to realize that there is no difference in XAML between a dependency property and a regular property; the syntax is the same. In fact, there is no way to tell if a certain property is a dependency property or not, just by looking at its use in XAML.



Dependency properties provide the following features (a detailed explanation is provided in *Chapter 6, Components, Templates, and Custom Elements*):

- Change notifications when the property value changes
- Visual inheritance for certain properties (mostly the font-related properties)
- Multiple providers that may affect the final value (one wins out)
- Memory conservation (value not allocated unless changed)

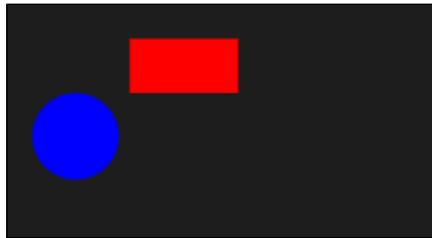
Some WinRT features, such as data binding, styles, and animations are dependent on that support.

Another kind of dependency properties is attached properties. Again, a detailed discussion is deferred until *Chapter 5, Data Binding*, but essentially an attached property is contextual—it's defined by one type (with a registration mechanism that will be discussed in *Chapter 6, Components, Templates, and Custom Controls*), but can be used by any type that inherits from `DependencyObject` (as all elements and controls do). Since this kind of property is not defined by the object it's used on, it merits a special syntax in XAML. The following is an example of a Canvas panel that holds two elements:

```
<Canvas>
  <Rectangle Fill="Red" Canvas.Left="120" Canvas.Top="40"
    Width="100" Height="50" />
  <Ellipse Fill="Blue" Canvas.Left="30" Canvas.Top="90"
    Width="80" Height="80" />
</Canvas>
```

The `Canvas.Left` and `Canvas.Top` are attached properties. They were defined by the `Canvas` class, but they are attached to the `Rectangle` and `Ellipse` elements. Attached properties only have meaning in certain scenarios. In this case, they indicate the exact position of the elements within the canvas. The canvas is the one that looks for these properties in the layout phase (discussed in detail in the next chapter). This means that if those same elements were placed in, say a `Grid`, those properties would have no effect, because there is no interested entity in those properties (there is no harm in having them, however). Attached properties can be thought of as dynamic properties that may or may not be set on objects.

This is the resulting UI:



Setting an attached property in code is a little verbose. Here's the equivalent C++ code for setting the `Canvas.Left` and `Canvas.Top` properties on an element named `_myrect`:

```
Canvas::SetLeft(_myrect, 120);  
Canvas::SetTop(_myrect, 40);
```

The reason why the preceding calls will become apparent will be discussed when we will learn how to create attached properties in *Chapter 6, Components, Templates, and Custom Elements*.

Content properties

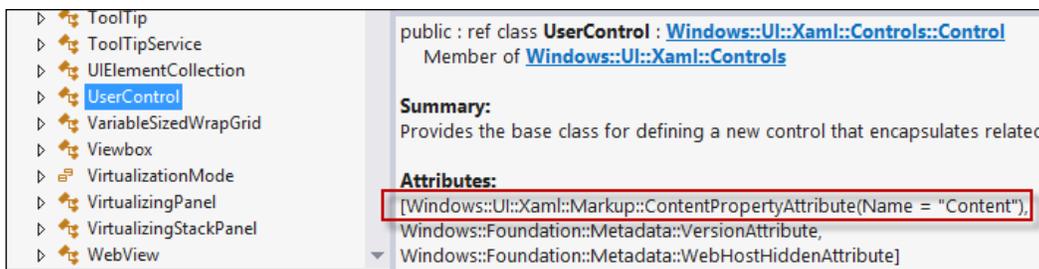
The relationship between a `Page` object and a `Grid` object is not obvious. `Grid` seems to be inside the `Page`. But how would that translate to code? The `Page/Grid` markup can be summed up as follows (ignoring the detailed markup):

```
<Page>  
  <Grid Background="...">  
  </Grid>  
</Page>
```

This is actually a shortcut for the following markup:

```
<Page>
  <Page.Content>
    <Grid Background="...">
  </Grid>
</Page.Content>
</Page>
```

This means the `Grid` object is set as the `Content` property of the `Page` object; now the relationship is clear. The XAML parser considers certain properties (no more than one per type hierarchy) as the default or content properties. It doesn't have to be named `Content`, but it is in the `Page` case. This attribute is specified in the control's metadata using the `Windows::UI::Xaml::Markup::ContentAttribute` class attribute. Looking at the Visual Studio object browser for the `Page` class shows no such attribute. But `Page` inherits from `UserControl`; navigating to `UserControl`, we can see the attribute set:



Attributes are a way to extend the metadata for a type declaratively. They can be inserted in C++/CX by an attribute type name in square brackets before the item that attribute is applied to (can be a class, interface, method, property, and other code element). An attribute class must derive from `Platform::Metadata::Attribute` to be considered as such by the compiler.

Some of the common `ContentProperty` properties in WinRT types are as follows:

- `Content` of `ContentControl` (and all derived types)
- `Content` of `UserControl`
- `Children` of `Panel` (base class for all layout containers)
- `Items` of `ItemsControl` (base class for collection-based controls)
- `GradientStops` of `GradientBrush` (base class of `LinearGradientBrush`)

Collection properties

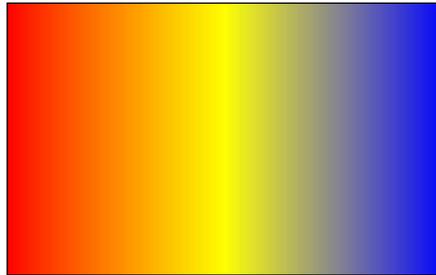
Some properties are collections (of type `IVector<T>` or `IMap<K, V>`, for instance). Such properties can be filled with objects, and the XAML parser will call the `IVector<T>::Append` or `IMap<K, V>::Insert` methods. Here's an example for a `LinearGradientBrush`:

```
<Rectangle>
  <Rectangle.Fill>
    <LinearGradientBrush EndPoint="1,0">
      <GradientStop Offset="0" Color="Red" />
      <GradientStop Offset=".5" Color="Yellow" />
      <GradientStop Offset="1" Color="Blue" />
    </LinearGradientBrush>
  </Rectangle.Fill>
</Rectangle>
```

Two rules are at work here. The first is the `ContentProperty` of `LinearGradientBrush` (`GradientStops`), which need not be specified. It's of the type `GradientStopCollection`, which implements `IVector<GradientStop>`, and thus is eligible for automatic appending. This is equivalent to the following code:

```
auto r = ref new Rectangle;
auto brush = ref new LinearGradientBrush;
brush->EndPoint = Point(1.0, 0);
auto stop = ref new GradientStop;
stop->Offset = 0; stop->Color = Colors::Red;
brush->GradientStops->Append(stop);
stop = ref new GradientStop;
stop->Offset = 0.5; stop->Color = Colors::Yellow;
brush->GradientStops->Append(stop);
stop = ref new GradientStop;
stop->Offset = 1; stop->Color = Colors::Blue;
brush->GradientStops->Append(stop);
r->Fill = brush;
```

This is perhaps the first clear sign of XAML syntax advantage over C++. Here's the rectangle in all its glory:



In the case of `IMap<K, V>`, an attribute named `x:Key` must be set on each item to indicate the key sent to the `IMap<K, V>::Insert` method. We'll see an example of such a map later in this chapter, when we will discuss resources.

Markup extensions

Markup extensions are special instructions to the XAML parser that provide ways of expressing things that are beyond object creation or setting some property. These instructions are still declarative in nature, but their code equivalent usually entails calling methods, which is not directly possible in XAML.

Markup extensions are placed inside curly braces as property values. They may contain arguments and properties, as we'll see in later chapters. The only markup extension used by default in a blank page is `{StaticResource}`, which will be discussed later in this chapter.


 WPF and Silverlight 5 allow developers to create custom markup extensions by deriving classes from `MarkupExtension`. This capability is unavailable in the current WinRT implementation.

One simple example of a markup extension is `{x:Null}`. This is used in XAML whenever the value `nullptr` needs to be specified, as there's no better way to use a string for this. The following example makes a hole in the `Rectangle` element:

```
<Rectangle Stroke="Red" StrokeThickness="10" Fill="{x:Null}" />
```

Naming elements

Objects created through XAML can be named using the `x:Name` XAML attribute. Here's an example:

```
<Rectangle x:Name="r1">
...
</Rectangle>
```

The net result is a private member variable (field) that is created by the XAML compiler inside `MainPage.g.h` (if working on `MainPage.xaml`):

```
private: ::Windows::UI::Xaml::Shapes::Rectangle^ r1;
```

The reference in itself must be set in the implementation of `MainPage::InitializeComponent` with the following code:

```
// Get the Rectangle named 'r1'
r1 = safe_cast<::Windows::UI::Xaml::Shapes::Rectangle^>(
    static_cast<Windows::UI::Xaml::IFrameworkElement^>(
        this)->FindName(L"r1"));
```

The mentioned file and method are discussed further in the section *XAML compilation and execution*. Regardless of how it works, `r1` is now a reference to that particular rectangle.

Connecting events to handlers

Events can be connected to handlers by using the same syntax as setting properties, but in this case the value of the property must be a method in the code behind class with the correct delegate signature.

Visual Studio helps out by adding a method automatically if *Tab* is pressed twice after entering the event name (in the header and implementation files). The default name that Visual Studio uses includes the element's name (`x:Name`) if it has one, or its type if it doesn't, followed by an underscore and the event name, and optionally followed by an underscore and an index if duplication is detected. The default name is usually not desirable; a better approach that still has Visual Studio creating the correct prototype is to write the handler name as we want it, and then right-click on the handler name and select **Navigate to Event Handler**. This has the effect of creating the handler (if it does not exist) and switching to the method implementation.

Here's an example of an XAML event connection:

```
<Button Content="Change" Click="OnChange" />
```

And the handler would be as follows (assuming the XAML is in `MainPage.xaml`):

```
void MainPage::OnChange(Platform::Object^ sender, Windows::UI::Xaml::RoutedEventArgs^ e)
{
}
```



Visual Studio also writes the namespace name in front of the class name (deleted in the preceding code example); this can be deleted safely, since an in-use namespace statement exists at the top of the file for the correct namespace. Also, the usage of `Platform::Object` instead of just `Object` (and similarly for `RoutedEventArgs`) is less readable; the namespace prefixes can be removed, as they are set up at the top of the file by default.

All events (by convention) use delegates that are similar. The first argument is always the sender of the event (in this case a `Button`) and the second parameter is the extra information regarding the event. `RoutedEventArgs` is the minimum type for events, known as routed events. A detailed discussion of routed events is covered in the next chapter.

XAML rules summary

This is a summary of all XAML rules:

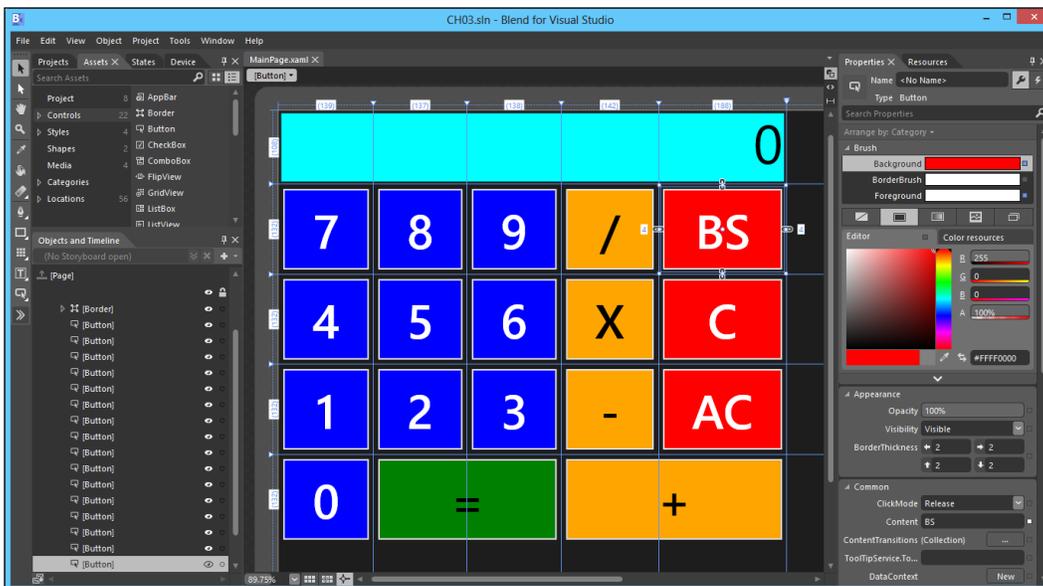
- An XAML element means creating an instance.
- An XAML attribute sets a property or an event handler. For properties, a type converter may execute depending on the property's type.
- Complex properties are set with the `Type.Property` element syntax.
- Attached properties are set with the `Type.Property` syntax, where `Type` is the declaring type of the attached property.
- `ContentPropertyAttribute` sets a `Content` property that need not be specified.
- Properties that are collections cause the XAML parser to call `Append` or `Insert`, as appropriate, automatically.
- Markup extensions allow for special (predefined) instructions.

Introducing the Blend for Visual Studio 2012 tool

Visual Studio 2012 is installed with the Blend for Visual Studio 2012 tool. This tool is typically used by UI designers to create or manipulate the user interface for XAML-based applications.

 The initial release of Blend for Visual Studio 2012 only supported Windows 8 Store Apps and Windows Phone 8 projects. The support for WPF 4.5 and Silverlight was added in Update 2 for Visual Studio 2012.

Blend can be used alongside Visual Studio 2012, as both understand the same file types (such as solution .sln files). It's not atypical to switch back and forth between the two tools – using each tool for its strengths. Here's a screenshot of Blend with the CH03.sln solution file open (the solution that holds all the samples for this chapter):



The preceding screenshot shows a particular XAML file open, with one button selected. Several windows comprise Blend, some of which are similar to their Visual Studio counterparts, namely **Projects** and **Properties**. Some of the new windows include:

- **Assets**: Holds the elements and controls available in WinRT (along with some other useful shortcuts)

- **Objects and Timeline:** Include all objects in the visual tree and also animations
- **Resources:** Holds all resources (refer to the next section) within the application

Blend's design surface allows manipulating elements and controls, which is also possible to do in Visual Studio. Blend's layout and some special editing features make it easier for UI/graphic designers to work with as it mimics other popular applications, such as Adobe Photoshop and Illustrator.

Any changes made using the designer are immediately reflected by the changed XAML. Switching back to Visual Studio and accepting the reload option synchronizes the files; naturally, this can be done both ways.

It's possible to work from within Blend entirely. Pressing *F5* builds and launches the app in the usual way. Blend, however, is not Visual Studio, and breakpoints and other debugging tasks are not supported.

Blend is a non-trivial tool, and is well beyond the scope of this book. Experimentation can go a long way, however.

XAML compilation and execution

The XAML compiler that runs as part of the normal compilation process, places the XAML as an internal resource within the EXE or DLL. In the constructor of a XAML root element type (such as `MainPage`), a call is made to `InitializeComponent`. This method uses a static helper method `Application::LoadComponent` to load the XAML and parse it – creating objects, setting properties, and so on. Here's the implementation created by the compiler for `InitializeComponent` (in `MainPage.g.hpp`, with some code cleaning):

```
void MainPage::InitializeComponent() {
    if (_contentLoaded)
        return;

    _contentLoaded = true;

    // Call LoadComponent on ms-appx:///MainPage.xaml
    Application::LoadComponent(this,
        ref new ::Windows::Foundation::Uri(
            L"ms-appx:///MainPage.xaml"),
        ComponentResourceLocation::Application);
}
```

Connecting XAML, H, and CPP files to the build process

From a developer's perspective, working with a XAML file carries with it two other files, the H and CPP. Let's examine them in a little more detail. Here's the default `MainPage.xaml.h` (comments and namespaces removed):

```
#include "MainPage.g.h"

namespace BasicXaml {
    public ref class MainPage sealed {
    public:
        MainPage();

    protected:
        virtual void OnNavigatedTo(NavigationEventArgs^ e)
        override;
    };
}
```

The code shows a constructor and a virtual method override named `OnNavigatedTo` (unimportant for this discussion). One thing that seems to be missing is the `InitializeComponent` method declaration mentioned in the previous section. Also the inheritance from `Page` that was hinted at earlier is missing. It turns out that the XAML compiler generates another header file named `MainPage.g.h` (g stands for generated) based on the XAML itself (this is evident with the top `#include` declaration). This file contains the following (it can be opened easily by selecting the **Project | Show All Files**, or the equivalent toolbar button, or right clicking on `#include` and selecting **Open Document...**):

```
namespace BasicXaml {
    partial ref class MainPage : public Page,
    public IComponentConnector {
    public:
        void InitializeComponent();
        virtual void Connect(int connectionId, Object^ target);

    private:
        bool _contentLoaded;

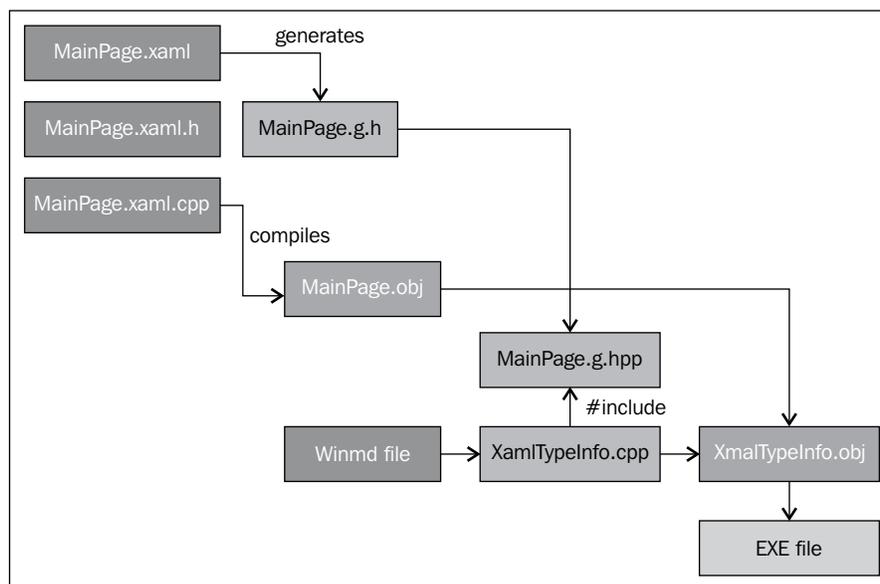
    };
}
```

Here we find the missing pieces. Here we find `InitializeComponent`, as well as the derivation from `Page`. How can there be more than one header file per class? A new C++/CX feature called partial classes allows this. The `MainPage` class is marked as `partial`, meaning there are more parts to it. The last part should not be marked as `partial`, and should include at least one header so that a chain forms, eventually including all the partial headers; all these headers must be part of the same compilation unit (a CPP file). This `MainPage.g.h` file is generated before any compilation happens; it's generated on the fly while editing the XAML file. This is important because named elements are declared in that file, providing instance intellisense.

During the compilation process, `MainPage.cpp` is finally compiled, producing an object file, `MainPage.obj`. It still has some unresolved functions, such as `InitializeComponent`. At this time, `MainPage.obj` (along with other XAML object files, if exist) is used to generate the metadata (`.winmd`) file.

To complete the build process, the compiler generates `MainPage.g.hpp`, which is actually an implementation file, created based on the information extracted from the metadata file (the `InitializeComponent` implementation is generated in this file). This generated file is included just once in a file called `XamlTypeInfo.g.cpp`, which is also generated automatically based on the metadata file (its job is related to data binding, as discussed in *Chapter 5, Data Binding*), but that's good enough so that `MainPage.g.hpp` is finally compiled, allowing linking to occur correctly.

The entire process can be summarized with the following diagram:



Resources

The term "resources" is highly overloaded. In classic Win32 programming, resources refer to read-only chunks of data, used by an application. Typical Win32 resources are strings, bitmaps, menus, toolbars, and dialogs, but custom resources can be created as well, making Win32 treat those as unknown chunks of binary data.

WinRT defines binary, string, and logical resources. The following sections discuss binary and logical resources (string resources are useful for localization scenarios and will not be discussed in this section).

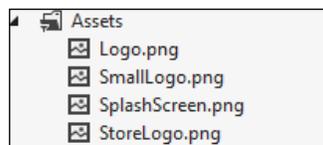
Binary resources

Binary resources refer to chunks of data, provided as part of the application's package. These typically include images, fonts, and any other static data needed for the application to function correctly.

Binary resources can be added to a project by right-clicking on the project in Solution Explorer, and selecting **Add Existing Item**. Then, select a file that must be in the project's directory or in a subdirectory.

 Contrary to C# or VB projects, adding an existing item from a location does not copy the file to the project's directory. This inconsistency is a bit annoying for those familiar with C#/VB projects, and hopefully will be reconciled in a future Visual Studio version or service pack.

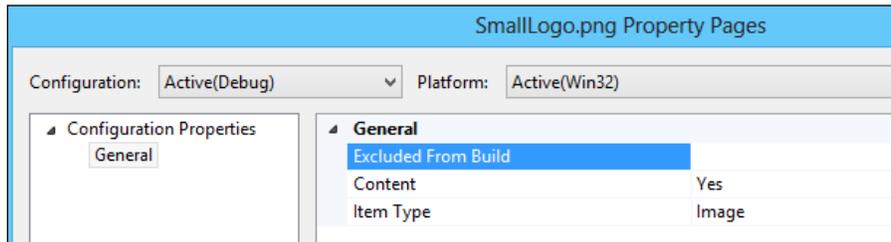
A typical Store app project already has some binary resources stored in the `Assets` project folder, namely images used by the application:



Using folders is a good way to organize resources by type or usage. Right-clicking on the project node and selecting **Add New Filter** creates a logical folder, to which items may be dragged.

 Again, contrary to C#/VB projects, project folders are not created in the filesystem. It's recommended that these are actually created in the filesystem for better organization.

The added binary resource is packaged as part of the application's package and is available in the executable folder or subfolder, keeping its relative location. Right-clicking on such a resource and selecting **Properties** yields the following dialog:



The **Content** attribute must be set to **Yes** for the resource to be actually available (the default). **Item Type** is typically recognized by Visual Studio automatically. In case, it doesn't, we can always set it to **Text** and do whatever we want with it in code.

 Don't set **Item Type** to **Resource**. This is unsupported in WinRT and will cause compile errors (this setting is really for WPF/Silverlight).

Binary resources can be accessed in XAML or in code, depending on the need. Here's an example of using an image named `apple.png` stored in a subfolder in the application named `Images` under the `Assets` folder by an `Image` element:

```
<Image Source="/Assets/Images/apple.png" />
```

Note the relative URI. The preceding markup works because of a type converter that's used on the `Image::Source` property (which is of the type `ImageSource`). That path is really a shortcut for the following, equivalent, URI:

```
<Image Source="ms-appx:///Assets/Images/apple.png" />
```

Other properties may require a slightly different syntax, but all originate through the `ms-appx` scheme, indicating the root of the application's package.

Binary resources that are stored in another component referenced by the application can be accessed with the following syntax:

```
<Image Source="/ResourceLibrary/jellyfish.jpg" />
```

The markup assumes that a component DLL named `ResourceLibrary.dll` is referenced by the application and that a binary resource named `jellyfish.jpg` is present in its root folder.

Logical resources

Binary resources are not new or unique to Store apps. They've been around practically forever. Logical resources, on the other hand, is a more recent addition. First created and used by WPF, followed by the various versions of Silverlight, they are used in WinRT as well. So, what are they?

Logical resources can be almost anything. These are objects, not binary chunks of data. They are stored in the `ResourceDictionary` objects, and can be easily accessed in XAML by using the `StaticResource` markup extension.

Here's an example of two elements that use an identical brush:

```
<Ellipse Grid.Row="0" Grid.Column="1">
  <Ellipse.Fill>
    <LinearGradientBrush EndPoint="0,1">
      <GradientStop Offset="0" Color="Green" />
      <GradientStop Offset=".5" Color="Orange" />
      <GradientStop Offset="1" Color="DarkRed" />
    </LinearGradientBrush>
  </Ellipse.Fill>
</Ellipse>
<Rectangle Grid.Row="1" Grid.Column="1" StrokeThickness="20">
  <Rectangle.Stroke>
    <LinearGradientBrush EndPoint="0,1">
      <GradientStop Offset="0" Color="Green" />
      <GradientStop Offset=".5" Color="Orange" />
      <GradientStop Offset="1" Color="DarkRed" />
    </LinearGradientBrush>
  </Rectangle.Stroke>
</Rectangle>
```

The problem should be self-evident. We're using the same brush twice. This is bad for two reasons:

- If we want to change the brush, we need to do it twice (because of the duplication). Naturally, this is more severe if that brush is used by more than two elements.
- Two different objects are created, although just one shared object is needed.

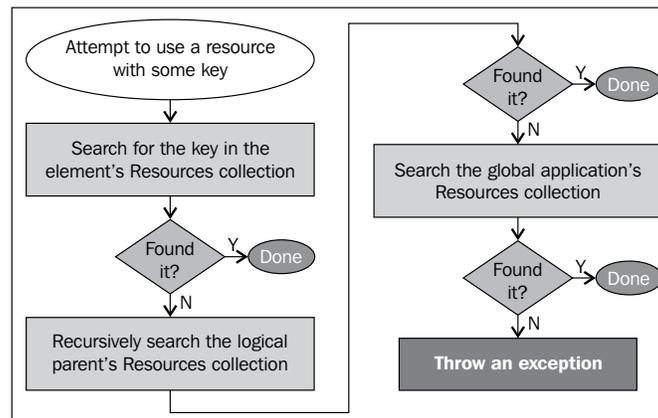
`LinearGradientBrush` can be turned into a logical resource (or simply a resource) and referenced by any element that requires it. To do that, the brush must be placed in a `ResourceDictionary` object. Fortunately, every element has a `Resources` property (of type `ResourceDictionary`) that we can use. This is typically done on the root XAML element (typically a `Page`), or (as we'll see in a moment) in the application's XAML (`App.Xaml`):

```
<Page.Resources>
  <LinearGradientBrush x:Key="brush1" EndPoint="0,1">
    <GradientStop Offset="0" Color="Green" />
    <GradientStop Offset=".5" Color="Orange" />
    <GradientStop Offset="1" Color="DarkRed" />
  </LinearGradientBrush>
</Page.Resources>
```

Any logical resource must have a key, because it's in a dictionary. That key is specified by the `x:Key` XAML directive. Once placed, a resource can be accessed from any element within that `Page` with the `StaticResource` markup extension as follows:

```
<Ellipse Fill="{StaticResource brush1}" />
<Rectangle Stroke="{StaticResource brush1}" StrokeThickness="40" />
```

The `StaticResource` markup extension searches for a resource with the specified key starting from the current element. If not found, the search continues on the resources with its parent element (say a `Grid`). If found, the resource is selected (it is created the first time it's requested), and `StaticResource` is done. If not found, the parent's parent is searched and so on. If the resource is not found at the top level element (typically a `Page`, but can be a `UserControl` or something else), the search continues in the application resources (`App.Xaml`). If not found, an exception is thrown. The search process can be summarized by the following diagram:



 Why is the markup extension called `StaticResource`? Is there a `DynamicResource`? `DynamicResource` exists in WPF only, which allows a resource to be replaced dynamically, with all those bound to it noticing the change. This is currently unsupported by WinRT.

There is no single call that is equivalent to `StaticResource`, although it's not difficult to create one if needed. The `FrameworkElement::Resources` property can be consulted on any required level, navigating to the parent element using the `Parent` property. The `Application::Resources` property has special significance, since any resources defined within it can be referenced by any page or element across the entire application. This is typically used to set various defaults for a consistent look and feel.

 It may be tempting to store actual elements as resources (such as buttons). This should be avoided because resources are singletons within their usage container; this means referencing that button more than once within the same page will cause an exception to be thrown on the second reference, because an element can be in the visual tree just once.

Resources are really intended for sharable objects, such as brushes, animations, styles, and templates.

Resources can be added dynamically by using the `ResourceDictionary::Insert` method (on the relevant `ResourceDictionary`) and removed by calling `ResourceDictionary::Remove`. This only has an effect on subsequent `{StaticResource}` invocations; already bound resources are unaffected.

 A `StaticResource` markup extension can be used by a resource as well. For this to work, any `StaticResource` must reference a resource that was defined earlier in the XAML; this is due to the way the XAML parser works. It cannot find resources that it has not yet encountered.

Managing logical resources

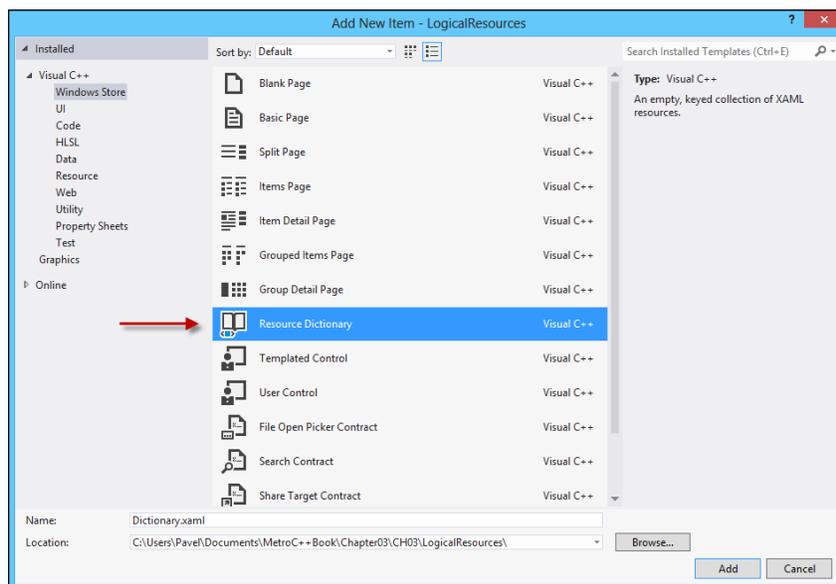
Logical resources may be of various types, such as brushes, geometries, styles, templates, and more. Placing all those resources in a single file, such as `App.xaml`, hinders maintainability. A better approach would be to separate resources of different types (or based on some other criteria) from their own files. Still, they must be referenced somehow from within a common file such as `App.xaml` so that they are recognized.

`ResourceDictionary` can incorporate other resource dictionaries using its `MergedDictionaries` property (a collection). This means a `ResourceDictionary` can reference as many resource dictionaries as desired and can have its own resources as well. The `Source` property must point to the location of `ResourceDictionary`. The default `App.xaml` created by Visual Studio contains the following (comments removed):

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

Indeed, we find a file called `StandardStyles.xaml` in the `Common` folder, which hosts a bunch of logical resources, with `ResourceDictionary` as its root element. For this file to be considered when `StaticResource` is invoked, it must be referenced by another `ResourceDictionary`, from a `Page` or the application (the application is more common). The `ResourceDictionary.MergedDictionaries` property contains other `ResourceDictionary` objects, whose `Source` property must point to the required XAML to be included (that XAML must have `ResourceDictionary` as its root element).

We can create our own `ResourceDictionary` XAML by using Visual Studio's **Add New Item** menu option and selecting **Resource Dictionary**:



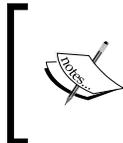
Duplicate keys

No two objects can have the same key in the same `ResourceDictionary` instance. `StaticResource` takes the first resource it finds with the specified key, even if that key already exists in a `ResourceDictionary`. What about merged dictionaries?

Merging different resource dictionaries may cause an issue – two or more resources with the same key that originate from different merged dictionaries. This is not an error and does not throw an exception. Instead, the selected object is the one from the last resource dictionary added (which has a resource with that key). Furthermore, if a resource in the current resource dictionary has the same key as any of the resources in its merged dictionaries, it always wins out. Here's an example:

```
<ResourceDictionary>
  <SolidColorBrush Color="Blue" x:Key="brush1" />
  <ResourceDictionary.MergedDictionaries>
    <ResourceDictionary Source="Resources/Brushes2.xaml" />
    <ResourceDictionary Source="Resources/Brushes1.xaml" />
  </ResourceDictionary.MergedDictionaries>
</ResourceDictionary>
```

Given this markup, the resource named `brush1` is a blue `SolidColorBrush` because it appears in the `ResourceDictionary` itself. This overrides any resources named `brush1` in the merged dictionaries. If this blue brush did not exist, `brush1` would be looked up in `Brushes1.xaml` first, as this is the last entry in the merged dictionaries collection.



XAML containing a `ResourceDictionary` as its root can be loaded dynamically from a string using the static `XamlReader::Load` method and then added as a merged dictionary, where appropriate.

Styles

Consistency in user interface is an important trait; there are many facets of consistency, one of which is the consistent look and feel of controls. For example, all buttons should look roughly the same – similar colors, fonts, sizes, and so on. Styles provide a convenient way of grouping a set of properties under a single object, and then selectively (or automatically, as we'll see later) apply it to elements.

Styles are always defined as resources (usually at the application level, but can also be at the `Page` or `UserControl` level). Once defined, they can be applied to elements by setting the `FrameworkElement::Style` property.

Here's a style defined as part of the `Resources` section of a `Page`:

```
<Page.Resources>
  <Style TargetType="Button" x:Key="style1">
    <Setter Property="FontSize" Value="40" />
    <Setter Property="Background">
      <Setter.Value>
        <LinearGradientBrush >
          <GradientStop Offset="0" Color="Yellow" />
          <GradientStop Offset="1" Color="Orange" />
        </LinearGradientBrush>
      </Setter.Value>
    </Setter>
    <Setter Property="Foreground" Value="DarkBlue" />
  </Style>
</Page.Resources>
```

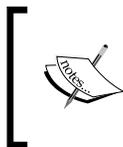
The style has a key (`style1`), and must have `TargetType`. This is the type the style may be applied to (and any derived types). The XAML parser has a type converter that converts `TargetType` to a `TypeName` object.

The main ingredient in `Style` is its `Setters` collection (which is also its `ContentProperty`). This collection accepts `Setter` objects, which need `Property` and `Value`. The property must be a dependency property (not usually a problem, as most element properties are dependency properties); these are provided as simple strings thanks to type converters used behind the scene.

The above markup sets up the properties `FontSize`, `Background` (with a complex property syntax because of the `LinearGradientBrush`), and `Foreground`—all for the `Button` controls.

Once defined, the style can be applied to elements using the usual `StaticResource` markup extension in XAML by setting the `FrameworkElement::Style` property, as in the following example:

```
<Button Content="Styled button" Style="{StaticResource style1}" />
```



Readers familiar with WPF may be wondering if the `TargetType` property can be omitted so that a greater control range can be covered. This is unsupported in the current version of WinRT.

Setting the style on an incompatible element type (such as a `CheckBox` control in this example) causes an exception to be thrown at page load time. If a `CheckBox` should also be able to use the same style, the `TargetType` can be changed to `ButtonBase` (which covers all button types).

 Use different styles for different elements, even if a base type seems to cover several controls. It's very likely that later some properties may need to be tweaked for a particular type, making it difficult to change the style. Build a different style for different concrete types. You can also use style inheritance (as described later) to shorten some of the markup.

What happens if an element with an applied style sets a property to a different value than the one from `Style`? The local value wins out. This means that the following button has a font size of 30 and not 40:

```
<Button Content="Styled button" FontSize="30"
        Style="{StaticResource style1}" />
```

Implicit (automatic) styles

The previous section showed how to create a style that has a name (`x:Key`) and how to apply it to elements. Sometimes, however, we would like a style to be applied automatically to all elements of a certain type, to give the application a consistent look. For example, we may want all buttons to have a certain font size or background, without the need for setting the `Style` property of each and every button. This makes creating new buttons easier, as the developer/designer doesn't have to know what style to apply (if any, the implicit style in scope will be used automatically).

To create a `Style` that is applied automatically, the `x:Key` attribute must be removed:

```
<Style TargetType="Button">
...
</Style>
```

The key still exists, as the `Style` property is still part of `ResourceDictionary` (which implements `IMap<Object, Object>`), but is automatically set to a `TypeName` object for the specified `TargetType`.

Once the `Style` property is defined and any `Button` element (in this example) in scope for `ResourceDictionary` of the `Style` property is in, that style will be applied automatically. The element can still override any property it wishes by setting a local value.



Automatic styles are applied to the exact type only, not to derived types. This means that an automatic style for `ButtonBase` is useless, as it's an abstract class.

An element may wish to revert to its default style and not have an implicit style applied automatically. This can be achieved by setting `FrameworkElement::Style` to `nullptr` (`x:Null` in XAML).

Style inheritance

Styles support the notion of inheritance, somewhat similar to the same concept in object orientation. This is done using the `BasedOn` property that must point to another style to inherit from. The `TargetType` of the derived style must be the same as in the base style.

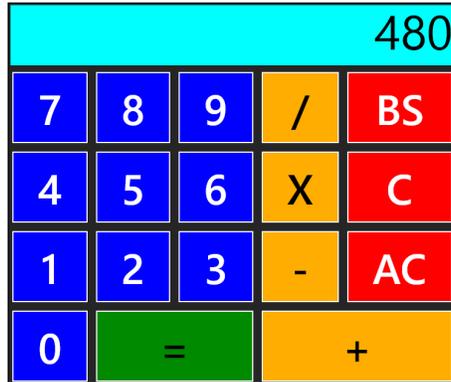
An inherited style can add `Setter` objects for new properties to set, or it can provide a different value for a property that was set by the base style. Here's an example for a base style of a button:

```
<Style TargetType="Button" x:Key="buttonBaseStyle">
  <Setter Property="FontSize" Value="70" />
  <Setter Property="Margin" Value="4" />
  <Setter Property="Padding" Value="40,10" />
  <Setter Property="HorizontalAlignment" Value="Stretch" />
</Style>
```

The following markup creates three inherited styles:

```
<Style TargetType="Button" x:Key="numericStyle"
  BasedOn="{StaticResource buttonBaseStyle}">
  <Setter Property="Background" Value="Blue" />
  <Setter Property="Foreground" Value="White" />
</Style>
<Style TargetType="Button" x:Key="operatorStyle"
  BasedOn="{StaticResource buttonBaseStyle}">
  <Setter Property="Background" Value="Orange" />
  <Setter Property="Foreground" Value="Black" />
</Style>
<Style TargetType="Button" x:Key="specialStyle"
  BasedOn="{StaticResource buttonBaseStyle}">
  <Setter Property="Background" Value="Red" />
  <Setter Property="Foreground" Value="White" />
</Style>
```

These styles are part of a simple integer calculator application. The calculator looks like this when running:



Most of the elements comprising the calculator are buttons. Here is the numeric button markup:

```
<Button Style="{StaticResource numericStyle}" Grid.Row="1"
        Content="7" Click="OnNumericClick" />
<Button Style="{StaticResource numericStyle}" Grid.Row="1"
        Grid.Column="1" Content="8" Click="OnNumericClick"/>
<Button Style="{StaticResource numericStyle}" Grid.Row="1"
        Grid.Column="2" Content="9" Click="OnNumericClick"/>
```

The operator buttons simply use a different style:

```
<Button Style="{StaticResource operatorStyle}" Grid.Row="3"
        Grid.Column="3" Content="-" Click="OnOperatorClick"/>
<Button Style="{StaticResource operatorStyle}" Grid.Row="4"
        Grid.Column="3" Content="+" Grid.ColumnSpan="2"
        Click="OnOperatorClick"/>
```

The = button uses the same style as operators, but changes its background by setting a local value:

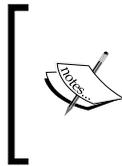
```
<Button Style="{StaticResource operatorStyle}" Grid.Row="4"
        Grid.Column="1" Grid.ColumnSpan="2" Content="="
        Background="Green" Click="OnCalculate"/>
```

The complete project is named `StyledCalculator` and can be found as part of the downloadable source for this chapter.

Style inheritance may seem very useful, but should be used with caution. It suffers from the same issues as object oriented inheritance in a deep inheritance hierarchy – a change in a base style up in the hierarchy can affect a lot of styles, being somewhat unpredictable, leading to a maintenance nightmare. Thus, a good rule of thumb to use is to have no more than two inheritance levels. Any more than that may cause things to get out of hand.

Store application styles

A Store app project created by Visual Studio has a default style file named `StandardStyles.xaml` in the `Common` folder. The file includes styles for all common elements and controls the set up for a common look and feel that is recommended as a starting point. It's certainly possible to change these styles or to inherit from them if needed.



WinRT styles are similar in concept to CSS used in web development to provide styling to HTML pages. The cascading part hints to the multilevel nature of CSS, much like the multilevel nature of WinRT styles (application, page, panel, specific element, and so on).

Summary

This chapter was all about XAML, the declarative language used to build user interfaces for Windows Store apps. XAML takes some time getting used to it, but its declarative nature and markup extensions cannot easily be matched by procedural code in C++ (or other languages). Designer-oriented tools, such as Expression Blend and even the Visual Studio designer make it relatively easy to manipulate XAML without actually writing XAML, but as developers and designers working with other XAML-based technologies have already realized, it's sometimes necessary to write XAML by hand, making it an important skill to acquire.

In the next chapter, we'll continue to use XAML heavily, while covering elements and controls, as well as layout, used in Windows 8 Store applications.

4

Layout, Elements, and Controls

The last chapter discussed XAML, a neutral language used to create objects and set up their properties. But XAML is just a tool—the content is what matters. Building an effective user interface involves at least selecting the best elements and controls to achieve the usability and required user experience.

In this chapter, we'll take a look at the WinRT layout system and discuss the major elements and controls that comprise most user interfaces.

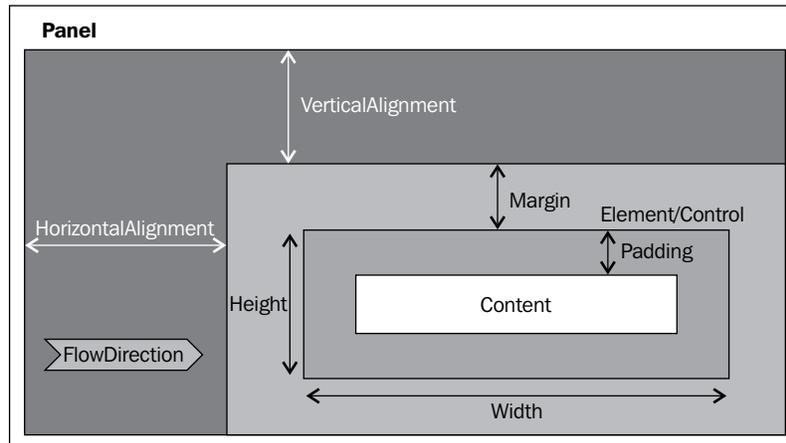
Introducing layout

Layout is the process of element placement and their size and position changes in face of user interactions or content changes. In the Win32/MFC world layout is mostly very simple and limited. Controls are placed using distance from the top-left point of a window and their size is explicitly specified. This model has very limited flexibility; if a control's content changes (for example, becomes bigger), the control can't compensate automatically. Other similar changes have no effect on the UI layout.

WinRT, on the other hand, provides a much more flexible model based around a set of layout panels that provide different ways to lay out elements. By combining those panels in various ways, complex and adaptive layouts can be created.

Layout is a two-step process. First, the layout container asks each of its children for their desired size. In the second step, it uses whatever logic is applicable (for that panel type) to determine at what position and what size each child element be, and places each child in that rectangular area.

Each element indicates to its parent its size requirements. The following figure summarizes the most important properties related to these requirements:



Here's a quick breakdown of these important properties:

- **Width/Height** - the width and height of the element in question. This is not typically set (unset value being the default value - "Auto" - in XAML; more on that in a moment), meaning the element would like to be as big as it needs to be. Nevertheless, these may be set if needed. The actual (rendered) width and height of an element is available using the `FrameworkElement::ActualWidth` and `ActualHeight` read-only properties.
- **MinWidth/MaxWidth/MinHeight/MaxHeight** - the minima and maxima for the element size (not shown in the figure). Default values are 0 for the minima, and infinite for the maxima.
- **Margin** - a "breathing space" around the element. This is of type `Thickness` that has four fields (`Left`, `Top`, `Right`, and `Bottom`) that determine the amount of space around the element. It can be specified in XAML using four values (left, top, right, bottom), two values (the first is left and right, the second top and bottom), or a single number (the same distance in all four directions).
- **Padding** - same idea as `Margin`, but determines the space between the outer edge of the element and its content (if any). This is typed as `Thickness` as well, and is defined by the `Control` base class and some other special elements, such as `Border` and `TextBlock`.
- **HorizontalAlignment/VerticalAlignment** - specifies how to align the element against its parent if extra space is available. Possible values are `Left`, `Center`, `Right`, and `Stretch` (for `HorizontalAlignment`) and `Top`, `Center`, `Bottom`, and `Stretch` for `VerticalAlignment`.

- `HorizontalAlignment/VerticalContentAlignment` (not shown in the figure) – same idea as `Horizontal/VerticalAlignment`, but for the Content of the element (if any).
- `FlowDirection` – can be used to switch the layout direction from the default (`LeftToRight`) to `RightToLeft`, suitable for right to left languages, such as Hebrew or Arabic. This effectively turns every "left" to "right" and vice versa.

After the layout panel collects the required size of each child element (by calling `UIElement::Measure` on each one), it moves on to the second stage of the layout – arranging. In this stage, the panel calculates the final positions and sizes of its child elements based on the element's desired size (`UIElement::DesiredSize` read-only property) and whatever algorithm is appropriate for that panel and informs each element of the resulting rectangle by calling `UIElement::Arrange`. This procedure can go on recursively, because an element can be a layout panel in itself, and so on. The result is known as a visual tree.



Interested readers may be wondering how to specify the "Auto" XAML value for, for example, `Width` in code, given that this is a double value. This is done by including `<limits>` and then using the expression `std::numeric_limits<double>::quiet_NaN()`. Similarly, to specify an infinite value, use `std::numeric_limits<double>::infinity()`.

Layout panels

All layout panels must derive from the `Windows::UI::Xaml::Controls::Panel` class, itself deriving from `FrameworkElement`. The main addition `Panel` is the `Children` property (also its `ContentProperty` for easier XAML authoring), which is a collection of elements implementing the `IVector<UIElement>` interface. By using the `Children` property, elements can be dynamically added or removed from a `Panel`. WinRT provides a bunch of specific panels, each with its own layout logic, providing flexibility in creating the layout. In the following sections, we'll take a look at some of the built-in layout panels.



All panel classes, as well as elements and controls described later, are assumed to exist in the `Windows::UI::Xaml::Controls` namespace, unless noted otherwise.

StackPanel

`StackPanel` is one of the simplest layout panels. It lays out its children in a *stack*, one after the other, horizontally or vertically, based on the `Orientation` property (`Vertical` being the default).

When used for vertical layout, each element gets the height it wants and all the available width, and vice versa. Here's an example of `StackPanel` with some elements:

```
<StackPanel Orientation="Horizontal" >
  <TextBlock Text="Name:" FontSize="30" Margin="0,0,10,0"/>
  <TextBox Width="130" FontSize="30"/>
</StackPanel>
```

This is how it looks at runtime (after some text is entered):



`StackPanel` is useful for small layout tasks, as part of other, more complex layout panels.

Grid

`Grid` is probably the most useful layout panel because of its flexibility. It creates a table-like layout of cells. Elements can occupy single or multiple cells, and cell size is customizable. We've used `Grid` to create the calculator layout from the previous chapter. Here's another `Grid` example (wrapped in a `Border` element), a piece of markup for a login page:

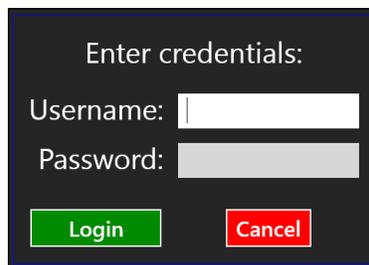
```
<Border HorizontalAlignment="Center" VerticalAlignment="Center"
  BorderThickness="1" BorderBrush="Blue" Padding="10">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <TextBlock Text="Enter credentials:" Grid.ColumnSpan="2"
```

```

        TextAlignment="Center" FontSize="40" Margin="20"/>
<TextBlock Text="Username:" TextAlignment="Right"
    Margin="10" Grid.Row="1" FontSize="40"
    VerticalAlignment="Bottom"/>
<TextBox HorizontalAlignment="Left" Width="250"
    Grid.Row="1" Grid.Column="1" Margin="10"
    FontSize="30" />
<TextBlock Text="Password:" TextAlignment="Right"
    Margin="10" Grid.Row="2" FontSize="40"
    VerticalAlignment="Bottom" />
<PasswordBox HorizontalAlignment="Left" Width="250"
    Grid.Row="2" Grid.Column="1" Margin="10"
    FontSize="30" />
<Button Content="Login" HorizontalAlignment="Stretch"
    Grid.Row="3" FontSize="30" Margin="10,30,10,10"
    Background="Green" />
<Button Content="Cancel" HorizontalAlignment="Center"
    Grid.Row="3" Grid.Column="1" FontSize="30"
    Margin="10,30,10,10" Background="Red" />
</Grid>
</Border>

```

This is how it looks at runtime:



The number of rows and columns is not specified by simple properties. Instead, it's specified using `RowDefinition` objects (for rows) and `ColumnDefinition` objects (for columns). The reason has to do with the size and behavior that can be specified on a row and/or column basis.

`RowDefinition` has a `Height` property, while `ColumnDefintion` has a `Width` property. Both are of type `GridLength`. There are three options for setting `GridLength`:

- A specific length
- A star-based (relative) factor (this is the default, and factor equals 1)
- Automatic length

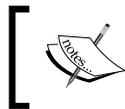
Setting `Height` (of a `RowDefinition`) or `Width` (of a `ColumnDefinition`) to a specific number makes that row/column that particular size. In code it's equivalent to `ref new GridLength(len)`.

Setting `Height` or `Width` to "Auto" (in XAML) makes the row/column as high/wide as it needs to be based on the tallest/widest element placed within that row/column. In code, it's equivalent to the static property `GridLength::Auto`.

The last option (which is the default) is setting `Height/Width` to `n*` in XAML, where `n` is a number (1 if omitted). This sets up a relationship with other rows/columns that have a "star" length. For example, here are three rows of a `Grid`:

```
<RowDefinition Height="2*" />
<RowDefinition />
<RowDefinition Height="3*" />
```

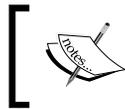
This means that the first row is twice as tall as the second row (`Height="*"`). The last row is three times taller than the second row, and is one-and-a-half times taller than the first row. These relations are maintained, even if the `Grid` is dynamically resized due to layout changes.



The value for the "star" factor need not be a whole number; it can be a floating point value as well. The ratio is what matters, not the actual numbers.

Elements are placed in specific grid cells using the attached `Grid.Row` and `Grid.Column` properties (both default to zero, meaning the first row and column).

Elements occupy one cell by default. This can be changed by using the `Grid.RowSpan` and `Grid.ColumnSpan` properties (this was set for the first `TextBlock` in the previous XAML).



It's ok to specify `ColumnSpan` or `RowSpan` with a large number to ensure an element will occupy all cells in a given direction. The `Grid` automatically will use the actual row/column count.

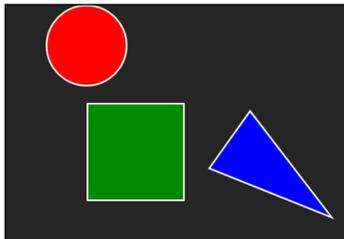
Canvas

`Canvas` models the classic Win32/MFC layout – exact positioning. This type of layout is useful if exact coordinates are required, such as the case with graphs, animations, graphic games, and other complex drawings. `Canvas` is the fastest layout panel, just because it does very little layout (in fact almost none at all).

Here's an example of Canvas hosting some shapes:

```
<Canvas x:Name="_canvas" >
  <Ellipse Stroke="White" StrokeThickness="2" Fill="Red"
    Width="100" Height="100" Canvas.Left="50"/>
  <Rectangle Stroke="White" StrokeThickness="2" Fill="Green"
    Canvas.Left="100" Canvas.Top="120" Width="120"
    Height="120"/>
  <Polygon Points="0,0 150,60 50,-70" Canvas.Left="250"
    Canvas.Top="200" Fill="Blue" Stroke="White"
    StrokeThickness="2" />
</Canvas>
```

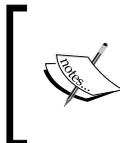
The output is shown as follows:



The placement coordinates are set using the `Canvas.Left` and `Canvas.Top` attached properties (both default to zero, meaning the top-left corner of the Canvas). The only other attached property that Canvas defines is `ZIndex`. This specifies the relative order of rendering the elements inside the Canvas, where a large value places the element on top. By default, elements defined later in XAML are higher in the Z-order.

As a more complex example, suppose we wanted to allow the user to drag the shapes around the Canvas using the mouse or a finger. First, we'll add event handlers for pointer pressing, releasing, and movement:

```
<Canvas x:Name="_canvas" PointerPressed="OnPointerPressed" PointerReleased="OnPointerReleased" PointerMoved="OnPointerMoved">
```



The concept of "pointer" replaces the "mouse" event names that may be familiar from Win32/MFC/WPF/Silverlight; pointer is generic, representing any pointing device, be it a mouse, stylus, or a finger.

The pointer-related events use the bubbling strategy, meaning any pressing on elements (such as the shapes used) will raise `PointerPressed` on that shape first, and if unhandled (as in this case), bubble up to its parent (the `Canvas`) where it does get handled.

The `PointerPressed` event may be handled like so:

```
void MainPage::OnPointerPressed(Platform::Object^ sender,
    PointerRoutedEventArgs^ e) {
    _element = (FrameworkElement^)e->OriginalSource;
    if(_element == _canvas) return;
    _lastPoint = e->GetCurrentPoint(_canvas)->Position;
    _lastPoint.X -= (float)Canvas::GetLeft(_element);
    _lastPoint.Y -= (float)Canvas::GetTop(_element);
    _canvas->CapturePointer(e->Pointer);
    e->Handled = true;
    _isMoving = true;
}
```

Since this event fires on the `Canvas`, even though the original elements are the children of one of the `Canvas`, how do we get to that child element? The sender argument is the actual object that sent the event—the `Canvas` in this case. The child element is indicated by the `PointerRoutedEventArgs::OriginalSource` property (inherited from `RoutedEventArgs`). First, a check is made to see if the pointer pressing is in fact on the `Canvas` itself. If so, the method returns immediately.

 With the preceding `Canvas`, this will never happen. The reason is that the default `Background` of the `Canvas` (or any other `Panel` for that matter) is `nullptr`, so that no events can register on it—they propagate to its parent. If events on the `Canvas` itself are desirable, `Background` must be some non-`nullptr` `Brush`; using `ref new SolidColorBrush(Colors::Transparent)` is good enough if the parent's background `Brush` is to show through.

Next, the position of the press is extracted in two steps, first using `PointerRoutedEventArgs::GetCurrentPointer()` (this is a `PointerPoint` object) and second, with the `PointerPoint::Position` property (of type `Windows::Foundation::Point`). Then the point is adjusted to become the offset of the pressing point to the top-left corner position of the element, this helps in making the later movement accurate.

Capturing the pointer (`UIElement::CapturePointer`) ensures the `Canvas` continues to receive pointer-related events no matter where the pointer is. Setting `PointerRoutedEventArgs::Handled` to `true` prevents further bubbling (as there's no need here), and a flag indicating movement should occur from now on, until the pointer released is set (another private member variable).

 Pointer capturing is similar in concept to mouse capturing that exists in other UI technologies (Win32/MFC/WPF/Silverlight).

When the pointer moves, the element in question needs to move as well, as long as the pointer has not yet been released:

```
void MainPage::OnPointerMoved(Platform::Object^ sender,
    PointerRoutedEventArgs^ e) {
    if(_isMoving) {
        auto pos = e->GetCurrentPoint(_canvas)->Position;
        Canvas::SetLeft(_element, pos.X - _lastPoint.X);
        Canvas::SetTop(_element, pos.Y - _lastPoint.Y);
        e->Handled = true;
    }
}
```

The main idea here is moving the element by setting the attached Canvas properties `Canvas.Left` and `Canvas.Top` (using the static `Canvas::SetLeft` and `Canvas::SetTop` methods).

When the pointer is finally released, we just need to do some cleaning up:

```
void MainPage::OnPointerReleased(Platform::Object^ sender,
    PointerRoutedEventArgs^ e) {
    _isMoving = false;
    _canvas->ReleasePointerCapture(e->Pointer);
    e->Handled = true;
}
```

The complete code is in a project called `CanvasDemo`, part of the downloadable code for this chapter.

 The pointer-related methods may seem more complex than needed, but they're not. Since touch input is (more often than not) multi-touch, what happens if two fingers press on two different elements at the same time, trying to move them? Multiple `PointerPressed` events may trigger, and there should be a way to distinguish one finger from another. The previous code is implemented while assuming only one finger is used at a time.

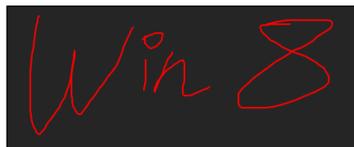
Adding children to a panel dynamically

The `Panel.Children` property can be manipulated programmatically (for any `Panel` type). For example, using a `Canvas` as a drawing surface, we can use the previous pointer events to add `Line` elements that connect to each other to create a drawing. When the pointer is moving (after being pressed), `Line` objects may be added with code like the following:

```
void MainPage::OnPointerMoved(Object^ sender,
    PointerRoutedEventArgs^ e) {
    if(_isDrawing) {
        auto pt = e->GetCurrentPoint(_canvas);
        auto line = ref new Line();
        line->X1 = _lastPoint->Position.X;
        line->Y1 = _lastPoint->Position.Y;
        line->X2 = pt->Position.X;
        line->Y2 = pt->Position.Y;
        line->StrokeThickness = 2;
        line->Stroke = _paintBrush;
        _canvas->Children->Append(line);
        _lastPoint = pt;
    }
}
```

A `Line` object is constructed, its properties set as appropriate, and finally it's added to the `Children` collection of the `Canvas`. Without this last step, the `Line` object is not attached to anything and simply dies out when its reference goes out of scope. `_paintBrush` is a `Brush` field maintained by the hosting page (not shown).

The complete source is in a project called `SimpleDraw`, part of the downloadable code for this chapter. Here's a sample drawing done with this application:



VariableSizedWrapGrid

`StackPanel`, `Grid`, and `Canvas` are fairly straightforward; they are not much different from their counterparts in WPF or Silverlight. WinRT has some more interesting panels, starting with `VariableSizedWrapGrid`.

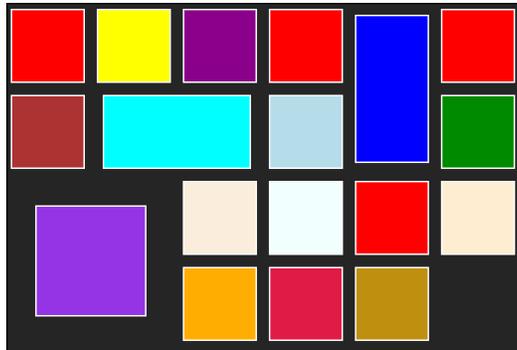
As its name suggests, it's essentially a grid, where items are arranged in rows or columns (depending on the `Orientation` property). When space runs out, or if the number of items in a row/column reaches the limit set by the `MaximumRowsOrColumns` property, the layout continues on the next row/column.

A final twist with `VariableSizedWrapGrid` is that it is available with two attached properties, `RowSpan` and `ColumnSpan`, that can change the size of an item to take up more than one cell. Here's an example `VariableSizedWrapGrid` with a bunch of `Rectangle` elements:

```
<Grid Background=
  "{StaticResource ApplicationPageBackgroundThemeBrush}">
  <Grid.Resources>
    <Style TargetType="Rectangle">
      <Setter Property="Stroke" Value="White" />
      <Setter Property="StrokeThickness" Value="2" />
      <Setter Property="Margin" Value="8" />
      <Setter Property="Width" Value="100" />
      <Setter Property="Height" Value="100" />
      <Setter Property="Fill" Value="Red" />
    </Style>
  </Grid.Resources>
  <VariableSizedWrapGrid x:Name="_grid"
    Orientation="Horizontal"
    MaximumRowsOrColumns="6">
    <Rectangle />
    <Rectangle Fill="Yellow" />
    <Rectangle Fill="Purple"/>
    <Rectangle />
    <Rectangle Fill="Blue" VariableSizedWrapGrid.RowSpan="2"
      Height="200"/>
    <Rectangle />
    <Rectangle Fill="Brown"/>
    <Rectangle VariableSizedWrapGrid.ColumnSpan="2"
      Width="200" Fill="Aqua"/>
    <Rectangle Fill="LightBlue"/>
    <Rectangle Fill="Green"/>
    <Rectangle VariableSizedWrapGrid.ColumnSpan="2"
      VariableSizedWrapGrid.RowSpan="2" Width="150"
      Height="150" Fill="BlueViolet"/>
    <Rectangle Fill="AntiqueWhite"/>
    <Rectangle Fill="Azure"/>
    <Rectangle />
    <Rectangle Fill="BlanchedAlmond"/>
```

```
<Rectangle Fill="Orange"/>
<Rectangle Fill="Crimson"/>
<Rectangle Fill="DarkGoldenrod"/>
</VariableSizedWrapGrid>
</Grid>
```

This is the result:



Panel virtualization

All the previously discussed panels create their child elements as soon as they are added. For most scenarios, this is acceptable. However, if the item count is very high (hundreds or more), the panel's performance may degrade, as many elements need to be created and managed, taking up memory and wasting CPU cycles upon creation, or when layout changes occur. A virtualizing panel does not create all the elements up front for items it holds; instead, it only creates actual elements that are currently visible. If the user scrolls to see more data, elements are created as needed. Elements that scroll out of view may be destroyed. This scheme conserves memory and CPU time (at creation time).

The `VirtualizingPanel` class is an abstract base class for all virtualization panel implementations in WinRT. A further refinement of `VirtualizingPanel` is `OrientedVirtualizingPanel`, indicating a panel with an inherent orientation. WinRT provides the three virtualizing panels, as we'll see in a moment.

All virtualizing panels have one more interesting trait, they can only be used to customize the panel used for controls based on `ItemsControl` (typically with data binding); they cannot be used as normal panels are used – by placing items inside them (in XAML or programmatically). Full discussion of `ItemsControl` and its derivatives is reserved for a later part of this chapter; for now we'll take a quick look at the way the existing virtualizing panels work; we'll see examples of usage later, when `ItemsControl` is discussed.

Virtualizing panels

The easiest virtualizing panel to understand is `VirtualizingStackPanel`. It acts just like a regular `StackPanel`, but it virtualizes elements that are currently not visible.

`WrapGrid` is similar to `VariableSizedWrapGrid`, but without the "variable" part (it has no attached properties that can change an individual element's size). It's used in a `GridView` as the default panel (`GridView` is one of the many types derived from `ItemsControl`). It can be customized with properties such as `Orientation`, `ItemHeight`, `ItemWidth`, and `MaximumRowsOrColumns`, which are mostly self-explanatory.

`CarouselControl` is similar to `VirtualizingStackPanel`, with the added capability to roll over to the first item when the last one is reached. It's used as the default panel for a `ComboBox` and, in fact, cannot be used by other control, making it pretty useless generally.

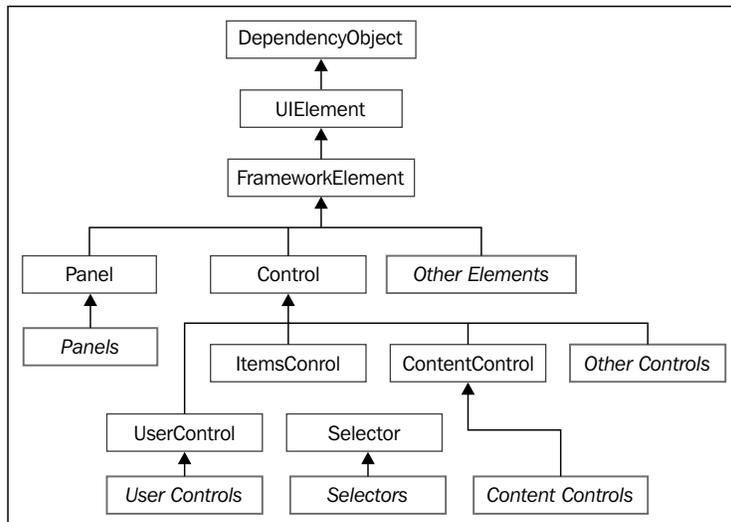
Working with elements and controls

The difference between "elements" and "controls" is not that important in practice, but it is useful to understand the distinction.

Elements derive from `FrameworkElement` (directly or indirectly), but not from `Control`. They have some look and provide some functionality that is customizable mostly by changing properties. For example, `Ellipse` is an element. There's no way to change the fundamental appearance of an `Ellipse` (and it would be illogical to be able to turn an `Ellipse` into, for example, a rectangle). It can still be customized in some ways using properties such as `Stroke`, `StrokeThickness`, `Fill`, and `Stretch`.

Controls, on the other hand, derive (directly or indirectly) from the `Control` class. `Control` adds a bunch of properties, of which the most significant is the `Template` property. This allows for completely changing the control's appearance without affecting its behavior. Furthermore, all that can be achieved with XAML alone, without code or any class derivation. We'll discuss control templates in *Chapter 6, Components, Templates, and Custom Elements*.

The following class diagram shows some of the fundamental element-related classes in WinRT:



In the following sections, we'll go over the various groups of elements and controls (based on derivation and usage categories), studying their main features and usage. In each group, we'll look at some of the more useful or unique controls. These sections are by no means complete (and are not intended as such); further information can be found in the official MSDN documentation and samples.

Content controls

Content controls derive from the `ContentControl` class (itself deriving from `Control`). `ContentControl` adds two important properties: `Content` (also its `ContentProperty` attribute, making it easy to set in XAML) and `ContentTemplate`. A simple example of a `ContentControl` is `Button`:

```
<Button Content="Login" FontSize="30" />
```

This `Content` property may seem like a string, but in fact it's typed as `Platform::Object^`, meaning it can be anything at all.



It may seem odd that "anything" is specified using `Platform::Object`; after all, WinRT is based on COM, so there must be an interface behind this. And there is, `Platform::Object` is, in fact, a projected replacement for the `IInspectable` interface pointer.

A type derived from `ContentControl` renders its `Content` using the following rules:

- If it's a string, `TextBlock` is rendered with its `Text` set to the string value.
- If it derives from `UIElement`, it's rendered as is.
- Otherwise (`Content` does not derive from `UIElement` and is not a string), if `ContentTemplate` is `nullptr`, then the content is rendered as a `TextBlock` with its `Text` set to a string representation of the `Content`. Otherwise, the supplied `DataTemplate` is used for rendering.

The preceding rules are used for any type derived from `ContentControl`. In the case of the previous button, the first rule is used, as the `Content` of the `Button` is the string **Login**. Here's an example that uses the second rule:

```
<Button>
  <StackPanel Orientation="Horizontal">
    <Image Source="assets/upload.png" Stretch="None" />
    <TextBlock Text="Upload" FontSize="35"
      VerticalAlignment="Center" Margin="10,0,0,0" />
  </StackPanel>
</Button>
```

The resulting button is shown as follows:



The resulting control is still a button, but its `Content` is set to a type derived from `UIElement` (in this case a `StackPanel`).

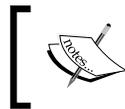
The third rule is the most interesting. Suppose we have a simple data object implemented like so:

```
namespace ContentControlsDemo {
  public ref class Book sealed {
  public:
    property Platform::String^ BookName;
    property Platform::String^ AuthorName;
```

```
        property double Price;
    };
}
```

Given this implementation, let's create a `Book` instance in XAML as a resource:

```
<Page.Resources>
    <local:Book x:Key="book1" BookName="Windows Internals"
        AuthorName="Mark Russinovich" Price="50.0" />
</Page.Resources>
```



To make this compile without errors, `#include "book.h"` must be added to `MainPage.xaml.h`. The reason for that will become clear in the next chapter.

Now, we can set the `Content` of a type derived from the `ContentControl` (such as `Button`) to that `Book` object:

```
<Button Content="{StaticResource book1}" FontSize="30"/>
```

Running the application shows the following result:

ContentControlsDemo.Book

The result is simply the fully qualified type name of the class (including namespace); this is not always the case, it depends on the default control template of the control in question. In any case, it's obvious it's not usually what we want. To get custom rendering for the object, a `DataTemplate` is needed, plugged into the `ContentTemplate` property.

Here's an example that creates a `DataTemplate` for use within the `Button` in question:

```
<Button Margin="12" Content="{StaticResource book1}" >
    <Button.ContentTemplate>
        <DataTemplate>
            <Grid>
                <Grid.RowDefinitions>
                    <RowDefinition Height="Auto" />
                    <RowDefinition Height="Auto" />
                </Grid.RowDefinitions>
                <Grid.ColumnDefinitions>
                    <ColumnDefinition />
                    <ColumnDefinition Width="15" />
                    <ColumnDefinition Width="Auto" />
                </Grid.ColumnDefinitions>
            </Grid>
        </DataTemplate>
    </Button.ContentTemplate>
</Button>
```

```

</Grid.ColumnDefinitions>
<TextBlock FontSize="35" Foreground="Yellow"
  Text="{Binding BookName}" />
<TextBlock Grid.Row="1" FontSize="25"
  Foreground="Orange"
  Text="{Binding AuthorName}" />
<TextBlock FontSize="40" Grid.Column="2"
  Grid.RowSpan="2" TextAlignment="Center"
  VerticalAlignment="Center">
  <Span FontSize="25">Just</Span><LineBreak />
  <Span FontSize="40">${</Span>
  <Run Text="{Binding Price}" FontSize="40" />
</TextBlock>
</Grid>
</DataTemplate>
</Button.ContentTemplate>
</Button>

```

There are a few things to note here:

- A `DataTemplate` can contain a single element (typically a `Panel` — `Grid` in this example), and can build any required UI.
- Using the properties of the actual content is done via data binding expressions, expressed with the `{Binding}` markup extension, with the property name. A complete treatment of data binding is found in the next chapter.
- To make the properties work with a data object (a `Book` in this case), the class (`Book`) must be decorated with the `Bindable` attribute like so:

```

[Windows::UI::Xaml::Data::Bindable]
public ref class Book sealed {

```

The result looks like the following:



Data templates are powerful tools for visualizing data objects; we'll encounter more of those later. For now, it's important to realize that every type deriving from `ContentControl` has that customization ability.

In the following sections, we'll discuss some of the common `ContentControl`-derived types.

Buttons

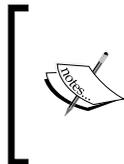
As we've already seen, the classic `Button` control is a `ContentControl`, meaning it can have any content, but still act like a button. Most of the functionality of the `Button` is derived from its abstract base class, `ButtonBase`. `ButtonBase` declares the ubiquitous `Click` event, along with some other useful properties:

- `ClickMode` – indicates what constitutes a "click": `Release`, `Press`, or `Hover`. Naturally, this is mostly applicable to a mouse.
- `Command` – indicates which command (if any) to invoke when the button is clicked (commands will be discussed in the next chapter).
- `CommandParameter` – an optional parameter that is sent with the invoked command.

`Button` derives from `ButtonBase` and adds nothing in terms of members, except being concrete, rather than abstract.

Another `ButtonBase` derivative is `HyperlinkButton`. It renders as a web hyperlink by default, and adds a `NavigationUri` property that causes automatic navigation to the specified URI; the `Click` event is typically not handled.

`RepeatButton` (in the `Windows::UI::Xaml::Controls::Primitives` namespace) is another `ButtonBase` derivative. It raises the `Click` event, as long as the button is pressed; the rate of `Click` events can be specified using the `Delay` (first `Click` event) and `Interval` (period of `Click` event raises) properties.



`RepeatButton` is less useful by itself; it's mostly useful as a building block of other, more complex, controls. This is hinted by placing the control in the `Primitives` subnamespace. As an example, `RepeatButton` composes several parts of a `ScrollBar` (in itself in the `Primitives` namespace).

Two other useful button controls are `CheckBox` and `RadioButton`. Both derive from a common base, `ToggleButton`. `ToggleButton` defines the `IsChecked` property, which can have three values (`true`, `false`, or `nullptr`). The latter indicates an indeterminate state, supported by `CheckBox` (but not by `RadioButton`). `ToggleButton` also declares the `IsThreeState` property, to indicate whether the third state should be allowed. Finally, it defines three events, `Checked`, `Unchecked`, and `Indeterminate`.

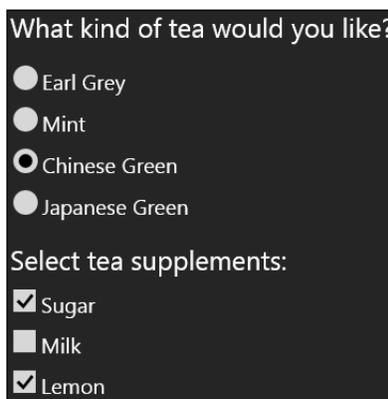
CheckBox adds nothing to ToggleButton except becoming concrete. RadioButton adds just one property, `GroupName` (a string). This allows grouping of RadioButton controls, to be used as an exclusion group. By default, all RadioButton controls under the same immediate parent become a group (only one in that group can have the `IsChecked` property set to true at a time). If `GroupName` is specified, all RadioButtons with the same `GroupName` are considered a group.

Here's a simple example that uses CheckBox and RadioButton controls:

```
<StackPanel>
  <TextBlock Text="What kind of tea would you like?"
    FontSize="25" Margin="4,12"/>
  <RadioButton Content="Earl Grey" IsChecked="True" Margin="4"
    FontSize="20" />
  <RadioButton Content="Mint" Margin="4" FontSize="20"/>
  <RadioButton Content="Chinese Green" Margin="4"
    FontSize="20"/>
  <RadioButton Content="Japanese Green" Margin="4"
    FontSize="20"/>

  <TextBlock Text="Select tea supplements:" FontSize="25"
    Margin="4,20,4,4" />
  <CheckBox Content="Sugar" Margin="4" FontSize="20" />
  <CheckBox Content="Milk" Margin="4" FontSize="20" />
  <CheckBox Content="Lemon" Margin="4" FontSize="20" />
</StackPanel>
```

The resulting display, after some selections, is shown as follows:



ScrollViewer

`ScrollViewer` is a content control that hosts a single child (its `Content` property, just like any other `ContentControl`) and uses a pair of `ScrollBar` controls to support scrolling. The most important properties are `VerticalScrollBarVisibility` and `HorizontalScrollBarVisibility`, which indicate the way scrolling should work and the way the scroll bars present themselves. There are four options (`ScrollBarVisibility` enumeration):

- `Visible` - the scroll bar is always visible. If the content does not require scrolling, the scroll bar is disabled.
- `Auto` - the scroll bar appears if needed and disappears if not needed.
- `Hidden` - the scroll bar is not shown, but scrolling is still possible using the keyboard, touch, or programmatically.
- `Disabled` - the scroll bar is hidden and no scrolling is possible. The `ScrollViewer` does not give more space than it has to the content (in that dimension).

The default values are `Visible` for `VerticalScrollBarVisibility` and `Disabled` for `HorizontalScrollBarVisibility`.

Another useful feature of the `ScrollViewer` is its ability to allow zooming in or out for the `Content` with a zoom/pinch touch gestures. This is controlled through the `ZoomMode` property (`Enabled` or `Disabled`).

The `HorizontalScrollBarVisibility`, `VerticalScrollBarVisibility`, and `ZoomMode` properties are exposed as attached properties as well, so they are relevant to other controls that internally use a `ScrollViewer`, such as `ListBox` or `GridView`. Here's a simple example that changes the way a horizontal scroll bar is presented in a `ListBox`:

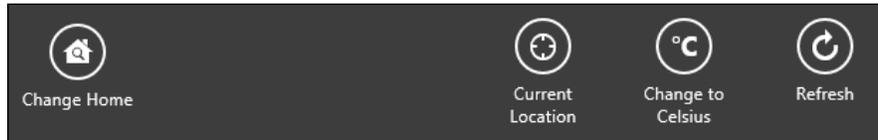
```
<ListBox ScrollViewer.HorizontalScrollBarVisibility="Hidden">
```

Other content controls to note

The following is a brief description of some of the other `ContentControl`-derived types in WinRT.

AppBar

`AppBar` is a `ContentControl` that is used for the application bar, typically appearing at the bottom (sometimes at the top) if the user swipes from the bottom (or top) or right clicks the mouse. It typically hosts a horizontal `StackPanel` with buttons for various options. Here's an example from the Weather app that is available with any Windows 8 installation:



Frame

`Frame` is a `ContentControl` that's used for navigation among controls derived from `Page`. Calling the `Navigate` method with a `Page` type "navigates" to that page, by creating an instance and calling some virtual methods: `OnNavigatedFrom` on the old page (if any) and `OnNavigatedTo` on the new page. By default, the application wizard creates a `Frame` object in the `App::OnLaunched` method (event handler for the `Launched` event), and then quickly navigates to the `MainPage` with the following code:

```
rootFrame->Navigate (TypeName (MainPage::typeid) , args->Arguments)
```

The second parameter to `Navigate` is an optional context argument available in the `OnNavigatedTo` override (in `NavigationEventArgs::Parameter`).

The `Frame` object maintains a back stack of pages which can be navigated using methods such as `GoBack` and `GoForward`. The `CanGoBack` and `CanGoForward` read-only properties can help maintain state on buttons that serve navigation purposes.

Navigating to previously visited pages can create new instances of those pages or reuse instances. The `CacheSize` property enables setting a maximum number of cached pages that are retained in memory during navigation. To enable any kind of caching for a particular `Page` instance, its `Page::NavigationCacheMode` property must be set to `Enabled` or `Required` (`Disabled` being the default). `Enabled` works with the cache, while `Required` always maintains the page state in memory (the `Required` setting does not count against the `Frame::CacheSize` value).

SelectorItem

`SelectorItem` is an abstract base class for items that may be selectable in `ItemsControl` controls (see the next section for a description of `ItemsControl`). It adds just one property: `IsSelected`. Derived types are containers for items in their respective collection-based control: `ListBoxItem` (in a `ListBox`), `GridViewItem` (in a `GridView`), `ListViewItem` (in a `ListView`), and so on.

Collection-based controls

The following sections discuss controls that hold more than one data item. These all derive from the `ItemsControl` class that provides the basic structure of all derived types.

The `Items` read-only property is the collection of objects hosted in this `ItemsControl` (of type `ItemCollection`, also its `ContentProperty`). Objects can be added with the `Append` and `Insert` methods, and removed with the `Remove` and `RemoveAt` methods (any kind of object can be part of the `ItemsControl`). Although this may sound appealing, this is not the typical way of working with an `ItemsControl` or its derived types; usually a collection of objects is set to the `ItemsSource` property (typically with a data binding expression) and that automatically uses the `Items` property behind the scenes to populate the control. We'll see this in action in *Chapter 5, Data Binding*.

The `ItemsPanel` property allows changing the default `Panel` hosting the items in the particular `ItemsControl`. For example, a `ListView` uses a vertical `VirtualizingStackPanel` as its default `Panel`. This can be changed to `WrapGrid` with the following XAML snippet inside the `ListView` element:

```
<ListView.ItemsPanel>
  <ItemsPanelTemplate>
    <WrapGrid Orientation="Horizontal"/>
  </ItemsPanelTemplate>
</ListView.ItemsPanel>
```

The `ItemTemplate` property may be set to a `DataTemplate`, as a way to show an object that is part of the collection. `ItemTemplate` has the same purpose and rules as `ContentControl::ContentTemplate`, but is applicable to each and every object in the `ItemsControl`. We'll see examples of `ItemTemplate` usage in the next chapter.

`DisplayMemberPath` is a `String` property that may be used if `ItemTemplate` is `nullptr` to show some property (or subproperty) of objects in this `ItemsControl`. As an example, suppose we use the following `Book` class (defined earlier):

```
[Bindable]
public ref class Book sealed {
```

```
public:
    property Platform::String^ BookName;
    property Platform::String^ AuthorName;
    property double Price;
};
```

Creating an array of such `Book` objects and placing it in the `ItemsControl::ItemsSource` property (or adding them manually through `Items->Append` method calls), would show, by default, the `Book` type name (assuming no `ItemTemplate` has been set). Setting `DisplayMemberPath` to `"BookName"` would show every object's `BookName` within the `ItemsControl`.

The `ItemContainerStyle` property may be used to place a `Style` on the particular container item for this `ItemsControl`. For example, a `ListView` setting its `ItemContainerStyle` property affects the `ListViewItem` controls, each holding the data object in question (based on the usual rules for content).

We'll see some more properties of `ItemsControl` in the next chapter. The following sections discuss briefly some of the common types derived from `ItemsControl`. Technically, there's just one such class: `Selector`, adding the notion of selection with the `SelectedItem` (the actual data object) and `SelectedIndex` (an integer index) properties. The `SelectedValue` property indicates the "value" of the selected item, based on the `SelectedValuePath` property. For example, if the control holds `Book` objects, as shown previously, and `SelectedValuePath` is `"BookName"`, then `SelectedValue` will hold the actual book name for the `SelectedItem` (which holds the entire `Book` object).

`Selector` also defines a single event, `SelectionChanged`, fired when the selected item changes.

ListBox and ComboBox

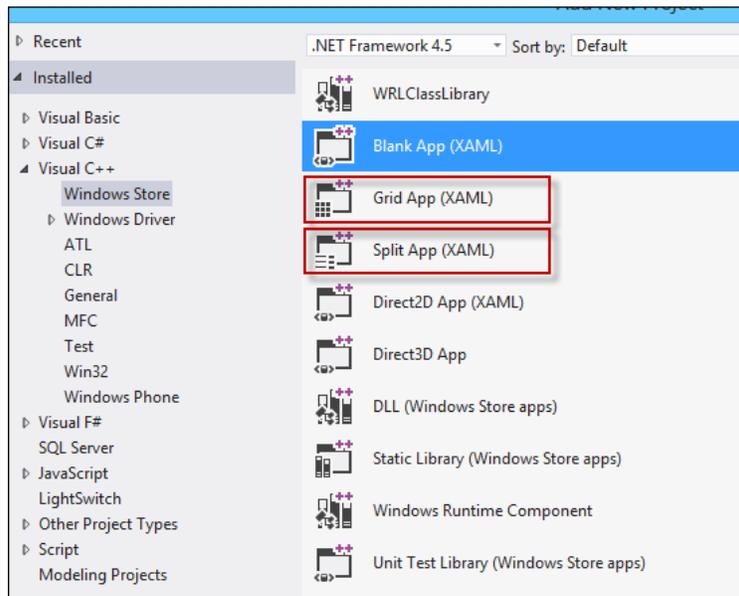
`Listbox` and `ComboBox` are the WinRT versions of the classic Windows controls. `Listbox` shows a collection of objects (vertically, by default), with scroll bars if necessary. `Listbox` also adds the notion of multiple selected items with the `SelectedItems` property and a `SelectionMode` property (`Single`, `Multiple`—each click/touch selects/deselects items, and `Extended`—pressing *Shift* selects multiple consecutive objects and *Ctrl* for nonadjacent group selection).

`ComboBox` shows just one item, selected from a drop-down list. Usage of both these controls is discouraged in Store apps, because their touch behavior is not as good as it should be, and they have no interesting visual transitions, making them a bit dull; that said, they still may be useful at times, particularly the `ComboBox`, which has no similar alternative.

Listview and GridView

Listview and GridView both derive from ListViewBase (that derives from Selector), and they're the preferred controls for hosting multiple items. Listview and GridView add nothing to ListViewBase – they just have different defaults for their ItemsPanel property and some other tweaks.

Both have been designed with a lot of thought for touch input, transition animations and the like; these are the workhorses for showing collections of objects. In fact, Visual Studio has some project templates that build sample Listview and GridView controls to help developers get started:



FlipView

The FlipView control adds nothing to Selector, but has a somewhat unique appearance, showing just one (selected) item at a time (similar to a ComboBox), but allows "flipping" through the items by swiping left or right, or by clicking two arrows on its sides. The classic example being flipping through image objects:



Text-based elements

Text is an all important part of any user interface. Naturally, WinRT provides several elements and controls that have text as their main visual appearance. With text, font-related properties are typically involved. These include:

- `FontSize` – the size of the text (a double value).
- `FontFamily` – the font family name (such as "Arial" or "Verdana"). This can include fallback font families (separated by commas) in case that particular font is unavailable.
- `FontStretch` – indicates stretch characteristics for the font, such as `Condensed`, `Normal` (the default), `ExtraCondensed`, `Expanded`, and so on.
- `FontWeight` – indicates the font weight, such as `Bold`, `ExtraBold`, `Medium`, `Thin`, and so on (all taken from the static properties of the `FontWeights` class).
- `FontStyle` – one of `Normal`, `Oblique` or `Italic`.

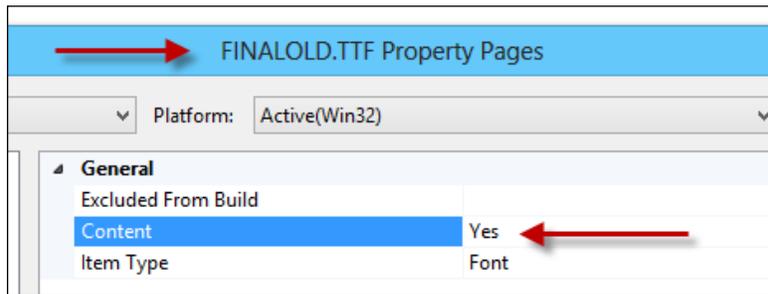
All font-related properties have one notable attribute, they set a "default" font for all elements that exist as children of the element in question (direct or indirect). This means that setting a font-related property on the `Page` object effectively sets the default font for all elements in the page (barring two exceptions: font properties set by a control template explicitly and a local font property set by a particular element; both override the default font settings).

Another property that is common to most text elements is `Foreground`. This sets the `Brush` that draws the actual text. There are several `Brush` types, `SolidColorBrush` is the simplest, but there are others, such as `LinearGradientBrush` and `TileBrush`.

Other text-related properties common to most text-related elements include `TextAlignment` (`Left`, `Right`, `Center`, `Justify`), `TextTrimming` (`None` and `WordEllipsis`), and `TextWrapping` (`NoWrap` and `Wrap`), all pretty self-explanatory.

Using custom fonts

Custom fonts can be used with WinRT. This involves adding a font file to the project (with a .TTF extension), and making sure its **Content** property in Visual Studio is set to **Yes**:



All that's needed now is to use the `FontFamily` property with a special value, consisting of the font URI (filename and any logical folder it's in), a hash (#) and the font name itself, visible when double-clicking the font file in Windows. Here's an example of two lines using a standard font and a custom font:

```
<StackPanel>
  <TextBlock Text="This text is in a built in font"
    FontFamily="Arial" FontSize="30" Margin="20"/>
  <TextBlock Text="This text is in old Star Trek style"
    FontFamily="Finalold.ttf#Final Frontier Old Style"
    FontSize="30" Margin="20" />
</StackPanel>
```

The result is shown as follows:



The following sections discuss some of the common text-related elements and controls.

TextBlock

`TextBlock` is probably the most useful text-related element. It displays text that cannot be changed by the user interactively (only programmatic changes are possible). This is useful for showing static text, which should not be edited by the user.



Although text cannot be edited in a `TextBlock`, it can still be selected by the user (and even copied by pressing `Ctrl + C`) if `IsTextSelectionEnabled` is true. In case it is, other properties can be used, namely `SelectedText`, `SelectionStart`, and `SelectionEnd` (the later returns `TextPointer` objects).

The most straightforward way of working with a `TextBlock` is by setting the `Text` property (a `String`) and the font-related properties if needed. As an alternative to `Text`, `TextBlock` supports a collection of objects called inlines (through the `Inlines` property, which is also its `ContentProperty` for XAML purposes), that allow the building a more complex `TextBlock`, but still uses just one element (the `TextBlock`).

Inlines include (all deriving from `Inline`) `Span`, `Run`, `LineBreak`, and `InlineUIContainer` (all in the `Windows::UI::Xaml::Documents` namespace). `Span` is a container for more inlines with the same properties set by `Span`. `Run` has a `Text` property and adds a `FlowDirection`. `LineBreak` is exactly that. `InlineUIContainer` can't be used in a `TextBlock`, but only in a `RichTextBlock` (discussed later).

Here's an example `TextBlock`:

```
<TextBlock>
  <Run FontSize="30" Foreground="Red" Text="This is a run" />
  <LineBreak />
  <Span Foreground="Yellow" FontSize="20">
    <Run Text="This text is in yellow" />
    <LineBreak />
    <Run Text="And so is this" />
  </Span>
</TextBlock>
```

The result is shown as follows:



If the `Text` property is used along with inlines, `Text` wins and the inlines are not displayed.

TextBox

TextBox is the classic text-entering control and provides all the expected capabilities of such a control. Common properties include (in addition to the font properties and others discussed at the beginning of this section):

- `Text` – the actual text shown or edited by the user.
- `MaxLength` – the maximum character length allowed for user input (this setting is not used when programmatically manipulating the `Text` in the `TextBox`).
- `SelectedText`, `SelectedLength`, `SelectionStart`, `SelectionEnd` – selection related properties (self-explanatory).
- `IsReadOnly` – indicates whether text can actually be edited (default is `false`).
- `AcceptsReturn` – if `true`, indicates a multiple line `TextBox` (default is `false`).
- `InputScope` – indicates what kind of virtual keyboard should pop up on a touch-based device that is not using a physical keyboard. This can help with text entering. Values (from the `InputScopeNameValue` enumeration) include: `Url`, `Number`, `EmailSmtAddress` (e-mail address), and others. Here is a keyboard screenshot for `InputScope` of `Number`:



This is the keyboard for `InputScope` of `Url`:



This one is for `InputScope` of `EmailSmtAddress`:



`TextBox` defines several events, `TextChanged` being the most useful.

PasswordBox

`PasswordBox` is used for entering passwords (no surprise here). The text is shown with a single repeating character, that can be changed with the `PasswordChar` property (default is '*' appearing as a circle). The `Password` property is the actual password, typically read in code.

A nice feature of `PasswordBox` is a "reveal" button that can show the actual password when the button is pressed, useful to make sure entered password is what was intended; this feature can be turned off by setting `IsPasswordRevealButtonEnabled` to `false`.

RichTextBlock and RichEditBox

The "rich" versions of `TextBlock` and `TextBox` provide richer formatting capabilities with respect to their "poor" counterpart. For example, it's possible to set font-related properties to any text within the control.

For `RichTextBlock`, the actual content of the controls is in a collection of block objects (`Blocks` property), with just one derived type - `Paragraph`. `Paragraph` has its own formatting properties and can host `Inline` objects (similar to a `TextBlock`); the `InlineUIContainer` inline is supported by `RichTextBlock`, providing the ability to embed elements (such as images, or anything else for that matter) as part of the text.

`RichEditBox` allows for richer editing capabilities that can embed *rich* content, such as hyperlinks. The `Document` property (of type `ITextDocument`) provides the gateway to the object model behind the `RichEditBox`. This object model supports saving and loading the document in text and rich text (RTF) formats, multiple undo/redo capability, and other features.

Images

Images can be displayed with the `Image` element. The `Source` property indicates what should be displayed. The simplest possibility is an image added to the project as content:

```
<Image Source="penguins.jpg" />
```

The `Source` property is of type `ImageSource`; this markup works only because a type converter exists to turn the relative URI into a type derived from `ImageSource`.

The simplest derived type is `BitmapImage` (actually derives from `BitmapSource` that derives from `ImageSource`). `BitmapImage` can be initialized from a URI (with the `UriSource` property), which is exactly what happens with the type converter used in the preceding XAML.

A more interesting type is `WriteableBitmap` (derives from `BitmapSource` as well), that exposes the ability to change the bitmap bits dynamically.

To create a `WriteableBitmap`, we need to specify its dimensions in pixels, as the following code demonstrates:

```
_bitmap = ref new WriteableBitmap(600, 600);
```

`_bitmap` is a `WriteableBitmap` reference. Next, we set it as the `Source` property of an `Image` element:

```
_image->Source = _bitmap;
```

To access the actual bits, we'll need to use a native interface with WRL. First, two `includes` and a `using` statement:

```
#include <robuffer.h>
#include <wrl.h>

using namespace Microsoft::WRL;
```

`robuffer.h` defines the `IBufferByteAccess` interface, used with the `WriteableBitmap::PixelBuffer` property like so:

```
ComPtr<IUnknown> buffer((IUnknown*)_bitmap->PixelBuffer);
ComPtr<IBufferByteAccess> byteBuffer;
buffer.As(&byteBuffer);
byte* bits;
byteBuffer->Buffer(&bits);
```

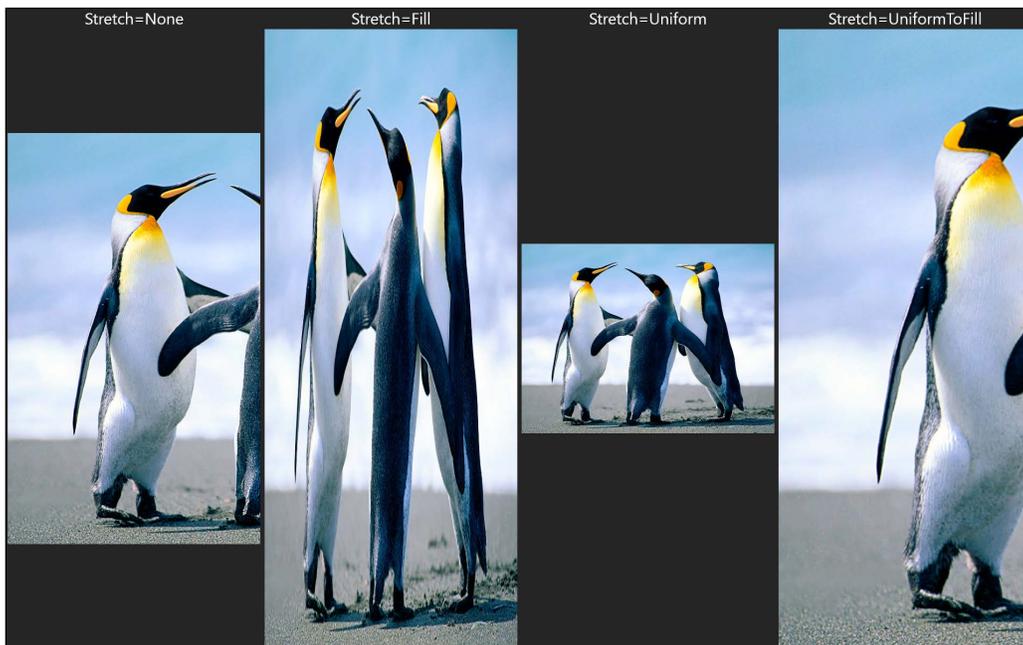
Finally, the bits can be used. Here's a simple example that draws the first line in the bitmap with a random color:

```
RGBQUAD* bits2 = (RGBQUAD*)bits;
RGBQUAD color = {
    ::rand() & 0xff, ::rand() & 0xff, ::rand() & 0xff
};
for(int x = 0; x < 600; x++)
    bits2[x] = color;
_bitmap->Invalidate();
```

The call to `WritableBitmap::Invalidate` is necessary, ensuring the bitmap is redrawn, and so connected `Image` elements are updated.

The Stretch property

The `Image::Stretch` property sets the way the `ImageSource` is stretched given the size of the `Image` element. Here's how the `Stretch` property affects displayed images:



With `Stretch=None`, the image is displayed in its original size. In the shown image, the penguins are clipped because the image is too big to fit. `Uniform` and `UniformToFill` preserve the aspect ratio (the original image width divided by the height), while `Fill` simply stretches the image to fill the available space for the Image. `UniformToFill` may cut out content if the available space has a different aspect ratio than the original.



Do not confuse `Image` with `ImageSource`. `Image` is an element, so can be placed somewhere in the visual tree. An `ImageSource` is the actual data, an `Image` element simply shows the image data in some way. The original image data (`ImageSource`) is unchanged.

The SemanticZoom control

The `SemanticZoom` control deserves a section for itself, as it's pretty unique. It combines two views in one control, one as a "zoomed out" view and another as a "zoomed in" view. The idea behind `SemanticZoom` is two related views – one more general (zoomed out) and the other more specific (zoomed in). The classic example is the Start screen. Doing the pinch/zoom touch gesture (or holding `Ctrl` and scrolling the mouse wheel) changes between the two views. The following is the zoomed in view:



This is the zoomed out view:



The `ZoomedInView` and `ZoomedOutView` properties hold the views—typically `ListView` or `GridView`, but technically anything that implements the `ISemanticZoomInformation` interface.

`SemanticZoom` is an effective way to handle mater/detail scenarios in an easily accessible and intuitive way.

Summary

Building an effective and addictive user interface is an art in itself, and is beyond the scope of this book. The modern design guidelines, as they relate to Windows Store apps are relatively new, but a lot of information can be found on the web, on Microsoft websites and others.

This chapter's goal was to introduce the C++ developer to the UI landscape, making it a more comfortable zone. Even if eventually the C++ developer will be more concerned with the application logic, infrastructure, and other low-level activities, it's still useful to understand the landscape of user experience and user interface.

In the next chapter, we'll tie user interface and data via data binding to create robust and scalable applications, at least where UI and data are concerned.

5

Data Binding

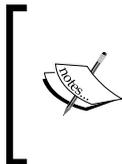
In the preceding two chapters, we looked at XAML and the way user interface elements are constructed and laid out with the help of the layout panels. The user interface, however, is just the first step. Some data must be set upon the UI to make up an application.

There are several ways of getting data to the controls. The simplest, perhaps, and most direct is the one we've been using so far; getting a reference to a control and changing the relevant property. If we needed some text to be placed in a `TextBox`, we would simply change its `Text` property when needed.

This certainly works and when working with the Win32 API for UI purposes, there is really no other way. But this is tedious at best, and involves an unmanageable maintenance headache at worst. Data is not the only thing that needs to be taken care of. Element states, such as enabled/disabled and checked/unchecked need to be examined and perhaps dynamically changed. In WinRT, much of this chore is handled with data binding.

Understanding data binding

Data binding is essentially simple – one property changes in some object (source) and another property in another object (target) reflects the change in some meaningful way. Coupled with data templates, data binding provides a compelling and powerful way to visualize and interact with data.



Those familiar with WPF or Silverlight will find WinRT data binding very familiar. There are some changes, mostly omissions, from WinRT, making data binding a bit less powerful than in WPF/Silverlight. Still, it's much better than manually transferring and synchronizing data.

Data binding in WinRT leads to one of the well-known patterns for working with data and UI in a seamless manner, known as **Model-View-ViewModel (MVVM)**, which we'll briefly discuss at the end of this chapter.

Data binding concepts

We'll start by examining some of the basic terms associated with data binding, with the WinRT specifics added:

- **Source:** The object whose property is monitored for changes.
- **Source path:** The property on the source object to monitor.
- **Target:** The object whose property is changed when the source changes. In WinRT, the target property must be a dependency property (as we'll see later on).
- **Binding mode:** Indicates the direction of the binding.

Possible values are (all from the `Windows::UI::Xaml::Data::BindingMode` enumeration) as follows:

- `OneWay`: Source changes update the target
- `TwoWay`: Source and target update each other
- `OneTime`: Source updates the target just once

Data binding is specified (most of the time) in XAML, providing a declarative and convenient way to connect to data. This has the direct effect of minimizing code writing for managing the element state and exchanging data between controls and data objects.

Element-to-element binding

The first binding scenario we'll examine is the way in which we can connect elements together without writing any code—by performing data binding between required properties. Consider the following two elements:

```
<TextBlock Text="This is a sizing text"
  TextAlignment="Center" VerticalAlignment="Center"/>
<Slider x:Name="_slider" Grid.Row="1" Minimum="10" Maximum="100"
  Value="30"/>
```

Suppose we wanted the `FontSize` of the `TextBlock` to be changed, based on the current `Value` of the `Slider`. How would we go about doing that?

The obvious approach would be to use events. We can react to the `ValueChanged` event of the `Slider` and modify the `FontSize` property value of the `TextBlock` to be equal to the `Value` of the `Slider`.

This certainly works, but it has a few drawbacks:

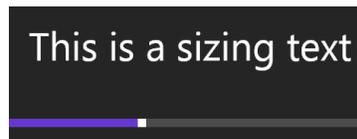
- C++ code needs to be written for this to work. This is a shame, because no real data is used – this is just UI behavior. Perhaps a designer could take care of that if he/she could just use XAML and not code.
- Such logic may change in the future, creating a maintenance headache – remember that a typical user interface will contain many such interactions – the C++ developer does not really want to be concerned about every such little detail.

Data binding provides an elegant solution. Here's the `FontSize` setting of `TextBlock` that's needed to make this idea work, without requiring any C++ code:

```
FontSize="{Binding Path=Value, ElementName=_slider}"
```

The data binding expression must be specified on the target property, using the `{Binding}` markup extension. The `Path` property indicates the source property to look for (`Slider::Value` in this case), and `ElementName` is the property to use if the source object is an element on the current page (in this case, the `Slider` is named `_slider`).

Running this shows the following:



Dragging the slider changes the text size automatically; this is the power of data binding.



The `Path` property of `Binding` can be omitted if its value is the first argument. This means the previous binding expression is equivalent to the following:

```
FontSize="{Binding Value, ElementName=_slider}"
```

This is more convenient and will be used most of the time.

The same expression can be achieved with a code, such as the following:

```
auto binding = ref new Binding;  
binding->Path = ref new PropertyPath("Value");  
binding->ElementName = "_slider";  
BindingOperations::SetBinding(_tb, TextBlock::FontSizeProperty,  
binding);
```

The code assumes `_tb` is the name of the `TextBlock` in question. This is certainly more verbose, and is actually used only in specific scenarios (which we'll examine in *Chapter 6, Components, Templates, and Custom Elements*).

Let's add another element, a `TextBox`, whose `Text` should reflect the current font size of the `TextBlock`. We'll use data binding as well:

```
<TextBox Grid.Row="2" Text="{Binding Value, ElementName=_slider}"  
FontSize="20" TextAlignment="Center"/>
```

This works. But if we change the actual text of the `TextBox` to a different number, the font size does not change. Why?

The reason is that the binding is by default one way. To specify a two-way binding, we need to change the `Mode` property of the binding:

```
Text="{Binding Value, ElementName=_slider, Mode=TwoWay}"
```

Now, changing the `TextBox` and moving the focus to another control (such as by pressing *Tab* on the keyboard or touching some other element), changes the `FontSize` value of the `TextBlock`.

Object-to-element binding

Although an element-to-element binding is sometimes useful, the classic data binding scenario involves a source, which is a regular, non-UI object, and a target, which is a UI element. The binding expression itself is similar to the element-to-element binding case; but naturally the `ElementName` property cannot be used.

The first step is to create an object that can support data binding. This must be a WinRT class that's decorated with the `Bindable` attribute. The bindings themselves are on properties (as always). Here's a simple `Person` class declaration:

```
[Windows::UI::Xaml::Data::BindableAttribute]  
public ref class Person sealed {  
public:  
property Platform::String^ FirstName;
```

```

    property Platform::String^ LastName;
    property int BirthYear;
};

```

The preceding code uses auto-implemented properties, which will suffice for now.

We can create such an object in XAML as a resource, and then use the `Binding::Source` property to hook up the binding itself. First, two `Person` objects are created as resources:

```

<Page.Resources>
  <local:Person FirstName="Albert" LastName="Einstein"
    BirthYear="1879" x:Key="p1" />
  <local:Person FirstName="Issac" LastName="Newton"
    BirthYear="1642" x:Key="p2" />
</Page.Resources>

```

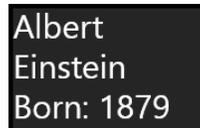
Next, we can bind these objects to elements as follows (all inside a `StackPanel`):

```

<TextBlock Text="{Binding FirstName, Source={StaticResource p1}}"
  FontSize="30" />
<TextBlock Text="{Binding LastName, Source={StaticResource p1}}"
  FontSize="30" />
<TextBlock FontSize="30" >
  <Span>Born: </Span>
  <Run Text="{Binding BirthYear, Source={StaticResource p1}}" />
</TextBlock>

```

The `Source` property refers to the object being bound; a `Person` instance in this case. Here's the resulting UI:



Albert
Einstein
Born: 1879

Note that `Source` is specified in each and every binding expression. Without it, the binding would simply fail, as there is no source object to bind to.

Since the source is identical for all three elements, it would be beneficial to be able to specify the source just once, and allow all relevant elements to bind to it automatically without needing to specify the source explicitly. Fortunately, this is possible using the `FrameworkElement.DataContext` property. The rule is simple, if a source is not specified explicitly in a binding expression, a `DataContext` that is not a `nullptr` is searched in the visual tree, from the target element until one is found or the root of the visual tree is reached (typically a `Page` or a `UserControl`). If a `DataContext` is found, it becomes the source of the binding. Here's an example that sets the `DataContext` on a parent `StackPanel` to be used by its children (whether immediate or not):

```
<StackPanel Margin="4" DataContext="{StaticResource p2}">
  <TextBlock Text="{Binding FirstName}" />
  <TextBlock Text="{Binding LastName}" />
  <TextBlock>
    <Span>Born: </Span>
    <Run Text="{Binding BirthYear}" />
  </TextBlock>
</StackPanel>
```

This is the result (after some font size tweaks):



Issac
Newton
Born: 1642

The binding expressions work correctly, because the implicit source is the `Person` object whose key is `p2`. Without the `DataContext`, all these bindings would silently fail.

 Notice how the data binding expressions are simplified with `DataContext`. They're saying something such as, "I don't care what the source is, as long as there's a property named `<Fill in property name>` on some `DataContext` that's in scope."

The idea of a `DataContext` is a powerful one and, in fact, using the `Source` property is rare.

Of course, setting a `Source` or `DataContext` in XAML to a predefined resource is rare as well. Setting the `DataContext` is usually done via code by getting relevant data from a data source, such as a local database or a web service. But it works regardless of where or how the `DataContext` is set.

Binding failures

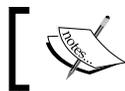
Bindings are loosely typed – properties are specified as strings, and can be misspelled. For example, writing `FirstNam` instead of `FirstName` in the previous examples does not cause any exception to be thrown; the binding silently fails. The only other indication that something went wrong can be found in the **Visual Studio Output** window (**View** | **Output** from the menu) if the program is run under a debugger:

```
Error: BindingExpression path error: 'FirstNam' property not found
on 'ElementObjectBinding.Person'. BindingExpression: Path='FirstNam'
DataItem='ElementObjectBinding.Person'; target element is 'Windows.
UI.Xaml.Controls.TextBlock' (Name='null'); target property is 'Text'
(type 'String')
```

This text pinpoints the exact problem, specifying the property name to bind, the source object type, and the details about the target. This should help to fix the spelling mistake.

Why is no exception thrown? The reason is that a data binding may fail at some point in time, and that's ok, because the conditions for this binding have not been met yet; there may be some information that is retrieved from a database, or from a web service, for instance. When the data is finally available, those bindings suddenly begin to work correctly.

This means true debugging of data binding expressions is not possible. A nice feature would be the ability to place a breakpoint in the XAML binding expression. This is currently unsupported, although it is graphically possible to place a breakpoint on a binding, it would simply never fire. This feature is available in Silverlight 5; hopefully, it will be supported in a future release of WinRT.



One way to debug data bindings is by using value converters, discussed later in this chapter.

Change notifications

Data binding supports three modes for binding: one way, two way, and one time. Up until now the binding occurred when the page first loads and remains unchanged after that. What happens if we change a property value on a `Person` object after the bindings are in place?

After adding a simple button, the `Click` event handler does the following:

```
auto person = (Person^)this->Resources->Lookup("p1");
person->BirthYear++;
```

Since the `Person` instance was defined as a resource (uncommon, but possible), it's extracted from the `Resources` property of the page by using the specified key (`p1`). Then the `BirthYear` property is incremented.

Running the application shows no visual change. Setting a breakpoint in the `Click` handler confirms it's actually called, and the `BirthYear` is changed, but the binding seems to have no effect.

The reason for this is the way the `BirthYear` property is currently implemented:

```
property int BirthYear;
```

This is a trivial implementation that uses a private field behind the scenes. The problem is that when the property changes, nobody knows about it; specifically, the binding system has no idea anything has happened.

To change that, a data object should implement the `Windows::UI::Xaml::Data::INotifyPropertyChanged` interface. This interface is queried by the binding system and, if found, registers for the `PropertyChanged` event (the only member of that interface). Here's a revised declaration of the `Person` class, focusing on the `BirthYear` property:

```
[Bindable]
public ref class Person sealed : INotifyPropertyChanged {
public:
    property int BirthYear {
        int get() { return _birthYear; }
        void set(int year);
    }

    virtual event PropertyChangedEventHandler^ PropertyChanged;

private:
    int _birthYear;
    //...
};
```

The getter is implemented inline, and the setter is implemented in the CPP file, as follows:

```
void Person::BirthYear::set(int year) {
    _birthYear = year;
    PropertyChanged(this,
        ref new PropertyChangedEventArgs("BirthYear"));
}
```

The `PropertyChanged` event is raised, accepting a `PropertyChangedEventArgs` object that accepts the changed property name. Now, running the application and clicking on the button shows an incremented birth year, as expected.

This effectively means that every property should be implemented in a similar fashion; declaring a private field and raising the `PropertyChanged` event in the setter. Here's a revised `FirstName` property implementation (this time implemented inline):

```
property String^ FirstName {
    String^ get() { return _firstName; }
    void set(String^ name) {
        _firstName = name;
        PropertyChanged(this,
            ref new PropertyChangedEventArgs("FirstName"));
    }
}
```

`_firstName` is a private `String^` field defined within the class as well.

Binding to collections

The previous examples used a property that binds to a single object. As we've seen in the previous chapter, a bunch of controls deriving from `ItemsControl` can present information for more than one data item. It stands to reason that these controls should bind to a collection of data items, such as a collection of the `Person` objects.

The property to be used for binding purposes is `ItemsSource`. This should be set to a collection, typically `IVector<T>`. Here's an example of some `Person` objects bound to a `ListView` (a constructor was added to `person` for convenient initialization):

```
auto people = ref new Vector<Person^>;
people->Append(ref new Person(L"Bart", L"Simpson", 1990));
people->Append(ref new Person(L"Lisa", L"Simpson", 1987));
people->Append(ref new Person(L"Homer", L"Simpson", 1960));
people->Append(ref new Person(L"Marge", L"Simpson", 1965));
people->Append(ref new Person(L"Maggie", L"Simpson", 2000));
```

To set up the binding, we can use an explicit assignment to the `ListView::ItemsSource` property:

```
_theList->ItemsSource = people;
```

An (elegant and preferred) alternative is to bind `ItemsSource` to something related to the `DataContext`. For example, the `ListView` markup can start with this:

```
<ListView ItemsSource="{Binding}" >
```

This means that `ItemsSource` is bound to whatever the `DataContext` is (should be a collection in this case). The lack of a property path means the object itself. With this markup, the binding is done with the following simple code:

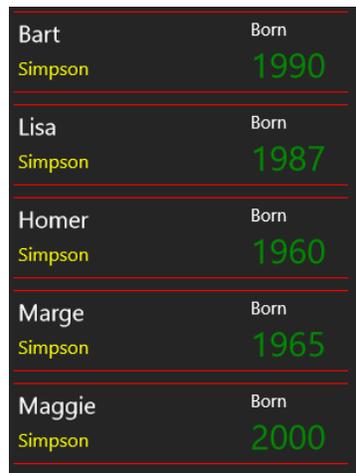
```
DataContext = people;
```

To view the actual `Person` objects, `ItemsControl` provides the `ItemTemplate` property, which is a `DataTemplate` object that defines how `Person` objects are displayed. By default (without a `DataTemplate`), the type name is shown or another string representation of the object (if it has one) is used. This is rarely useful. A simple alternative is using the `DisplayMemberPath` property to show a specific property on the data object (such as `FirstName` for `Person` objects). A much more powerful way would be to use a `DataTemplate`, providing a customizable user interface for each item connected to the actual object via data binding. Here's an example for our `ListView` of people:

```
<ListView ItemsSource="{Binding}">
  <ListView.ItemTemplate>
    <DataTemplate>
      <Border BorderThickness="0,1" Padding="4"
        BorderBrush="Red">
        <Grid>
          <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
          </Grid.RowDefinitions>
          <Grid.ColumnDefinitions>
            <ColumnDefinition Width="200"/>
            <ColumnDefinition Width="80" />
          </Grid.ColumnDefinitions>
          <TextBlock Text="{Binding FirstName}"
            FontSize="20" />
          <TextBlock FontSize="16" Foreground="Yellow"
            Grid.Row="1" Text="{Binding LastName}" />
          <TextBlock Grid.Column="1" Grid.RowSpan="2">
            <Span FontSize="15">Born</Span>
            <LineBreak />
            <Run FontSize="30" Foreground="Green"
              Text="{Binding BirthYear}" />
          </TextBlock>
        </Grid>
      </DataTemplate>
    </ListView.ItemTemplate>
  </ListView>
```

```
</Grid>
</Border>
</DataTemplate>
</ListView.ItemTemplate>
</ListView>
```

The binding expressions inside the `DataTemplate` reach out to the relevant properties on the data object itself. Here's the resulting `ListView`:



Bart Simpson	Born 1990
Lisa Simpson	Born 1987
Homer Simpson	Born 1960
Marge Simpson	Born 1965
Maggie Simpson	Born 2000

Customizing a data view

Data templates provide a powerful way to visualize and interact with data, partly because of the power of data binding. Sometimes, however, more customization is needed. For example, in a list of the `Book` objects, every book that's currently on sale should be displayed with a different color, or have some special animation, and so on.

The following sections describe some ways of customizing data templates.

Value converters

Value converters are types that implement the `Windows::UI::Xaml::Data::IValueConverter` interface. This interface provides a way to convert one value to another value, which can be of different types. Suppose we want to show a collection of books, but those that are on sale should have a slightly different look. With plain data templates, this is difficult, unless there are specific `Book` properties that have visual impact (such as a color or brush); this is unlikely, as data objects should be concerned about the data and not about how to display the data.

Here's the definition of the Book class (change notifications are not implemented to simplify the example):

```
[Windows::UI::Xaml::Data::BindableAttribute]
public ref class Book sealed {
public:
    property Platform::String^ BookName;
    property double Price;
    property Platform::String^ Author;
    property bool IsOnSale;

internal:
    Book(Platform::String^ bookName, Platform::String^ author,
        double price, bool onSale) {
        BookName = bookName;
        Author = author;
        Price = price;
        IsOnSale = onSale;
    }
};
```

Value converters provide an elegant solution that keeps the object (Book in this example) decoupled from the way it's presented. Here's a basic Book data template:

```
<ListView.ItemTemplate>
<DataTemplate>
<Border BorderThickness="1" BorderBrush="Blue" Margin="2"
    Padding="4">
<Grid>
<Grid.ColumnDefinitions>
<ColumnDefinition Width="400" />
<ColumnDefinition Width="50" />
</Grid.ColumnDefinitions>
<TextBlock VerticalAlignment="Center"
    FontSize="20">
<Run Text="{Binding BookName}" />
<Span> by </Span>
<Run Text="{Binding Author}" />
</TextBlock>
<TextBlock Grid.Column="1" FontSize="25">
<Span>${</Span>
<Run Text="{Binding Price}" />
</TextBlock>
</Grid>
</Border>
</DataTemplate>
</ListView.ItemTemplate>
```

This is how the books are shown:

Windows Internals by Mark Russinovich	\$ 50
Essential COM by Don Box	\$ 40
WPF 4.5 Cookbook by Pavel Yosifovich	\$ 60
Windows via C/C++ by Jeffrey Richter	\$ 45
Programming MFC by Jeff Prosise	\$ 30

Suppose we would like to use a green background for books that are on sale. What we don't want to do is add a `Background` property to the `Book` class. Instead, a value converter will be used to convert the `IsOnSale` property (a `Boolean`) to a `Brush` object, suitable for a property related to the `Background` property.

First, the declaration of our value converter is as follows:

```
public ref class OnSaleToBrushConverter sealed : IValueConverter {
public:
    virtual Object^ Convert(Object^ value, TypeName targetType,
        Object^ parameter, String^ language);
    virtual Object^ ConvertBack(Object^ value, TypeName
        targetType, Object^ parameter, String^ language);

    OnSaleToBrushConverter();

private:
    Brush^ _normalBrush;
    Brush^ _onSaleBrush;
};
```

There are two methods to implement:

- `Convert`: Used when binding from source to target (the usual way)
- `ConvertBack`: Relevant for two-way bindings only

In our case, the binding we use is the one-way binding only, so `ConvertBack` can simply return `nullptr` or throw an exception. Here's the implementation:

```
OnSaleToBrushConverter::OnSaleToBrushConverter() {
    _normalBrush = ref new SolidColorBrush(Colors::Transparent);
    _onSaleBrush = ref new SolidColorBrush(Colors::Green);
}

Object^ OnSaleToBrushConverter::Convert(Object^ value, TypeName
targetType, Object^ parameter, String^ culture) {
    return (bool)value ? _onSaleBrush : _normalBrush;
}

Object^ OnSaleToBrushConverter::ConvertBack(Object^ value, TypeName
targetType, Object^ parameter, String^ culture) {
    throw ref new NotImplementedException();
}
```

Two brushes are created in the constructor; one for regular books (transparent) and the other for books on sale (green). The `Convert` method is called with the `value` parameter being the `IsOnSale` property of the book in question. How this happens will become clear soon enough. The method simply looks at the Boolean value and returns the appropriate brush. This conversion is from a Boolean value to a `Brush`.

The next step would be to actually create an instance of the converter. This is typically done in XAML, making the converter a resource:

```
<Page.Resources>
    <local:OnSaleToBrushConverter x:Key="sale2brush" />
</Page.Resources>
```

Now, for the final connection, use an appropriate property to bind to `IsOnSale` and supply a converter for the operation. In our case, `Border` (part of the `DataTemplate`) will do very nicely:

```
<Border BorderThickness="1" BorderBrush="Blue" Margin="2"
    Padding="4" Background="{Binding IsOnSale,
    Converter={StaticResource sale2brush}}">
```

Without the converter, the binding would simply fail, as there's no way a Boolean can be converted to a `Brush` automatically. The converter has been passed the value of `IsOnSale` and should return something appropriate to the target property for the conversion to succeed.



It's possible to use a Binding expression without Path (without `IsOnSale` in this example). The result is that the entire object (Book) is passed as the value argument for the converter. This can help in making decisions that are based on more than one property.

Here's the result:

Windows Internals by Mark Russinovich	\$ 50
Essential COM by Don Box	\$ 40
WPF 4.5 Cookbook by Pavel Yosifovich	\$ 60
Windows via C/C++ by Jeffrey Richter	\$ 45
Programming MFC by Jeff Prosise	\$ 30

Let's add a small image next to a book that is on sale. How would we do that? We can certainly add an image, but it must be shown only when the book is on sale. We can use a (somewhat classic) converter, changing from Boolean to the `Visibility` enumeration and vice versa:

```
Object^ BooleanToVisibilityConverter::Convert(Object^ value, TypeName
targetType, Object^ parameter, String^ culture) {
    return (bool)value ? Visibility::Visible :
        Visibility::Collapsed;
}

Object^ BooleanToVisibilityConverter::ConvertBack(Object^ value,
TypeName targetType, Object^ parameter, String^ culture) {
    return (Visibility)value == Visibility::Visible;
}
```

With this in place, we can create an instance as a resource in the usual way:

```
<local:BooleanToVisibilityConverter x:Key="bool2vis" />
```

Then, we can add an image to the third column, visible only when needed:

```
<Image Source="Assets/sun.png" VerticalAlignment="Center"
HorizontalAlignment="Center" Height="24" Grid.Column="2"
Visibility="{Binding IsOnSale, Converter={StaticResource
bool2vis}}" />
```

This is the result:

Windows Internals by Mark Russinovich	\$ 50
Essential COM by Don Box	\$ 40 ☀
WPF 4.5 Cookbook by Pavel Yosifovich	\$ 60
Windows via C/C++ by Jeffrey Richter	\$ 45
Programming MFC by Jeff Prosise	\$ 30 ☀

Value converters are extremely powerful, because they can leverage code to achieve visual changes.

Other parameters for Convert and ConvertBack

`Convert` and `ConvertBack` accept more parameters, not just the value. Here's the complete list:

- `value`: The value argument (first) is the most important one for the `Convert/ConvertBack` method. There are three other arguments as well.
- `targetType`: This indicates what's the expected object type that should be returned. This can be used for checking if the converter is used correctly (in our example, `OnSaleToBrushConverter` would have a `Brush` type as `targetType` for the `Convert` method). The other possible use of this argument is in the case of a more complex value converter that may deal with multiple return types and may need to know about the current request.
- `parameter`: This is a free parameter that can be passed via the `Binding` expression with the `ConverterParameter` property. This can be useful for customizing a value converter on a binding expression basis.
- `culture`: This receives whatever the `ConverterLanguage` property of the `Binding` expression has. This can be used to return different values based on a language, which is really just another string that can be passed along to the converter.

Data template selectors

In more extreme cases, the changes needed from `DataTemplate` may be too dramatic for value converters to be useful. If very different templates are needed by different objects (in the same collection), data template selectors may be a better alternative.

A data template selector is a class that derives from `Windows::UI::Xaml::Controls::DataTemplateSelector` (not a control, despite the namespace) and overrides the `SelectTemplateCore` method defined as follows:

```
protected:
    virtual DataTemplate^ SelectTemplateCore(Object^ item,
        DependencyObject^ container);
```

The method needs to return a `DataTemplate` that corresponds to the `item` argument. In the preceding examples, each `item` is `Book`; the code would look at some `Book` properties and conclude which `DataTemplate` should be used. This could also be based in some way on the `container` argument, which is the actual control hosting these objects (`ListView` in our example).

Next, an instance of this class is created in XAML (similar to a value converter), and the instance is set to the `ItemsControl::ItemTemplateSelector` property. If this is set, the `ItemTemplate` cannot be set at the same time, as it would conflict with the logic the template selector uses.

Commands

The traditional way of connecting a piece of user interface to some logic is through events. The canonical example is a button – when clicked, some action is undertaken, hopefully accomplishing some goal the user has intended. Although WinRT supports this model completely (as other UI frameworks do), it has its drawbacks:

- The event handler is part of the "code behind" where the UI is declared, typically a `Page` or a `UserControl`. This makes it difficult to call from other objects that may want to invoke the same piece of logic.
- The aforementioned button may disappear and be replaced by a different control. This would require the event hooking code to potentially change. What if we wanted more than one control to invoke the same functionality?
- An action may not be allowed at some state – the button (or whatever) needs to be disabled or enabled at the right time. This adds management overhead to the developer – the need to track the state and change it for all UI elements that invoke the same functionality.
- An event handler is just a method – there's no easy way to pick it up and save it somewhere, such as for undo/redo purposes.
- It's difficult to test application logic without using an actual user interface, because logic and UI are intertwined.

These and other more subtle issues make working with event handlers less than ideal, especially when application logic is involved. If some event is just intended for usability enhancement or for otherwise serving the UI alone, this is not usually a concern.

The typical solution to this UI logic coupling is the concept of commands. This follows the famous "command design pattern" that abstracts away the application logic into distinct objects. Being an object, a command can be invoked from multiple locations, saved in lists (for example, for undo purposes), and so on. It can even indicate whether it's allowed in certain times or not, freeing other entities from taking care of the actual enabling or disabling of controls that may be bound to that command.

WinRT defines a basic command support with the `Windows::UI::Xaml::Input::ICommand` interface. `ICommand` has two methods and an event:

- **The `Execute` method:** This executes the command in question. It accepts a parameter that can be anything that can be used as an argument to the command.
- **The `CanExecute` method:** This method indicates whether this command is available at this time or not. WinRT uses this as a hint for enabling or disabling the command source.
- **The `CanExecuteChanged` event:** This is raised by the command to let WinRT know that it should call `CanExecute` again, because the availability of the command may have changed.

Various controls have a `Command` property (of the type `ICommand`) that can be set up (typically using data binding) to point to an object implemented by `ICommand` (and a `CommandParameter` that allows passing some information to the command). The canonical example being the classic `Button`. When the button is clicked, the hooked command's `Execute` method is called. This means no `Click` handler is needed to set this up.

WinRT does not provide any implementation for `ICommand`. It's up to the developer to create appropriate implementations. Here's a simple implementation of a command that's used for incrementing the birth year of a person:

```
public ref class IncreaseAgeCommand sealed : ICommand {
public:
    virtual void Execute(Platform::Object^ parameter);
    virtual bool CanExecute(Platform::Object^ parameter);
    virtual event EventHandler<Object^>^ CanExecuteChanged;
};
```

And the implementation is as follows:

```
void IncreaseAgeCommand::Execute(Object^ parameter) {
    auto person = (Person^)parameter;
    person->BirthYear++;
}

bool IncreaseAgeCommand::CanExecute(Object^ parameter) {
    return true;
}
```

To make this work we can create a command source, such as a button and fill in the command details as follows:

```
<Button Content="Inrease Birth Year With Command"
    CommandParameter="{StaticResource p1}">
    <Button.Command>
        <local:IncreaseAgeCommand />
    </Button.Command>
</Button>
```

Creating a command within a `Command` property is unusual, the typical way of being bound to an appropriate property on a `ViewModel`, as we'll see in the next section.

Introduction to MVVM

Commands are just one aspect of more general patterns for dealing with user interface in non-trivial applications. To that end, a number of UI design patterns are available, such as **Model View Controller (MVC)**, **Model View Presenter (MVP)**, and **Model-View-ViewModel (MVVM)**. All have something in common: the separation of the actual UI (view) from the application logic (controller, presenter, and view model) and the underlying data (model).

The MVVM pattern popularized by WPF and Silverlight leverages the power of data binding and commands to create decoupling between the UI (View) and the data (Model) by using an intermediary (View Model).

MVVM constituents

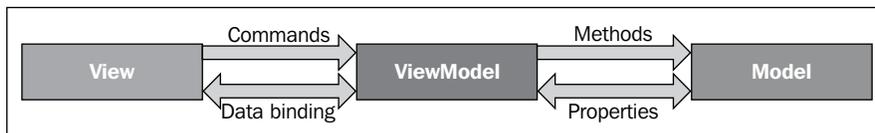
MVVM has three participants. The model represents the data or business logic. This may consist of types that may be written in standard C++, without any regard to WinRT. It's typically neutral; that is, it knows nothing of how it's going to be used.

The view is the actual UI. It should display relevant parts of the model and provide the required interactive functionality. The view should not have a direct knowledge of the model, and this is where data binding comes in. All bindings access a property without explicitly knowing what type of object sits at the other end. This magic is satisfied at runtime by setting the view's `DataContext` to the object providing the data; this is the `ViewModel`.

The `ViewModel` is the glue that hands out the required data to the view (based on the model). The `ViewModel` is just that – a model for the view. It has several responsibilities:

- Expose properties that allow binding in the view. This may be just by accessing properties on the model (if it's written with WinRT), but may be more involved if the model exposes data in another way (such as with methods) or types that need translation, such as `std::vector<T>` that needs to be returned as `IVector<T>`.
- Expose commands (`ICommand`) to be invoked by elements in the view.
- Maintain a relevant state for the view.

The entire relationship between model, view, and view model can be summarized with the following diagram:



Building an MVVM framework

It should be clear at this point that MVVM-based applications have a lot of elements in common, such as change notifications and commands. It would be beneficiary to create a reusable framework that we can simply leverage in many applications. Although there are several good frameworks out there (most are free), they are based around .NET, meaning they cannot be used in a C++ app because they are not exposed as a WinRT component, and even if they did, a C++ app must pay the price of the .NET CLR. Building such a framework ourselves is not too difficult and will enhance our understanding.

The first thing we'll tackle is the desired ability of objects to implement the `INotifyPropertyChanged` interface so that they can raise the `PropertyChanged` event when any property is changed. We can achieve this with the following WinRT class:

```
public ref class ObservableObject :
    DependencyObject, INotifyPropertyChanged {
public:
    virtual event PropertyChangedEventHandler^ PropertyChanged;
protected:
    virtual void OnPropertyChanged(Platform::String^ name);
};
```

And the implementation is as follows:

```
void ObservableObject::OnPropertyChanged(String^ name) {
    PropertyChanged(this, ref new PropertyChangedEventArgs(name));
}
```

The inheritance from `DependencyObject` may seem redundant, but it's actually necessary to circumvent one deficiency in current WinRT support – any regular class has to be sealed, making it useless as a base class. Any class inheriting from `DependencyObject` can remain unsealed – exactly what we want.

The `ObservableObject` class seems very simple and perhaps not worth as a separate class. But we can add to it common functionalities that any derived classes can benefit from. For example, we can support the `ICustomPropertyProvider` interface – this interface allows the object to support dynamic properties that are not statically part of the type (the interested reader can find more information in the MSDN documentation).

Concrete types may use `ObservableObject` with code similar to the following:

```
public ref class Book : ObservableObject {
public:
    property Platform::String^ BookName {
        Platform::String^ get() { return _bookName; }
        void set(Platform::String^ name) {
            _bookName = name;
            OnPropertyChanged("BookName");
        }
    }

    property bool IsOnLoan {
        bool get() { return _isOnLoan; }
        void set(bool isLoaned) {
```

```
        _isOnLoan = isLoaned;
        OnPropertyChanged("IsOnLoan");
    }
}

private:
    Platform::String^ _bookName;
    bool _isOnLoan;
    //...
};
```

The next thing to take care of is commands. As we've seen, we can create a command by implementing `ICommand`, and this is sometimes necessary. An alternative would be to create a more generic class that uses delegates to call whatever code we want in response to the `Execute` and `CanExecute` methods of a command. Here's an example of such a command:

```
public delegate void ExecuteCommandDelegate(Platform::Object^
    parameter);
public delegate bool CanExecuteCommandDelegate(Platform::Object^
    parameter);

public ref class DelegateCommand sealed : ICommand {
public:
    DelegateCommand(ExecuteCommandDelegate^ executeHandler,
        CanExecuteCommandDelegate^ canExecuteHandler)
        : _executeHandler(executeHandler),
          _canExecuteHandler(canExecuteHandler) { }

    virtual bool CanExecute(Platform::Object^ parameter) {
        if (_canExecuteHandler != nullptr)
            return _canExecuteHandler(parameter);

        return true;
    }

    virtual void Execute(Platform::Object^ parameter) {
        if (_executeHandler != nullptr && CanExecute(parameter))
            _executeHandler(parameter);
    }

    virtual event EventHandler<Platform::Object^>^
        CanExecuteChanged;
};
```

```

private:
    ExecuteCommandDelegate^ _executeHandler;
    CanExecuteCommandDelegate^ _canExecuteHandler;
};

```

The class takes advantage of delegates, accepting two in the constructor; the first for executing a command, and the second to indicate whether the command is enabled.

Here's a ViewModel that exposes a command to make a book loaned:

```

public ref class LibraryViewModel sealed : ObservableObject {
public:
    property IVector<Book^>^ Books {
        IVector<Book^>^ get() { return _books; }
    }

    property ICommand^ LoanBookCommand {
        ICommand^ get() { return _loanBookCommand; }
    }

internal:
    LibraryViewModel();

private:
    Platform::Collections::Vector<Book^>^ _books;
    ICommand^ _loanBookCommand;
};

```

The command is created in the constructor of a ViewModel:

```

LibraryViewModel::LibraryViewModel() {
    _loanBookCommand = ref new DelegateCommand
    (ref new ExecuteCommandDelegate([](Object^ parameter) {
        // execute the command
        auto book = (Book^)parameter;
        book->IsOnLoan = true;
    }), nullptr); // command is always enabled
}

```

The ViewModel is control (view) free, meaning we can construct it without any visible user interface. It exposes properties for data binding to a relevant view and commands to execute actions from the view. The actual action would typically modify some state in the appropriate model.



There is typically a one-to-one mapping between a view and a ViewModel. Although sometimes it's possible to share, this is not recommended.

More on MVVM

This was a quick tour of MVVM. Since this is a well-known pattern (thanks to its use in WPF and Silverlight), there's a lot of material on the web. Some things that can be added include a navigation-aware ViewModel (so that a `Frame` control is not accessed directly), a ViewModel locator service that allows easier binding between a view, and its corresponding ViewModel, and more.



A good start for more information on MVVM is available on Wikipedia at http://en.wikipedia.org/wiki/Model_View_ViewModel.

An implementation of a WinRT MVVM framework in C++ is somewhat awkward, as it's not (currently) possible to expose such a framework as a Windows Runtime Component, but only as a C++ static or dynamic library.

Regardless, the separation between data and view is an important one and will benefit all but the simplest of applications.

Summary

In this chapter, we saw what data binding is and how to use it. Data binding is an extremely powerful concept and its implementation in WinRT is pretty strong. Developers coming from a Win32 or MFC background should realize that a different approach is required for connecting display to data. Data binding provides a declarative model that supports separation between data and display, so the application logic only works with the data and will not really care which controls (if any) bind to that data.

MVVM concepts make this separation clearer and establish a foundation for incrementally enhancing an application without increasing maintenance headaches and logical complexity.

In the next chapter, we'll take a look at building reusable WinRT components, as well as custom elements.

6

Components, Templates, and Custom Elements

In the previous chapters, we looked at the fundamentals of building a user interface by looking at the way layout and elements work hand in hand to create flexible UIs. Data binding provides a detached way of writing and reading data to and from controls.

In this chapter, we'll explore ways to customize controls in a fundamental and powerful way using control templates. This is useful when a control's functionality is what's needed, but its looks are not. In other cases, the built-in controls may not have the required behavior; in these cases, custom and user controls can be created for application-specific needs. But first we should consider the more general concept of components built using C++, and how these can be used in C++ and non-C++ projects.

Windows Runtime Components

As we've seen in *Chapter 2, COM and C++ for Windows 8 Store Apps*, the Windows Runtime is based on COM classes implementing COM interfaces. Any such class that is also written to metadata (a `winmd` file) can be exported from a DLL and thus used by any other WinRT-compliant language or environment; the ones currently supported are C++, .NET languages (C# and VB), and JavaScript.

Such components must only use WinRT types in their public interface. For C++, this means that STL-based classes can only be used in the nonpublic area of a WinRT class. When passed in public methods or properties, these must be converted to WinRT types.

One typical scenario is an existing C++ type, perhaps written sometime in the past, and needs to be used in WinRT for data binding purposes, or at least exposed to WinRT clients beyond the current project. Let's see how this transition can be achieved.

Converting C++ to WinRT

Let's take a concrete example, and then discuss things more generally. Suppose we have the following standard C++ classes:

```
#include <string>
#include <vector>

class book_review {
public:
    book_review(const std::wstring& name,
               const std::wstring& content,
               int rating);

    int rating() const { return _rating; }
    void set_rating(int rating) { _rating = rating; }
    const std::wstring& name() const { return _name; }
    const std::wstring& content() const { return _content; }

private:
    std::wstring _name;
    std::wstring _content;
    int _rating;
};

class book {
public:
    book(const std::wstring& name, const std::wstring& author);
    void add_review(const book_review& review);
    size_t reviews_count() const { return _reviews.size(); }
    const book_review& get_review(size_t index) const {
        return _reviews[index];
    }
    const std::wstring& name() const { return _name; }
    const std::wstring& author() const { return _author; }

private:
    std::wstring _name;
    std::wstring _author;
    std::vector<book_review> _reviews;
};
```

Simply stated, a `book` class is defined and has a name, an author, and a collection of reviews (`book_review` class). Each review consists of a name, the review content, and a numeric rating.

These classes are written in standard C++ and have no knowledge of WinRT (or C++/CX for that matter).

As these stand, they can only be used internally in a C++ project. They cannot be exported to other WinRT environments (for example, .NET), and even in a C++ project they cannot benefit from features such as data binding, as these are not WinRT classes in any way.

These (and similar) classes need to be wrapped in a WinRT class. With C++, this can be done in two ways. The first is by using WRL; the benefit is that standard C++ is being used (and not Microsoft-specific extensions), but this benefit is somewhat diminished, as WinRT is Microsoft specific anyway (at least at the time of writing). The second possible benefit is more control of the resulting WinRT types. Although this may sound appealing, it's also much harder to do, and unnecessary for most cases, so most of the time we'll take the easier approach by leveraging C++/CX.



Using WRL to create a WinRT component is sometimes necessary. One example is when a single class needs to implement a WinRT interface and a native COM interface. For instance, a media encoder or decoder must be COM/WinRT classes that must implement not only the `Windows::Media::IMediaExtension` interface, but also the Media Foundation non-WinRT interface, `IMFTTransform`. WRL is the only way to achieve this.

To wrap the book-related classes, we'll create a Windows Runtime Component project (we'll call it `BookLibrary`). Then, we'll add a C++/CX WinRT class to wrap `book` and `book_review`. Let's start with the `book_review` wrapper:

```
[Windows::UI::Xaml::Data::BindableAttribute]
public ref class BookReview sealed {
public:
    BookReview(Platform::String^ name, Platform::String^ content,
        int rating);

    property Platform::String^ Name { Platform::String^ get(); }
    property Platform::String^ Content { Platform::String^ get(); }
    property int Rating {
        int get() { return _review.rating(); }
        void set(int rating) { _review.set_rating(rating); }
    }
private:
    book_review _review;
};
```

A few things to note:

- The `Bindable` attribute is applied to the class so that proper code is generated for data binding to work.
- All the public stuff is WinRT only. The `book_review` wrapped instance is in the private section of the class. Any attempt to make it public would cause a compilation error. The error states, "**a non-value type cannot have any public data members**"; that's the first issue – since WinRT is based on COM, and COM is based on interfaces, which are defined by virtual tables, they can only contain methods (functions) and not data members.

If the data member would turn into a method that returns a non-WinRT type, the compiler would issue a different error, "**(MethodName): signature of public member contains native type 'book_review'**". The net result is that only WinRT types can be used in public members.

- Standard C++ has no concept of properties. Data members are sometimes wrapped by getters and/or setters. These should be turned into WinRT properties, as was done with `Name`, `Content`, and `Rating` in the preceding code.

WinRT coding conventions are to use Pascal casing for class and member names, so these may need to change slightly to reflect this (for example, name in `book_review` is changed to `Name` in `BookReview`, and so on).

- One thing that's missing from the `BookReview` class is implementation of `INotifyPropertyChanged`, as described in *Chapter 5, Data Binding*. This is needed because the `Rating` property can be changed after a `BookReview` is constructed. The implementation was omitted for easier focus on the fundamentals, but should be implemented in a real scenario.

The header file does not implement the constructor and the properties `Name` and `Content`. Here's the constructor (implemented in the corresponding CPP file):

```
BookReview::BookReview(String^ name,  
String^ content, int rating) :  
    _review(name->Data(), content->Data(), rating) { }
```

The constructor (like any other method) must accept WinRT types, using `Platform::String^` for any string that's required. This is used to initialize the wrapped `book_review` instance (which requires a standard `std::wstring`) by using the `Data` method.

The `Name` and `Content` properties are read only, but must return WinRT types – a `Platform::String^` in this case (which as you may recall wraps a WinRT `HSTRING`):

```
String^ BookReview::Name::get() {
    return ref new String(_review.name().c_str());
}

String^ BookReview::Content::get() {
    return ref new String(_review.content().c_str());
}
```

The implementation is straightforward, this time going in the other direction, by using a `Platform::String` constructor that accepts a `const wchar_t*`.

Next, we need to take a look at the wrapper created for the `book` class. This is a bit more complicated, as a book holds a `std::vector` of `book_review` objects; `std::vector` is not a WinRT type, so it must be projected with another type, representing a collection:

```
[Windows::UI::Xaml::Data::BindableAttribute]
public ref class Book sealed {
public:
    Book(Platform::String^ name, Platform::String^ author);
    void AddReview(BookReview^ review);

    property Platform::String^ Name {
        Platform::String^ get() {
            return ref new Platform::String(_book.name().c_str());
        }
    }

    property Platform::String^ Author {
        Platform::String^ get() {
            return ref new Platform::String(_book.author().c_str());
        }
    }

    property Windows::Foundation::Collections::
        IVectorView<BookReview^>^ Reviews {
        Windows::Foundation::Collections::
            IVectorView<BookReview^>^ get();
    }

private:
```

```

    book _book;
    Windows::Foundation::Collections::
        IVectorView<BookReview^>^ _reviews;
};

```

The Name and Author properties are straightforward, and implemented inline. The constructor initializes these, and they remain read only throughout the lifetime of the object.

The original book class has a `std::vector<book_review>` instance. In WinRT, a collection such as vector should be projected as a `Windows::Foundation::Collections::IVector<BookReview>` or `IVectorView<BookReview>` (in the same namespace, the latter being a read-only view of the former).



The namespace prefixes may be a bit confusing. Why is `IVector<T>` in `Windows::Foundation::Collections` but `Vector<T>` is in `Platform::Collections`? The rule is simple. WinRT types go into the `Windows::*` namespaces, while the specific C++ implementation go into the `Platform::*` namespace. Generally speaking, `Platform::*` types cannot be exported in WinRT types, as they are C++-specific implementations of WinRT interfaces (mostly). The notable exceptions are `Platform::String` and `Platform::Object`, which are understood as the replacement for `HSTRING` and `IInspectable` pointers respectively, and so are used in public methods and properties.

The `Book` class provides the `Reviews` read-only property as `IVectorView<BookReview^>^`. It can return any object implementing this interface. The `Platform::Collections::Vector<T>` provides an implementation of `IVector<T>`. `IVector<T>` provides the `GetView` method, returning `IVectorView<T>`:

```

IVectorView<BookReview^>^ Book::Reviews::get() {
    if(_reviews == nullptr) {
        auto reviews = ref new Vector<BookReview^>();
        for(size_t i = 0; i < _book.reviews_count(); i++) {
            auto review = _book.get_review(i);
            reviews->Append(
                ref new BookReview(
                    ref new String(review.name().c_str()),
                    ref new String(review.content().c_str()),
                    review.rating()));
        }
    }
}

```

```

    _reviews = reviews->GetView();
}
return _reviews;
}

```

The property implementation tries to optimize by caching the `IVectorView<BookReview>` result if no new reviews are added, or the property is never called (indicated by a `nullptr` in `_reviews`). Otherwise, `Vector<BookReview>` is created, and `BookReview` objects are added with `IVector<BookReview>::Append`.

The last interesting method to implement is `AddReview`:

```

void Book::AddReview(BookReview^ review) {
    book_review br(review->Name->Data(),
        review->Content->Data(), review->Rating);
    _book.add_review(br);
    _reviews = nullptr;
}

```

The `_reviews` data member is set to `nullptr` to force future calls to the `Reviews` property to regenerate the returned collection.



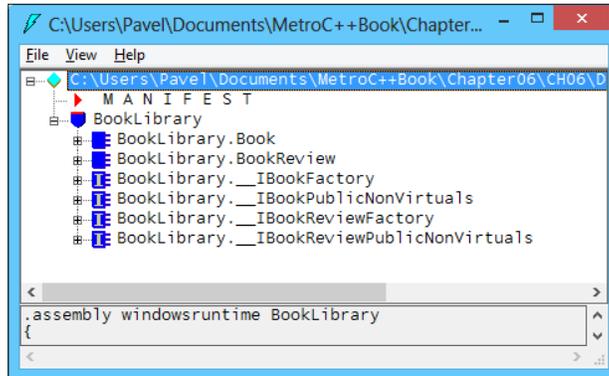
When working with collections such as `std::vector` and its WinRT wrappers (such as `Vector<T>`), try to use `std::vector` as much as possible. Only use `Vector<T>` when exporting from a WinRT class. Make all collection manipulations on native C++ types as they have less overhead than WinRT types (because of the WinRT interface based nature).

Crossing the ABI

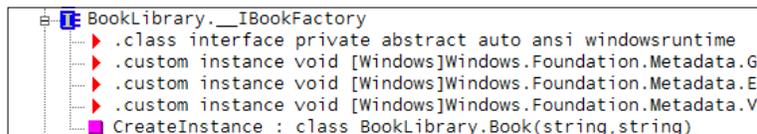
The **Application Binary Interface (ABI)** is the boundary between standard C++ and WinRT. Any C++ class that is not implemented as a WinRT class cannot cross the ABI. The previously used types `std::wstring` and `std::vector<>` are perfect examples of types that require projection when crossing the ABI. The compiler will not allow non-WinRT types to be used in public sections in `public ref class` declarations. See *Chapter 2, COM and C++ for Windows 8 Store Apps* for further discussions on mapping native C++ types to WinRT types.

Consuming Windows Runtime Components

Once a Windows Runtime Component is built, a metadata file (.winmd) is created that indicates which types, interfaces, enums, and so on are exported from the library. For example, our BookLibrary component DLL produces BookLibrary.winmd. Opening it in ILDASM shows this:

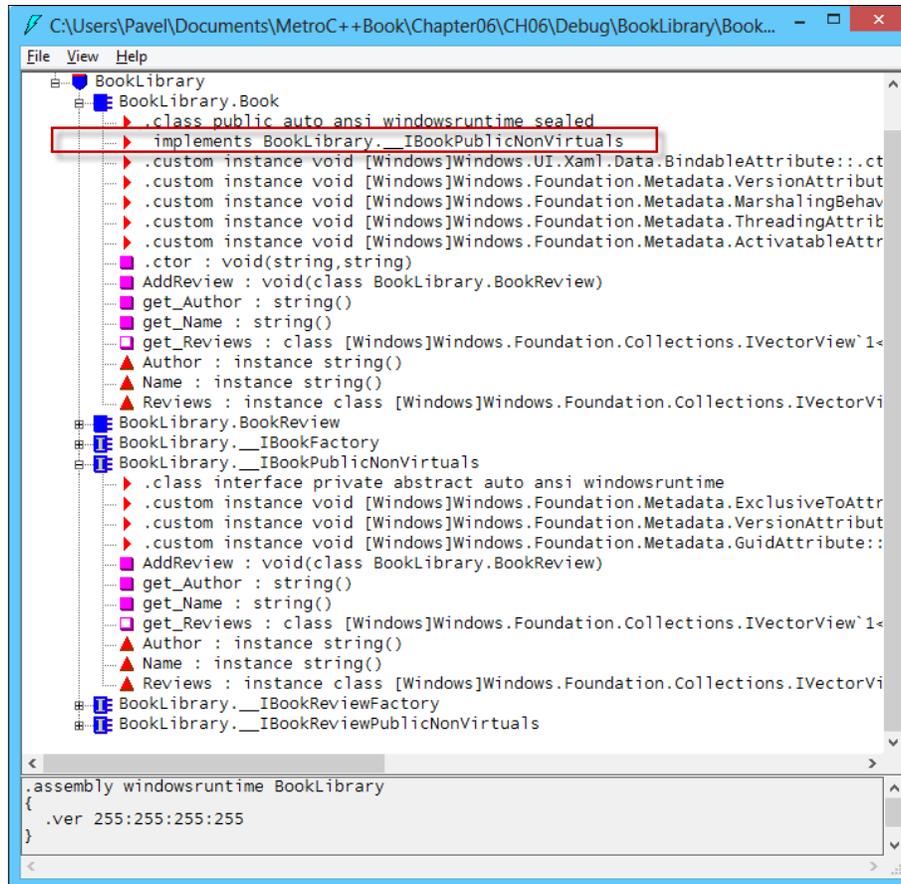


This clearly shows the exported types, Book and BookReview. The strange interface names represent the internal WinRT implementation provided by the compiler—WinRT is all about interfaces. The *Factory interfaces exist if there are any non-default constructors. For example, opening __IBookFactory shows this:

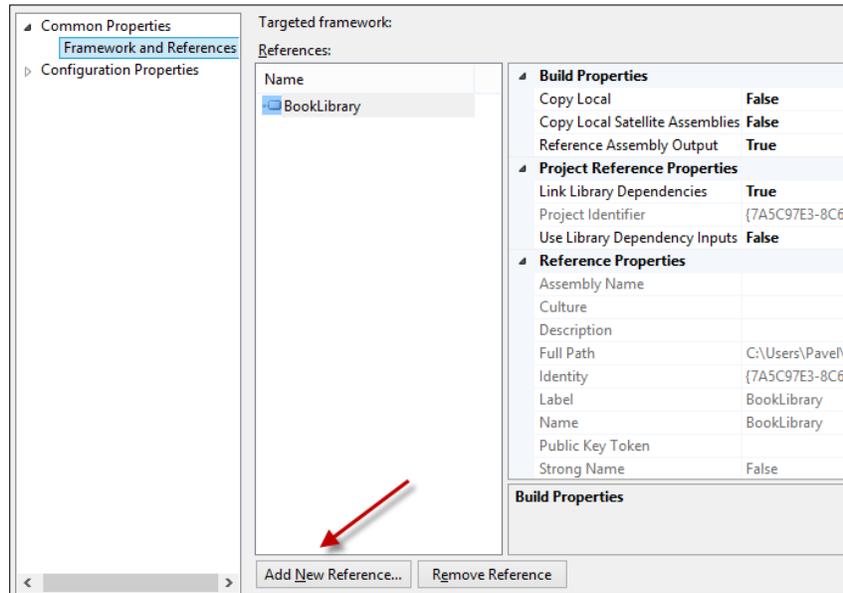


Note the CreateInstance method, modelled after the single constructor of Book. This interface is implemented by the activation factory that creates Book instances (implemented behind the scenes by C++/CX for any public ref class).

The `__IBookPublicNonVirtuals` interface is the one implemented by the `Book` class:



Consuming the resulting DLL is possible from any WinRT compliant environment. In a C++ project, a reference to the winmd file needs to be added. For that right-click on the project node in Solution Explorer and select **References....** And then select **Add New Reference** in the **Common Properties** or **Framework and References** node (or get to the same location from the project properties):



After the reference is added (by selecting the BookLibrary project, or browsing for a winmd file in the general case), all exported types can be used immediately, just like any other WinRT type. Here's an example of creating a Book with some reviews:

```
using namespace BookLibrary;

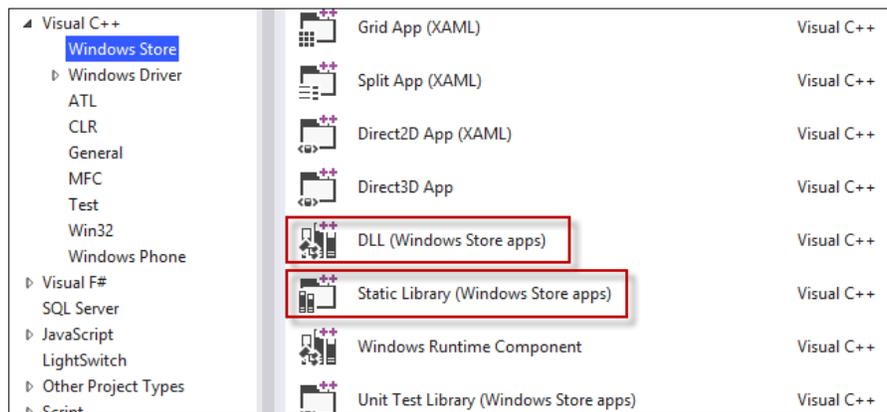
auto book = ref new Book("Windows Internals", "Mark Russinovich");
book->AddReview(
    ref new BookReview("John Doe",
        "Great book! Lots of pages!", 4));
book->AddReview(
    ref new BookReview("Mickey Mouse",
        "Why two parts? This makes my ears spin!", 3));
book->AddReview(
    ref new BookReview("Clark Kent",
        "Big book. Finally something to stretch the muscles!", 5));
```

Consuming the `BookLibrary` DLL from other environments, such as .NET can be similarly accomplished, as demonstrated in *Chapter 2, COM and C++ for Windows 8 Store Apps*. Each environment performs whatever projection is needed, all based on the metadata (`winmd`) file.

 WinRT components created with C++ are the only ones that guarantee not involving the .NET CLR. Components created by C# will always require the CLR, even if used from a C++ client.

Other C++ library projects

Looking at the available project types in Visual Studio 2012 shows two more options for creating reusable libraries:



The outlined projects create a classic DLL or static library, but do not, by default, generate a `winmd` file. These components can only be consumed by other C++ Store projects (WinRT components or other Store-enabled libraries). What's the difference in relation to a regular, classic C++ DLL, or static library? First, any usage of a forbidden Win32 API will cause a compiler error. Second, these projects cannot use C++/CX unless specific steps are performed, such as adding a reference to the `platform.winmd` and `windows.winmd` files.

Custom control templates

In *Chapter 4, Layout, Elements, and Controls*, we discussed various elements and controls provided by WinRT. Customizing the appearance of elements and controls can be done using the following levels (from simple to complex). Of course, not all elements/controls support all levels:

- Change property values; by far the simplest customizations are achieved by changing properties. Common examples are the font-related properties (`FontSize`, `FontFamily`, and so on), `Foreground` and `Background`, and many others.
- For content controls (those deriving from `ContentControl`), the `Content` property can be set to any element, as complex as required. For example, this can make a `Button` show images, text, and anything else that's required, while still maintaining the expected button behavior.
- Data templates may be used for properties that support it, to display data objects in a rich and meaningful way. `ContentControl::Content` supports this, as it's typed as `Platform::Object^`, meaning it can accept anything. If this is a derived type that is not `UIElement`, the `DataTemplate` is used if provided (in this case through the `ContentControl::ContentTemplate` property). This is also used by all `ItemsControl` derivatives through the `ItemTemplate` property.
- Types derived from `ItemsControl` have the `ItemContainerStyle` and `ItemsPanel` properties that can further customize the way data is presented.

Although the preceding list is impressive, there are times when these customizations are not enough. For example, a `Button` is always rectangular; and although it can contain anything (it's a `ContentControl`), it can never be elliptic. Some things are just "baked" into the control's appearance. This is where control templates come in.

The fundamental difference between elements and controls is the existence of the `Control::Template` property that defines the way the control appears. Elements don't have this property. For example, an `Ellipse` is an ellipse, it cannot look like anything else, because that would violate its very definition. Thus, an `Ellipse` is an element and not a control.

Controls (deriving from `Control`) can change their `Template` property and have a different look (but preserve functionality). In fact, all controls have default templates that WinRT provides (otherwise, controls would have no "look").

Building a control template

A control template is of the type `ControlTemplate`. It's very similar to a `DataTemplate` (both derive from `FrameworkTemplate`), and can contain a single `UIElement` (typically a `Panel`) that comprises the control's look.

As an example, we'll build an alternate control template for the `ProgressBar` control. We'll start with simple steps, and add features as we go along.

A control template is typically created as a resource, so it can be reused more easily. Here's a first simple attempt:

```
<ControlTemplate TargetType="ProgressBar" x:Key="progTempl">
  <Grid>
    <Rectangle Fill="DarkBlue" />
    <Rectangle RadiusX="10" RadiusY="4" HorizontalAlignment="Left"
      Fill="Yellow" Margin="2" />
  </Grid>
</ControlTemplate>
```

To use the template, we simply set it to the `Template` property:

```
<ProgressBar Value="30" Height="40" Margin="10"
  Template="{StaticResource progTempl}" />
```

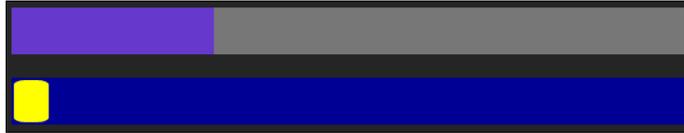
The idea here is to create a dark blue rectangle on top of which another rectangle (with rounded corners) will show the current progress. The result, however, is less than ideal (the top `ProgressBar` is using the default template):



The `ProgressBar` doesn't seem to show any progress (`Value="30"` should have shown 30 percent filled `ProgressBar`, as the default `Maximum` is 100, just like the top `ProgressBar`). And why would it? We just created a `Rectangle`, which has a default `Width` of 0. One way to get around this is to bind the `Width` property of the second `Rectangle` to the `Value` property of the `ProgressBar`. Here's one way to do it:

```
Width="{TemplateBinding Value}"
```

`TemplateBinding` is a markup extension that binds to the control being templated. This is necessary, as we can't use the `Source` or `ElementName` with a regular `Binding`. Here's the result:



This is certainly better, but the progress indicator seems small compared to the reference `ProgressBar` at the top. The reason is simple, `Value` is taken as `width`, but it really should be proportional to the width of the entire `ProgressBar`. We can solve this by using a value converter, but there is a better way.

The `ProgressBar` already has the smarts to set the `Width` property of some elements to the required proportional value. We just need to tell it which element it should be. It turns out this element must have a specific name, in this case `ProgressBarIndicator`. All we need to do is set the `x>Name` property to this value on the relevant element, our second `Rectangle`:

```
<Rectangle RadiusX="10" RadiusY="4" x>Name="ProgressBarIndicator"
  HorizontalAlignment="Left" Fill="Yellow" Margin="2" />
```

Here's the result:



Now it looks exactly right. Where did that special name come from? The secret is looking at the control's default template, looking for specially named parts. All the default control templates can be found in the file `C:\Program Files (x86)\Windows Kits\8.0\Include\WinRT\Xaml\Design\Generic.xaml` (on 32-bit Windows systems, the directory starts with `C:\Program Files`). The control templates are part of the default styles for the controls.

Looking at the `ProgressBar` control template, most elements are named with uninteresting names, for example, `e1`, `e2`, and so on—`ProgressBarIndicator` stands out.



In WPF and Silverlight, the `TemplatePart` attribute placed on controls indicate which named parts are looked up by the control and what their type should be. Although WinRT defines the `TemplatePart` attribute, it doesn't seem to be used in the current version of WinRT, so we resign ourselves to doing some "guesswork".

Using the control's properties

The template now functions correctly (or so it seems). Changing properties, such as `Foreground` or `Background`, has no effect when using our new template. That's because the template doesn't use them in any way. Sometimes, this is what we want, but typical templates want to provide ways to customize their own appearance; one way is to leverage existing properties on the control. This was already briefly demonstrated with `TemplateBinding` to the `Value` property, but here's a more interesting template, which uses several properties from `ProgressBar`:

```
<ControlTemplate TargetType="ProgressBar" x:Key="progTemp2">
  <Grid>
    <Rectangle Fill="{TemplateBinding Background}" />
    <Rectangle RadiusX="10" RadiusY="4"
      x:Name="ProgressBarIndicator"
      HorizontalAlignment="Left" Fill=
        "{TemplateBinding Foreground}"
      Margin="2"/>
    <TextBlock HorizontalAlignment="Center" Foreground="White"
      VerticalAlignment="Center" >
      <Run Text="{Binding Value, RelativeSource=
        {RelativeSource TemplatedParent}}" />
      <Span>%</Span>
    </TextBlock>
  </Grid>
</ControlTemplate>
```

In the preceding code snippet, there are several interesting things to note. The `TemplateBinding` markup extension is used to bind to the templated control's properties (`Background` and `Foreground`); `TemplateBinding` works for one-way bindings only (source to target, but not the other way around). For two-way binding properties, the longer syntax must be used, that is, a `Binding` expression with the `RelativeSource` property set to another markup extension, named `RelativeSource` (which should not be confused with the `Binding::RelativeSource` property name), which accepts `Mode` (also as a constructor parameter) that can be either `Self` (target and source are the same object, not useful here) or `TemplatedParent`, which means the control that is being templated, which is exactly what we want.

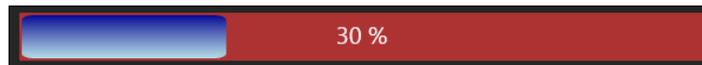


TemplateBinding should have worked here as well, as we're only interested in the one-way binding. But, since Value can be bound two way, TemplateBinding fails. This seems to be a bug in the current WinRT implementation.

Here's a `ProgressBar` that uses this template:

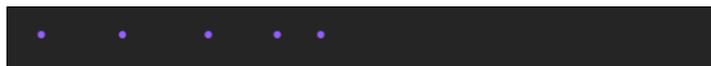
```
<ProgressBar Value="30" Height="40" Margin="10" FontSize="20"
  Template="{StaticResource progTemp2}"
  Background="Brown">
  <ProgressBar.Foreground>
    <LinearGradientBrush EndPoint="0,1">
      <GradientStop Offset="0" Color="DarkBlue" />
      <GradientStop Offset="1" Color="LightBlue" />
    </LinearGradientBrush>
  </ProgressBar.Foreground>
</ProgressBar>
```

This is the result:



Handling state changes

A `ProgressBar` normally shows the progress of an operation. Sometimes, however, the application doesn't know the progress of an operation – it only knows that it's underway. A `ProgressBar` can indicate this by setting its `IsIndeterminate` property to `true`. Here's how a standard `ProgressBar` appears in this mode:



It's actually difficult to capture a static image of that, as the `ProgressBar` shows an interesting nonlinear animation consisting of small circles.

Setting `IsIndeterminate` to `true` on the `ProgressBar` that uses our template has no effect on the way the `ProgressBar` is shown. That's because our control didn't take into account this property.

How can we solve that? One way would be to add something to the control template that would be hidden by default, but if `IsIndeterminate` turns to `true`, something will become visible and indicate that the `ProgressBar` is in that special mode (with a value converter, for instance). Although this is technically possible, that's not the way it's typically done. One of the reasons is that some state changes can be difficult to monitor with just bindings and value converters—for example, if the mouse cursor hovers over the control (not relevant for a `ProgressBar`, but relevant for many other controls), a property may not be enough. And how would we start an animation?

All these state changes and reactions are handled through an auxiliary object, known as `VisualStateManager`. A control transitions between states; these states and their transitions can be captured by the `VisualStateManager`. For each change, a set of `Storyboard` objects can be provided; these `Storyboard` objects represent animations in the general case, or simple state changes in a particular case.

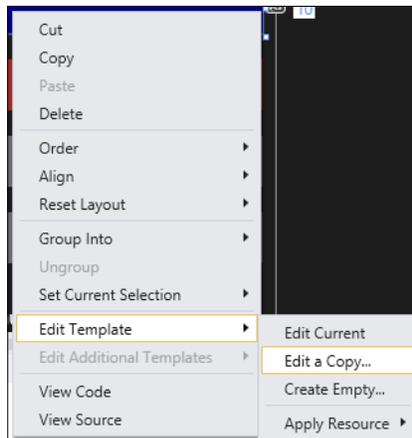
Here's an extended template that deals with the effects of the `IsIndeterminate` property:

```
<ControlTemplate TargetType="ProgressBar" x:Key="progTemp4">
  <Grid>
    <VisualStateManager.VisualStateGroups>
      <VisualStateGroup x:Name="CommonStates">
        <VisualState x:Name="Indeterminate">
          <Storyboard>
            <DoubleAnimation Storyboard.TargetProperty="Opacity"
              Storyboard.TargetName="IndetRect" To="1"
              Duration="0:0:1"
              AutoReverse="True" RepeatBehavior="Forever"/>
          </Storyboard>
        </VisualState>
        <VisualState x:Name="Determinate">
        </VisualState>
      </VisualStateGroup>
    </VisualStateManager.VisualStateGroups>
    <Rectangle Fill="{TemplateBinding Background}" />
    <Rectangle RadiusX="10" RadiusY="4"
      x:Name="ProgressBarIndicator" HorizontalAlignment="Left"
      Fill="{TemplateBinding Foreground}" Margin="2"/>
    <Rectangle x:Name="IndetRect" Opacity="0">
    <Rectangle.Fill>
      <LinearGradientBrush EndPoint=
        ".1,.3" SpreadMethod="Repeat">
        <GradientStop Offset="0" Color="Yellow" />
        <GradientStop Offset="1" Color="Red" />
      </LinearGradientBrush>
    </Rectangle.Fill>
  </Grid>
</ControlTemplate>
```

```
</Rectangle.Fill>
</Rectangle>
</Grid>
</ControlTemplate>
```

The `VisualStateManager` has one interesting property, which is an attached property, `VisualStateGroups`. For each group, one state is always active; this means a control can be in several states at once. For example, a button can be in a pressed state and in a keyboard-focused state. The `VisualStateGroups` property must be set on the top-level `Panel` comprising the control template (a `Grid` in our case).

Each `VisualStateGroup` consists of `VisualState` objects that indicate what to do (which animations to run) for each state. The state names must be the correct names, as the control changes to these states based on its internal logic. How did we know which state groups exist and which states are in each group? This is done by looking at the default control template. This can be done by looking in the aforementioned file, but can also be achieved with Visual Studio 2012 by right-clicking on a control, and selecting **Edit Template | Edit a Copy...**:



In the control template, a third rectangle named `IndetRect` was created with an initial opacity of zero, making it invisible. When the `ProgressBar` moves into the `Indeterminate` state, an animation is executed using the `DoubleAnimation` class (animating a property of the type `double`) that changes the opacity of that rectangle to 1 (fully shown) in one second (the `Duration` property), with auto reverse (`AutoReverse="true"`) and animating forever (`RepeatBehavior="Forever"`). Here's the result:



And the opacity is animating, fading in and out of this rectangle.



Complete cover of animations is beyond the scope of this book, but this should give you a sense of it. A `Storyboard` represents a time line, in which animation objects are played, in this case a `DoubleAnimation` object, but there are many others.

How does the state actually change? The control, with its own logic, calls the static `VisualStateManager::GoToState` method, setting the new state within a particular group. For a control template's author, that does not matter; the only thing that matters is setting the required animations according to the expected state changes.



The `VisualStateManager` also allows specifying the actual transition to take place when state changes occur. This is in contrast to the actual state itself. What this means is that a transition can be temporary when moving to a particular state, but the state itself may have different animations. For further information, refer to the MSDN documentation, starting with the `VisualStateManager::Transitions` property and the `VisualTransition` class.

Customizing using attached properties

The `ProgressBar` template created thus far used properties set on the `ProgressBar` itself using the `TemplateBinding` markup extension, or a regular `Binding` with a `Source` specified with the `RelativeSource` markup extension and with `TemplatedParent` as its `Mode`. What about adding properties that only make sense for our template? For example, in the preceding template definition, the `ProgressBar` shows a text string of its value. What if we wanted to allow the template user to hide the text or change its color?

The `ProgressBar` was not created with all this in mind. And why would it? It was created with the needed properties for some customization level; this is acceptable for the default `ProgressBar` template.

One way to get around this is to create a new class that derives from `ProgressBar` and add the required properties. Although this works (and we'll discuss custom controls in the next section), this is somewhat inelegant—we don't need any new functionality from the `ProgressBar`, rather we need some properties to tweak its template.

A more elegant solution is to use attached properties, which are defined on one class, but can be used by any other class (it must be derived from `DependencyObject`, though). Technically, we can look for appropriate attached properties defined elsewhere in WinRT, but it's better to create a new class that defines these attached properties and use them in the `ProgressBar` template.

Defining an attached property

Attached properties are dependency properties (which we'll discuss in detail in the next section) that are registered by calling the static `DependencyProperty::RegisterAttached` method. This sets up a static field that manages this property for all objects using it. Two static methods accompany the registration that actually sets and gets the attached property value on an object. Here's the declaration of a class, `ProgressBarProperties` that defines a single attached property, `ShowText`:

```
public ref class ProgressBarProperties sealed {
public:
    static bool GetShowText(DependencyObject^ obj) {
        return (bool)obj->GetValue(ShowTextProperty);
    }

    static void SetShowText(DependencyObject^ obj, bool value) {
        obj->SetValue(ShowTextProperty, value);
    }

    static property DependencyProperty^ ShowTextProperty {
        DependencyProperty^ get() { return _showTextProperty; }
    }

private:
    static DependencyProperty^ _showTextProperty;
};
```

The static field must be initialized and this is done in the CPP file:

```
DependencyProperty^ ProgressBarProperties::_showTextProperty =
    DependencyProperty::RegisterAttached(L"ShowText",
        TypeName(bool::typeid),
        TypeName(ProgressBarProperties::typeid),
        ref new PropertyMetadata(false));
```

The `RegisterAttached` method accepts the property name, its type (as a `TypeName` structure), the type of its owner, and a `PropertyMetadata` instance that can accept the default value of the property (if not set on an actual object and that property is queried). A more detailed explanation of `PropertyMetadata` can be found in the next section, where dependency properties are described; for now, we'll focus on the attached property usage in control templates.

The `TextBlock` inside the `ProgressBar` template can use the attached property as follows:

```
<TextBlock HorizontalAlignment="Center" Foreground="White"
  VerticalAlignment="Center"
  Visibility="{Binding (local:ProgressBarProperties.ShowText),
  RelativeSource={RelativeSource TemplatedParent},
  Converter={StaticResource bool2vis}}">
  <Run Text="{Binding Value, RelativeSource=
    {RelativeSource TemplatedParent}}" />
  <Span>%</Span>
</TextBlock>
```

The parenthesis around the property path is required, otherwise the XAML parser fails to understand the expression correctly, which results in a runtime binding failure. The converter used is for converting a `Boolean` to a `Visibility` enumeration, as was demonstrated in *Chapter 5, Data Binding*.

Clearly, defining and registering an attached property is simple, yet verbose. One solution would be to define macros to automate this boilerplate code. The downloadable source for this chapter has some macros for defining and registering dependency and attached properties that should make these easier to work with (in a file called `DPHelper.h`). Here's an example of another attached property, defined using the aforementioned macros. First, inside the `ProgressBarProperties` class:

```
DECLARE_AP(TextForeground, Windows::UI::Xaml::Media::Brush^);
```

And in the implementation file (to initialize the static field):

```
DEFINE_AP(TextForeground, Brush, ProgressBarProperties, nullptr);
```

This property can be used inside the template on the `TextBlock` as follows:

```
Foreground="{TemplateBinding
  local:ProgressBarProperties.TextForeground}"
```

Custom elements

Control templates provide a powerful and complete way to change the control's appearance. But that's just appearance – the control still behaves in the same way. If a new functionality is required, templates are not enough, and a new class should be created. This is where custom elements come in.

There are several ways of authoring custom elements in WinRT, we'll take a look at the two most frequently used controls – user controls and custom controls. Then, we'll briefly discuss how to create custom panels and custom shapes.

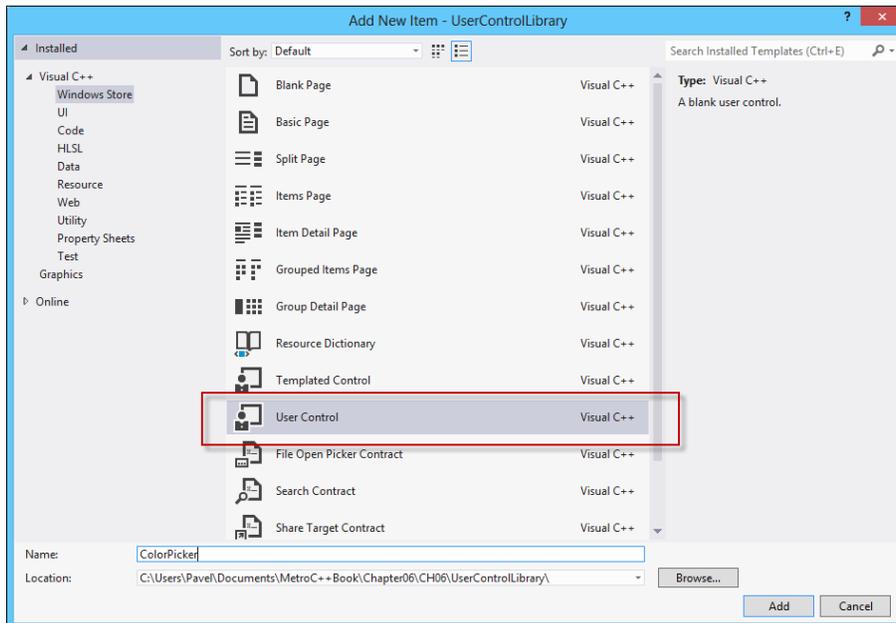
User controls

A user control is typically used to group related elements and controls together, for the purpose of re-use. Appropriate properties and events are exposed from this control, providing easy access to its functionality. As an added bonus, Visual Studio supports user control UI design, just as it does for a regular page.

User controls derive from the `UserControl` class. The UI design is effectively the `Content` property of the control, just like a `ContentControl`. They are typically placed in their own Windows Runtime Component project so that they can be used in any WinRT project, using C++ or another language.

Creating a color picker user control

As an example of a user control, we'll create a color picker control, which allows for selecting a solid RGB color by manipulating three sliders for the red, green, and blue color components (RGB). To begin, after creating a Windows Runtime Component project, we can add a new item of the type **User Control** to the project:



A design surface is opened with the usual pair of files created, `ColorPicker.h` and `ColorPicker.cpp`.

The first thing we want to do is define properties that would provide easy access to the user control's functionality. Most of the time, these will not be simple properties that wrap some private field, but rather dependency properties.

Dependency properties

Simple properties that wrap a field (perhaps with some validation in the setter) lack certain features that are desirable when working with a UI framework. Specifically, WinRT dependency properties have the following characteristics:

- Change notifications when the property's value is changed.
- Various providers can attempt to set the property's value, but only one such provider wins at a time. Nevertheless, all values are retained. If the winning provider goes away, the property's value is set to the next winner in the line.
- Property value inheritance down the visual tree (for some predefined set of properties).
- No memory is allocated for a property's value if that value is never changed from its default

These features provide the basis of some of WinRT's powerful capabilities, such as data binding, styles, and animation.

On the surface, these properties look the same as any other property—a getter and a setter. But no private fields are involved. Instead, a static field manages the property values for all instances using that property.

Defining dependency properties

Here's the way to define a dependency property (must be done in a class that derives from `DependencyObject`, which is always the case with `UserControl`). A private static field manages the property, which is exposed publicly as a read-only property. A setter and getter exist as easy access to the actual `set` and `get` methods, implemented in the `DependencyObject` base class. The following code demonstrates the creation of a dependency property of the type `Windows::UI::Color` named `SelectedColor` that is exposed by the `ColorPicker` user control:

```
public ref class ColorPicker sealed {
public:
//...
property Windows::UI::Color SelectedColor {
    Windows::UI::Color get() {
        return (Windows::UI::Color)GetValue(SelectedColorProperty);
    }
}
```

```
    void set(Windows::UI::Color value) {
        SetValue(SelectedColorProperty, value); }
}

property DependencyProperty^ SelectedColorProperty {
    DependencyProperty^ get() { return _selectedColorProperty; }
}

private:
    static DependencyProperty^ _selectedColorProperty;
};
```

A few things to note:

- The `GetValue` and `SetValue` properties are inherited from `DependencyObject`.
- The static property's name should be suffixed with `Property`.
- It's never a good idea to add more code to the `get()` or `set()` parts of the property, because these are sometimes not used and one can call the `GetValue` and `SetValue` methods directly; this is done by the XAML parser, for example.

The missing part is the initialization of the static field, typically done in the `.cpp` file:

```
DependencyProperty^ ColorPicker::_selectedColorProperty =
    DependencyProperty::Register(
        "SelectedColor", TypeName(Color::typeid),
        TypeName(ColorPicker::typeid),
        ref new PropertyMetadata(Colors::Black,
            ref new PropertyChangedCallback(
                &ColorPicker::OnSelectedColorChanged)));
```

A **dependency property (DP)** is registered by calling the static `DependencyProperty::Register` method, passing the property name, its type (as a `TypeName` structure), the containing type, and the `PropertyMetadata` object, which can accept the property's default value (`Colors::Black` in this case) and an optional callback to invoke when the property changes. This will be useful in the `ColorPicker` case.

This code can repeat numerous times, once for each DP. This clearly calls out for some helper macros. Here are three other properties defined on the `ColorPicker` using the macros. First, in the header file:

```
DECLARE_DP(Red, int);
DECLARE_DP(Green, int);
DECLARE_DP(Blue, int);
```

And the .cpp file:

```
DEFINE_DP_EX(Red, int, ColorPicker, 0, OnRGBChanged);
DEFINE_DP_EX(Green, int, ColorPicker, 0, OnRGBChanged);
DEFINE_DP_EX(Blue, int, ColorPicker, 0, OnRGBChanged);
```

This is considerably shorter (and less error prone) than the verbose version. These macros can be found in the `DPHelper.h` file, available with the downloadable source code for this chapter.

The next thing to do is implement the change notification methods, if they exist. In this case, `Red`, `Green`, and `Blue` should reflect the `SelectedColor` property color components, and vice versa. First, if `Red`, `Green`, or `Blue` changes, use the following code snippet:

```
void ColorPicker::OnRGBChanged(DependencyObject^ obj,
    DependencyPropertyChangedEventArgs^ e) {
    ((ColorPicker^)obj)->OnRGBChangedInternal(e);
}

void ColorPicker::OnRGBChangedInternal(
    DependencyPropertyChangedEventArgs^ e) {
    auto color = SelectedColor;
    auto value = safe_cast<int>(e->NewValue);
    if(e->Property == RedProperty)
        color.R = value;
    else if(e->Property == GreenProperty)
        color.G = value;
    else
        color.B = value;
    SelectedColor = color;
}
```

Since the registered handler must be static, it's easier to delegate the actual work to an instance method (`OnRGBChangedInternal` in the preceding code). The code updates the `SelectedColor` property based on the changed RGB property.

The other direction implementation is along the same lines:

```
void ColorPicker::OnSelectedColorChanged(DependencyObject^ obj,
    DependencyPropertyChangedEventArgs^ e) {
    ((ColorPicker^)obj)->OnSelectedColorChangedInternal(
        safe_cast<Color>(e->NewValue));
}
```

```
void ColorPicker::OnSelectedColorChangedInternal(Color newColor) {  
    Red = newColor.R;  
    Green = newColor.G;  
    Blue = newColor.B;  
}
```



The preceding code snippets may seem to create an infinite loop—if Red changes, SelectedColor changes, which changes Red again, and so on. Fortunately, this is handled automatically by the dependency property mechanism, which invokes the callback if the property value actually changes; setting to the same value does not invoke the callback.

Building the UI

The next step is to create the actual UI of the user control using regular XAML. Binding expressions can be used to bind to properties exposed by the control (since these are DPs, they provide automatic change notification for bindings). Here's a UI for the ColorPicker with the sliders bound to the Red, Green, and Blue properties and a Rectangle that binds to the SelectedColor property of the control (default XAML namespaces are omitted):

```
<UserControl  
    x:Class="UserControlLibrary.ColorPicker"  
    x:Name="uc">  
    <UserControl.Resources>  
    </UserControl.Resources>  
    <Grid>  
        <Grid.RowDefinitions>  
            <RowDefinition Height="Auto" />  
            <RowDefinition Height="Auto" />  
            <RowDefinition Height="Auto" />  
        </Grid.RowDefinitions>  
        <Grid.ColumnDefinitions>  
            <ColumnDefinition />  
            <ColumnDefinition Width="150" />  
        </Grid.ColumnDefinitions>  
        <Slider Maximum="255" Margin="4" TickFrequency="20"  
            Value="{Binding Red, ElementName=uc, Mode=TwoWay}"/>  
        <Slider Maximum="255" Margin="4" TickFrequency="20"  
            Value="{Binding Green, ElementName=uc, Mode=TwoWay}"  
            Grid.Row="1"/>  
        <Slider Maximum="255" Margin="4" TickFrequency="20"  
            Value="{Binding Blue, ElementName=uc, Mode=TwoWay}"
```

```

        Grid.Row="2"/>
<Rectangle Grid.Column="1" Grid.RowSpan="3" Margin="10"
    Stroke="Black" StrokeThickness="1">
    <Rectangle.Fill>
        <SolidColorBrush Color="{Binding SelectedColor,
            ElementName=uc}" />
    </Rectangle.Fill>
</Rectangle>
</Grid>
</UserControl>

```

Adding events

Events can be added to the user control to notify interested parties of interesting events. Here's an event added in the header file of the control:

```
event EventHandler<Windows::UI::Color>^ SelectedColorChanged;
```

The event uses the `EventHandler<T>` delegate that requires the client to provide a method that accepts a `Platform::Object^` and a `T` (`Color` in this case). We'll raise the event when the `SelectedColor` property changes:

```

void ColorPicker::OnSelectedColorChangedInternal(Color newColor) {
    Red = newColor.R;
    Green = newColor.G;
    Blue = newColor.B;

    SelectedColorChanged(this, newColor);
}

```

Using the ColorPicker

Now we can use the `ColorPicker` in another project, by adding a reference in the usual way and adding an XML namespace mapping. Then just use the control like any other. Here's an example:

```

<StackPanel VerticalAlignment="Center">
    <Border Margin="10" Padding="6" Width="500" BorderBrush="White"
        BorderThickness="2" >
        <controls:ColorPicker SelectedColorChanged="OnColorChanged" />
    </Border>
    <TextBlock FontSize="30" HorizontalAlignment="Center">
        <Span>Color: #</Span>
        <Run x:Name="_color" />
    </TextBlock>
</StackPanel>

```

The control is placed inside a border and its `SelectedColorChanged` event is handled as follows:

```
void MainPage::OnColorChanged(Object^ sender, Color color) {
    wstringstream ss;
    ss.fill(L'0');
    ss << hex << uppercase << setw(2) << color.R << setw(2) <<
    color.G << setw(2) << color.B;
    _color->Text = ref new String(ss.str().c_str());
}
```

This changes the `TextBlock` at the bottom of the control. This is how it looks at runtime:



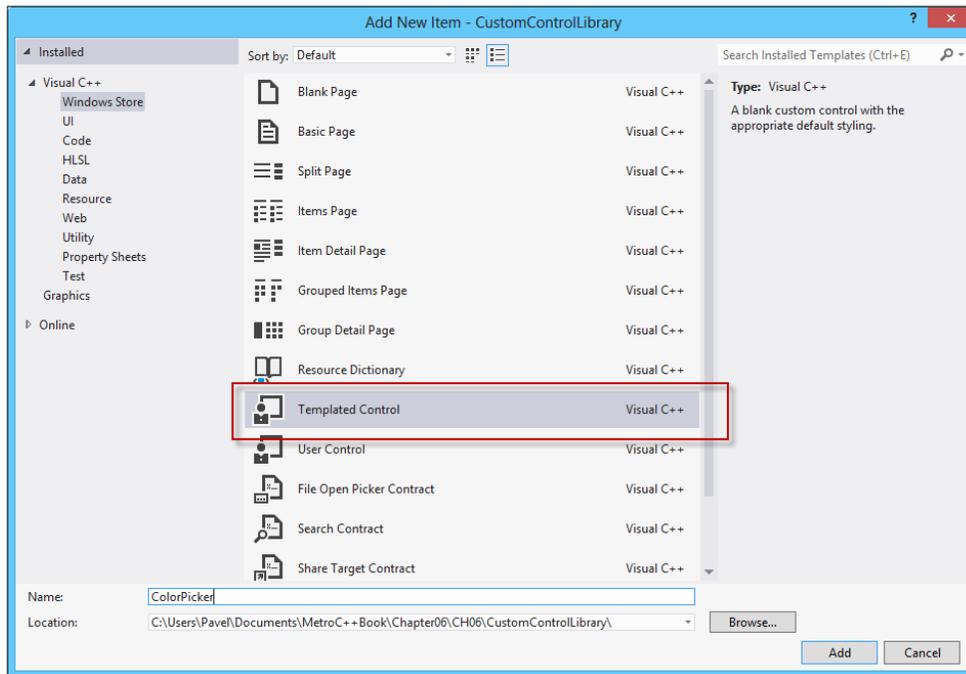
Custom controls

User controls are great for encapsulating a piece of UI functionality that can be easily reused. Their potential disadvantage is the lack of deep customization. Suppose in the `ColorPicker` example, we would like to have the sliders placed vertically rather than horizontally, or we want an ellipse rather than a rectangle. Although it's possible to add properties that would allow some customization, there's no way we can anticipate everything.

The solution is to create a custom control with a default control template that can be changed completely as desired, while still retaining the original functionality. This is exactly how the regular WinRT controls are built.

Creating a ColorPicker custom control

A custom (also known as templated) control derives from the `Control` class. A good starting point is the Visual Studio Templated Control template:



The result is a pair of files, `ColorPicker.h` and `ColorPicker.cpp`, and an XAML file named `Generic.xaml` that holds the default style for the `ColorPicker` that includes the default template as follows:

```
<Style TargetType="local:ColorPicker">
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="local:ColorPicker">
        <Border
          Background="{TemplateBinding Background}"
          BorderBrush="{TemplateBinding BorderBrush}"
          BorderThickness="{TemplateBinding BorderThickness}">
        </Border>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```



All custom control styles must reside in the same `Generic.xaml` file. Its name and origin lie in WPF, which supports different styles for different Windows UI Themes. This is not relevant to WinRT, but the convention remains.

Practically, when authoring multiple custom controls, using the same file is inconvenient at best. This can be remedied by using the `ResourceDictionary::MergedDictionaries` property to include other XAML files into `Generic.xaml`.

The default template looks much the same as the default UI created for the user control, with one important difference; no data binding expressions. The reason is that if there were bindings, a custom template would have to duplicate those to maintain functionality and this puts an unreasonable burden on custom template authors; the alternative is binding in code. Here's the revised markup for the default template of `ColorPicker`:

```
<ControlTemplate TargetType="local:ColorPicker">
  <Border
    Background="{TemplateBinding Background}"
    BorderBrush="{TemplateBinding BorderBrush}"
    BorderThickness="{TemplateBinding BorderThickness}">
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
        <RowDefinition Height="Auto" />
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition />
        <ColumnDefinition Width="150" />
      </Grid.ColumnDefinitions>
      <Slider Maximum="255" Margin="4" TickFrequency="20"
        x:Name="PART_Red"/>
      <Slider Maximum="255" Margin="4" TickFrequency="20"
        x:Name="PART_Green" Grid.Row="1"/>
      <Slider Maximum="255" Margin="4" TickFrequency="20"
        x:Name="PART_Blue" Grid.Row="2"/>
      <Rectangle Grid.Column="1" Grid.RowSpan="3" Margin="10"
        Stroke="Black" StrokeThickness="1">
        <Rectangle.Fill>
          <SolidColorBrush x:Name="PART_Color" />
        </Rectangle.Fill>
      </Rectangle>
    </Grid>
  </Border>
</ControlTemplate>
```

```

    </Grid>
  </Border>
</ControlTemplate>

```

The interesting parts of the template are assigned names. These names will be looked up by the control and bound in code. These are the named parts discussed at the beginning of this chapter.

Binding in code

Defining the dependency properties and events in a custom control is exactly the same as with a user control.

When a template is applied to a control, the virtual `Control::OnApplyTemplate` method is called. This is the best opportunity for the control to hunt down its named parts and connect to them using bindings, or event handlers, as appropriate.

To bind the three sliders, a helper method is created as follows:

```

void ColorPicker::BindSlider(String^ name, String^ propertyName) {
    auto slider = (RangeBase^)GetTemplateChild(name);
    if(slider != nullptr) {
        auto binding = ref new Binding;
        binding->Source = this;
        binding->Path = ref new PropertyPath(propertyName);
        binding->Mode = BindingMode::TwoWay;
        BindingOperations::SetBinding(slider,
            RangeBase::ValueProperty, binding);
    }
}

```

The method uses `GetTemplateChild()` to get a named element. If that element does not exist, `nullptr` is returned. A typical control simply moves on and does not throw an exception.



Note that the code casts to `RangeBase` rather than `Slider`. This is possible because the required property is `Value` defined on `RangeBase`. This means that this can be something other than a `Slider`, as long as it derives from `RangeBase` (for example, `ScrollBar` or `ProgressBar`).

Next, a binding is created in code by instantiating a `Binding` object, setting the source object (the `Source` and `Path` properties), binding mode (the `Mode` property), and converter (if needed, using the `Converter` property), and then finally calling `BindingOperations::SetBinding` with the target object, the target DP, and the binding instance.

The complete `OnApplyTemplate` would be as follows:

```
void ColorPicker::OnApplyTemplate() {
    BindSlider("PART_Red", "Red");
    BindSlider("PART_Green", "Green");
    BindSlider("PART_Blue", "Blue");
    auto color = (SolidColorBrush^)GetTemplateChild("PART_Color");
    if(color != nullptr) {
        auto binding = ref new Binding;
        binding->Source = this;
        binding->Path = ref new PropertyPath(L"SelectedColor");
        BindingOperations::SetBinding(color,
            SolidColorBrush::ColorProperty, binding);
    }
}
```

The three potential sliders (actually the controls derived from `RangeBase`) are bound, and then the `SolidColorBrush` is bound, if it exists. This means it can be the `Fill` of a `Rectangle`, an `Ellipse`, or the `BorderBrush` of a `Border`—as long as it's a `SolidColorBrush`.

Using the custom control is the same as using the user control. But, it's possible to replace the control template (as was done at the beginning of this chapter with the `ProgressBar`) to create a `ColorPicker` that looks different, yet has the same functionality, all with no code at all—just XAML.

Custom panels

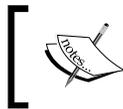
WinRT provides the standard panels deriving from `Panel`. It's possible to create new panels that arrange their children in unique ways, such as a radial panel, whose children are arranged along the circumference of an ellipse.

The layout process is a two-step process—measure and arrange. This is modelled precisely by the methods of `Panel` to override two methods for this exact purpose, `MeasureOverride` and `ArrangeOverride`.

`MeasureOverride` asks the panel (or any element that overrides it for that matter) what size it requires. For a panel, the main concern is the requirements of its child elements. The panel should call `UIElement::Measure` for each child element, causing its own `MeasureOverride` to be called (and this may go on if that child is a panel, or acts like a panel, in itself).

The panel needs to decide what size it requires based on its children's requirements and the layout logic it wants to employ. The parameter sent to `MeasureOverride` is the available size provided by that panel's container. This can be an infinite size in either one or both dimensions (for example, a `ScrollViewer` indicates it has infinite space in directions where scrolling is available). It's important to return a finite size; otherwise WinRT has no way of knowing how much space to leave for the panel and throws an exception.

`ArrangeOverride` is a more interesting method that actually implements a special layout logic, for which the panel was created. The panel calls `UIElement::Arrange` on each element that forces that element to be placed within a specific rectangle.



This procedure is almost identical to the way it's done in WPF or Silverlight; many such examples are on the web, and can be converted to WinRT with little difficulty.

Custom drawn elements

Custom drawn elements can be created in WinRT by deriving them from the `Windows::UI::Xaml::Path` class, which is a kind of `Shape`. A `Path` is based on a `Geometry`—a mathematical abstraction of a 2D layout that can be a `PathGeometry`, which in itself can be built from various `PathSegment` objects. These shapes are beyond the scope of this book, but, again, they are similar to the ones existing in Silverlight, so a lot of information about them is available.



WinRT currently does not support WPF's `OnRender` method that uses `DrawingContext` to do free-style drawing of various kinds. Hopefully, this will be supported in a future version.

Many new controls exist as part of the WinRT XAML toolkit, available for free on Microsoft's CodePlex site at <http://winrtxamltoolkit.codeplex.com/>. The problem with the toolkit is that it's written as a .NET class library, and so can only be used by .NET projects.

Summary

Components are the backbone of module re-use. True WinRT components use WinRT types only and so can be exported to any WinRT-compatible environment, such as C++/CX, .NET, and JavaScript.

Control templates provide the ultimate control customization mechanism that can be done in XAML alone, with little or no code (code may be needed if value converters are used). Templates are appropriate if the control's appearance needs to change, but its functionality should remain intact, and is what's needed.

Custom and user controls are used when new functionality is needed that is not provided by any built-in control. By deriving from `UserControl` and `Control`, dependency properties and events can be added to create a new reusable control.

User controls and custom controls should be packaged in such WinRT components for easy re-use by C++ and other projects.

In the next chapter, we'll take a look at some of the special features of Windows Store apps, such as Live Tiles and Push Notifications. These (and other) capabilities can make your Store app unique and appealing.

7

Applications, Tiles, Tasks, and Notifications

Windows Store applications are different from traditional desktop applications in a number of aspects. Store apps live inside a secure container, with well-defined ways of interacting with the outside world, such as other applications, the operating system, or something on the network. These apps are also imposed by several restrictions, unlike anything in the desktop app world. Knowing these restrictions and ways to handle them by cooperation with Windows, is key to a successful and well-behaving Windows Store application.

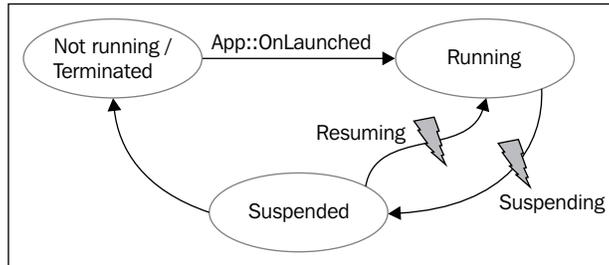
We'll start by examining the execution model of Store apps, and the way it differs from classic desktop apps. Then we'll take a look at some of the unique features of Store apps, such as live tiles and other notification mechanisms. Finally, we'll look at ways in which an application can do work even if it's not the currently running app, by using various forms of background tasks.

Application lifecycle

Store applications are managed by the Windows operating system with strict rules that need to be considered when developing apps:

- Only one app can be in the foreground at any single time (the notable exception being the "snap view": one app takes most of the screen, while another takes a width of 320 pixels; this is discussed in *Chapter 9, Packaging and the Windows Store*).
- Other apps are automatically suspended by Windows, meaning they get no CPU time; the memory they occupy, however, is preserved.
- If Windows detects shortage of memory, it may terminate the first suspended app; if memory is still tight, it will terminate the second suspended app, and so on.

These rules are there to ensure that the foreground application has full access to the CPU and other resources, while also conserving battery power as much as possible. The complete app lifecycle can be illustrated with the following state diagram:



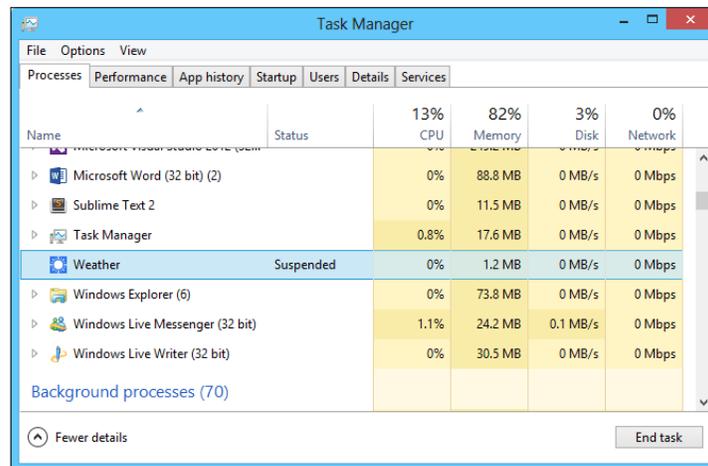
At first, an app is not running. Then the user launches the application, typically by clicking or tapping on its tile on the Start Screen. This causes the `Application::OnLaunched` virtual method to be invoked; this is where the app should initialize and present the main user interface.



The default code provided by Visual Studio for the `OnLaunched` method creates a `Frame` element that becomes the `Content` of the current `Window` (the one and only application window). Then a call is made to `Frame::Navigate` with the type name of `MainPage`, which causes `MainPage` to appear as expected.

The application is now in the running state, and the user can interact with the app. If the user switches to another application by pressing `Alt + Tab`, or going to the Start Screen and activating another app tile (or switches to another app by swiping from the left), our app is no longer in the foreground. If after 5 seconds, the user does not switch back to the app, it's suspended by the OS. Before that happens, the `Application::Suspended` event is fired. This is a chance for the app to save state, in case the app is terminated later. The app is given no more than 5 seconds to save state; if it takes longer, the app is terminated. Assuming all is well, the app is suspended.

The current state of the app can be viewed in **Task Manager**:



To view the application's status in **Task Manager**, first select the **View** menu and then **Status Values**, and click on **Show suspended status** (it's off by default).

Once suspended, the application may be resumed because the user switched back to it. This causes the `Resuming` event to fire on the `Application` object. In most cases, the app has nothing to do, because the app was retained in memory, so nothing has been lost. In cases where the UI should refresh due to stale data, the `Resuming` event can be used for that (for example, an RSS reader would refresh data because the app could have been suspended for hours, or even days).

While suspended, the application may be terminated by Windows because of low memory resources. The application gets no notification that this happened; this makes sense, as the app cannot use any CPU cycles while it's suspended. If the user activates the app again, `OnLaunched` is called, providing the opportunity to restore state with the help of the `LaunchActivatedEventArgs::PreviousExecutionState` property. One possible value is `ApplicationExecutionState::Terminated`, indicating the app was closed from the suspended state, and so state restoration should be attempted.



Individual pages within the app may want to be notified when the app is about to be suspended or resumed. This can be done in a constructor of `Page` by accessing the global `Application` object with `Application::Current`. A typical suspension registration may look like the following:

```
Application::Current->Suspending += ref new
SuspendingEventHandler(this, &MainPage::OnSuspending);
```

Saving and restoring the state

If, and when, the application is suspended, it's the application's responsibility to save whatever state is needed, in case Windows decides to terminate the application before it's resumed. This is done in response to the `Application::Suspending` event that can be handled on the application level and/or the page level.

Suppose we have a movie review application that allows users to review movies. A simple UI may exist that looks something like the following:



If the user switches to another app, the app will be suspended after 5 seconds have elapsed and the user does not switch back. We can use the `Windows::Storage::ApplicationData` class to access a local settings store or a local folder (for more complex storage requirements) to save the state of the preceding `TextBox` elements so that it can be restored should the application be terminated unexpectedly by Windows. First, we need to register for the `Suspending` event in the `MainPage` constructor:

```
MainPage::MainPage() {
    InitializeComponent();

    DataContext = _review = ref new MovieReview;

    Application::Current->Suspending +=
        ref new SuspendingEventHandler(
            this, &MainPage::OnSuspending);
}
```

The `MovieReview` class represents a review (implements `INotifyPropertyChanged` as discussed in *Chapter 5, Data Binding*), and the `TextBox` elements bind to its three properties. If the app is suspended, the following is executed:

```
void MainPage::OnSuspending(Object^ sender, SuspendingEventArgs^ e) {
    ApplicationData::Current->LocalSettings->Values->
        Insert("MovieName", _review->MovieName);
    ApplicationData::Current->LocalSettings->Values->
```

```

        Insert("ReviewerName", _review->ReviewerName);
    ApplicationData::Current->LocalSettings->Values->
        Insert("Review", _review->Review);
    }

```

The code uses the `ApplicationData::LocalSettings` property (an `ApplicationDataContainer` object), which manages a collection of key/value pairs (with optional inner containers), exposed through the `Values` property.



The types that can be stored in this way are limited to the basic WinRT types, and do not include custom types, such as `MovieReview`. It's possible to create some code that serializes such an object to XML or JSON and then saves it as a string.

If the application is indeed terminated, state needs to be restored. This can be done on the `Page::OnNavigatedTo` override, as follows:

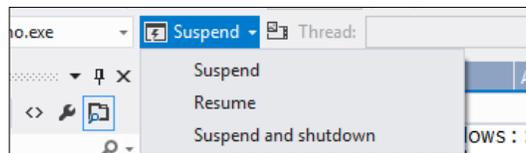
```

void MainPage::OnNavigatedTo(NavigationEventArgs^ e) {
    auto settings = ApplicationData::Current->LocalSettings->Values;
    if(settings->HasKey("MovieName"))
        _review->MovieName = safe_cast<String^>(
            settings->Lookup("MovieName"));
    if(settings->HasKey("ReviewerName"))
        _review->ReviewerName = safe_cast<String^>(
            settings->Lookup("ReviewerName"));
    if(settings->HasKey("Review"))
        _review->Review = safe_cast<String^>(
            settings->Lookup("Review"));
}

```

To test this, we can run the app without the Visual Studio debugger. However, if we need to debug the code, there is a slight problem. When the app is being debugged, it will never enter a suspended state. This is to allow the developer to switch to Visual Studio and look at code while the app is in the background and still be able to switch to it at any time.

We can force the app to go into suspension by using the Visual Studio toolbar button that allows suspending, resuming, and terminating the app (and invoking background tasks, as we'll see in the section *Background tasks* later in this chapter):



Determining application execution states

When the application is activated, it may be because it's being launched by the user (there are other options, such as implemented contracts, as we'll see in the next chapter). It's usually important to learn why the application has been closed the last time around. If it was terminated, state should have restored. If, on the other hand, it was closed by the user, perhaps the state should have cleared, as the user is expecting the app to start afresh.

We can determine this previous state using the `LaunchActivatedEventArgs::PreviousExecutionState` property available on the application's `OnLaunched` method override:

```
ApplicationData::Current->LocalSettings->Values
    ->Insert("state", (int)args->PreviousExecutionState);

if (args->PreviousExecutionState ==
    ApplicationExecutionState::Terminated) {
    // restore state
}
else if (args->PreviousExecutionState ==
    ApplicationExecutionState::ClosedByUser) {
    // clear state
}
```

It's useful to write the state to the `LocalSettings` container so that other entities (typically pages) can access this information after `OnLaunched` is done. This allows our restore code to query this state and act accordingly:

```
auto settings = ApplicationData::Current->LocalSettings->Values;
auto state = safe_cast<ApplicationExecutionState>(
    safe_cast<int>(settings->Lookup("state")));
if (state == ApplicationExecutionState::Terminated) {
    // restore state...
```

 Enums are also forbidden from being stored directly, but can be cast to an int and then stored.

State store options

The previous code samples used the `ApplicationData::LocalSettings` property. This uses a store local to the machine (and the current user and application), meaning that on another device running Windows 8, even if the same user is logged in, the same state will not be available.

WinRT provides an alternative, which allows settings to roam across devices by being stored within the Microsoft cloud service using the `ApplicationData::RoamingSettings` property. Working with this property is exactly the same as with `LocalSettings`; it's automatically synchronized with the cloud.

 Synchronization with the cloud can only work if the user has logged in to the system with his Microsoft ID (formerly Live ID), and not a "normal" username/password.

`LocalSettings` and `RoamingSettings` are useful for simple key/value pairs. If more complex data needs to be stored, we can create a folder (`StorageFolder` object) that we can use by creating `StorageFile` objects, more folders, and so on, as needed. This is available by accessing other `ApplicationData` properties: `LocalFolder`, `RoamingFolder`, and `TemporaryFolder` (`TemporaryFolder` stores information until the app terminates, and is not usually useful for application state management).

 Files stored in the local application store (`ApplicationData::LocalFolder`) can be accessed with a URI that starts with `ms-appdata:///local/` followed by the relative path of the file; replace `local` with `roaming` to access the roaming store. These URIs can be used in XAML, as well as in code.

Helper classes

Some of the Visual Studio 2012 project templates, such as Grid App, provide two classes that can help in state management: `SuspensionManager` and `LayoutAwarePage`. These provide the following features:

- Can manage a navigation pages stack, saved as XML in a local folder
- `LayoutAwarePage` must be used as the base page class
- Can save/restore this state automatically

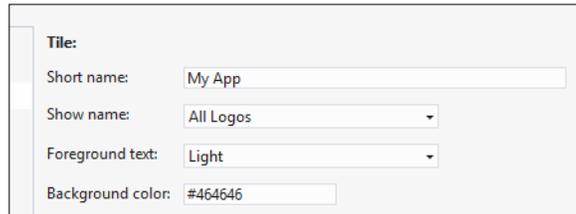
The interested readers should refer to the source code for more information on these classes.

Live tiles

One of the unique characteristics of Windows Store apps is the use of tiles on the Start Screen. These tiles can contain images and text, but these need not be constant and can change. Through various mechanisms to provide live and meaningful information, draw the user to tapping/clicking on the tile, to access the app itself. In this section, we'll take a look at creating and manipulating tiles.

Setting application tile defaults

The defaults of an application tile can be set in the application manifest, which is easily accessible through the Visual Studio user interface:



The screenshot shows a settings window titled "Tile:" with the following fields:

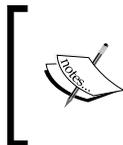
- Short name: My App
- Show name: All Logos (dropdown menu)
- Foreground text: Light (dropdown menu)
- Background color: #464646

Two sizes of tiles are available, the standard and the wide. If a wide logo image is available, it will be shown by default and the user will be able to change that to a standard tile (and vice versa) by right clicking on the tile (or swiping from the bottom) and selecting the relevant option:



The standard tile image should be 150 px x 150 px and the wide tile image should be 310 px x 150 px. If these sizes are not supplied, Visual Studio will issue a warning and the images will be stretched/shrunk as needed.

The **Short name** will appear on top of the tiles selected within the **Show name** combobox (**All Logos**, **No logos**, **Standard logo only**, and **Wide logo only**). The **Foreground text** option selects light or dark text and the background color selected will be used for transparent images (PNG files) and for some other dialogs, as the default background color.



An application should not define a wide tile unless the app plans to provide meaningful and interesting content within that tile. Just using a large static image is a bad idea; users will expect the tile to provide something more.

Updating the tile's contents

A tile can be updated by the running application. The updated tile will retain its contents even if the application is closed. Updating a tile involves creating some XML that specifies parts of the tile, which can include images and text in various layouts. There are also peek tile options that alternate between two tile sets. The first thing we need to do is select an appropriate tile template from an extensive, predefined set of templates. Each template is represented by an XML string that needs to be sent as the actual update.

Tile templates exist for standard and wide tiles; these are represented by the `Windows::UI::Notifications::TileTemplateType` enumeration. Here's an example of the generic XML needed for a wide tile with one text item, known as `TileWideImageAndText01` (the enum value):

```
<tile>
  <visual>
    <binding template="TileWideImageAndText01">
      <image id="1" src="image1.png" alt="alt text"/>
      <text id="1">Text Field 1</text>
    </binding>
  </visual>
</tile>
```

The highlighted element and inner text need to be updated with the required new content.



The complete template list and XML schemas can be found at [http://msdn.microsoft.com/EN-US/library/windows/apps/hh761491\(v=vs.10\).aspx](http://msdn.microsoft.com/EN-US/library/windows/apps/hh761491(v=vs.10).aspx).

After selecting the required template, we can retrieve the associated XML with the following code (no need to build the whole XML manually):

```
auto xml = TileUpdateManager::GetTemplateContent(
    TileTemplateType::TileWideImageAndText01);
```

The returned value is a `Windows::Data::Xml::Dom::XmlDocument`, representing the resulting XML. Now, we need to tweak the XML with the required updates. In this example, we'll change the image and text:

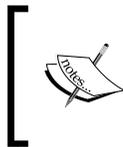
```
((XmlElement^)xml->GetElementsByTagName("image")->GetAt(0))
->SetAttribute("src", "assets\\bug.png");
xml->GetElementsByTagName("text")->GetAt(0)->AppendChild(
    xml->CreateTextNode("You have a bug!!!"));
```

The code uses the WinRT XML DOM API to manipulate the XML. The image is set to a local image, but remote images (`http://...`) work just as well.

The last step would be to create a tile updater for the app, build the tile notification and do the actual update:

```
auto update = TileUpdateManager::CreateTileUpdaterForApplication();
auto tile = ref new TileNotification(xml);
update->Update(tile);
```

Here's the resulting wide tile:



The preceding code updates the wide tile only, leaving the standard tile unchanged. To change the standard tile as well, we can add another `<binding>` element to the `<visual>` element, with the appropriate XML for the required standard tile. This will make both changes.



Enabling cycle updates

One interesting feature of live tiles is the ability to cycle up to five tile updates, working with the last five by default. The following code will enable tile cycling:

```
auto update = TileUpdateManager::CreateTileUpdaterForApplication();
update->EnableNotificationQueue(true);
```

If a particular tile should be replaced (instead of the first update being dropped), a tile may be tagged with a unique value, using the `TileNotification::Tag` property, identifying the exact tile to replace.

Tile expiration

A tile can be set to expire at some point in the future, by setting the `TileNotification::ExpirationTime` property. When the time comes, the tile reverts to its default state.

Badge updates

A **badge** is a small notification symbol, located at the lower-right part of a tile. It can be a number in the range 1 to 99, or one of a set of predefined glyphs. It's typically used to show a status, such as network connectivity (if applicable for the app) or the number of pending messages (in a messaging app).

Updating a badge is very similar to updating a tile—it's based on an XML string that contains a single element (<badge>), manipulated to get the desired result. Here's the required code that updates a badge with a numeric value:

```
auto xml = BadgeUpdateManager::GetTemplateContent(
    BadgeTemplateType::BadgeNumber);
auto element = (XmlElement^)xml->SelectSingleNode("/badge");
element->SetAttribute("value", (++count).ToString());

auto badge = ref new BadgeNotification(xml);
BadgeUpdateManager::CreateBadgeUpdaterForApplication()
    ->Update(badge);
```

The variable `count` is used as the numeric value.

Creating secondary tiles

The application tile (primary tile) can be accompanied by secondary tiles. These typically represent deep links into the application. For example, a weather app may use secondary tiles for extra locations where weather is important to update or a Store app can use secondary tiles as links to specific products.

In any case, the user is the only entity that can allow secondary tiles to actually be pinned to the Start Screen or unpinned from the Start Screen. Usually, some UI within the app allows the user to pin a secondary tile, but this can only happen if the user provides consent—otherwise the tile is not pinned.

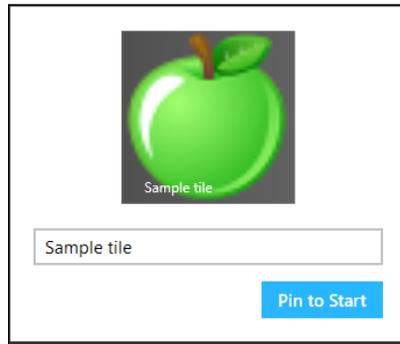
The following code snippet creates a secondary tile and asks the user if he/she wants it pinned to the Start Screen:

```
using namespace Windows::UI::StartScreen;
auto tile = ref new SecondaryTile("123", "Sample tile",
    "This is a sample tile", "123",
    TileOptions::ShowNameOnLogo, ref new Uri(
        "ms-appx:///assets/apple.png"));
create_task(tile->RequestCreateAsync()).then([](bool ok) {
    // do more stuff
});
```

The `SecondaryTile` constructor used by the preceding code accepts the following arguments in order (they can also be set using properties):

- A unique tile ID that can later be used to identify the tile (for unpinning purposes, for example)
- A short name (required) that will show in the system-provided consent dialog
- A display name (recommended)
- Tile activation arguments that help in determining what to do if the application is invoked through the secondary tile (more on that in a moment)
- Logo URI

Calling `SecondaryTile::RequestCreateAsync` presents a standard system dialog (based on the tile's creation arguments), asking the user's permission to actually create and pin the tile:



A secondary tile can be retrieved given its unique ID, by using a `SecondaryTile` constructor that accepts an ID only. Other options include calling the static `SecondaryTile::FindAllAsync` to get a list of all secondary tiles created by the app.

The `SecondaryTile::RequestDeleteAsync` method shows a system dialog, requesting the user to consent to a tile deletion.

Updating a secondary tile is done in much the same way as it is with the primary tile (tile and badge). The only difference is with the updater, created with `TileUpdateManager::CreateTileUpdaterForSecondaryTile` (for tile updates) and `BadgeUpdateManager::CreateBadgeUpdaterForSecondaryTile` (for badge updates).

Activating a secondary tile

When a secondary tile is tapped or clicked, the application launches as usual. Since a secondary tile is supposed to provide a shortcut to a specific location within the app, this scenario must be recognized and handled in the `Application::OnLanuched` override. Here's an example code that looks for arguments passed at launch time:

```
if(args->Arguments != nullptr) {
    // assume arguments are from secondary tiles only
    rootFrame->Navigate(TypeName(DeepPage::typeid),
        args->Arguments);
}
```

The code assumes that `DeepPage.xaml` is the relevant page to show, in case a secondary tile activation is detected.

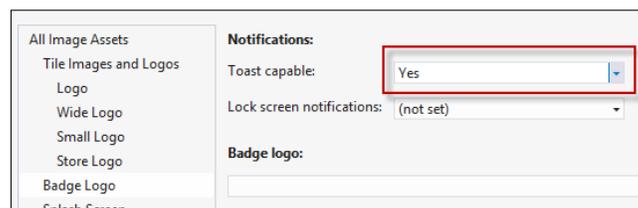
Using toast notifications

Toasts are small pop-up windows that show important information pertaining to an application that may or may not be running at the time. It appears at the top-right corner of the screen—the user can tap (or click) on it to run or switch to the application, or the user can close (dismiss) the toast as being unimportant now, or if the user is not in front of the computer right now, the toast will be gone after a few seconds, making the user miss the toast altogether.

Toasts are somewhat intrusive, as they pop up regardless of the currently executing application (the current app can be the classic desktop or even the lock screen). This means toasts should be used sparingly, where they really make sense. A typical usage is to notify the user for an incoming message in a chat app or a new e-mail.

Toast notifications can be turned off on an app by app basis, by selecting the **Settings** charm and then selecting **Permissions**. Toasts can also be disabled globally, by going to the **Windows PC** settings and selecting **Notifications**.

To make toast notifications work, the app should declare in its manifest that it's toast capable (the **Application UI** tab in the manifest view in Visual Studio):



Raising a toast notification is somewhat similar to tiles. First, we select a predefined template using the `ToastTemplateType` enumeration and then build an appropriate XML based on that template (the content is available through the `ToastNotificationManager::GetTemplateContent` method). Next, we create a `ToastNotification` object, passing the final XML. Finally, we call `ToastNotificationManager::CreateToastNotifier()->Show`, passing the `ToastNotification` object.

Toast options

Toasts can be scheduled to a future point in time by using the `ScheduledToastNotification` class instead of `ToastNotification`. The second argument to the constructor is a `DateTime` value indicating when the toast should be raised. The `Show` method must be replaced with `AddToSchedule` for this to compile and work correctly.

A second constructor to `ScheduledToastNotification` provides a way to show a recurring toast, with a time interval between pop ups (between 1 minute and 60 minutes) and a number of times to show the toast (1 to 5).

Toasts can be standard (showing for 7 seconds) or long (showing for 25 seconds). Long toasts are appropriate when there is a human on the other side of the toast, such as an incoming call. To set it up, a `duration` attribute must be set to `long` within the toast XML.

Toasts play a default sound effect when shown. This effect can be changed to one of the set of predefined sounds available in Windows for this purpose. Again, this is accomplished by adding an `audio` element, with an `src` attribute set to one of the predefined sound strings (check the documentation for a complete list).

Push notifications

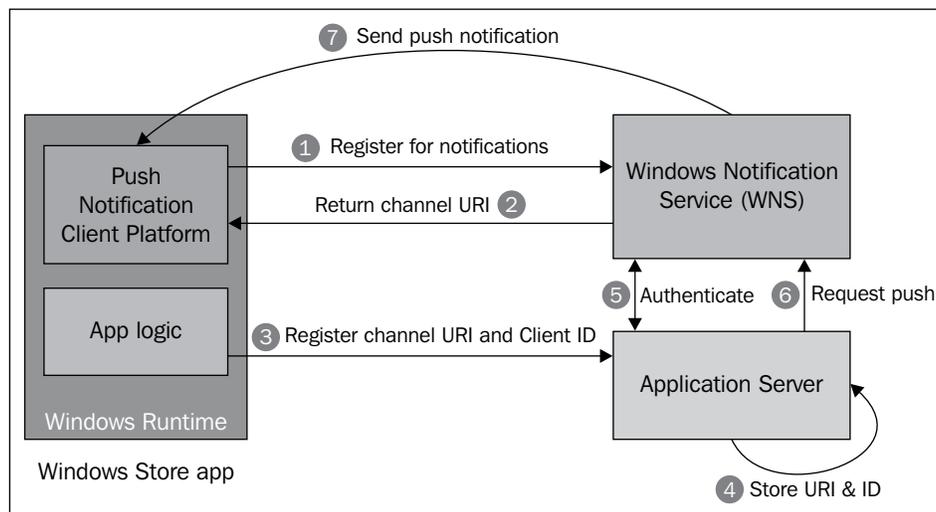
As we've seen, an application can set its tile (and optional secondary tiles) in any way that's reasonable; it can even update the tiles if new information is received. But what happens if the application is suspended? How can it update its tile? Worse, the application may not be running at all. How would its tile get updated? Imagine a news app that may want its tile to reflect recent news.

One way to do this is with push notifications. As the name suggests, the notifications are pushed by a server onto the device that may or may not be running the app at the time. This is in contrast to a pull model, where some part of the app polls some server for new information. Push notifications are energy efficient and don't require the app to do anything special (apart from first registering for notifications as we'll see in a moment) to get the notifications.

Push notification architecture

Push notifications involve several participants, the application being just one of them. The push notification itself is sent from a Microsoft-provided service, the **Windows Notification Service (WNS)** hosted on Windows Azure. The other major entity is an application server that has the logic, or is controlled appropriately, to actually initiate push notifications. In the news app example, this would be a server that receives news updates and then spreads them around using push notifications to all registered client apps.

The push notifications architecture is summarized in the following diagram:



The basic steps to set up push notifications are as follows:

1. The Windows Store application must register to receive notifications. It uses a WinRT API to call WNS and request a unique channel URI that identifies this app (technically the primary tile) for this user on this device.
2. WNS returns a unique channel URI to the application.
3. The app needs to pass the unique channel URI and some unique client identifier to the application server. A unique client ID is usually required because the channel URI can expire and will have to be renewed. The client ID remains the identity as far as the app server is concerned.
4. The application server stores a list of all registered clients with their channel URIs. Later, when a notification needs to be sent, it will loop over the list and send the notifications.

5. The application server needs to authenticate with WNS and get an authentication token in return, to be used as part of the push notification payload. This is a one-time operation, but may need to repeat as the token may expire in the future.
6. Finally, when the application server logic decides to send a push notification (or is instructed by some external management app), it sends the notification as an HTTP POST request.
7. WNS receives the request and does the actual push notification to the client device.

A push notification can change a live tile (the primary tile or a secondary tile), change the badge, or make a toast notification appear. It can even send raw, application-specific notifications that can run a background task registered for the app (background tasks will be discussed later in this chapter).

In the next section, we'll see examples for implementing the preceding steps to get push notifications up and running.

Building a push notification application

The first step for receiving push notifications is to get a unique URI from WNS. This is a fairly simple operation, involving a single method call:

```
create_task(PushNotificationChannelManager::
    CreatePushNotificationChannelForApplicationAsync()).then(
    [this](PushNotificationChannel^ channel) {
        _channel = channel;
```

The call to `returns` a `PushNotificationChannel` object that is stored for later use in the `_channel` member variable. These types reside in the `Windows::Networking::PushNotifications` namespace.

The next step is to register this URI with the application server, so let's take a look at that server first.

The application server

The application server can be built with any server-side technology, within the Microsoft stack or outside it. A typical server will expose some kind of service that clients can connect to and register their unique URIs for push notification (and perhaps other) purposes.

As an example, we'll build a WCF service hosted in IIS that will expose an appropriate operation for this purpose. The example assumes that the server manages movie information and wants to notify registered clients that a new movie is available. The WCF service interface would be as follows:

```
[DataContract (Namespace="")]
public class ClientInfo {
    [DataMember]
    public string Uri { get; set; }
    [DataMember]
    public string ClientID { get; set; }
}

[ServiceContract]
public interface IMovieService {
    [OperationContract, WebInvoke (UriTemplate="add")]
    void AddNewMovie (Movie movie);

    [OperationContract, WebInvoke (UriTemplate="register")]
    void RegisterForPushNotification (ClientInfo info);
}
```

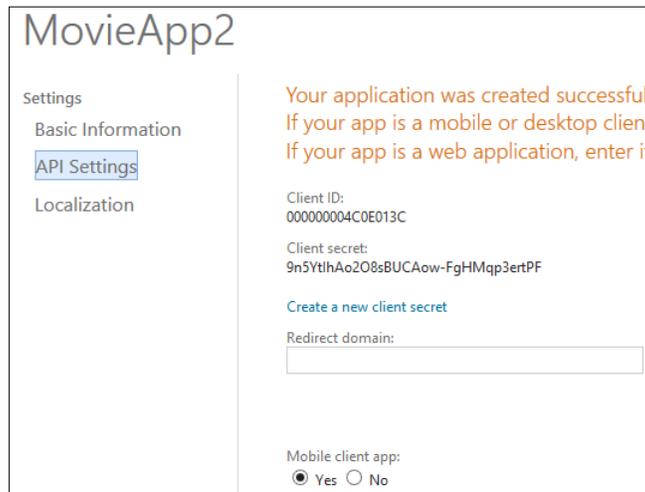
The `IMovieService` has two operations (modelled as methods):

- `RegisterForPushNotification` is used to enlist an interested client as a target for push notifications. It passes a `ClientInfo` object that has the unique channel URI (obtained from the previous step) and some unique client ID.
- The `AddNewMovie` operation will be called later by some controller application to indicate that a new movie is available and consequently to invoke the push operation (we'll look at that in a moment).



WCF (Windows Communication Foundation) is a .NET-based technology for writing services and service clients and is beyond the scope of this book, as it has no direct relation to Windows 8 Store apps. WCF will be used for the server-side code, as it's fairly well-known and easy enough to use, at least for these purposes; the code, naturally, is written in C#.

The first thing such a service must do is obtain an authentication token from WNS, so it can actually perform push notifications. The first step to achieve this is to register the Windows 8 application and obtain two pieces of information: a security ID and a secret key. With these information in hand, we can contact WNS and request a token. To register the app we have to browse to <https://manage.dev.live.com>, log in with our Microsoft ID (formerly Live ID), click on **Create application**, enter some unique application name, and then click on **Yes**:



The result is a **security ID (SID)** and a secret key:



We'll copy these and store them as simple constant or static fields in the service class implementation. The application name itself must be copied to the application manifest (in the **Packaging** tab), with some other details outlined on the web page. To make some of this easier, right-click on the project, select **Store**, and then select **Associate App with Store**. This will enter most information to the correct locations:

Package name:	63046ScorpioSoftware.MovieApp			
Package display name:	MovieApp			
Version:	Major:	Minor:	Build:	Revision:
	1	0	0	0
Publisher:	CN=2723A664-E75E-4B08-B983 [REDACTED]			
Publisher display name:	Scorpio Software			
Package family name:	63046ScorpioSoftware.MovieApp [REDACTED]			

The code to obtain the authentication token would be as follows:

```
private static void GetToken() {
    var body = string.Format
        ("grant_type=client_credentials&client_id={0}&client_secret={1}
        &scope=notify.windows.com",
        HttpUtility.UrlEncode(SID), HttpUtility.UrlEncode(Secret));

    var client = new WebClient();
    client.Headers.Add("Content-Type",
        "application/x-www-form-urlencoded");
    string response = client.UploadString(new Uri(AuthUri), body);

    dynamic data = JsonConvert.DeserializeObject(response);
    _token = data.access_token;
}
```

The code is fairly uninteresting. It uses the specific format required for the authentication procedure. The `WebClient` class provides a simple way in .NET to make HTTP calls. The result of the call is a string representation of a JSON object, which is deserialized by the `Newtonsoft.Json.JsonConvert` class. Finally, the `access_token` field is the actual token we need, saved in the static variable `_token`.



`JsonConvert` is part of the free `Json.NET` package, which can easily be installed using Nuget (right-click on the project in Visual Studio, select **Manage Nuget packages...**, search for `Json.Net`, and click on **Install**).

The dynamic C# keyword allows (among other things) untyped access to objects, by late binding to the actual member (if it exists). The compiler is happy to defer type checking to runtime, thus an unrecognized member throws a runtime exception, as opposed to the usual compile time error.

Now that a token is obtained, it can be used to send push notifications.

 The authentication token can actually expire, which can be discovered by examining the response from an actual push notification POST request, looking for the `Token expired` value for the `WWW-Authenticate` header. In that case, just call `GetToken` again to get a new token.

Now that the server is ready, the client app needs to register its unique channel URI with the application service.

Registering for push notifications

Theoretically, this step is easy. Just call the `RegisterForPushNotification` method on the service, pass the required arguments, and you're done. Unfortunately, this is not as easy as we would like in C++.

The app needs to make the correct network call (typically over HTTP) to the service. The simplest HTTP calls are based on REST, so it will be simpler if our service is configured to accept REST over HTTP.

 **REST (Representational State Transfer)** is beyond the scope of this book. For our purposes, it means encoding the information on the HTTP URL as simple strings, with more complex information passed with the request body. This is in contrast to more complicated protocols, such as SOAP.

The WCF service we created is configured to accept REST calls because of the `[WebInvoke]` attributes, setting the URL suffix for each request. This also requires configuring a service host to use the `WebHttpBinding` WCF binding and the `WebHttp` behavior. This is accomplished via the `MovieWorld.svc` file, where the service is declared:

```
<%@ ServiceHost Language="C#" Debug="true"
    Service="MoviesWorld.MovieService"
    CodeBehind="MovieService.svc.cs"
    Factory= "System.ServiceModel.Activation.WebServiceHostFactory" %>
```

The `Factory` attribute is the important (non-default) part.

The next challenge is to make a REST (or any HTTP) call from the C++ client application.

Unfortunately, at the time of writing, there is no easy way to use a WinRT class that can make HTTP calls, something along the lines of `WebClient` and `HttpClient` in .NET. The documentation recommends using the low-level `IXMLHttpRequest2` COM interface for this purpose.

Although it's certainly possible, it's not easy. Fortunately, Microsoft created a C++ wrapper class, `HttpRequest`, which does most of the work for us. I've copied the class into the project almost as it is (made one slight modification), so now it's much easier to make HTTP calls.

 `HttpRequest` is implemented in the `HttpRequest.h` or `HttpRequest.cpp` files, part of the `MovieApp` project, available with the downloadable source for this chapter.

Here's the HTTP request to register the app for push notifications:

```
Web::HttpRequest request;
wstring body = wstring(L"<ClientInfo><ClientID>123</ClientID><Uri>" +
channel->Uri->Data() + L"</Uri></ClientInfo>");

return request.PostAsync(ref new Uri(
    "http://localhost:36595/MovieService.svc/register"),
    L"text/xml", body);
```

The body consists of a `ClientInfo` object, serialized as XML with the `Uri` element having the unique channel URI obtained in the first step. The client ID here is encoded as a constant 123 as an example; in a real app, this would be generated as something unique for the app on this machine for this user. The strange port number is the local port IIS that is listening on where my service is hosted. Again, in a real app, this would be over port 80 (regular HTTP) or 443 (HTTPS).

 An alternative way to issue an HTTP request is by using the C++ REST SDK (Casablanca) library; this was published to CodePlex at the time these lines were written. This library allows (among other things) working with HTTP requests in an easy and customizable way, somewhat similar to the .NET `HttpClient` class. The SDK can be found at <http://casablanca.codeplex.com/>.

Issuing the push notification

When the application server gets a call to its `AddNewMovie` method (as part of some logic in the server itself, or because some management application invoked the operation), it needs to send push notifications to all registered clients:

```
public void AddNewMovie(Movie movie) {
    _movies.Add(movie);
    foreach(var uri in _pushData.Values) {
        // push new movie to registered clients
        SendPushTileNotification(uri, movie);
    }
}
```

The `SendPushTileNotification` method looks like the following:

```
private async Task SendPushTileNotification(string uri, Movie movie) {
    string body =
        "<tile>" +
        "<visual>" +
        "<binding template=\"TileSquareText01\">" +
        "<text id=\"1\">" + movie.Year + "</text>" +
        "<text id=\"2\">" + movie.Name + "</text>" +
        "</binding>" +
        "</visual>" +
        "</tile>";

    var client = new HttpClient();
    var content = new StringContent(body);
    content.Headers.ContentType = new MediaTypeHeaderValue(
        "text/xml");
    client.DefaultRequestHeaders.Add("X-WNS-Type", "wns/tile");
    client.DefaultRequestHeaders.Add("Authorization",
        string.Format("Bearer {0}", _token));
    await client.PostAsync(uri, content);
}
```

The body of the message is a regular XML tile. In this case, it includes two lines of text:

- The first contains the year the movie was published
- The second includes the movie name

The notification is an HTTP `POST` request, based on the unique channel URI, with some specific headers that must be set correctly. Also, notice the use of the authentication token obtained earlier from WNS.

 The `await` C# keyword allows waiting for an asynchronous operation without blocking the calling thread. This is similar to our use of the `task<T>` class with the `then` method. C# still looks easier to use.

The type of notification can change to `toast` or `badge` by changing the `X-WNS-Type` header to `wns/toast` and `wns/badge` respectively. The body, naturally, must be modified accordingly.

 The sample code for this chapter includes a project named `MovieManager` that is used to add new movies that generate push notifications.

Here's the original application tile (left) and the tile after a push notification of a new movie:



 The recently available Windows Azure Mobile Services provide easier ways of maintaining push notification clients and sending the notifications themselves (and other useful capabilities). Mobile Services are outside the scope of this book, but more information can be found at <http://www.windowsazure.com/en-us/develop/mobile/>.

Push notifications for secondary tiles

Secondary tiles can also be targets for push notifications. The main difference is the way a unique channel URI is obtained by the client app. It uses `CreatePushNotificationChannelForSecondaryTileAsync` with the tile ID, instead of `CreatePushNotificationChannelForApplicationAsync` (both static methods of the `PushNotificationChannelManager` class).

Background tasks

Windows Store applications may be suspended when the user switches to another app. The app may still want some work to happen even while the app is suspended and even terminated. This is the job of background tasks.

What is a task?

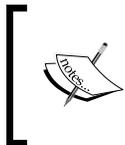
A **task** is just a class that implements the `Windows::ApplicationModel::Background::IBackgroundTask` interface, having just one method, `Run`. This class must be placed in a separate project from the main application, in a **Windows Runtime Component** type project. This is essential, as tasks run in a separate process, and thus cannot be bound with the main app (so they are not suspended if the main app is suspended).

The main application needs to reference the project where the task(s) are located, and indicate via its manifest that these are indeed its tasks.



An application can have any number of tasks implemented in one or more Windows Runtime Component projects.

A task must have exactly one trigger, specifying what triggers the task execution. A task can also have zero or more conditions that must be specified for the trigger to be usable.



Only one trigger can be associated with a task, but it's possible to register another task that runs the same code, but configured with a different trigger. This effectively creates a task that can run with multiple triggers.

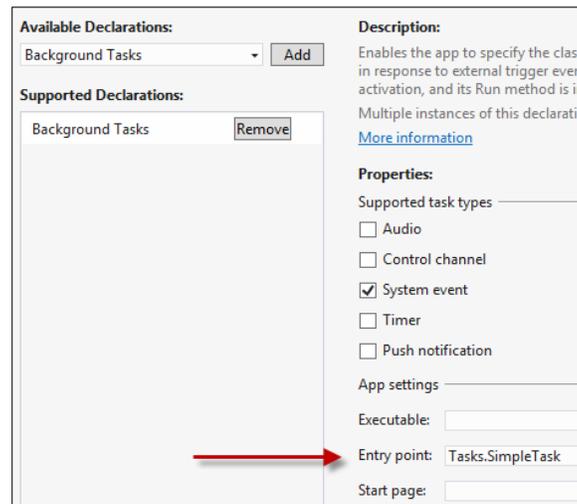
Creating and registering a task

The first step for creating a task is to create a Windows Runtime Component project with a class that implements the `IBackgroundTask` interface, as follows:

```
namespace Tasks {
    using namespace Windows::ApplicationModel::Background;

    [Windows::Foundation::Metadata::WebHostHidden]
    public ref class SimpleTask sealed : IBackgroundTask {
    public:
        virtual void Run(IBackgroundTaskInstance^ taskInstance);
    };
}
```

Next, we need to add a reference to the task component from the main application project. The last prerequisite is to add the task to the main application's manifest. This is done in the **Declarations** tab:



A background task declaration is selected with the appropriate task type, which roughly means trigger type, as will be used later for actual registration in code. We'll discuss triggers in a moment.

The **Entry point** field must be set to the full class name implementing the background task (`Tasks::SimpleTask` in this case).

The result is an entry in the manifest XML in the `<extensions>` section:

```
<Extensions>
  <Extension Category="windows.backgroundTasks"
    EntryPoint="Tasks.SimpleTask">
    <BackgroundTasks>
      <Task Type="systemEvent" />
    </BackgroundTasks>
  </Extension>
</Extensions>
```

The main application must do the actual task registration on startup and it must do so only once. Registering a task with an existing task name throws an exception.

Registration involves the `BackgroundTaskBuilder` class and a trigger class with optional conditions. Here's a piece of code registering the `SimpleTask` defined in the preceding code snippet to execute whenever an Internet connection is available:

```
auto trigger = ref new SystemTrigger(  
    SystemTriggerType::InternetAvailable, false);  
auto condition = ref new SystemCondition(  
    SystemConditionType::InternetAvailable);  
  
auto builder = ref new BackgroundTaskBuilder();  
builder->Name = "Simple";  
builder->TaskEntryPoint = "Tasks.SimpleTask";  
builder->SetTrigger(trigger);  
builder->AddCondition(condition);  
auto task = builder->Register();
```

A trigger must be selected for a task; in this case, it's the generic `SystemTrigger`, based on a `SystemTriggerType` enumeration, which has values such as `InternetAvailable`, `UserPresent`, `UserAway`, `SmsReceived`, `TimeZoneChange`, and more.

Conditions are optional; `SystemCondition` is currently the only one available, but it's generic as well, using the `SystemConditionType` enumeration. Values include `InternetAvailable`, `InternetUnavailable`, `UserPresent`, `UserNotPresent`, and others.

`BackgroundTaskBuilder` holds the trigger and condition information, along with the task name and entry point. Then a call to `Register` does the actual registration with the system (returning a `BackgroundTaskRegistration` object).

Implementing a task

Let's use an application that allows the user to enter data, and that data is saved in the local folder. If the user is connected to the Internet, a background task should do some processing on the resulting files, such as uploading them to a server, doing some calculations, and so on. Eventually, the task will delete the files after being processed.

Here's some simple code that the main app uses to save some data to files:

```
auto root = ApplicationData::Current->LocalFolder;  
  
create_task(root->CreateFolderAsync("Movies",  
    CreationCollisionOption::OpenIfExists)).then([](  
    StorageFolder^ folder) {
```

```

        return folder->CreateFileAsync("movie",
            CreationCollisionOption::GenerateUniqueName);
    }).then([](StorageFile^ file) {
        // build data to write
        return file->OpenAsync(FileAccessMode::ReadWrite);
    }).then([this](IRandomAccessStream^ stm) {
        wstring data = wstring(L"<Movie><Name>") +
            _movieName->Text->Data() + L"</Name><Year>" +
            _year->Text->Data() + L"</Year></Movie>";
        auto writer = ref new DataWriter(stm);
        writer->WriteString(ref new String(data.c_str()));
        return writer->StoreAsync();
    }).then([this](size_t) {
        _movieName->Text = "";
        _year->Text = "";
    });
}

```

The files are saved to a subfolder named `Movies` under the `LocalFolder`.

Tasks share the application's local folder, effectively making it a communication mechanism. Here's the task's `Run` method implementation:

```

void SimpleTask::Run(IBackgroundTaskInstance^ taskInstance) {
    auto root = ApplicationData::Current->LocalFolder;
    Platform::Agile<BackgroundTaskDeferral^> deferral(
        taskInstance->GetDeferral());
    create_task(root->GetFolderAsync("Movies")).
        then([](StorageFolder^ folder) {
            return folder->GetFilesAsync(
                CommonFileQuery::DefaultQuery);
        }).then([](IVectorView<StorageFile^>^ files) {
            int count = files->Size;
            for(int i = 0; i < count; i++) {
                auto file = files->GetAt(i);
                // process each file...
                file->DeleteAsync();
            }
        }).then([deferral](task<void> t) {
            t.get();
            // error handling omitted
            deferral->Complete();
        });
}
}

```

The task begins by obtaining the `LocalFolder` location. Before the actual process begins, it obtains a *deferral* object by calling `IBackgroundTaskInstance::GetDeferral`. Why?

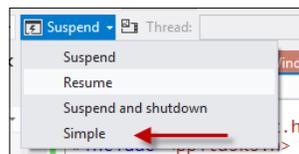
A task is normally considered done when the `Run` method terminates. If, however, the implementation invokes any asynchronous operations, the method will return to its caller sooner, making the task complete. Obtaining a deferral effectively holds off task completion until that time when a call to `BackgroundTaskDeferral::Complete` is made.

Next, comes the actual file processing. All the files in the `Movies` folder are enumerated using `StorageFolder::GetFilesAsync` and after each dummy processing, the file is deleted. Only then the entire task is done, and the deferral's `Complete` method can be invoked to indicate that the task is complete.

Task debugging

The available triggers and conditions are not easily satisfied in a development environment. We don't want to disconnect and reconnect the Internet; nor do we want to wait 15 minutes until a task with a `TimeTrigger` is executed.

Visual Studio provides a way to invoke a task at any point for debugging purposes. This functionality is located in the same toolbar button as suspension and resumption:



If we place a breakpoint within the task's `Run` method, we can debug the task as usual. The `BackgroundTaskHost.exe` is the process instance that is hosting the application's tasks. This fact can be viewed in the debugger **Process** toolbar button or in Windows Task Manager.

Task progress and cancellation

A background task can run when the main app is running as well. One thing that's possible to do is indicate its progress from the background task's perspective. This is done using the `IBackgroundTaskInstance::Progress` property. If the main app is not running, nobody cares. If it is running, it can register for the `Progress` event (part of `IBackgroundTaskRegistration` that's returned upon successful registration of the task) and update the UI based on that progress.

When the task finishes, the `IBackgroundTaskRegistration::Completed` event fires so that the main app knows the task is done. If the main app is currently suspended, it will be notified when it resumes.

In some situations, Windows may cancel a running task. The `IBackgroundTaskInstance` exposes a `Canceled` event that a task can register for. If the task is cancelled, it must return within 5 seconds, or it will be terminated. The `Canceled` event provides a `BackgroundTaskCancellationReason` that specifies why the task was cancelled. Examples include `ServiceUpdate` (the main app is being updated) and `LoggingOff` (the user is logging off the system).

For example, we can use a Win32 event to notify our task that cancellation has been requested. First, we create the event object and register for the `Canceled` event:

```
void SimpleTask::Run(IBackgroundTaskInstance^ taskInstance) {
    if(_hCancelEvent == nullptr) {
        _hCancelEvent = ::CreateEventEx(nullptr, nullptr, 0,
            EVENT_ALL_ACCESS);
        taskInstance->Canceled +=
            ref new BackgroundTaskCanceledEventHandler(
                this, &SimpleTask::OnCancelled);
    }
}
```

`_hCancelEvent` is a `HANDLE` type, created with `CreateEventEx`. Then the `Canceled` event is associated with a private `OnCancelled` method.



The classic Win32 API `CreateEvent` can't be used, as it's illegal in WinRT. `CreateEventEx` was introduced in Windows Vista, and can be considered as a superset of `CreateEvent`.

If the task is being cancelled, we set the Win32 event:

```
void SimpleTask::OnCancelled(IBackgroundTaskInstance^ instance,
    BackgroundTaskCancellationReason reason) {
    ::SetEvent(_hCancelEvent);
}
```

The task main processing code should examine the Win32 event and bail out as quickly as possible, if it's signaled:

```
for(int i = 0; i < count; i++) {
    auto file = files->GetAt(i);
    if(::WaitForSingleObjectEx(_hCancelEvent, 0, FALSE) ==
        WAIT_OBJECT_0)
        // cancelled
        break;
}
```

```
        // process each file...
        file->DeleteAsync();
    }
}
```

Calling `WaitForSingleObject` with a zero timeout just examines the state of the event. If it's signaled, the return value is `WAIT_OBJECT_0` (otherwise, the return value is `WAIT_TIMEOUT`).

Playing background audio

Some applications play audio, and users expect the audio to keep playing even if the user switches to another app; for example, a music playing app is expected to continue playing until told to stop by the user. A voice over IP app (such as Skype) is expected to maintain the other party's audio even if the user switches to another app. This is where the background audio task comes in.

Playing audio

Audio playing can be easily achieved by using the `MediaElement` control (which can also play video). It should be placed somewhere in XAML so that it's part of a visual tree, although it has no visible parts when playing audio.

Actual playing is done by setting a URI to play using the `Source` property (or for files obtained from a `FileOpenPicker` by calling the `SetSource` method). Playing starts immediately unless the `AutoPlay` property is set to `false`.

Controlling playback is done with the `Play`, `Pause`, and `Stop` methods of `MediaElement`. Here's an example of an audio file obtained from a `FileOpenPicker`. First, some basic UI for the `MediaElement` and playback control:

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
  <MediaElement x:Name="_media" />
  <Button Content="Select Audio File..." FontSize="30" Margin="10"
    Click="OnSelectMediaFile" />
  <StackPanel Orientation="Horizontal" Grid.Row="2"
    Margin="10,30">
    <Button Content="Play" FontSize="40" Click="OnPlay"
      Margin="8"/>
    <Button Content="Pause" FontSize="40" Click="OnPause"
```

```

        Margin="8"/>
        <Button Content="Stop" FontSize="40" Click="OnStop"
        Margin="8"/>
    </StackPanel>
</Grid>

```

The `OnSelectedMediaFile` is implemented as follows:

```

auto picker = ref new FileOpenPicker();
picker->FileTypeFilter->Append(".mp3");
create_task(picker->PickSingleFileAsync()).
    then([this](StorageFile^ file) {
        if(file == nullptr)
            throw ref new OperationCanceledException();
        return file->OpenReadAsync();
    }).then([this](IRandomAccessStreamWithContentType^ stm) {
    _media->SetSource(stm, "");
    }).then([](task<void> t) {
    try {
        t.get();
    }
    catch(Exception^ ex) {
    }
    });

```

Most of the code should be familiar by now. The filter of `FileOpenPicker` is set for MP3 files, and once selected, the call to `MediaElement::SetSource` readies the audio stream for playback.

Playing the stream is just a matter of calling `MediaElement::Play` at the `Play` button's `Click` handler:

```

void MainPage::OnPlay(Object^ sender, RoutedEventArgs^ e) {
    _media->Play();
}

```

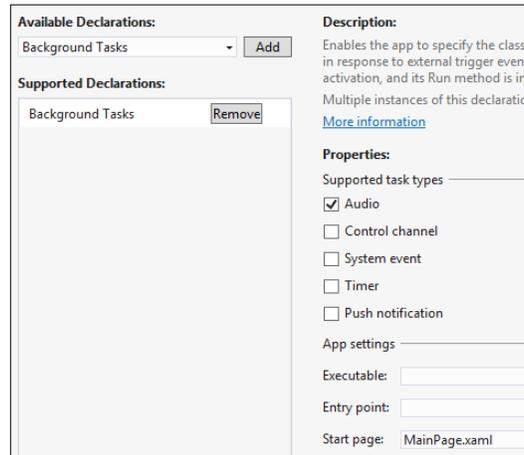
`OnPause` and `OnStop` are similarly implemented by calling `MediaElement::Pause` and `MediaElement::Stop` respectively.

Running the application now allows for selecting an MP3 file and playing it. Switching to another app, however, immediately stops playback.

Maintaining background audio

Making the application continue to play automatically in the background requires several steps.

First, the app manifest must be modified to indicate background audio is needed; this is done by adding a **Background Task** declaration and setting the **Audio** checkbox:



The other required step is setting the **Start** page, as shown in the preceding screenshot. The next steps require adding some code:

- The `MediaElement::AudioCategory` property must be set to `AudioCategory::BackgroundCapableMedia` (for general background playback) or `AudioCategory::Communications` (for peer to peer communications, such as a chat).
- Register for static events of the `Windows::Media::MediaControl` class that indicate changes that may result from other applications using audio playback.

First, we'll change the `AudioCategory` property of the `MediaElement`:

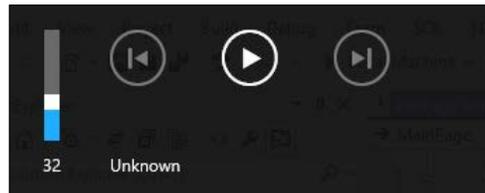
```
<MediaElement x:Name="_media" AudioCategory="BackgroundCapableMedia"/>
```

The result of this setting makes the app never enter suspension.

Next, we'll register for the required events:

```
MediaControl::PlayPressed += ref new EventHandler<Object^>(
    this, &MainPage::OnPlayPressed);
MediaControl::PausePressed += ref new EventHandler<Object^>(
    this, &MainPage::OnPausePressed);
MediaControl::StopPressed += ref new EventHandler<Object^>(
    this, &MainPage::OnStopPressed);
MediaControl::PlayPauseTogglePressed +=
    ref new EventHandler<Object^>(
    this, &MainPage::OnPlayPauseTogglePressed);
```

These events are fired by the system when the playback state changes due to a system-provided media control that is accessible with some keyboards and perhaps other gestures:



Reacting to these events is not difficult. Here's the code for the `PlayPressed` and `PlayPauseTogglePressed` events:

```
void MainPage::OnPlayPressed(Object^ sender, Object^ e) {
    Dispatcher->RunAsync(CoreDispatcherPriority::Normal,
        ref new DispatchedHandler([this]() {
            MediaControl::IsPlaying = true;
            _media->Play();
        }));
}

void MainPage::OnPlayPauseTogglePressed(Object^ sender, Object^ e) {
    Dispatcher->RunAsync(CoreDispatcherPriority::Normal,
        ref new DispatchedHandler([this]() {
            if(_media->CurrentState == MediaElementState::Playing) {
                MediaControl::IsPlaying = false;
                _media->Pause();
            }
            else {
                MediaControl::IsPlaying = true;
                _media->Play();
            }
        }));
}
```

The notifications are handled as commands for the app to play or pause playback as needed; correct implementation ensures consistent behavior across all audio playbacks on the system.


 Note that the events arrive on a thread pool thread, and since the `MediaElement` needs to be touched, the call must be marshalled to the UI thread using the `CoreDispatcher::RunAsync` method.

Handling the `PausePressed` and `StopPressed` events is similar.

Other events from the `MediaControl` class can be handled if appropriate, such as `NextTrackPressed` and `PreviousTrackPressed`.

Sound-level notifications

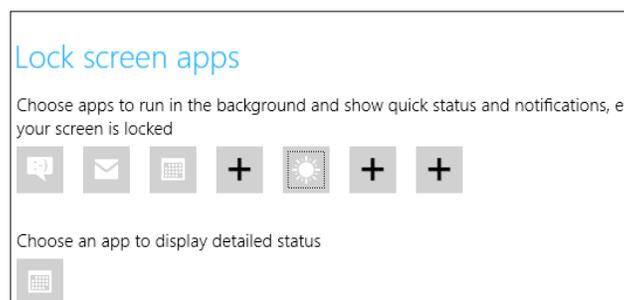
If a background app is playing audio and another foreground app starts playing audio, the system sends a `MediaControl::SoundLevelChanged` event to the background app. This event indicates what happened to the background app's sound by looking at the `MediaControl::SoundLevel` property. Possible values are:

- `Muted`: The app's sound has been muted, so the app should pause its playback. This usually means a foreground app is playing audio.
- `Low`: The app's sound level has gone low. This indicates a VoIP call came in, decreasing the app's sound level. The app may want to pause playback until another `SoundLevelChanged` event fires that indicates full volume.
- `Full`: The app's sound is at full volume. If the app was playing audio and had to pause it, now it's time to resume playing.

Registering for this event is optional, but can enhance the user experience and it indicates a well-behaving app.

Lock screen apps

The lock screen (before a user logs in, or when the device is locked) allows for up to seven applications to be on it—these can have an icon (and even a text message); these apps are known as lock screen apps. The seven possible apps can be configured via control panel | **PC Settings** | **Personalize**:



Lock screen apps are considered more important by the system (because they are more important to the user), and consequently have some capabilities that cannot be obtained by non-lock screen apps. For example, some trigger types only work with lock screen apps:

- `TimeTrigger` can be used to do periodic work (minimum is 15 minutes).
- `PushNotificationTrigger` can be used to get a raw push notification that causes the task to execute (raw means any string, unrelated to tile, toast, or badge).
- `ControlChannelTrigger` can be used to keep a live connection to a remote server, even when the application is suspended; useful for Instant Messaging or video chat applications.

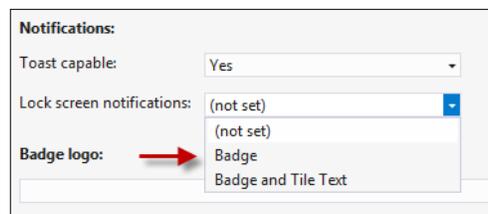


This last two trigger types make the task actually run inside the app process and not in the standard task hosting process.

There is actually another time-related trigger, `MaintenanceTrigger`. This one does not require a lock screen app, but it only functions if the device is connected to AC power. If it's disconnected, the task will not run. If it's disconnected while the task executes, the task will be cancelled.

To make an application lock screen capable, a few things need to be set up:

- A wide logo must be set for the app.
- A badge logo must be set up as well; this is the default image that appears on the lock screen for the app.
- At least one background task must be declared (using the manifest's **Declarations** tab in Visual Studio) that uses a push notification trigger, a time trigger or a control channel trigger.
- The **Lock screen notifications** option must be set to **Badge** or **Badge with Tile Text**:



Requesting to set a lock screen app

Although the user can go to the app's settings and set it as a locked screen app, or go to the Windows personalization section and do the same. It's easier if the app asks the user (via a system-provided dialog), if it's ok for the app to become a lock screen app. This is accomplished with the `BackgroundExecutionManager::RequestAccessAsync` static method call. The result of the asynchronous call specifies whether the user accepted the suggestion or not (`BackgroundAccessStatus` enumeration).



If the user denies, the dialog won't pop up when the application is run again; it will pop up again if the app is reinstalled.

Other common operations for lock screen apps

Lock screen apps typically perform the following operations during the app's lifetime:

- Sending badge updates (shown on the lock screen)
- Sending tile updates
- Receiving and processing raw push notifications that are used to execute app-specific logic
- Raising a toast notification (that's shown even on the lock screen)

The exact details of these operations are beyond the scope of this book (although the badge, tile, and toast update mechanisms are similar to those already discussed); more details can be found in the MSDN documentation.

Background tasks limits

When executing, background tasks compete for CPU and network resources with the currently running foreground application, so they cannot run for an arbitrary length of time; the foreground app is the most important. Tasks are subject to the following constraints:

- Lock screen app tasks get 2 seconds of CPU time every 15 minutes (actual running time, not wall clock time).
- Non-lock screen apps receive 1 second of CPU time every 2 hours (again, actual running time).
- Network resources are unlimited when the device is running on AC power. Otherwise, some limitations may be in place depending on energy consumption.

- Tasks configured with `ChannelControlTrigger` or `PushNotificationTrigger` receive some resource guarantees, as they are deemed more important.

Additionally, there is a global pool of CPU and network resources that can be used by any application. This pool is refilled every 15 minutes. This means that even if a task needs more than 1 second to run (non-lock screen app), it may get the extra CPU time, provided the pool is not exhausted. Naturally, the task can't rely on this pool, as other tasks may have already exhausted it.

Background transfers

It should be clear at this point that a suspended application can't do anything by itself, unless it has some background tasks working on its behalf. One of the operations that an app may need to perform is a lengthy download or upload of files. If the app becomes suspended, the download or upload operations cannot proceed. If the app is terminated, whatever was already downloaded goes away. Clearly, there must be a better way.

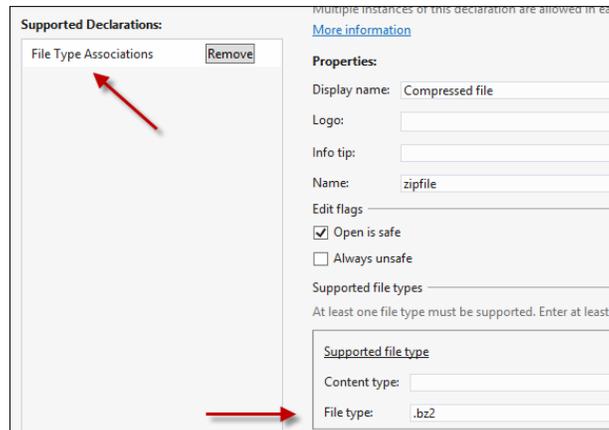
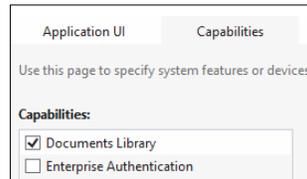
WinRT provides a way to do background transfers (downloads and uploads) even if the app is suspended, by using a separate process to do the actual transfers (`BackgroundTransferHost.exe`). This capability allows the app to make a lengthy transfer without requiring the user to stay with the app for the entire transfer time.

Example – downloading a file

Here's a simple example that starts a download operation targeted to a file in the user's documents location (error handling omitted):

```
wstring filename(_url->Text->Data());
auto index = filename.rfind(L'/');
filename = filename.substr(index + 1);
create_task(
    KnownFolders::DocumentsLibrary->CreateFileAsync(
        ref new String(filename.c_str()),
        CreationCollisionOption::GenerateUniqueName) .then([this](
    StorageFile^ file) {
        auto downloader = ref new BackgroundDownloader();
        auto operation = downloader->CreateDownload(
            ref new Uri(_url->Text), file);
        return operation->StartAsync();
    });
```

The code assumes `_url` is a `TextBox` where the user entered a URL for a file that he/she wants to download. First, the filename is created based on the end phrase of the URL after the last slash. Then, the file is created in the documents folder of the user. Note that to get this capability, it must be declared in the manifest, and for the documents library, at least one file extension must be selected:



[ For the Video, Music, and Pictures libraries, declaring file associations is not needed.]

Next, a `BackgroundDownloader` instance is created, and a call to its `CreateDownload` is made, passing the URL to download and the target file. This call returns a `DownloadOperation` object, without the download actually being started. To start the download, `DownloadOperation::StartAsync` is called.

While the download is away, it's useful to know how it progresses. Here's a revised code that sets up progress reporting (the difference is from the `StartAsync` call):

```
auto async = operation->StartAsync();
async->Progress =
    ref new AsyncOperationProgressHandler<DownloadOperation^,
        DownloadOperation^>(this, &MainPage::OnDownloadProgress);
return async;
```

In this case, we're actually looking at the result of `StartAsync`, returning an object implementing `IAsyncOperationWithProgress<DownloadOperation, DownloadOperation>` and we set up the `Progress` property with an appropriate delegate:

```
void
MainPage::OnDownloadProgress (IAsyncOperationWithProgress
<DownloadOperation^, DownloadOperation^>^ operation,
DownloadOperation^ download) {
    auto progress = download->Progress;
    Dispatcher->RunAsync (CoreDispatcherPriority::Normal,
        ref new DispatchedHandler ([progress, this] () {
            _progress->Maximum =
double (progress.TotalBytesToReceive >> 10);
            _progress->Value = double (progress.BytesReceived >> 10);
            _status->Text = progress.Status.ToString();
        }));
}
```

The `DownloadOperation::Progress` property returns a simple structure (`BackgroundDownloadProgress`) with fields such as `TotalBytesToReceive`, `BytesReceived`, and `Status` (`Running`, `Completed`, `Cancelled`, and so on). The preceding code uses these values for a `ProgressBar` control (`_progress`) and `TextBlock` (`_status`).

Note that the notification is not arriving on the UI thread, so any updates to the UI must be marshalled to the UI thread by using the `Page::Dispatcher` property (of the type `Windows::UI::Core::CoreDispatcher`) with a call to `RunAsync` that accepts a delegate that is guaranteed to execute on the UI thread.

If the app is terminated, the transfer stops as well, but the data downloaded so far is not lost. When the app is started again, its job is to look up all its incomplete transfers and resume them. This can be done by calling the static `BackgroundDownloader::GetCurrentDownloadsAsync`, getting back a list of those incomplete downloads, and then attaching to each one (for example, progress report) and, of course, resuming the downloads.



You can find a complete example of this in the Background Transfer sample available at <http://code.msdn.microsoft.com/windowsapps/Background-Transfer-Sample-d7833f61>.

Summary

Windows Store applications are unlike desktop applications in many ways. This chapter dealt with the application lifecycle – the app may get suspended and even terminated, all controlled by the OS.

Live tiles, badge updates, and toast notifications are some of the more unique features of Windows Store apps – desktop apps don't have these powers (although desktop apps can create their own toast-like pop ups). Used wisely, these can add a lot of traction to a Store app, luring the user into the app frequently.

Background tasks provide a way around the involuntary suspension/termination scenarios, so that some control is maintained even when the app is not in the foreground. Still, this is pretty restricted, to keep the main app responsive and maintain good battery life. Tasks are an important ingredient in non-trivial apps, and so should be used wisely.

In the next chapter, we'll take a look at ways a Windows Store app can better integrate with Windows and indirectly communicate with other applications by implementing contracts and extensions.

8

Contracts and Extensions

Windows Store applications run in a tight sandbox known as **AppContainer**. This container does not allow applications to communicate directly with other applications on the machine (such as Win32 kernel object handles and shared memory). This makes sense in a way, because an app can't assume anything about the computing environment in which it's installed from the Store, except for the CPU architecture and capabilities that were requested by the app. There's no way to know for sure that an app exists, for instance, and even if there was a way, there's no good way to make sure it can actually talk to this app.

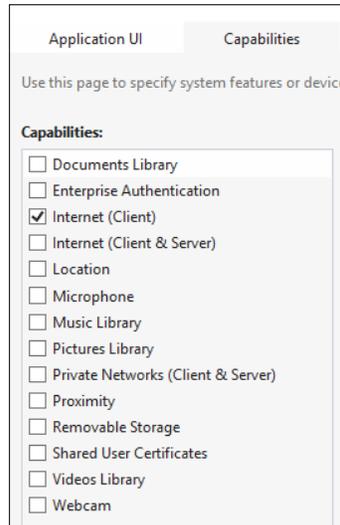
Instead, Windows defines a set of contracts for app to app communication. Such contracts allow applications to implement some functionality without knowing which other app is going to use it. The contracts are well defined, and with the help of the operating system to make the necessary connections, they allow apps to communicate indirectly. We'll examine some of the common contracts in this chapter.

Applications can also provide "plugins" into some of the operating system's provided functionality. These "plugins" are called extensions, and we'll take a look at one of them, the Settings extension.

Capabilities

Windows Store apps cannot directly communicate with other apps, but what about the system itself? What about files, folders, or devices? It turns out that, by default, these are also restricted, and must be given permission by the user at the application install time.

Applications must declare all intended system usage that is defined by Windows, as something the user must agree to. These are **capabilities**, and are part of the application manifest. Visual Studio provides a graphical view of capabilities in its manifest view which we have already used in previous chapters:



The image shows all currently supported capabilities. By default, just one capability is enabled: the ability to make outbound network calls.

Accessing the user's "My" libraries (documents, pictures, videos, and music) must be explicitly requested, otherwise an "access denied" exception will be thrown upon access; the Documents library also demands the app specifies which file types it accepts.

Device access is naturally an issue, represented by capabilities such as **Microphone**, **Webcam**, **Location**, and **Proximity**.

Note, that there is no capability that can grant an application access to Windows system folders, such as `Program Files`, `System32`, and so on; this is simply beyond a Windows Store app – and so it should be. No app should need such high privilege access.

Contracts

Contracts are defined by Windows for app to app communication; it's a kind of agreement between apps, mediated by the operating system, that allows apps to communicate indirectly. Let's look at two common examples of contracts.

Share contract

The **share contract** operates between an app that is a **share source** (has something to share) and a **share target** app (that wants to do something with the shared data). An application can be a share source, a share target, or both.

Sharing is usually initiated by using the Share charm. When activated from a share source app, a list of possible target apps is provided by Windows—all the installed apps that implement the share target contract, and accept at least one of the data types provided by the source. Let's see how to create a share source and a share target.

Share source

Becoming a share source is easier than being a share target. A share source needs to notify Windows of any potential data it can provide. Most of the work required for share resides in the `Windows::ApplicationMode::DataTransfer` namespace.

A share source must register for the `DataTransferManager::DataRequested` event when the application or the main page initializes, with code such as the following:

```
DataTransferManager::GetForCurrentView()->DataRequested +=
    ref new TypedEventHandler<DataTransferManager^,
        DataRequestedEventArgs^>( this, &MainPage::OnDataRequested);
```

The code registers the `OnDataRequested` private method as the handler that is invoked by the Share Broker Windows component that manages the sharing operation. When the method is called, the application needs to provide the data. Here's a simple app that shows the flags of the world:



This app wants to share a selected flag image and some text, being the name of the selected country. The `OnDataRequested` method is implemented as follows:

```
void MainPage::OnDataRequested(DataTransferManager^ dtm,
    DataRequestedEventArgs^ e) {
    int index = _gridFlags->SelectedIndex;
    if(index < 0) return;
```

```
auto data = e->Request->Data;
auto flag = (CountryInfo^)_gridFlags->SelectedItem;

data->SetText(ref new String(L"Flag of ") + flag->CountryName);
auto bitmap = RandomAccessStreamReference::CreateFromUri(
    flag->FlagUri);
data->SetBitmap(bitmap);
data->Properties->Title = "Flags of the world";
data->Properties->Thumbnail = bitmap;
}
```

The first thing the method does is check whether any flag is selected (`_gridFlags` is a `GridView` holding all flags). If nothing is selected then the method simply exits. If the user tries to share when nothing is selected, Windows displays a message, **There's nothing to share right now.**

It's possible to set another text line to indicate to the user the exact reason why sharing was unavailable. Here's an example:



```
if(index < 0) {
    e->Request->FailWithDisplayText(
        "Please select a flag to share.");
    return;
}
```

The `DataRequestedEventArgs` has a single property (`Request`, of type `DataRequest`), which has a `Data` property (a `DataPackage` object), that's used to fill in sharing data. In the preceding code snippet, a string is set with the `DataPackage::SetText` method. Next, `DataPackage::SetBitmap` is used to set an image (with the helper `RandomAccessStreamReference` class).

A package also contains a bunch of properties that can be set, of which `Title` is the only one that's required. The example sets the thumbnail to the same flag image.

Other formats are acceptable by the `DataPackage`, with methods such as `SetHtmlFormat`, `SetUri`, `SetRtf`, and `SetStorageItems` (sharing of files/folders).

Another method, `SetDataProvider`, allows the app to register a delegate that will be queried when the data is actually needed, and not before that. This may be useful if obtaining the data is expensive, and should only be done if actually needed; also, it provides a way to share custom data.

Once the method completes, the data is available to a share target.

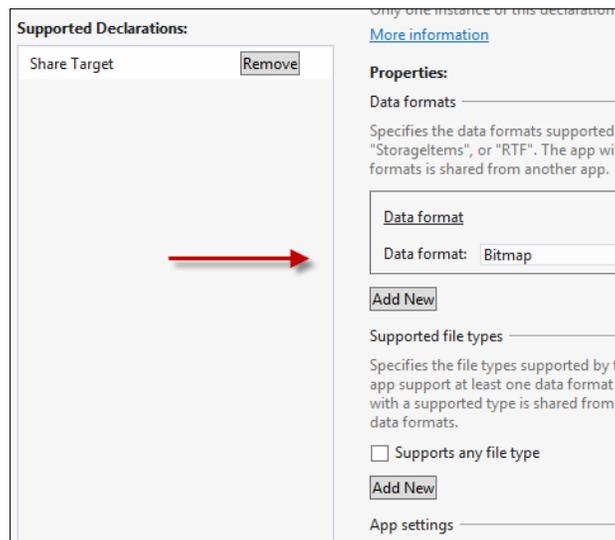


`DataRequest` has a `GetDeferral` method that allows the app to make asynchronous calls without the sharing broker considering the data ready when the method returns (a similar mechanism to the one we've seen for background tasks). Calling `DataRequestDeferral::Complete` signals the actual readiness of the data to be shared.

Share target

Becoming a share target is more difficult than a share source. One reason for this, is that the share target application may not be running when sharing is requested. This means the system must know beforehand which applications are capable of being share targets and what kind of data these applications can receive.

The first step in becoming a share target is declaring in the application manifest that the app is, in fact, a share target, and specifying the kinds of data it's willing to accept. Here's a screenshot of the manifest showing an application that is willing to receive bitmaps:



A share target app must support at least one data format (**Bitmap** in this example), or at least one file type (such as `.doc`).

Here's how this app appears in the share pane when a flag is selected:



The application named **ShareTargetDemo** is part of the downloadable code for this chapter, which is a simple image viewer for the Pictures library.

Once the user selects our application, it's activated (executed) if not already in memory. The system calls the virtual method `Application::OnShareTargetActivated`. This method indicates the app is being activated as a share target, and must respond appropriately.

Specifically, the application must provide some user interface for the share pane, indicating what data it's about to consume and provide some button control to allow the user to actually confirm the share.

Here's a simple share page UI that allows for some text labels, an image, and a **Share** button:

```
<StackPanel>
  <TextBlock Text="{Binding Text}" FontSize="20" Margin="10"/>
  <TextBlock Text="{Binding Description}" FontSize="15"
    TextWrapping="Wrap" Margin="4" />
  <Image Margin="10" Source="{Binding Source}" />
  <Button Content="Share" FontSize="25" HorizontalAlignment="Right"
    Click="OnShare"/>
</StackPanel>
```

The bindings expect a relevant `ViewModel` to use, which is defined like so:

```
[Windows::UI::Xaml::Data::BindableAttribute]
public ref class ShareViewModel sealed {
```

```

public:
    property Platform::String^ Text;
    property Windows::UI::Xaml::Media::ImageSource^ Source;
    property Platform::String^ Description;
};

```

The target app in this case is willing to accept images. The Image element would show a preview of the image to accept. Once the user clicks the **Share** button, the sharing operation is executed and the entire sharing operation is deemed complete.

The Application::OnShareTargetActivated override is responsible for activating the share page UI:

```

void App::OnShareTargetActivated(ShareTargetActivatedEventArgs^ e) {
    auto page = ref new SharePage();
    page->Activate(e);
}

```

SharePage is the class holding the share UI defined previously. The Activate method is an application-defined method that should extract the sharing information and initialize the UI as appropriate:

```

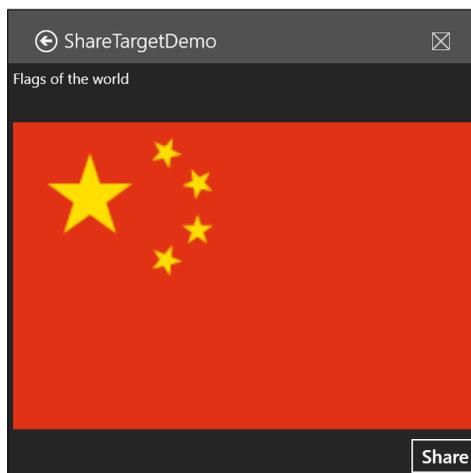
void SharePage::Activate(ShareTargetActivatedEventArgs^ e) {
    _operation = e->ShareOperation;
    auto data = _operation->Data;
    auto share = ref new ShareViewModel();
    share->Text = data->Properties->Title;
    share->Description = data->Properties->Description;
    auto ref = data->Properties->Thumbnail;
    if(ref != nullptr) {
        create_task(ref->OpenReadAsync()).then(
            [share, this](IRandomAccessStream^ stm) {
                auto bmp = ref new BitmapImage();
                bmp->SetSource(stm);
                share->Source = bmp;
                DataContext = nullptr;
                // INotifyPropertyChanged is not implemented
                DataContext = share;
            });
    }
    DataContext = share;
    Window::Current->Content = this;
    Window::Current->Activate();
}

```

The first thing to do is to save the operation object for later use when the **Share** button is clicked (`_operation` is a field of type `ShareOperation` located in the `Windows::ApplicationModel::DataTransfer::ShareTarget` namespace). The sharing data itself is in the `ShareOperation::Data` property (a `DataPackageView` object, similar to a `DataPackage` object on the share source side, but a read-only view of that data).

Next, the required information is extracted from the data object and placed in the `ShareViewModel` instance. If a thumbnail is provided, it's extracted by opening the `RandomAccessStreamReference` object and using a `BitmapImage` to load the image before placing it into the `ImageSource` used by the `ShareViewModel`.

Finally, the `DataContext` is set to the `ShareViewModel` instance and the page is made the current window content before actual activation. Here's how this looks when the share source is the flags application (China's flag is selected before sharing):



Now, the user can interact with the sharing pane. If closed, nothing happens and the target application is terminated, if it wasn't running prior to share activation. If, on the other hand, the user decides to do the actual share (by clicking the **Share** button), the application needs to do whatever is appropriate for such a case. For example, the built-in Mail application shows a new e-mail UI that adds the shared data (typically text) to an empty e-mail that can be sent.

Our sharing target app wants to save the provided image to the Pictures library for the current user. Here's the `Click` handler for the **Share** button for this app:

```
void SharePage::OnShare(Object^ sender, RoutedEventArgs^ e) {
    if(_operation->Data->Contains(StandardDataFormats::Bitmap)) {
        auto op = _operation;
        create_task(_operation->Data->GetBitmapAsync()).then(
```

```

        [op] (RandomAccessStreamReference^ sref) {
            return sref->OpenReadAsync();
        }).then([op] (IRandomAccessStream^ stm) {
            return BitmapDecoder::CreateAsync(stm);
        }).then([op] (BitmapDecoder^ decoder) {
            create_task(KnownFolders::PicturesLibrary->CreateFileAsync(
                "SharedImage.jpg", CreationCollisionOption::GenerateUniqueName))
                .then([decoder] (StorageFile^ file) {
                    return file->OpenAsync(
                        FileAccessMode::ReadWrite);}).then(
                        [decoder] (IRandomAccessStream^ stm) {
                            return BitmapEncoder::CreateForTranscodingAsync(
                                stm, decoder);
                        }).then([op] (BitmapEncoder^ encoder) {
                            return encoder->FlushAsync();
                        }).then([op] () {
                            op->ReportCompleted();
                        });
                });
        });
    }
}

```

The code may seem complex, because it attempts to save the provided image to a file, and because most of the operations are asynchronous, several tasks are involved to make sure the operations are performed in the right order. Here's a quick rundown of the performed operations:

- A check is made to make sure the data package indeed includes a bitmap; this is somewhat redundant in this case, as the application has indicated in the manifest that bitmaps is the only supported data. Still, this check may be useful in more complex scenarios.
- The bitmap is extracted using `DataPackageView::GetBitmapAsync`, returning a `RandomAccessStreamReference` object.
- `RandomAccessStreamReference::OpenReadAsync` is called to get the image data as an `IRandomAccessStream` object. This object is used to instantiate a `BitmapDecoder` object that is capable of decoding the image bits, by calling the static factory method `BitmapDecoder::CreateAsync`.



The `BitmapDecoder` and `BitmapEncoder` types are located in the `Windows::Graphics::Imaging` namespace. The factory method that creates the `BitmapDecoder` automatically identifies the stored bitmap format.

- Once the resulting decoder is obtained, a new file is created named `SharedImage.jpg` in the Pictures library (`KnownFolders::PicturesLibrary` returns a `StorageFolder`). Then the file is opened for read/write access.
- A `BitmapEncoder` is created based on the decoder information (`BitmapEncoder::CreateForTranscodingAsync`) and the image saving is completed by the call to `BitmapEncoder::FlushAsync`.
- The last thing to do (for any sharing operation) is to indicate to the system that the operation is complete by calling `ShareOperation::ReportComplete`.

Sharing files

Text, URLs, and images are not the only things that applications can share. Files can be shared as well, by calling the `DataPackage::SetStorageItems` from a source sharing app. These storage items can actually be files or folders (based on the `IStorageItem` interface).

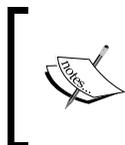
On the share target side, the storage items are available using the `DataPackageView::GetStorageItemsAsync` method, returning a read-only collection (`IVectorView`) of `IStorageItem` objects. The app can then access these files/folders in any way that's appropriate for the app.

Sharing page UI generation

Visual Studio provides a default page template for a share target operation:



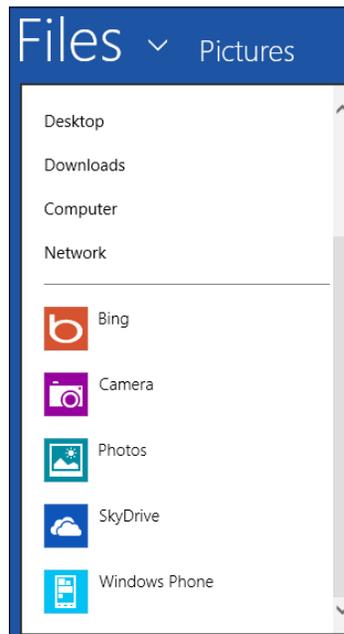
This adds a default UI for sharing, including a default `ViewModel` for data binding.



If the app project was created with the "Blank App" template, Visual Studio will add some helper classes that exist in other project templates, such as `SuspensionManager`, `LayoutAwarePage`, and so on because the share page it creates derives from `LayoutAwarePage`.

FileOpenPicker contract

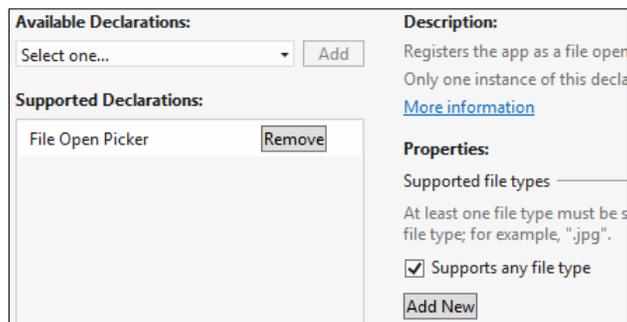
The `FileOpenPicker` class is used to pick a file from the filesystem – that's pretty obvious; what's not so obvious is that this same `FileOpenPicker` can be used to get a file from any application supporting the `FileOpenPicker` contract. When an application calls `FileOpenPicker::PickSingleFileAsync` or `PickMultipleFilesAsync`, a hosting process running the `image PickerHost.exe` is launched, where the `FileOpenPicker` is created. Besides the list of folders and files we can see applications:



The listed applications (**Bing**, **Camera**, and so on) are implementing the `FileOpenPicker` contract, so they can be contacted to get back a file. For example, SkyDrive allows browsing through the user's files and selecting a file or files to download. The Camera application provides a user interface that allows taking a picture right here and now with some camera connected, or embedded, in the device, returning the resulting image file to the calling application.

Implementing a FileOpenPicker contract

The first step in implementing the `FileOpenPicker` contract is to declare this in the app manifest. This is required, as the implementing app may not be running at the time a `FileOpenPicker` is opened from some other app:



As can be seen from the image, the app can support any file type, or a set of predefined file types, such as `.jpg`, `.doc`, and so on. This limits the applications that are considered candidates to be listed in the `FileOpenPicker`, depending on the file types specified with the `FileOpenPicker::FileTypeFilter` property by the calling application.

If the user actually selects the app within the `FileOpenPicker`, the app is launched (if it's not already running), and the `Application::OnFileOpenPickerActivated` virtual method is called. This idea is similar to the share target scenario we've looked at previously in this chapter.

The `FileOpenPicker` window is built with a header with the app's name (this is customizable by the app) and a footer with the **Open** and **Cancel** buttons. The middle section is where the app's specific selection UI is located.

The following example makes the Flags application a `FileOpenPicker` provider. The app should provide a view of the flags, allowing selection when an image is requested. The user interface for the flag selection is built like so:

```
<GridView ItemsSource="{Binding}" SelectionMode="Single"
  x:Name="_gridFlags" Margin="10"
  SelectionChanged="OnFlagSelected">
  <GridView.ItemTemplate>
    <DataTemplate>
      <Grid>
        <Grid.ColumnDefinitions>
          <ColumnDefinition />
          <ColumnDefinition Width="350" />
        </Grid.ColumnDefinitions>
      </Grid>
    </DataTemplate>
  </GridView.ItemTemplate>
</GridView>
```

```

        </Grid.ColumnDefinitions>
        <Image Margin="10,0" Height="60" Width="100">
            <Image.Source>
                <BitmapImage UriSource="{Binding FlagUri}" />
            </Image.Source>
        </Image>
        <TextBlock Text="{Binding CountryName}" FontSize="25"
            Grid.Column="1" Margin="2" />
    </Grid>
</DataTemplate>
</GridView.ItemTemplate>
</GridView>

```

The `GridView` hosts the collection of flags, bound to a collection of objects of type `CountryInfo`, defined like so:

```

[Windows::UI::Xaml::Data::BindableAttribute]
public ref class CountryInfo sealed {
public:
    property Platform::String^ CountryName;
    property Windows::Foundation::Uri^ FlagUri;
};

```

The `DataTemplate` for the `GridView` uses both properties to show an image of the flag and its corresponding country name.

The `GridView` event `SelectionChanged` is handled to provide the `FileOpenPicker` files to select or deselect. Before we get to that, we need to implement the `Application::OnFileOpenPickerActivated` method:

```

void App::OnFileOpenPickerActivated(
    FileOpenPickerActivatedEventArgs^ e) {
    auto picker = ref new FileOpenPickerPage();
    picker->Activate(e);
}

```

The code simply instantiates our custom `FileOpenPickerPage` class and calls an app-specific method on that page named `Activate`, passing the activation information provided by the system.

The preceding `Activate` method does very little:

```

void FileOpenPickerPage::Activate(
    FileOpenPickerActivatedEventArgs^ e) {
    _filePickerArgs = e;
    OnNavigatedTo(nullptr);
}

```

```
Window::Current->Content = this;
Window::Current->Activate();
}
```

The `FileOpenPickerActivatedEventArgs` is saved in the `_filePickerArgs` field, to be used later when flags are actually selected or deselected. The call to `OnNavigatedTo` sets up all flag data and the new page becomes the window's content and is activated.

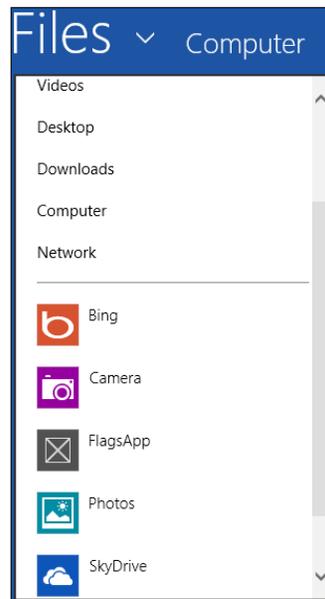
`OnNavigatedTo` does the following:

```
void FileOpenPickerPage::OnNavigatedTo(NavigationEventArgs^ e) {
    auto countries = ref new Vector<CountryInfo^>;

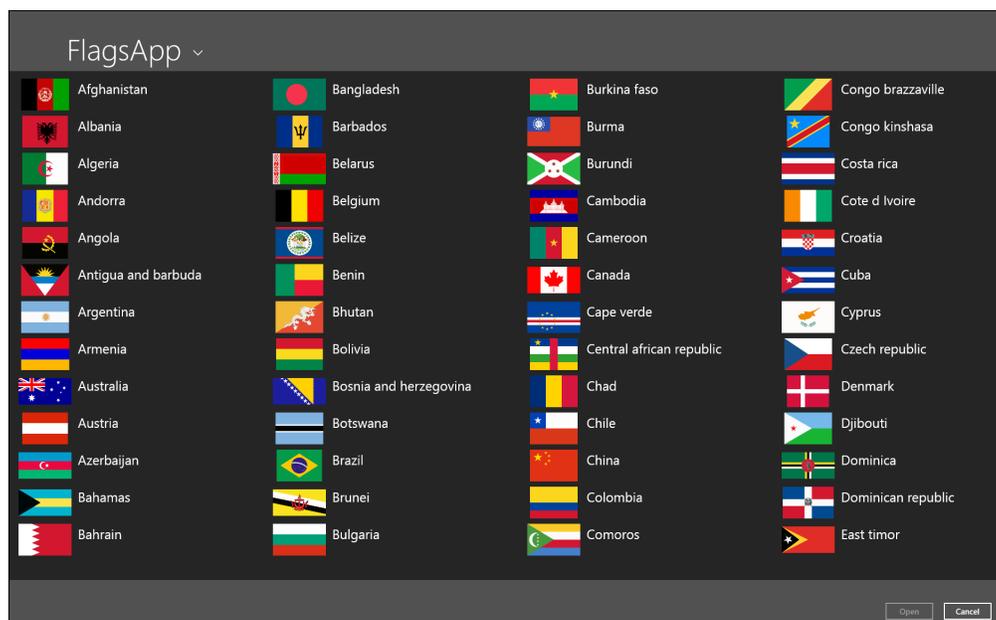
    create_task(Package::Current
        ->InstalledLocation->GetFolderAsync("Assets\\Flags")).then(
        [](StorageFolder^ folder) {
            return folder->GetFilesAsync(
                CommonFileQuery::DefaultQuery);
        }).then([this, countries](IVectorView<StorageFile^>^ files) {
            std::for_each(begin(files), end(files),
                [countries](StorageFile^ file) {
                    auto info = ref new CountryInfo;
                    info->FlagUri = ref new Uri(
                        L"ms-appx:///Assets/Flags/" + file->Name);
                    info->CountryName = MainPage::FlagUriToName(
                        file->Name->Data());
                    countries->Append(info);
                });
            DataContext = countries;
        });
}
```

The flag's image files are retrieved from the application installation location (`ms-appx:` scheme) and the country name is extracted from the image `Uri` by the helper method `FlagUriToName` (not shown); the country collection is updated, and finally the `DataContext` is set to the collection.

After the app is deployed (using a full build or by selecting **Build | Deploy Solution** in the Visual Studio menu), we can do a basic picker test by launching another app, such as the simple image viewer demonstrated in *Chapter 1, Introduction to Windows 8 Apps*. When clicking the **Open Image** button, the Flags application is displayed in the picker's custom apps:



If the Flags app is selected, the following is shown:



At this time, selecting any flag does nothing – the **Open** button remains disabled. We need to tell the `FileOpenPicker` that a file is selected. This is handled by the `SelectionChanged` event of the `GridView`:

```
void FileOpenPickerPage::OnFlagSelected(Object^ sender,
    SelectionChangedEventArgs^ e) {
    if(_gridFlags->SelectedIndex < 0 && _currentFile != nullptr) {
        _filePickerArgs->FileOpenPickerUI->RemoveFile(
            _currentFile);
        _currentFile = nullptr;
    }
    else {
        auto flag = (CountryInfo^)_gridFlags->SelectedItem;
        create_task(StorageFile::GetFileFromApplicationUriAsync(
            flag->FlagUri)).then([this, flag](
            StorageFile^ file) {
            AddFileResult result =
                _filePickerArgs->FileOpenPickerUI->AddFile(
                    _currentFile = flag->CountryName, file);
            // can check result of adding the file
        });
    }
}
```

The class keeps track of the currently selected file with the `_currentFile` field. If nothing is selected in the `GridView` and a file was previously selected, the `FileOpenPickerUI::RemoveFile` method is called to indicate this file should be removed from selection; if this is the last one selected, the **Open** button is disabled by the `FileOpenPicker`.

If a flag is selected (`GridView::SelectedIndex` is zero or greater), the file for the flag image is obtained by calling the static `StorageFile::GetFileFromApplicationUriAsync` method, and passed into `FileOpenPickerUI::AddFile`. The result (`AddFileResult` enumeration) indicates whether this succeeded or not. This can fail if the `FileOpenPicker` was not opened with that file type in mind. For example, in an image viewer that did not specify the GIF file extension, the addition would fail, as all flag images are in GIF format.



The simple code provided here is a bit too simple. One thing that is not handled is multiselection. The `GridView` is configured to use single selection only, but this should really be configured according to the way the `FileOpenPicker` was opened. This information is available in the `FileOpenPickerUI.SelectionMode` property (`Single` or `Multiple`).

If used as intended, the `SelectionChanged` event handler should use the `AddedItems` and `RemovedItems` properties of the `SelectionChangedEventArgs` object to manage selection and deselection.

Note, that just as with share targets, Visual Studio provides a page template for the `FileOpenPicker` contract.

Debugging contracts

Debugging contracts may seem difficult at first, as the application may not be running at the time, so setting a breakpoint will not automatically attach Visual Studio's debugger to the launched instance. This can be easily handled with one of the following two ways:

- Attach the Visual Studio debugger after the specific app is selected from the file picker. This is enough to hit a breakpoint when selecting or unselecting.
- Run the application with the debugger as usual, and then navigate to another app that shows a file picker. If the app is selected, any breakpoint will hit as usual.

Extensions

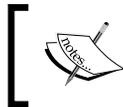
Extensions are a kind of contract between an app and the operating system. Extensions are implemented similarly to contracts, by overriding certain methods and providing certain functionality expected by the OS. Let's look at an example.



The distinction between contracts and extensions is not important in practice. They both have one important trait in common: implement some functionality defined by Windows, whether it's for app to app communication, or app to Windows.

Settings extension

The Settings charm opens up a Settings pane; its lower part shows standard Windows customization options, such as **Volume**, **Brightness**, **Power**, and so on. The top part of the settings pane can be used by applications to add app-specific settings.



Although Settings is documented as being a contract rather than extension, I feel it's an extension, as it does not involve another app—just the user.

For example, we'll add a settings extension to the Flags application, to allow the user to view the flags in three different sizes. The first thing to do is indicate to the system that the app is interested in supporting the settings extension:

```
SettingsPane::GetForCurrentView()->CommandsRequested +=  
    ref new TypedEventHandler<SettingsPane^,  
        SettingsPaneCommandsRequestedEventArgs^>(  
        this, &MainPage::OnCommandRequested);
```

This call registers the `SettingsPane::CommandRequested` event, raised when the user opens the Settings pane and the application is in the foreground.

When the event is fired, we can add commands to be displayed in the settings pane like so:

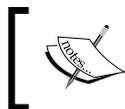
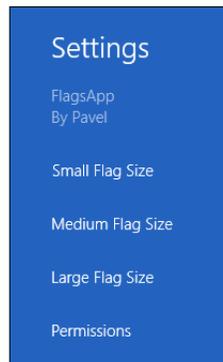
```
void MainPage::OnCommandRequested(SettingsPane^ pane,  
SettingsPaneCommandsRequestedEventArgs^ e) {  
    auto commands = e->Request->ApplicationCommands;  
    commands->Append(  
        ref new SettingsCommand("small", "Small Flag Size",  
            ref new UICommandInvokerHandler(  
                this, &MainPage::OnFlagSize)));  
    commands->Append(  
        ref new SettingsCommand("medium", "Medium Flag Size",  
            ref new UICommandInvokerHandler(  
                this, &MainPage::OnFlagSize)));  
    commands->Append(  
        ref new SettingsCommand("large", "Large Flag Size",  
            ref new UICommandInvokerHandler(  
                this, &MainPage::OnFlagSize)));  
}
```

The `SettingsCommand` constructor accepts an app-specific command ID that can be used to disambiguate commands in a common handler. The second argument is the text to show and the third is the handler to the command. In this example, all commands are handled by the same method:

```
void MainPage::OnFlagSize(UICommand^ command) {
    auto id = safe_cast<String^>(command->Id);
    if(id == "small") {
        ImageWidth = 60; ImageHeight = 40;
    }
    else if(id == "medium") {
        ImageWidth = 100; ImageHeight = 60;
    }
    else {
        ImageWidth = 150; ImageHeight = 100;
    }
}
```

The `UICommand` interface (in the `Windows::UI::Popups` namespace) provided is actually a `SettingsCommand` object which is currently the only class implementing that interface. It holds the properties of the command (`Id`, `Label`, and `Invoked` – the exact arguments to `SettingsCommand`, in that order).

`ImageWidth` and `ImageHeight` are properties bound to the `DataTemplate` that drives the flags image appearance. This is how the Settings pane looks when opened from the Flags' application:



The **Permissions** command is provided by the system and lists the capabilities that the application requires (such as internet connection, webcam, Pictures library, and so on).

Other contracts and extensions

Some other contracts and extensions, not shown here, include:

- File save picker – similar to a file open picker, but for save operations
- Search – provides a way for the app to participate in search, providing results that users can use to activate the app
- Cached file updater – used to track file changes (used by SkyDrive, for instance)
- Auto play – allows the app to be listed when new devices are plugged in to the machine
- File activation – allows the app to be registered as handling a file type
- Game explorer – allows the app to register as a game, providing a way for family safety features to be considered for the game

Summary

Contracts and extensions provide ways for an application to integrate better with Windows and other applications; for example, users can share data by using the Share charm, regardless of the kind of app. Contracts and extensions provide consistency in the user experience, not just the programming model. This makes the application more useful; it looks as though a lot of thought and care has been poured into the app. In general, this makes the app more likely to be used – which is one very important goal when building applications.

In the next (and last) chapter we'll take a quick look at application deployment and the Windows Store.

9

Packaging and the Windows Store

The previous chapters discussed various details for building Windows Store apps: from the basic Windows Runtime concepts, through building a user interface, to using unique Windows 8 features (for example live tiles, contracts). All that's left is to build your app and finally to submit it to the Store, so that everyone can enjoy it.

The Store, however, has its own rules and guidelines. Applications go through a certification process, to ensure they are of high quality, and will benefit users. "High quality" here encompasses many details, some related directly to quality experience (performance, touch experience, fluidity, and more) and some more subtle (such as responding appropriately to network fluctuations and adhering to modern UI design guidelines).

Windows Store apps provide many opportunities for developers. The Store is not yet saturated (as iOS and Android stores are), and so applications have a better chance of being noticed. Monetization is possible – apps may cost money, but other models exist: the Store model supports trial applications (with various expiration mechanisms), in-app purchases can be provided, so that the app may be downloaded for free, but items or services can be sold from within the app; the app can show ads – and get paid just because users are running the app – although the app itself is free.

In this chapter, we'll take a look at application packaging for the Windows Store and discuss some of the issues that need to be considered for the application to pass certification successfully.

The application manifest

We've already met the app manifest several times in previous chapters. This is the basic declarations the application makes, before it's even executed. Some important things to consider:

- Image logos, other than the defaults, must be supplied – the default images will automatically cause certification to fail; all image logos must be provided, and the image sizes must be as required (no scaling).
- In the capabilities tab, only the required capabilities should be checked. It's easy to check just almost everything, but this makes the app less secure and may even fail certification – the user will have to authorize capabilities that may not actually be used.
- Supported orientations may be supplied, leaving out some orientations that may make no sense for the particular app. Games, as an extreme example, can usually run with a particular orientation (landscape or portrait) only.
- For some capabilities, a privacy policy statement must be supplied as part of the app or through a web link. This should state what the app is doing with those capabilities. Examples that need a privacy statement are Internet (client, server) and Location (GPS).

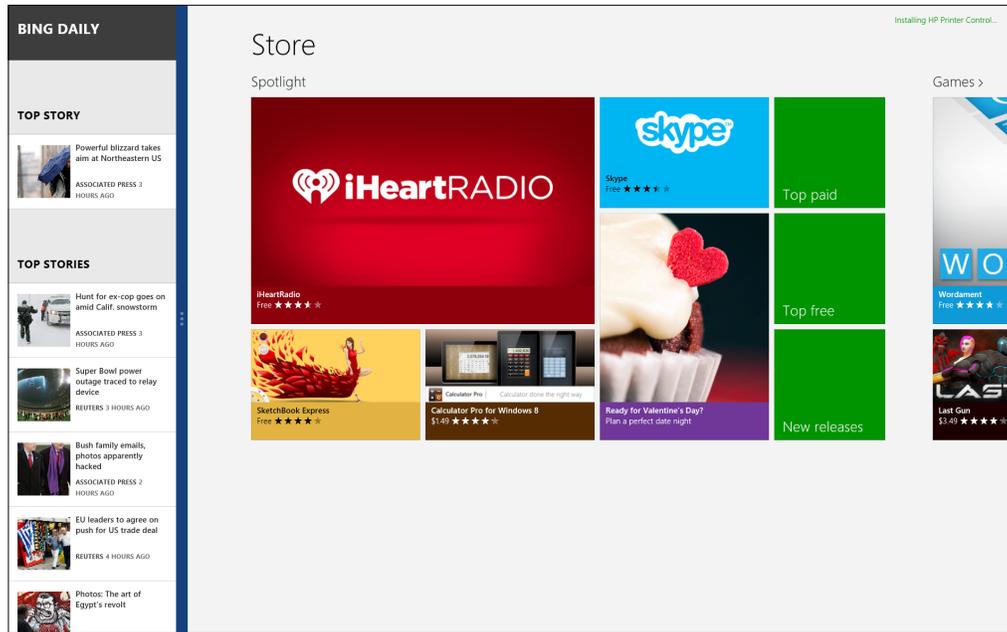
The question of orientation brings up a more general issue – application views. Besides the obvious landscape and portrait, the application (in landscape mode) can also share the screen with another application, in snapped or filled mode.

The application view state

An application can be viewed in four different ways:

- **Landscape** – screen width is larger than its height.
- **Portrait** – screen height is larger than its width.
- **Snapped** – the application takes up 320 pixels in width, while another application takes the rest of the screen width. This is only possible if the horizontal display resolution is at least 1366 pixels.
- **Filled** – the "mirror" of snapped. The application takes most of the horizontal space, leaving 320 pixels for another app.

Here's a screenshot of two apps being in the snapped/filled states:

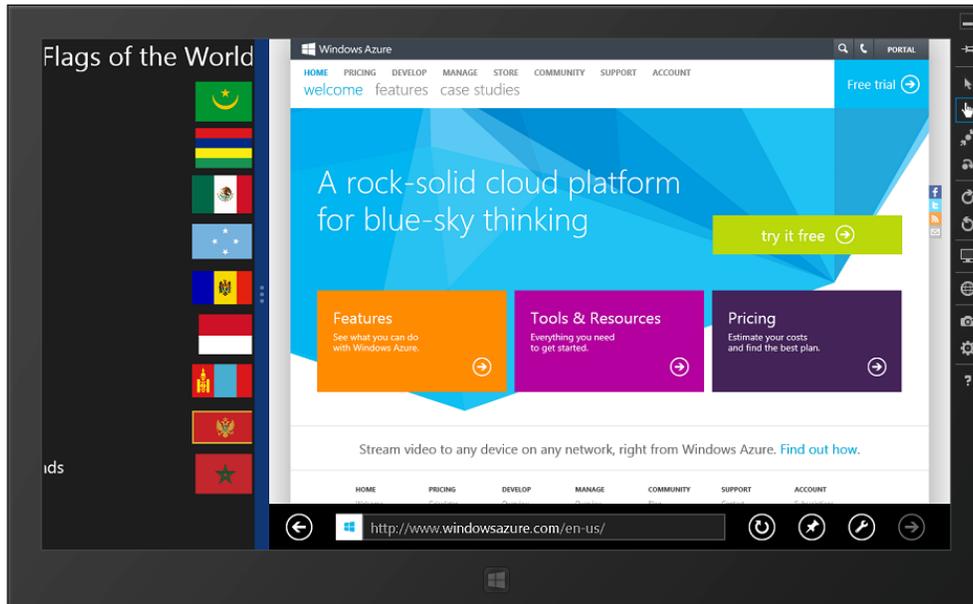


Users expect the application to change its view accordingly when in snap mode. In the preceding screenshot, the News application is snapped, so news articles appear as small items and not full text as they do in other modes.

Let's take the Flags app from the previous chapter and adapt it to display nicely, depending on its current view state.

Implementing view state changes

There are several ways to deal with view state changes. We'll take a simple, pragmatic approach. Currently, our Flags application presents the same view irrespective of the orientation or "snappiness". To test orientations and views we can use the simulator that's provided with the SDK (unless we happen to have a tablet-based device to do actual testing). Here's what the application looks like in the snapped mode inside the simulator:



Clearly, this is not the best user experience. The text next to each flag is too long, so not many flags are visible at the same time, and the text maybe cut off. A better approach would be to show just the flag images, without the country names.

The system raises the `SizeChanged` event on the active page – this is something we can handle and make the necessary view changes. First, we'll make the `ItemTemplate` of our `GridView` a bit more flexible by binding it to an additional property that we'll change as needed when the view changes. Here's the complete template:

```
<DataTemplate>
  <Grid Width="{Binding ColumnWidth, ElementName=_page}">
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto"/>
```

```

        <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Image Margin="10,0" HorizontalAlignment="Left"
    Height="{Binding ImageHeight, ElementName=_page}"
    Width="{Binding ImageWidth, ElementName=_page}">
        <Image.Source>
            <BitmapImage UriSource="{Binding FlagUri}" />
        </Image.Source>
    </Image>
    <TextBlock Text="{Binding CountryName}" FontSize="25"
    Grid.Column="1" Margin="2" />
</Grid>
</DataTemplate>

```

The change is in the topmost `Grid`—its `Width` is bound to a dependency property (`ColumnWidth`) on the `MainPage` object.



It would be more elegant to implement this in a separate `ViewModel` that implements `INotifyPropertyChanged`, as discussed in *Chapter 5, Data Binding*. The approach shown here is quicker, and is enough for demonstration purposes.

This `ColumnWidth` property would change, depending on the current view state.



This page's markup sustains both landscape and portrait orientations equally well. Sometimes, more drastic changes are required for good orientation support. Some layout panels are better suited for both orientations, such as the `StackPanel`. `Grid` is not suited for this, unless it's a very simple one. A complex `Grid` may have to undergo significant changes when switching orientations.

The `SizeChanged` event is registered in the `MainPage` constructor like so:

```

SizeChanged += ref new SizeChangedEventHandler(
    this, &MainPage::OnSizeChanged);

```

The handler just calls a helper method, `HandleSizeChanges`:

```

void MainPage::OnSizeChanged(Object^ sender,
    SizeChangedEventArgs^ e) {
    HandleSizeChanges();
}

```

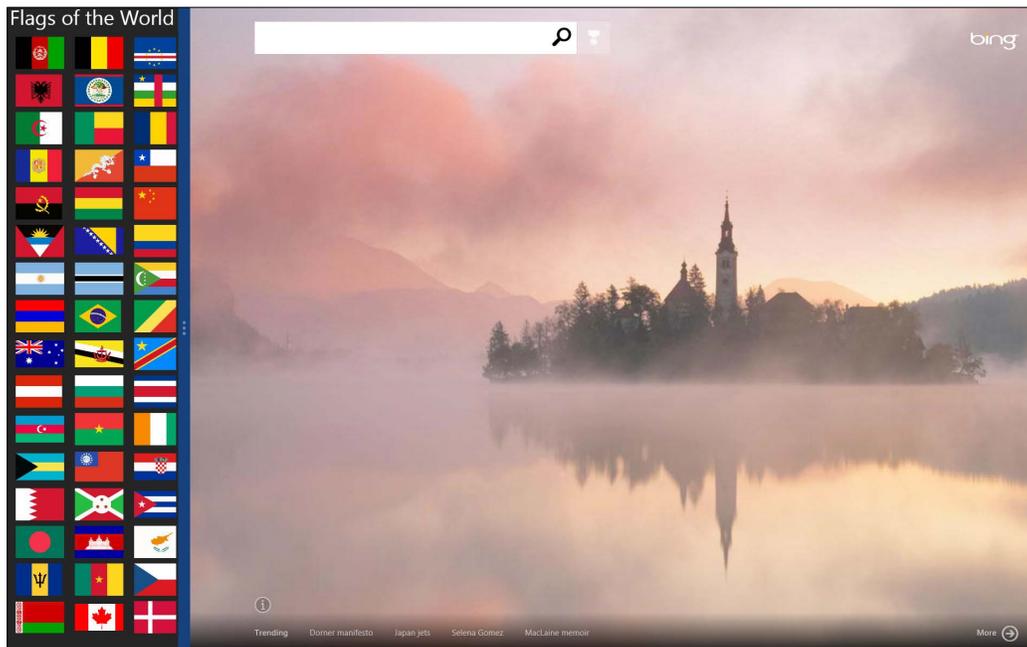
This same helper is called from the `OnNavigatedTo` override to make sure the view is adjusted when the page first loads. The basic idea is to inspect the `Windows::UI::ViewManagement::ApplicationView::Value` static property, and take appropriate actions based on the possible values:

```
void MainPage::HandleSizeChanges() {
    switch(ApplicationView::Value) {
    case ApplicationViewState::Filled:
    case ApplicationViewState::FullScreenLandscape:
        ColumnWidth = 300 + ImageWidth;
        break;

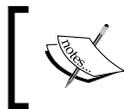
    case ApplicationViewState::FullScreenPortrait:
        ColumnWidth = 200 + ImageWidth;
        break;

    case ApplicationViewState::Snapped:
        ColumnWidth = ImageWidth;
        break;
    }
    // Force the GridView to re-evaluate its items
    auto ctx = DataContext;
    DataContext = nullptr;
    DataContext = ctx;
}
```

The code changes the `ColumnWidth` bound property as appropriate, based on the view state. The `Filled` and `Landscape` views are treated in the same way, but they could have been a bit different. In the `Portrait` mode the column width is narrower, so that more flags can be shown in a single screen. In the `snapped` view, the text portion is eliminated entirely, leaving the image only. This is the result in the `snapped` view:



Another common approach to deal with view state changes is using the `ViewStateManager` class. This allows for making changes in XAML and not requiring code, except the change to the correct view state, done with the `VisualStateManager::GoToState` static method. This approach is beyond the scope of this book, but many such examples can be found on the web.

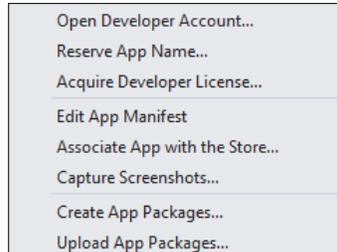


If nothing else makes sense, at the very least, an application should show some logo image or text when in the snapped view. Otherwise, the app may fail certification if the view is unprepared for the snap view.

Packaging and validating

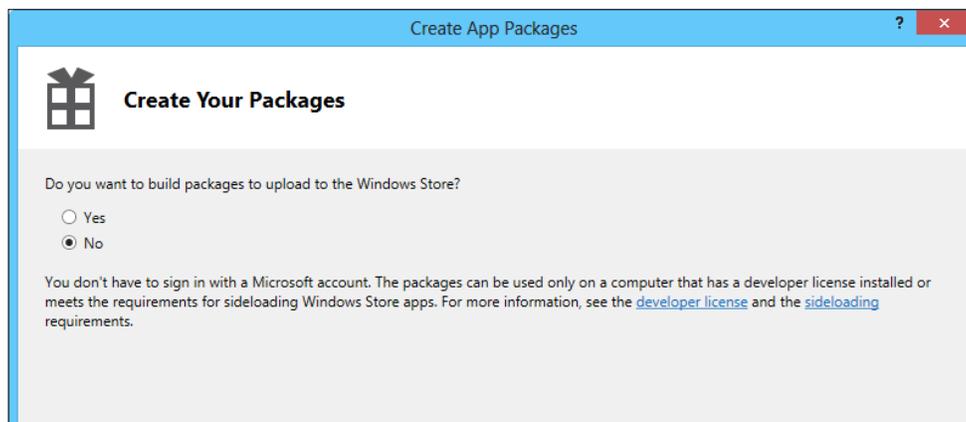
Once the application is done (or at least considered done by the developers), it's time to package and upload it to the Store. The first step should be testing the application for some types of errors that would fail certification, so these can be fixed right away.

To get started, we can use Visual Studio's **Project | Store** submenu:

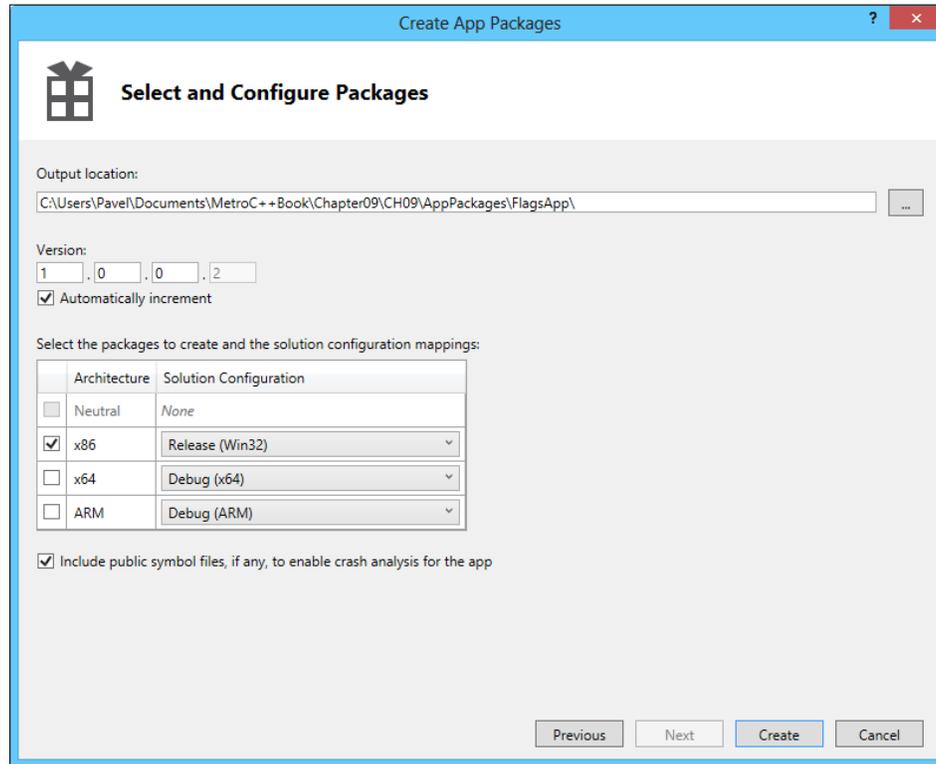


The menu allows opening a developer account, reserving an app name (must be unique, and will be reserved for one year) and do some other stuff (such as capture a screenshot – at least one is required); you can find information on these options in the developer portal for Windows Store apps. We'll now take a look at the **Create App Packages** option.

The first question the dialog asks is whether we want to create a package to upload to the Store. If **Yes** is selected, the developer must sign with his/her Microsoft ID, and then the packages will be built and later uploaded. We'll take the **No** answer route for now:

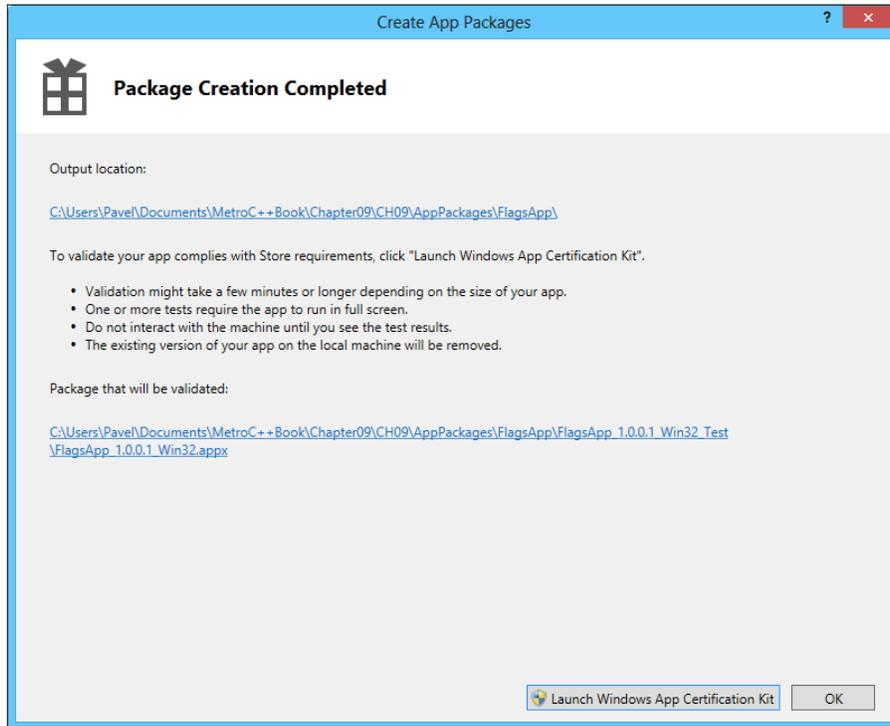


Clicking on **Next** shows a dialog that allows selecting the configurations to build (and to test):



We must select a Release configuration for package creation and testing. Testing a Debug configuration would fail, no matter what. Uploads to the Store must be Release builds only, otherwise the app would fail certification. We can select the required architectures that we want to create packages for. x86 and ARM are recommended – ARM is the only architecture available on Windows RT machines, so should be supported. x86 is a good choice for Intel/AMD-based architectures.

Clicking on **Create** builds the project in the selected configurations and shows the following dialog:



This shows the location of the created packages, where all the required files to be uploaded reside. The dialog further suggests to launch the **Windows App Certification Kit (WACK)** to do some automatic validation testing on the app.

Using the Windows App Certification Kit

Running the WACK is one of the first steps done in the Store; this means if the app fails the local WACK test, it will certainly fail certification in the Store. Once selected, the following dialog appears:



The validation process takes a few minutes, in which the application will be started and terminated automatically. Finally, the results will be shown with a "Passed" or "Failed" notification and a link to a report that was created for the test (warnings may appear as well, but they are not considered a failure).

 The Flags app fails the WACK test because it did not replace the default image logos.

If the application passes the WACK test, we can go ahead and upload the app to the Store. Passing WACK does not mean that the app is sure to pass Store certification. The Store does a lot more testing than WACK, including manual testing with a live person; but passing the WACK test is a good first step. You should never upload a package without passing the local WACK test.

 The complete requirement list for Store apps can be found at <http://msdn.microsoft.com/en-us/library/windows/apps/hh694083.aspx>.

Summary

In this chapter, we saw the basic procedure of packaging and uploading an app to the Store and discussed some of the issues that must be addressed for the app to pass certification successfully. The reader is encouraged to consult the official Microsoft certification guidelines to get the full details.

Since the certification process can take a few days, it's better if the app is as tested as possible before actual submission. Using the Windows App Certification Kit is a must to improve chances of successful certification.

Microsoft wants apps in the Store that are high quality and provide real value to users. This means the app has to behave "nicely", for sure, but that's not nearly enough; the content should be compelling and fun, so users return to the app again and again—and that's a challenge for a different book.

Index

Symbols

`__uuidof` operator 48
<visual> element 214

A

Activate method 251
Active Template Library (ATL) 11, 61, 78
AddNewMovie method 226
AddNewMovie operation 221
AddRef 42
AppBar 133
AppBar, Windows 8, 9
AppContainer 245
Application Binary Interface (ABI)
 about 48, 70, 177
 crossing 177
ApplicationData::LocalSettings
 property 210
ApplicationData::RoamingSettings
 property 211
application lifecycle
 about 205-207
 application execution states,
 determining 210
 helper classes 211
 state, restoring 208, 209
 state, saving 208
 state store options 210, 211
application manifest 266
Application::Suspending event 60
application tile defaults
 setting 212
application view state
 changing 267

 filled 266
 landscape 266
 portrait 266
 snapped 266
 snapped mode 268
 view state changes 268-271

ARM 273

ArrangeOverride method 203

asynchronous operations

 about 71-74
 cancelling 76, 77
 error handling 77
 tasks, using 75

attached properties, XAML

 about 89
 Canvas.Left 90
 Canvas.Top 90

attached property

 defining 190, 191
 using 189

audio

 playing 234, 235

AudioCategory property 236

auto 29, 30

Auto play 264

B

background audio

 maintaining 235-237
 playing 234
 sound-level notifications 238

BackgroundDownloader 242

background task

 background audio, playing 234

BackgroundTaskBuilder class 230

background task declaration 229
background tasks limits 240, 241
background transfers
 about 241
 file, downloading 241-243
badge
 about 215
 updating 215
BasicXaml 87
binary resources
 about 100, 101
 adding, to project 100, 101
binding failures 153
binding mode
 one time 148
 one way 148
 two way 148
BirthYear property 154
Blend
 about 96
 design surface 97
 introducing, for for Visual Studio 2012 tool
 96
blog post, Visual C++ team
 about 28
book_review wrapper
 about 173
boost library
 URL 27
ButtonBase class
 about 130
 ClickMode property 130
 CommandParameter property 130
 Command property 130
 properties 130
Button control
 about 130
 CheckBox 130
 RadioButton 130

C
C++
 about 27
 converting, to WinRT 172-177

C++11
 about 28
 auto 29, 30
 features 28
 lambdas 30
 nullptr 28, 29
 smart pointers 33
C++ Accelerated Massive Parallelism.
 See C++ AMP
Cached file updater 264
C++ AMP 79
CanExecuteChanged event 164
CanExecute method 164
CanvasDemo 121
Canvas panel
 about 118-121
 children, adding to panel 122
capabilities 246
CarouselControl 125
CarPtr class 34
C++ committee
 URL 27
C++/CX
 about 54
 member, accessing 56
 members, defining 61
 objects, creating 54
 objects, managing 55
 WinRT component project 62, 63
 WinRT types, defining 61
ChannelControlTrigger 241
Charms bar, Windows 8
 about 10
 devices charm 11
 search charm 10
 settings charm 11
 share charm 10
 start charm 11
CheckBox control 131
C++ libraries
 ATL 77, 78
 MFC 77
 STL 77
C++ library projects 181
ClientInfo object 225

collection-based controls

- about 134, 135
- ComboBox 135
- FlipView 136
- GridView 136
- ListBox 135
- ListView 136

collection properties, XAML 92, 93**collections type 81****ColorPicker custom control**

- creating 199-201

color picker user control

- creating 192, 193
- using 197

ColumnDefinition objects 117**COM**

- about 12, 37
- principles 37

ComboBox control 135**COM interface**

- about 37
- IInspectable interface 42
- implementing 38-41
- IUnknown interface 41, 42
- layout 37

commands

- about 163
- drawbacks 163

compilation process, XAML 97**complex properties, XAML 88****Component Object Model. *See* COM****ComPtr<T> 53****ContentControl 192****content controls**

- about 126
- AppBar 133
- Button control 130, 131
- Frame 133
- rules 127-129
- ScrollViewer 132
- SelectorItem 134

content property, XAML 90, 91**contracts**

- about 246
- debugging 261
- FileOpenPicker contract 255
- share contract 247

controls

- about 125
- collection-based controls 134
- Content controls 126-129
- SemanticZoom control 144

control's properties

- using 185

control template

- about 183
- building 183, 184
- customizing, attached properties used 189
- properties, using 185
- state changes, handling 186, 188

ControlTemplate 183**ConvertBack method 160****Convert method 160****C++ REST SDK 225****CRT**

- about 79
- URL 79

culture parameter, Convert and ConvertBack method 162**custom controls**

- about 198
- code, binding 201, 202
- ColorPicker custom control, creating 199-201

custom control templates 182**custom drawn elements**

- creating 203
- custom elements**
- about 191
 - custom controls 198
 - custom drawn elements 203
 - custom panels 202
 - user control 192

custom panels

- about 202
- creating 202

cycle updates, tiles

- enabling 214

D**data binding**

- about 147, 148
- binding, to collections 155-157

- concepts 148
- element-to-element binding 148
- failures 153
- notifications, changing 153-155
- object-to-element binding 150-152
- data binding concepts**
 - binding mode 148
 - source 148
 - source path 148
 - target 148
- DataPackage::SetText method 248**
- DataRequestedEventArgs 248**
- DataTemplate 134, 183**
- data templates**
 - customizing 157
- data template selectors 162**
- data view**
 - customizing 157
 - declaration 159
 - value converters 157, 158
- delegates, C++/CX 58**
- DependencyObject base class 193**
- dependency properties, XAML**
 - about 89
 - DependencyObject 89
- dependency property (DP). See WinRT**
- dependency properties**
- desktop apps**
 - versus, store apps 11
- devices charm 11**
- DirectX 15, 79**
- DisplayMemberPath 134**
- DownloadOperation object 242**
- DrawingContext 203**
- dynamic_cast<> operator**
 - using 41
- Dynamic Data Exchange (DDE) 37**

E

- elements**
 - about 125
 - Image element 142
 - text-based elements 137
- element-to-element binding**
 - about 148-150
 - drawbacks 149

- enums 210**
- events, C++/CX 60**
- events, XAML**
 - connecting, to handlers 94
- exception types 82, 83**
- Execute method 164**
- execution process, XAML 97**
- eXtensible Application Markup Language.**
 - See XAML
- extensions**
 - about 261
 - settings extension 262, 263

F

- File activation 264**
- FileOpenPickerActivatedEventArgs 258**
- FileOpenPicker contract**
 - about 255
 - implementing 256-260
- File save picker 264**
- FlipView control 136**
- Frame 133**
- FrameworkElement 125**
- fundamental element-related classes**
 - class diagram 126

G

- Game explorer 264**
- General Availability (GA) 8**
- GetIids 43**
- Global Unique Identifier (GUID) 42**
- Graphic Processing Unit (GPU) 15**
- Graphics Processing Unit (GPU) 79**
- Grid element 87**
- Grid panel 116, 118**
- GridView control 136**

H

- H and CPP files, with XAML**
 - build process, connecting to 98
- HSTRING 44**
- HttpRequest 225**

I

IAsyncOperation<T> interface 59
IBackgroundTask interface 228
ICalendar::AddMinutes() 57
ICalendar::Compare() 57
ICalendar::SetToNow() 57
ICustomPropertyProvider interface 167
IInspectable interface 42
IIterable<T> 81
Image element
 about 142, 143
 Stretch property 143, 144
IMoviesService 221
implicit (automatic) styles 108
InitializeComponent method 97
interface 37
Interface Definition Language (IDL) file 54
interface ID (IID) 48
IObservableVector<T> 81
IsIndeterminate property 187
ItemContainerStyle property 135
ItemsControl class 134
IUICommand interface 263
IUnknown interface 41
IVector<T> 81
IVector<UIElement> interface 115
IVectorView<T> 81
IXMLHttpRequest2 COM interface 225

J

JsonConvert 223

L

lambdas 30, 32
language projections
 about 14
 C++ 14
 C# and Visual Basic 14
 JavaScript language 14
layout
 about 113
 properties 114, 115
layout panels
 about 115
 Canvas 118

 Grid 116, 117
 StackPanel 116
 VariableSizedWrapGrid 122, 123
 virtualizing 124
layout properties 114
 FlowDirection 115
 HorizontalAlignment/
 VerticalAlignment 114
 HorizontalContentAlignment/
 VerticalContentAlignment 115
 Margin 114
 Padding 114
 Width/Height 114
ListBox control 135
ListView control 136
LocalSettings 211
lock screen apps
 about 238, 239
 ControlChannelTrigger 239
 operations 240
 PushNotificationTrigger 239
 settings 240
 TimeTrigger 239
logical resources
 about 102-104
 duplicate keys 106
 managing 104, 105

M

Markup extensions, XAML 93
MeasureOverride method 202
MediaControl class 238
MediaElement 234
member access, C++/CX
 about 56
 delegates 58, 59
 events 60, 61
 methods 57
 properties 57, 58
MFC 77
Microsoft Foundation Classes (MFC) 11
Model View Controller (MVC) 165
Model View Presenter (MVP) 165
Model-View-ViewModel. *See* MVVM
Multithreaded Apartment (MTA) 44

MVVM

- about 148, 165
- constituents 165
- overview 170

MVVM framework

- building 166-169

N

Name key class

- about 46
- ActivationType 46
- CLSID 46
- DllPath 46

naming elements, XAML 94

Newtonsoft.Json.JsonConvert class 223

NextTrackPressed event 238

nullptr 28, 29

O

Object Linking and Embedding (OLE) 37

object-to-element binding 150, 152

ObservableObject class 167

OnApplyTemplate 202

OnDataRequested method 247

OnLaunched method 207, 210

OnSelectedMediaFile 235

P

panel virtualization 124

parameter, Convert and ConvertBack method 162

PasswordBox 141

PathGeometry 203

PausePressed event 238

Permissions command 263

pInst 48

PlayPauseTogglePressed event 237

PlayPressed event 237

plugins 245

PointerPressed event 120

Porche class 39

ppvObject 42

PreviousTrackPressed event 238

ProgressBar 183

ProgressBarIndicator 184

project structure, Store app

- about 22, 23
- App.xaml 23
- App.xaml.cpp 23
- App.xaml.h 23
- Capabilities element 25
- MainPage.xaml 22
- Package.appxmanifest file 23

PropertyChanged event 155

PropertyChangedEventArgs object 155

push notification application

- application server 220-223
- building 220

push notifications

- about 218
- architecture 219
- issuing 226
- registering 224, 225
- setting up 219

push notifications architecture

- diagrammatic representation 219

push notifications, secondary tiles 227

PushNotificationTrigger 241

Q

QueryInterface 42

R

RadioButton control 130

reference counting

- about 34
- caveats 35

RegisterForPushNotification

- method 221, 224

Registry keys 48

RepeatButton class 130

Resource Acquisition Is Initialization (RAII) 33

ResourceDictionary 105

resources 100

resources, WinRT

- binary resources 100
- logical resources 102

REST (Representational State Transfer) 224

RichEditBox 141

RichTextBlock 141

RoActivateInstance 44-46
RoamingSettings 211
RoInitialize function 44
RowDefinition objects 117
rules, XAML 86
Runtime Callable Wrappers (RCWs) 14

S

ScrollViewer control
 about 132, 203
 features 132
 options 132
Search 264
search charm 10
secondary tiles
 activating 217
 creating 215, 216
 push notifications 227
security ID (SID) 222
SelectionChanged event 260
SelectorItem 134
SemanticZoom control
 about 144, 145
 ZoomedInView property 145
 ZoomedOutView property 145
SendPushTileNotification method 226
settings charm 11
SettingsCommand constructor 263
share charm 10
share contract
 about 247
 share source 247-249
 share target 249, 250
shared_ptr<>
 about 34
 initializing 34
SharePage 251
share source 247-249
share target
 becoming 249
ShareTargetDemo app
 about 250
 features 250, 251
 files, sharing 254
 operations 253

 page UI generation, sharing 254
 working 250, 252
SharpDX 15
Single Threaded Apartment (STA) 44
smart pointers 33, 34
source 148
source path 148
Source property, Image element 142, 143
StackPanel 116
Standard Template Library. *See* STL
start charm 11
start screen, Windows 8 8
StaticResource markup extension 103
STL 77
StopPressed event 238
Store app
 closing 19
 creating 15-19
 deploying 20, 21
 int.ToString 21
 project structure 22-25
store application styles 111
string type 80
style inheritance 109
styles
 about 106, 107
 implicit (automatic) styles 108, 109
 setting 108
 store application styles 111
 style inheritance 109, 110
SystemTriggerType enumeration 230

T

target 148
targetType parameter, **Convert** and **ConvertBack** method 162
task
 about 228
 cancelling 233
 creating 228
 debugging 232
 implementing 230-232
 progress 232
 registering 228-230
TemplateBinding 184

text-based elements

- about 137
- custom fonts, using 138
- font-related properties 137
- PasswordBox 141
- RichEditBox 141
- RichTextBlock 141
- TextBlock 138
- TextBox 140

TextBlock 138, 139

TextBox

- about 140
- font properties 140

tiles

- about 211
- application tile defaults, setting 212
- badge updates 215
- content, updating 213
- cycle updates, enabling 214
- expiration 214
- secondary tiles, activating 217
- secondary tiles, creating 215, 216

toast notifications

- about 217
- raising 218
- using 217

toast options 218

toasts 217

ToastTemplateType enumeration 218

ToggleButton 130

Type converters, XAML 88

U

UI, user control

- building 196

unique_ptr<> 34

user control

- about 192
- color picker user control, creating 192, 193
- ColorPicker, using 197
- dependency properties 193
- dependency properties, defining 193-195
- events, adding 197
- UI, building 196

user interface

- building 15
- building, with XAML 85

V

value converters

- about 157, 158, 162
- declaration 159
- methods, implementing 159

value parameter, Convert and ConvertBack method 162

VariableSizedWrapGrid 122-125

ViewModel

- about 166
- responsibilities 166

VirtualizingPanel class 124

VirtualizingStackPanel 125

VisualStateGroups property 188

VisualStateManager 189

W

WACK test 275

WCF (Windows Communication Foundation) 221

weak_ptr<>::lock function 36

WebHttpBinding 224

Win32 API 11, 78

Windows 8

- about 7
- AppBar 9
- Charms bar 10
- desktop apps, versus store apps 11, 12
- features 8
- language projections 14
- start screen 8
- Store app, creating 15
- touch-enabled devices 8
- UI, building with XAML 85
- Windows Runtime 12

Windows App Certification Kit (WACK)

- about 274
- running 274, 275

WindowsConcatString 80

WindowsCreateString 45, 80

WindowsGetStringLen 80
Windows Notification Service (WNS) 219
Windows Presentation Foundation (WPF) 86
WindowsReplaceString 80
Windows Runtime components 171
 consuming 178, 180
Windows Runtime Library (WRL)
 about 52, 53
 headers 52
Windows Runtime. *See* **WinRT**
Windows Store applications
 about 245, 265
 application lifecycle 205
 application packaging 265
 background tasks 228
 background tasks limits 240
 background transfers 241
 capabilities 246
 contracts 246
 extensions 261
 live tiles 211
 lock screen apps 238
 packaging 271-274
 push notifications 218
 toast notifications 217
 validating 275
WindowsSubString 80
Windows Workflow Foundation (WF) 86
WinRT
 about 12
 characteristics 13
 class library 80
 commands 164
 controls 125
 custom elements 191
 elements 125
 fundamental element-related classes 126
 resources 100
WinRT APIs
 about 80
 collections type 81
 exception type 82
 string type 80
WinRT Calendar class 43
WinRT class
 about 62, 63
 event, adding 66, 67
 methods, adding 63-66
 properties, adding 63-66
WinRT component
 C# client, building 69, 70
 C++ client, building 67, 68
 consuming 67
WinRT component project 62, 63
WinRT dependency properties
 characteristics 193
 defining 193, 194, 195
WinRT layout system 113
WinRT metadata
 about 49
 Calendar class 51
 viewing 50, 51
WinRT object
 about 43
 creating 43-48
WinRT XAML toolkit
 URL 203
WinRT XML DOM API 214
WrapGrid 125

X

x86 273
XAML
 about 15, 85
 attached properties 89, 90
 build process, connecting to 98, 99
 collection properties 92
 compilation 97
 complex properties 88, 89
 content property 90, 91
 dependency properties 89
 events, connecting to handlers 94
 execution 97
 features 85, 86
 Markup extensions 93
 namespace 87
 naming elements 94
 rules 86, 87
 rules summary 95
 Type converters 88
XAML rules summary 95



Thank you for buying **Mastering Windows 8 C++ App Development**

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

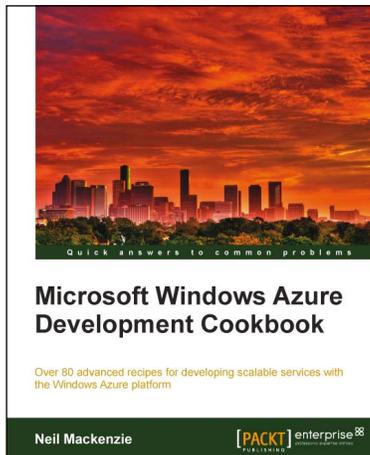


HTML5 Enterprise Application Development

ISBN: 978-1-84968-568-9 Paperback: 332 pages

A step-by-step practical introduction to HTML5 through the building of a real-world application, including common development practices

1. Learn the most useful HTML5 features by developing a real-world application
2. Detailed solutions to most common problems presented in an enterprise application development
3. Discover the most up-to-date development tips, tendencies, and trending libraries and tools



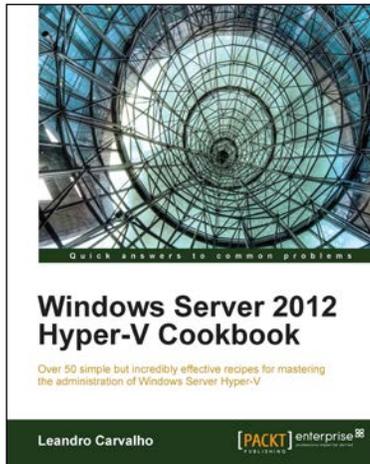
Microsoft Windows Azure Development Cookbook

ISBN: 978-1-84968-222-0 Paperback: 392 pages

Over 80 advanced recipes for developing scalable services with the Windows Azure platform

1. Packed with practical, hands-on cookbook recipes for building advanced, scalable cloud-based services on the Windows Azure platform explained in detail to maximize your learning
2. Extensive code samples showing how to use advanced features of Windows Azure blobs, tables, and queues.
3. Understand remote management of Azure services using the Windows Azure Service Management REST API

Please check www.PacktPub.com for information on our titles

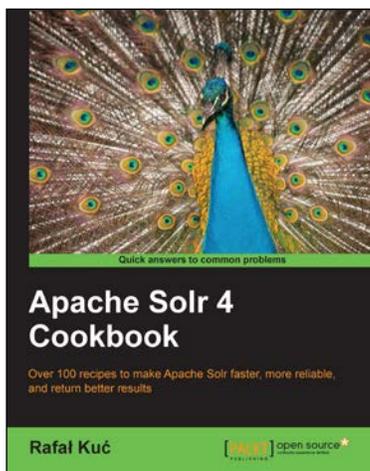


Windows Server 2012 Hyper-V Cookbook

ISBN: 978-1-84968-442-2 Paperback: 304 pages

Over 50 simple but incredibly effective recipes for mastering the administration of Windows Server Hyper-V

1. Take advantage of numerous Hyper-V best practices for administrators
2. Get to grips with migrating virtual machines between servers and old Hyper-V versions, automating tasks with PowerShell, providing a High Availability and Disaster Recovery environment, and much more
3. A practical Cookbook bursting with essential recipes



Apache Solr 4 Cookbook

ISBN: 978-1-78216-132-5 Paperback: 328 pages

Over 100 recipes to make Apache Solr faster, more reliable, and return better results

1. Learn how to make Apache Solr search faster, more complete, and comprehensively scalable
2. Solve performance, setup, configuration, analysis, and query problems in no time
3. Get to grips with, and master, the new exciting features of Apache Solr 4

Please check www.PacktPub.com for information on our titles