Delve inside the Windows Runtime—and learn best ways to design and build Windows Store apps. Guided by Jeffrey Richter, a well-known expert in Windows and .NET programming, along with principal Windows consultant Maarten van de Bospoort, you’ll master essential concepts. And you’ll gain practical insights and tips for how to architect, design, optimize, and debug your apps.

With this book, you will:
• Learn how to consume Windows Runtime APIs from C#
• Understand the principles of architecting Windows Store apps
• See how to build, deploy, and secure app packages
• Understand how apps are activated and the process model controlling their execution
• Study the rich features available when working with files and folders
• Explore how to transfer, compress, and encrypt data via streams
• Design apps that give the illusion of running using live tiles, background transfers, and background tasks
• Share data between apps using the clipboard and the Share charm
• Get advice for monetizing your apps through the Windows Store

Get Visual C# code samples
Download from:Http://Wintellect.com/Resource-WinRT-Via-CSharp

Microsoft Press
Celebrating 30 years!

About this Book
• Requires working knowledge of Microsoft .NET Framework, C#, and the Visual Studio IDE
• Targeted to programmers building Windows Store apps
• Some chapters also useful to those building desktop apps

Technologies Covered
• Windows 8.1
• Microsoft Visual Studio 2013

About the Authors
Jeffrey Richter is a cofounder of Wintellect (www.wintellect.com), a training and consulting firm dedicated to helping companies build better software faster. He is author of the classic CLR via C#, now in its fourth edition, Windows via C/C++, Fifth Edition, and other popular .NET programming books and courses.

Maarten van de Bospoort is a principal consultant with the Premier Services for Developers division in Microsoft. In this role, he teaches professional developers worldwide how to write Windows apps and helps independent software vendors architect and optimize their code for Windows.
Windows® Runtime via C#
Kristin, words cannot express how I feel about our life together.
I cherish our family and all our adventures. I’m filled each day
with love for you.

Aidan (age 10) and Grant (age 5), you both have been an
inspiration to me and have taught me to play and have fun.
Watching the two of you grow up has been so rewarding and
enjoyable for me. I am lucky to be able to partake in your lives.
I love and appreciate you more than you could ever know.

—JEFFREY RICHTER
To Jules and Joris. You guys have taught me so much. The two of you have been inspirational, each in your own particular way.

To Brigitte. For your tireless optimism, energy, love, and unwavering support.

—Maarten van de Bospoort

Maarten and family celebrate the publication of his first book.
## Contents at a glance

*Foreword*  
*Introduction*  

### PART I  CORE CONCEPTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Windows Runtime primer</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>App packaging and deployment</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>Process model</td>
<td>49</td>
</tr>
</tbody>
</table>

### PART II  CORE WINDOWS FACILITIES

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Package data and roaming</td>
<td>79</td>
</tr>
<tr>
<td>5</td>
<td>Storage files and folders</td>
<td>91</td>
</tr>
<tr>
<td>6</td>
<td>Stream input and output</td>
<td>119</td>
</tr>
<tr>
<td>7</td>
<td>Networking</td>
<td>145</td>
</tr>
<tr>
<td>8</td>
<td>Tile and toast notifications</td>
<td>183</td>
</tr>
<tr>
<td>9</td>
<td>Background tasks</td>
<td>205</td>
</tr>
<tr>
<td>10</td>
<td>Sharing data between apps</td>
<td>229</td>
</tr>
<tr>
<td>11</td>
<td>Windows Store</td>
<td>247</td>
</tr>
</tbody>
</table>

*Appendix: App containers*  
*Index*  
275
Contents

Foreword ................................................................. xiii
Introduction ............................................................. xvii
    Who should read this book ...................................... xvii
    Who should not read this book ............................... xviii
Organization of this book ......................................... xviii
Code samples ........................................................... xix
Acknowledgments .................................................... xix
Errata & book support ............................................... xx
We want to hear from you ............................................ xxi
Stay in touch ........................................................... xxi

PART I CORE CONCEPTS

Chapter 1 Windows Runtime primer 3
    Windows Store app technology stacks ....................... 6
    The Windows Runtime type system .......................... 10
    Windows Runtime type-system projections .................. 11
    Calling asynchronous WinRT APIs from .NET code ...... 16
        Simplifying the calling of asynchronous methods ...... 18
        Cancellation and progress ............................... 19
        WinRT deferrals ........................................... 21

What do you think of this book? We want to hear from you!
Microsoft is interested in hearing your feedback so we can improve our books and learning resources for you. To participate in a brief survey, please visit:

http://aka.ms/tellpress
Chapter 2  App packaging and deployment  25
  A Windows Store app’s project files ........................................... 25
  The app’s package manifest file ..................................................... 27
    Package identity ................................................................. 28
    Capabilities ........................................................................... 31
    App (not package) declarations (extensions/contracts) ............... 32
  Building a Windows Store app package ....................................... 34
    Contents of an .appx package file ............................................. 37
    Creating a bundle package file ................................................... 39
  Deploying a Windows Store package ......................................... 40
    Restricted deployments .......................................................... 40
    Enterprise deployments ........................................................... 41
    Windows Store deployments ..................................................... 43
  Package staging and registration ............................................... 44
  Wintellect’s Package Explorer desktop app .................................. 45
  Debugging Windows Store apps ................................................ 46

Chapter 3  Process model  49
  App activation ........................................................................... 49
  Managing the process model ....................................................... 55
  XAML page navigation ............................................................... 59
  Process lifetime management ..................................................... 64
    Windows Store app suspension ............................................... 65
    Windows Store app termination ............................................... 66
    How to best structure your app class’ code ............................... 70
    Debugging process lifetime management ................................. 75
Encrypting and decrypting data .................................................. 136
Populating a stream on demand .................................................. 138
Searching over a stream's content ............................................. 140

Chapter 7  Networking ................................................................. 145
Network information ............................................................... 145
Network isolation ..................................................................... 147
Network connection profile information ................................. 150
   How your app must use connectivity profile information ....... 152
   Network connectivity change notifications ....................... 153
Background transfer ................................................................. 154
   Debugging background transfers ........................................ 160
HttpClient: Client-side HTTP(S) communication .................... 161
   HttpBaseProtocolFilter ....................................................... 164
Windows Runtime sockets ......................................................... 168
   Socket addressing ............................................................. 169
   StreamSocket: Client-side TCP communication ................ 170
   StreamSocketListener: Server-side TCP communication .... 172
   StreamWebSocket: Streaming client-side WebSocket communication .................. 173
   MessageWebSocket: Messaging client-side WebSocket communication .................. 176
   DatagramSocket: Peer-to-peer UDP communication ............ 177
   DatagramSocket: Multicast UDP communication ................. 180
Encrypting data traversing the network with certificates ......... 181

Chapter 8  Tile and toast notifications ........................................ 183
Tiles and badges ..................................................................... 184
   Updating a tile when your app is in the foreground ............ 186
   Placing a badge on a tile .................................................... 188
   Animating a tile's contents ................................................ 190
   Updating a tile at a scheduled time ................................. 192
   Updating a tile periodically .............................................. 192
   Secondary tiles ............................................................... 192
Toast notifications ................................................. 194
Showing a toast notification at a scheduled time .......... 198
Using the Wintellect Notification Extension Library .... 199
Windows Push Notification Service (WNS) ............... 199

Chapter 9  Background tasks  205
Background task architecture .................................. 205
Step 1: Implement your background task’s code .......... 207
Step 2: Decide what triggers your background task’s code .. 208
  Maintenance and time triggers ......................... 209
  System triggers .............................................. 209
  Location triggers ....................................... 210
  Push notification triggers ............................ 211
  Control channel triggers ............................ 212
Step 3: Add manifest declarations ............................ 213
  Lock-screen apps ......................................... 214
Step 4: Register your app’s background tasks .......... 219
Debugging background tasks ................................ 222
Background task resource quotas ............................ 223
Deploying a new version of your app .................... 225
Background task progress and completion ............. 225
Background task cancellation .............................. 227

Chapter 10  Sharing data between apps  229
Apps transfer data via a DataPackage .................... 229
Sharing via the clipboard .................................... 231
Sharing via the Share charm ................................ 234
Implementing a share source app ......................... 237
  Delayed rendering of shared content .................. 239
Implementing a share target app ......................... 240
  Implementing an extended (lengthy) share operation .. 244
  Share target app quick links ........................ 244
Debugging share target apps .............................. 245
Chapter 11 Windows Store

Submitting a Windows Store app to the Windows Store ............... 248
  Submitting your app ........................................... 249
  Testing your app .............................................. 252
  Monitoring your app ......................................... 254
  Updating your app ........................................... 255

The Windows Store commerce engine .............................. 256
  The Windows Store commerce engine WinRT APIs .......... 257
  App trials and purchasing an app license ....................... 262
  Purchasing durable in-app product licenses ................. 264
  Purchasing consumable in-app products ....................... 266
  Purchasing consumable in-app product offers ............... 269

Appendix: App containers ........................................ 271

Index ....................................................................... 275

What do you think of this book? We want to hear from you!
Microsoft is interested in hearing your feedback so we can improve our books and learning resources for you. To participate in a brief survey, please visit:

http://aka.ms/tellpress
No kidding! Take your seats, everyone, so we can get started. If you haven’t a clue what is being discussed, you need to put this book down. Go back to the book store and buy Jeffrey Richter’s CLR via C#, Fourth Edition (Microsoft Press, 2012). Really, you need it anyway. Then after you read the Foreword, you may join us!

If you’re short on time, here is the CliffsNotes version: in Jeff’s previous two books, he vowed to never write another one. Well, here we all are again. No more empty promises. Jeff will probably write another book. After so many years of his lies about stopping, I can no longer support them. We are all here for the intervention. How much more can be said, right? I mean, aren’t there literally thousands of pages of stuff written on this already? Jeff claims that because Maarten came up with the initial research and prose, Jeff was cowriting the book, so it doesn’t count. We all see through this ruse. This is not our first rodeo.

Maybe you all can’t appreciate Jeff’s humble origins. He was never fully understood by his family. His parents didn’t believe there was a future in computers and hoped he would “get over it” and find a real career. When he quit his first job to write a book, they could not believe you could make a real living if you didn’t wear a tie every day. His mother never got over the fact that he wore jeans to work. His grandmother held a book of his in her hand and then decided that “windows” meant he dressed the mannequins at Macy’s. Like he was an expert on shopping and merchandising at the mall. I am not kidding; this is true. Let me just tell you something Jeffrey is not an expert on, and that is malls and shopping. So maybe that is why he must continually write, explaining over and over the importance of technology—this is just to justify his life to his family. It is the only explanation I can come up with.

The amazing thing is this new book does have stuff about the Windows Store! His grandma would be so excited—finally, she can go shopping for something in a store that has to do with Windows. Hopefully that will provide the validation he needs.

I will warn you. Jeff is becoming a bit of an old timer. Oh, it’s true. While I was trying to understand this book (which of course I don’t), he couldn’t stop himself from harking back to the day. When programs were real programs. They did meaningful things, like run medical software and financial software. Now we have applications and even that word is too complex, so we call them apps. They are available for $1.49, and they do things like pass gas or make a flashlight. There is nothing mission critical about this. Jeff feels a little like a sellout—with all his skills, the best he can do is try to create an app that will outsell Pet Rescue. He then talked about how this book will make
programming so easy. Windows 8.1 is so clean and smooth. There is not the same level of intensity to programming for it.

Although, between you and me, there is a little secret that should be shared. Under full NDA, just because we are friends. Jeff wrote a few of these apps for the Windows Store, and they were rejected. So maybe making a flashlight is not so easy, huh?

Really, this whole book thing is not even necessary. I mean, now you can hear him with WintellectNOW’s on-demand video training. It is like a lullaby—you can turn on Jeff’s videos anytime you need the comfort of another human’s voice. Reading is just a silly old-school skill that we used to need. Now we have the Internet and video feeds. So whenever you have issues with your code, you can invite Jeffrey into your office for a little lesson. If you happen to need a nap at the same time, well napping is one of the 7 habits of highly effective people.

So, with Windows 8.1 released, a new paradigm is in place. Jeffrey is clearly in front of this situation. He has his fingers on the pulse (or at least the touch-sensitive screen) of this situation. Who knows, someday he may even get me to update to this new version of Windows.

I would like to close with some thoughts from some old (I mean, longtime) friends, his business partners and fellow Wintellectuals.

John Robbins says:

*Jeffrey and I go way back. Back to the time when Steve Ballmer had
hair and modern applications used this amazing technology called
a “Windows message.” When Jeffrey started development with
Windows, you were doing really well if you could get two programs
running at the same time. After some detours through Windows
XP and the like, you could run dozens of applications concurrently.
Windows 8.1 brings us to the future of modern applications where
you can run two side by side.*

Jeff Prosise says:

*One of our favorite Jeffrey-isms: “This code is so bad, I feel sorry for
the compiler that has to compile it!”

*Jeffrey has an admitted inability to build user interfaces. Ergo Jeffrey-ism #2: “There is no UI problem that can’t be solved with a command prompt.”
And in closing, Mark Russinovich, author of the cyber thriller Zero Day, says:

*I have known Jeff since 1997 when he heckled me during a talk I was giving. He had a point, though, so we’ve been friends ever since. Jeff has come a long way since I first started mentoring him and he continues to impress me with his ability to solve Portal 2 puzzles.*

I hope you all enjoy this book! I am patiently awaiting the return of my husband.

Kristin Trace (Jeff’s wife)
October 2013

A typical father-and-son LEGO project.
Introduction

The Microsoft Windows operating system offers many features and capabilities to application developers. Developers consume these features by calling Windows Runtime (WinRT) APIs. This book explores many of the Windows Runtime APIs and how to best use them from within your own applications. An emphasis is placed on using WinRT APIs from Windows Store apps. Windows Store app developers will also find a lot of architectural guidance as well as performance and debugging advice throughout all the book’s chapters.

In addition, since many WinRT APIs are available to desktop apps too, much of this book is also useful to desktop app developers. In particular, desktop app developers will get a lot from the chapters that cover files, folders, streams, networking, toasts, and the clipboard.

Although WinRT APIs can be invoked from many different programming languages—including JavaScript, native C++, and Visual Basic—this book focuses on consuming them from C# because this language is expected to be the most-used language for consuming WinRT APIs due to its popularity with Microsoft-centric developers and the enormous productivity the language provides. However, if you decide to use a different programming language, this book still provides a lot of information and guidance about the WinRT APIs, and this information is useful regardless of the programming language used to invoke them.

Who should read this book

This book is useful to developers building applications for the Windows operating system. It teaches core WinRT API concepts and how to architect and design Windows Store apps, and it provides performance and debugging tips throughout. Much of the information presented in this book is also useful to developers building desktop apps for Windows.

Assumptions

This book expects that you have at least a minimal understanding of the Microsoft .NET Framework, the C# programming language, and the Visual Studio integrated development environment. For more information about C# and the .NET Framework, consider reading Jeffrey Richter’s CLR via C#, Fourth Edition (Microsoft Press, 2012).
Who should not read this book

This book does not focus on user-interface concepts and how to design an app’s user interface using technologies such as XAML or HTML. For information about using XAML to build user interfaces, consider reading Charles Petzold’s Programming Windows: Writing Windows 8 Apps with C# and XAML, Sixth Edition (Microsoft Press, 2013).

Organization of this book

This book is divided into two sections. Part I, “Core concepts,” focuses on concepts that all WinRT and Windows Store app developers must know.

■ Chapter 1, “Windows runtime primer,” defines the WinRT type system, its principles, and how to consume it from various programming languages. This chapter also addresses the importance of understanding asynchronous programming, which is pervasive throughout the WinRT API.

■ Chapter 2, “App packaging and deployment,” concentrates on the files that make up a Windows Store app, how those files get combined into a package file, and how the package file ultimately gets installed on users’ PCs. Package files are a new core concept in Windows, and understanding them is critical to being successful when using WinRT APIs.

■ Chapter 3, “Process model,” explains the core concepts related to how Windows Store apps execute. The chapter focuses on app activation, threading models, main view and hosted view windows, XAML page navigation, efficient memory management, process lifetime management, and debugging. All Windows Store apps must adhere to the architecture described in this chapter.

Part II, “Core Windows facilities,” contains chapters that explore various Windows facilities. The topics presented are key topics that almost all Windows Store app developers must know. Although the chapters can be read in any order, I recommend reading them in order because later chapters tend to reference topics presented in earlier chapters. Most of the chapters in Part II are about moving data around using settings, files, folders, streams, networking, and data sharing. However, there are also chapters explaining how apps can update tile content and display toasts. And there is a chapter explaining how apps can execute code when the user is not interacting with the app. The final chapter shows how to submit your app to the Windows Store and how to leverage the Windows Store commerce engine so that you can get paid for your development efforts.
Code samples

Most of the chapters in this book include code snippets showing how to leverage the various Windows features. Complete code samples demonstrating the features and allowing you to experiment with them can be downloaded from the following page:

http://Wintellect.com/Resource-WinRT-Via-CSharp

Follow the instructions to download the “WinRT via CS” .zip file.

Note In addition to the code samples, your system must be running Windows 8.1 and must have Visual Studio 2013 installed.

The Visual Studio solution contains several projects. Each project starts with a two-digit number that corresponds to the book’s chapter. For example, the “05a-Storage” project contains the code that accompanies Chapter 5, “Storage files and folders.”

Acknowledgments

I couldn’t have written this book without the help and technical assistance of many people. In particular, I’d like to thank my family. The amount of time and effort that goes into writing a book is hard to measure. All I know is that I could not have produced this book without the support of my wife, Kristin, and my two sons, Aidan and Grant. There were many times when we wanted to spend time together but were unable to due to book obligations. Now that the book project is completed, I really look forward to adventures we will all share together.

Of course, I also have to thank my coauthor, Maarten van de Bospoort. This book would not have existed at all if it were not for Maarten. Maarten started with my original course slides and demo code and turned that into the chapter text. Because books go into more technical depth and detail than courses, he had to research many areas in further depth and embellish the chapters quite a bit. Maarten would then hand the chapters over to me, and I would polish them by reorganizing a bit and add my own personal flair. It was a pleasure working with Maarten as he was always open to suggestions, and it was also really nice to have someone to discuss book organization and content with.

For technical content, there are many people on Microsoft’s Windows team who had one-on-one meetings with me so that I could learn more about the features and
their goals. In particular, I had two six-hour meetings with Howard Kapustein discussing packages, app containers, deployment, bundles, and so on. Talks with him changed my whole view of the system, and the chapters in this book reflect what I learned from these discussions. John Sheehan also spoke with me at length about package capabilities, declarations, and the resource system, which changed my whole view about app activation and contracts. Many others also had conversations with me about the WinRT type system, files, networking, background tasks, sharing, the Windows Store, tiles and toasts, and more. These people include Chris Anthony, Tyler Beam, Manoj Biswas, Arik Cohen, David Fields, Alain Gefflaut, Chris Guzak, Guanghui He, Scott Hoogerwerf, Suhail Khalid, Salahuddin Khan, Nathan Kuchta, Jon Lam, Nancy Perks, Hari Pulapaka, Brent Rector, Jamie Schwartz, Peter Smith, Ben Srou, Adam Stritzel, Henry Tappend, Pedro Teixeira, Dave Thaler, Marc Wautier, Sarah Waskom, and Terue Yoshihara.

As for editing and producing the book, I truly had some fantastic people helping me. Christophe Nasarre, who I’ve worked with on several book projects, has once again done just a phenomenal job ensuring that technical details are explained accurately. He has truly had a significant impact on the quality of this book. As always, the Microsoft Press team is a pleasure to work with. I’d like to extend a special thank you to Devon Musgrave and Carol Dillingham. Also, thanks to Curt Philips, Roger LeBlanc, and Andrea Fox for their editing and production support.

Errata & book support

We’ve made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed at:

http://aka.ms/WinRTviaCsharp/errata

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at:

mspinput@microsoft.com.

Please note that product support for Microsoft software is not offered through the addresses above.
**We want to hear from you**

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

http://aka.ms/tellpress

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

**Stay in touch**

Let’s keep the conversation going! We’re on Twitter: http://twitter.com/MicrosoftPress
In this chapter, we delve into a Windows Store app’s process model. Specifically, we’ll look at the various ways that an app gets activated as well as how it uses threads and windows. We’ll also talk about how to best architect your app so that it uses memory efficiently as it navigates the user from page to page. We’ll conclude with a discussion of process lifetime management (PLM) and how Microsoft Windows manages your app’s lifetime to further conserve memory, reduce CPU usage, and simplify the end-user experience.

Understanding this topic is critical to building Windows Store apps. If you are familiar with the Windows desktop app process model, you know that it is relatively easy to understand because you can usually get away with using just one thread, a main window, and then lots of child windows. However, the Windows Store app process model is substantially different and more complex because it uses several threads, each having at most one window, and child controls are simply drawn on a window. And this is just the tip of the iceberg in terms of complexity. The additional complexity is the result of two main factors:

- Windows Store apps are single instance. Windows allows only one instance of a Windows Store app to run at a time on the system. This conserves memory because multi-instance apps would each have their own memory. Because most apps have a single window, switching between apps is simpler for end users. Instead of seeing many windows they can switch to, users now see fewer windows. However, this makes your app more complex because you must now write the code to manage multiple documents or tabs yourself.

- Windows Store app activations. Windows Store apps can be activated for myriad reasons. All activations re-activate the already-running app and some activations cause other threads and windows to be created that your code has to manage.

### App activation

In this section, we talk about app activation. Specifically, we’ll discuss how Windows creates a process for your app and allows your app to initialize itself, and then we’ll look at how your app can start doing work on behalf of the user.
An app can be activated for several reasons. The most obvious is when the user taps your app’s tile from the Start screen. This kind of activation is called a launch activation, and all Windows Store apps must support launch activation; there is no way for a Windows Store app to opt out of it. But your Windows Store app can also be activated by the user tapping one of your app’s secondary tiles on the Start screen or if the user selects a toast notification that your app displays. (See Chapter 8, “Tiles and toast notifications,” for more information.) Activating your app due to a secondary tile or toast notification is also known as a launch activation. In addition to supporting launch activations, your app can optionally support other activations. For example, you can allow your app to be activated by the user opening a file in File Explorer, attaching a device (like a camera) to the PC, attempting to share content from another app with your app, and so on. There is a WinRT-enumerated type called Windows.ApplicationModel.Activation.ActivationKind that indicates all the ways an app can be activated. Table 3-1 shows the values offered by this enumeration and briefly describes each. Some of these activations are discussed in other chapters in this book, and some are very rarely used, so we will not discuss them at all.

**TABLE 3-1** ActivationKind values, their descriptions, and their view type.

<table>
<thead>
<tr>
<th>ActivationKind value</th>
<th>Activates your app when</th>
<th>View activation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Launch</td>
<td>User taps app's primary tile, a secondary tile, or a toast notification.</td>
<td>Main</td>
</tr>
<tr>
<td>Search</td>
<td>User uses the Search charm to search within your app while it's in the foreground.</td>
<td>Main</td>
</tr>
<tr>
<td>File</td>
<td>Another app launches a file whose file type is supported by your app.</td>
<td>Main</td>
</tr>
<tr>
<td>Protocol</td>
<td>Another app launches a URI whose scheme is supported by your app.</td>
<td>Main</td>
</tr>
<tr>
<td>Device</td>
<td>User attaches a device to the PC that is supported by your app (AutoPlay).</td>
<td>Main</td>
</tr>
<tr>
<td>Contact</td>
<td>User wants your app to post, message, call, video call, or map a contact.</td>
<td>Main</td>
</tr>
<tr>
<td>LockScreenCall</td>
<td>User taps a toast that answers a call when the user has locked her PC.</td>
<td>Main</td>
</tr>
<tr>
<td>AppointmentsProvider</td>
<td>Another app wants your app to show a time frame.</td>
<td>Main</td>
</tr>
<tr>
<td></td>
<td>Another app wants your app to add, replace, or remove an appointment.</td>
<td>Hosted</td>
</tr>
<tr>
<td>ShareTarget</td>
<td>User wants to share content from another app with your app.</td>
<td>Hosted</td>
</tr>
<tr>
<td>FileOpenPicker</td>
<td>Another app allows the user to open a file from a location your app has access to.</td>
<td>Hosted</td>
</tr>
<tr>
<td>FileSavePicker</td>
<td>Another app allows the user to save a file to a location your app has access to.</td>
<td>Hosted</td>
</tr>
<tr>
<td>CachedFileUpdater</td>
<td>Another app uses a file your app has cached.</td>
<td>Hosted</td>
</tr>
<tr>
<td>ContactPicker</td>
<td>Another app allows the user to access a contact maintained by your app.</td>
<td>Hosted</td>
</tr>
<tr>
<td>PrintTaskSettings</td>
<td>Your app is an app associated with a printer and exposes its settings.</td>
<td>Hosted</td>
</tr>
<tr>
<td>CameraSettings</td>
<td>Your app is an app associated with a camera and exposes its settings.</td>
<td>Hosted</td>
</tr>
</tbody>
</table>
**Note** The terms *app declaration, app extension, app activation,* and *contract* all relate to the exact same thing. That is, in your package, you must *declare* an app *extension,* allowing the system to *activate* your app. We say that your app implements a *contract* when it responds to an activation. The MSDN webpage that explains contracts and extensions, [http://msdn.microsoft.com/en-us/library/windows/apps/hh464906.aspx](http://msdn.microsoft.com/en-us/library/windows/apps/hh464906.aspx), is very inaccurate.

Figure 3-1 shows the relationship between various WinRT types that make up a running app, and Figure 3-2 shows a flowchart explaining how these various WinRT objects get created at runtime during app activation. You’ll want to periodically refer to these two figures as we continue the discussion.

**FIGURE 3-1** The relationship between various WinRT types that make up a running app.
When Windows needs to activate an app, it first displays a splash screen so that the user gets immediate feedback indicating that the app is starting. Windows gets the splash screen image and background color from the app’s manifest; this allows Windows to display the splash screen while the app is initializing. At the same time, Windows creates a process and loads the app’s code into it. After this, Windows creates the process’ primary thread and invokes a `Main` method. When you build a Windows Store app, a `Main` method is created for you automatically in an App.g.i.cs file. The `Main` method looks like this:

```
#if !DISABLE_XAML_GENERATED_MAIN
    public static class Program {
        static void Main(string[] args) {
            Windows.UI.Xaml.Application.Start((p) => new App);
        }
    }
#endif
```

As you can see, this method doesn’t do very much. When the process’ primary thread calls `Main`, it internally calls `Windows.UI.Xaml.Application's static Start method, which creates another thread called the main view thread. This thread then creates a `Windows.ApplicationModel.Core.CoreApplicationView` object that is your app’s main drawing surface. The `CoreApplicationView` object is associated with the main view thread and can be manipulated only by code executed by the main view thread. The main view thread then invokes the callback method passed as a parameter to `Application's Start method, which constructs an instance of your app’s `App` class. The

---

1 If you want to implement your own `Main` method and not use the XAML-generated one, you can do so by adding the `DISABLE_XAML_GENERATED_MAIN` conditional compilation symbol to your project’s build settings.
Application base class’ constructor stores a reference to your App object in a private static field, ensuring that it never gets garbage collected for the entire lifetime of the process. You can always get a reference to your app’s singleton App object by calling Application’s static Current property.

Important This App object is a singleton object that lives throughout the entire lifetime of the process. Because this object is never destroyed, any other objects directly or indirectly referred to by any static or instance fields will prevent those other objects from being garbage collected. Be careful about this because this can be a source of memory leaks.

After the App object singleton is created, the primary thread checks the ActivationKind value to see why the app is being activated. All the activations fall into one of two categories: main view activations or hosted view activations. (See the last column in Table 3-1.) Main view activations are what most developers are familiar with. A main view activation causes your app’s main window to become the foreground window and allows the user to interact with your app.

Hosted view activations are not as familiar to many people. In this case, an app wants to complete some operation leveraging some functionality provided by another app. The app the user is interacting with asks Windows to create a new window and then Windows activates the other app. This second app will create a small window that gets hosted inside Windows’ big window. This is why the activation is called a hosted view activation: the app is being activated to have its window hosted for use by another app. An example of a hosted view activation is when the user wants to share a webpage with a friend via the Mail app. Figure 3-3 shows the Bing News app as the main app the user is interacting with. If the user taps the Share charm and selects the Mail app, Windows creates a narrow, full-height window on the edge of the user’s screen. The header is displayed by Windows at the top of the window it created. The header contains the back arrow, app name (Mail), and logo. Underneath the header is a hosted view window created and managed by the Mail app itself.

Your App class is derived from the Windows.UI.Xaml.Application class, which defines some virtual methods as shown here:

```csharp
public class Application {
    // Override to know when the main view thread's or
    // a hosted view thread's window has been created
    protected virtual void OnWindowCreated(WindowCreatedEventArgs args);

    // Override any of these main view activations:
    protected virtual void OnLaunched(LaunchActivatedEventArgs args);
    protected virtual void OnSearchActivated(SearchActivatedEventArgs args);
    protected virtual void OnFileActivated(FileActivatedEventArgs args);

    // Override any of these hosted view activations:
    protected virtual void OnShareTargetActivated(ShareTargetActivatedEventArgs args);
    protected virtual void OnFileOpenPickerActivated(FileOpenPickerActivatedEventArgs args);
    protected virtual void OnFileSavePickerActivated(FileSavePickerActivatedEventArgs args);
    protected virtual void OnCachedFileUpdaterActivated(CachedFileUpdaterActivatedEventArgs args);
}
```
// Override this for less-frequently used main view (Protocol, Device,
// AppointmentsProvider, Contact, LockScreenCall) and hosted view (ContactPicker,
// PrintTaskSettings, CameraSettings) activations:
protected virtual void OnActivated(IActivatedEventArgs args);
}

FIGURE 3-3 The Bing News app sharing a news story via the Mail app's hosted view window.

Skype confirms it has developed 3D video calls.
The news was revealed by a senior executive during an exclusive interview with the BBC News.
There had been speculation about the development, but the company wanted to keep it confidential for workers unable to travel to the company's headquarters.
However, the executive warned that the technology was not yet ready for public consumption.
"We've done work in the labs for years developing a capability of 3D-screen and 3D video calls.

Skype confirms 3D tech research - Bing News

Add a message

Skype confirms 3D tech research

By Leo Kelion

BBC News - Thu Aug 29 07:30:00 UTC 2013

Skype has confirmed it has developed 3D video calls. The news was revealed by a senior executive ...
If you have Windows 8, open this in News.

http://www.bing.com/v7/id/KXKHp7ar1&m=en-us

Sent from Windows Mail

As soon as a main view or hosted view window is created, the thread creating the window calls the virtual OnWindowCreated method. If you override this method, the WindowsCreatedEventArgs object passed to it contains a reference to the thread's newly created window. In this method, you can register callback methods with any of the events (Activated, SizeChanged, VisibilityChanged, or Closed) it offers. After OnWindowCreated returns, one and only one of the other virtual methods is called, depending on why your app is being activated. The OnActivated method is called for the less-commonly used activation kinds.

Inside one of these virtual methods, you perform any initialization required for the specific kind of activation, create the desired user-interface element tree, set Window's Content property to the root of your user-interface element tree, and then activate the view's CoreApplicationView object, thereby bringing your app's window to the foreground so that the user can interact with it.

If your app is being activated due to a hosted view activation, your app's primary thread will create a hosted view thread. This thread then creates its own CoreApplicationView object that is your
app’s drawing surface while hosted. When the hosted view is no longer required by the hosting app,
your host CoreApplicationView window and the hosted view thread are destroyed. Every time your
app is activated with a hosted view activation, a new hosted view thread and CoreApplicationView
window are created. In fact, multiple apps could host your app simultaneously. For example, several
apps can host an app implementing the FileOpenPicker contract simultaneously. If this happens, your
app’s process will have one hosted view thread and CoreApplicationView window for each app
that is currently hosting your app. On the other hand, your app’s process will never have more than
one main view thread and main CoreApplicationView window.

While your app is running, it could be activated with more main view activations. This typically
happens if the user taps one of your app’s secondary tiles or a toast notification. In this case, the app
comes to the foreground but the act of tapping a tile or toast notification might direct the app to
show something special when brought to the foreground. When an already-running app is activated
with a new main view activation, the process’ primary thread will not create the main view thread
and its CoreApplicationView because these have already been created. Because the window has
already been created, the virtual OnWindowCreated method will not be called, but the proper virtual
method indicating why the main view is being re-activated will be called. This virtual method should
respond accordingly by deciding what UI to show and then activating the main view window so that
the user can interact with it.

![Important] Avoid registering event handlers inside a main view activation’s virtual method
because these methods can be called multiple times and you do not want to register mul-
tiple callbacks with a single event over the lifetime of your process. It can be OK to register
callback methods with events inside the OnWindowCreated method because this method is
called only once per thread/window.

Note that your app might not be running at all, and then a user can activate your app for a hosted
view. This causes your app’s primary thread to be created, and then a hosted view thread and its win-
dow are created. But your app’s main view thread and window are not created at this time. If the user
now activates your app with a main view activation, Windows will now create your app’s main view
thread and window, call the OnWindowCreated method, and then call the virtual method indicating
why your app is being activated with a main view activation.

**Managing the process model**

The previous section discussed how your app activates and initializes itself. In this section, we discuss
some core WinRT classes you should be aware of and how you can use them now that your app is
up and running. As you read this discussion, you might want to periodically refer back to Figure 3-1,
which shows the relationship between these classes.
WinRT offers a `Windows.ApplicationModel.Core.CoreApplication` class that looks like this:

```csharp
public static class CoreApplication {
    // Returns the CoreApplicationView associated with the calling thread
    public static CoreApplicationView GetCurrentView();

    // Returns all CoreApplicationViews existing within the process
    public static IReadOnlyList<CoreApplicationView> Views { get; }

    // Returns the main view thread's CoreApplicationView
    public static CoreApplicationView MainView { get; }

    // These events are discussed later in this chapter
    public static event EventHandler<Object> Resuming;
    public static event EventHandler<SuspendingEventArgs> Suspending;

    // These events are for debugging only
    public static event EventHandler<Object> Exiting;
    public static event EventHandler<UnhandledErrorDetectedEventArgs> UnhandledErrorDetected;

    // This method allows you to create multiple main view windows
    public static CoreApplicationView CreateNewView();

    // Some members not shown here...
}
```

As you can see, this class is a static class. This means that you cannot create instances of this class. So this static class manages your app as a whole. However, static classes don’t lend themselves to nice object-oriented programming features like inheritance and virtual methods. So, for XAML developers, WinRT also offers the `Windows.UI.Xaml.Application` class that we discussed earlier; this is the class that has all the virtual methods in it, making it easier for you to implement your activation code. In effect, the `Application` singleton object we discussed wraps the static `CoreApplication` class.

Now let me show you some of the other members of this `Application` class:

```csharp
public class Application {
    // Static members:
    public static void Start(ApplicationInitializationCallback callback);
    public static Application Current { get; }

    // The same Resuming & Suspending events offered by the CoreApplication class
    public event EventHandler<object> Resuming;
    public event SuspendingEventHandler Suspending;

    // XAML-specific properties and events:
    public DebugSettings DebugSettings { get; }
    public ApplicationTheme RequestedTheme { get; set; }
    public ResourceDictionary Resources { get; set; }
    public event UnhandledExceptionEventHandler UnhandledException;

    // The virtual methods shown earlier and some other members are not shown here...
}
```
Your App class derives from this Application class, inheriting all the instance members, and allows you to override the virtual methods.

Let’s go back to the CoreApplication class. This class has many members that return CoreApplicationView objects. Here is what the CoreApplicationView class looks like:

```csharp
public sealed class CoreApplicationView
{
    public CoreDispatcher Dispatcher { get; }
    public CoreWindow CoreWindow { get; }
    public Boolean IsMain { get; }
    public Boolean IsHosted { get; }

    public event TypedEventHandler<CoreApplicationView, IActivatedEventArgs> Activated;
}
```

As you can see, a CoreApplicationView object refers to a CoreDispatcher (the message pump that dispatches window messages) and a CoreWindow (the actual drawing surface), and it has an additional field indicating whether the CoreWindow is the app’s main window or one of the app’s hosted windows. There is also an Activated event that is raised when the window is being activated; the IActivatedEventArgs interface includes a Kind property, which returns one of the ActivationKind enumeration values (as shown in Table 3-1). Other members of this interface are described later in this chapter’s “Process lifetime management” section.

A CoreWindow object is a drawing surface, and it has associated with it the standard things you’d expect with a window. It has state (fields) indicating the bounding rectangle, whether input is enabled, which cursor to display, and whether the window is visible or not. It also offers events such as Activated, Closed, SizeChanged, VisibilityChanged, as well as keyboard and pointer (mouse, touch, and stylus) input events. And there are methods such as Activate, Close, Get(Async)KeyState, Set/ReleasePointerCapture, and a static GetForCurrentThread method.

For XAML developers, there is a sealed Windows.UI.Xaml.Window class that puts a thin wrapper around a CoreWindow object:

```csharp
public sealed class Window
{
    public static Window Current { get; } // Returns calling thread's Window
    public CoreWindow CoreWindow { get; }
    public CoreDispatcher Dispatcher { get; } // Same as CoreApplicationView.Dispatcher

    // The Content property is how XAML integrates with the window's drawing surface:
    public UIElement Content { get; set; }

    // This class exposes some of the same properties (Bounds, Visible)
    // This class exposes some of the same events (Activated, Closed,
    // SizeChanged, VisibilityChanged)
    // This class exposes some of the same methods (Activate, Close)
}
```
The final WinRT class to discuss here is the Windows.UI.Core.CoreDispatcher class, which looks like this:

```csharp
public sealed class CoreDispatcher {
    // Returns true if the calling thread is the same thread
    // that this CoreDispatcher object is associated with
    public Boolean HasThreadAccess { get; }

    // Call this to have the CoreDispatcher's thread execute the agileCallback
    // with a priority of Idle, Low, Normal, or High
    public IAsyncAction RunAsync(CoreDispatcherPriority priority, DispatchedHandler agileCallback);

    // Call this to get/set the priority of the code that dispatcher is currently executing
    public CoreDispatcherPriority CurrentPriority { get; set; }

    // Other members not shown...
}
```

Many .NET developers are already familiar with this CoreDispatcher class because it behaves quite similarly to the Dispatcher class found in Windows Presentation Foundation (WPF) and Silverlight. Because each CoreApplicationView has only one thread that manages it, its CoreDispatcher object lets you execute a method on that same thread, allowing the method to update that view's user interface. This is useful when some arbitrary thread calls one of your methods and you then need to update the user interface. I will talk more about the CoreDispatcher and show how to use it in other chapters.

A Windows Store app's main view can create additional views to show additional content. These views can be shown side by side on the same monitor and resized to the user's liking or shown on different monitors. For example, the Windows Mail app allows you to open new views, enabling you to refer to one mail message while composing another simultaneously. Apps can create new view threads and views by calling CoreApplication's static CreateNewView method. This method creates a new thread along with its own CoreDispatcher and CoreWindow, ultimately returning a CoreApplicationView. For this CoreApplicationView object, the IsMain and IsHosted properties both return false. Of course, when you create a new view, your App's OnWindowCreated virtual method is called via the new thread. Then you can create the UI for this new view using code like this:

```csharp
private async Task CreateNewViewWindow() {
    // Have Windows create a new thread, CoreDispatcher, CoreWindow, and CoreApplicationView
    CoreApplicationView cav = CoreApplication.CreateNewView();

    CoreWindow newAppViewWindow = null; // This will hold the new view's window

    // Have the new thread initialize the new view's content
    await cav.Dispatcher.RunAsync(CoreDispatcherPriority.Normal, () => {
        // Give the new thread's window back to the creating thread
        newAppViewWindow = Window.Current.CoreWindow;

        // Create the desired UI element tree and make it the content of the new window
        Window.Current.Content = new MyPage();
        Window.Current.Activate();
    });
}
```
// After the new thread initializes its view, the creating thread makes it appear
Int32 newAppViewId = ApplicationView.GetApplicationViewIdForWindow(newAppViewWindow);
await ApplicationViewSwitcher.TryShowAsStandaloneAsync(newAppViewId,
    ViewSizePreference.UseLess);
// The SDK documentation for Windows.UI.ViewManagement.ApplicationViewSwitcher explains
// its other methods, allowing you to control switching between your app's views.
}

The previous code leverages the Windows.UI.ViewManagement.ApplicationView class. This
class offers many dynamic properties related to a view. In other words, these properties' values
change frequently. The class looks like this:

```
public sealed class ApplicationView {
    // Gets the view for the calling thread
    public static ApplicationView GetForCurrentView();

    // Gets the unique window ID corresponding to a specific CoreWindow
    public static Int32 GetApplicationViewIdForWindow(ICoreWindow window);

    // Gets a unique ID identifying this view. NOTE: This ID is passed to
    // an XxxActivatedEventArgs' CurrentlyShownApplicationViewId property
    public Int32 Id { get; }

    // Gets/sets the view's title (shown in task switchers) & if PrtScn can capture its content
    public String Title { get; set; }
    public Boolean IsScreenCaptureEnabled { get; set; }

    // Read-only properties related to view's position & size
    public ApplicationViewOrientation Orientation { get; } // Landscape or Portrait
    public Boolean AdjacentToLeftDisplayEdge { get; }
    public Boolean AdjacentToRightDisplayEdge { get; }
    public Boolean IsFullScreen { get; }
    public Boolean IsOnLockScreen { get; }

    // Raised when the view is removed from task switcher (if user closes the view)
    public event TypedEventHandler<ApplicationView, ApplicationViewConsolidatedEventArgs>
        Consolidated;

    // Indicates if app terminates when all views close (Default=false)
    public static Boolean TerminateAppOnFinalViewClose { get; set; }
}
```

## XAML page navigation

Most XAML apps show the user a view with an initial page and then allow the user to navigate to
other pages within the view. This is similar to a website paradigm where users start at a website's
home page and then click on links to delve into specific sections of the website. Users are also quite
familiar with navigating back to pages they've seen before and, occasionally, after navigating back,
users navigate forward to a page they were just looking at. Windows Store apps typically offer this
same user experience. Of course, some Windows Store apps might just show a single page and, in this
case, navigation doesn't come into play at all.
In this section, I talk about the XAML support for page navigation and how to manage memory for this efficiently. Microsoft provides a WinRT class called Windows.UI.Xaml.Controls.Frame. An instance of this class manages a collection of UI pages allowing the user to navigate backward and forward through them. The class derives from ContentControl, which ultimately derives from UIElement, allowing you to assign a Frame object to Window's Content property to place XAML content on a drawing surface. The Frame class looks like this:

```csharp
public class Frame : ContentControl, INavigate {
    // Clears the stack from the next Page type to the end
    // and appends a new Page type to the stack
    public Boolean Navigate(Type sourcePageType, Object parameter);

    public Boolean CanGoBack { get; }    // True if positioned after the 1st Page type
    public void    GoBack();             // Navigates to the previous page type
    public Boolean CanGoForward { get; } // True if a Page type exists after the current position
    public void    GoForward();          // Navigates to the next Page type

    // These members return the stack's content and size
    public IList<PageStackEntry> BackStack { get; }
    public Int23 BackStackDepth { get; }

    // Member to serialize/deserialize the stack's types/parameters to/from a string
    public String GetNavigationState();
    public void   SetNavigationState(String navigationState);

    // Some members not shown
}
```

Frame objects hold a collection of Windows.UI.Xaml.Controls.Page-derived types. Notice that they hold Page-derived *types*, not Page-derived *objects*. To have the Frame object navigate to a new Page-derived object, you call the Navigate method, passing in a reference to a System.Type object that identifies the page you want to navigate to. Internally, the Navigate method constructs an instance of the Page-derived type and makes this object be the content of the Frame object, allowing the user to interact with the page's user interface. Your Page-derived types must derive from Windows.UI.Xaml.Controls.Page, which looks like this:

```csharp
public class Page : UserControl {
    // Returns the Frame that "owns" this page
    public Frame Frame { get; }

    // Invoked when the Page is loaded and becomes the current content of a parent Frame
    protected virtual void OnNavigatedTo(NavigationEventArgs e);

    // Invoked after the Page is no longer the current content of its parent Frame
    protected virtual void OnNavigatedFrom(NavigationEventArgs e);

    // Gets or sets the navigation mode that indicates whether this Page is cached,
    // and the period of time that the cache entry should persist.
    public NavigationCacheMode NavigationCacheMode { get; set; }

    // Other members not shown
}
```
After the Frame object constructs an instance of your Page-derived type, it calls the virtual OnNavigatedTo method. Your class should override this method and have it perform any initialization for the page. When you call Frame’s Navigate method, you get to pass an object reference as a parameter. Your Page-derived object can get the value of this parameter type by querying NavigationEventArgs’s read-only Parameter property. This gives you a way to pass some data from the code when navigating to a new page. For reasons that will be described later, in the “Process lifetime management” section, the value you pass should be serializable.

Page objects can be very expensive in terms of memory consumption because pages tend to have many controls and some of these controls are collection controls, which might manage many items. When the user navigates to a new Page, keeping all the previous Pages with all their child objects in memory can be quite inefficient. This is why the Frame object maintains Page types, not instances of Page objects. When the user navigates to another Page, the Frame removes all references to the previous page object, which allows the page object and all its child objects to be garbage collected, freeing up what can potentially be a lot of memory. Then, if the user navigates back to a previous page, the Frame constructs a new Page object and calls its OnNavigatedTo method so that the new Page object can initialize itself, reallocating whatever memory it needs.²

This is all fine and good but what if your Page needs to record some state in between being garbage collected and re-initialized? For example, the user might have entered some text in a TextBox control or scrolled to and selected a specific item in a ListView or GridView control. When the Page gets garbage collected, all of this state is destroyed by the garbage collector. So, when the user navigates away from a Page, in the OnNavigatedFrom method, you need to preserve the minimal amount of state necessary in order to restore the Page back to where it was before the user navigated away from it. And this state must be preserved in a place where it will not get garbage collected.

The recommended practice is to have your App singleton object maintain a collection of dictionaries; something like a List<Dictionary<String, Object>>. You have one dictionary for each page managed by the Frame, and each dictionary contains a set of key/value pairs; use one key/value pair for each piece of page state you need to persist. Now, because your App singleton object stays alive for the lifetime of your process, it keeps the collection alive and the collection keeps all the dictionaries alive.

When navigating to a new page, you add a new dictionary to the list. When navigating to a previous page, look up its dictionary in the list using Frame’sBackStackDepth property. Figure 3-4 shows what objects you should have in memory after the app navigates to Page_A. The Frame object has a single Page_A type in its collection along with its navigation parameter, and our list of dictionaries has just one dictionary in it. Notice that the Page_A object can reference the dictionary, but you must make sure that nothing in the App singleton object refers to any page object because this prevents the page object from being garbage collected. Also, avoid registering any of the page’s instance methods with external events because this also prevents the page object from ever being garbage.

² If you are less concerned about memory conservation, you can override this default behavior and have the Frame object keep your page objects in memory by setting your Page object’s NavigationCacheMode property. See the SDK documentation for details.
collected. Or, if you do register any instance methods with events, make sure you unregister them in the **OnNavigatedFrom** method.

![Diagram](image)

**FIGURE 3-4** The Page_A object persists its state in the first dictionary in the list.

Now, if the user navigates to Page_B, the Frame constructs a Page_B object, makes it the current contents of the Frame, and calls its **OnNavigatedTo** method. In the **OnNavigatedTo** method, we add another dictionary to the list, and this is where the page instance persists its state. Figure 3-5 shows what objects you should have in memory after the user navigates from Page_A to Page_B.

![Diagram](image)

**FIGURE 3-5** The Page_A object can be garbage collected, and the new Page_B object persists its state in the second dictionary in the list.

From here, the user might navigate from Page_B back to Page_A. Doing so would cause the Page_B object to be garbage collected, and a new Page_A object would be created, which would refer to the first dictionary in the list. Or, from Page_B, the user might navigate to a new Page_A object whose content is populated based on the navigation parameter passed to **OnNavigatedTo** and extracted via **NavigationEventArgs**'s **Parameter** property. Figure 3-6 shows what objects you should have in memory after the user navigates forward from Page_B to a new Page_A.
FIGURE 3-6 The second Page_A object persists its state in the third dictionary in the list.

Now, if the user navigates backward from the new Page_A to Page_B, the Frame object removes its reference to the Page_A object, allowing it to be garbage collected. But the dictionary maintains that instance of Page_A’s state so that it can restore its state should the user later navigate forward again from Page_B to a new Page_A object. Similarly, the user can navigate back and forth throughout all the page types in Frame’s collection. Navigating to a page constructs a new page, restoring its state from the dictionary. By the way, if the user is currently at the first Page_A and then, from this page, the app decides to navigate to Page_C, then the dictionaries beyond the current page must be removed from the list (allowing them to be garbage collected) because the user is navigating down a whole different branch of the app’s user interface now.

With this model in place, memory is used very efficiently by your app. There is another benefit we get when using this model, which is described later in the “Process lifetime management” section of this chapter. By the way, some of the Visual Studio templates for creating Windows Store apps spit out source code for a SuspensionManager class that manages page instance state. This class is not a WinRT class, and it is not part of Windows; the source code for the class is injected into your Visual Studio project when you create it.

Personally, I do not use the SuspensionManager class in my own projects. Instead, I created my own FramePageStateManager class that, in my opinion, is better. It has a cleaner interface and also leverages some helper classes that put a type-safety wrapper around each dictionary, giving you support for IntelliSense, compile-time type safety, and data binding. These additional features greatly simplify the effort of coding your app and managing its state. The code to manage it all is part of the Process Model app that is available with the downloadable code that accompanies this book; see http://Wintellect.com/Resource-WinRT-Via-CSharp.
Process lifetime management

Back when the Windows operating system (OS) was first created (in the early 1980s), there were no computers that ran on battery power. Instead, all computers were plugged into an AC power source, which effectively meant that there was an infinite amount of power to draw on. Because power was in infinite supply, Windows allowed apps to run all the time. Even when the user was not interacting with the app, the app was allowed to consume power-consuming resources such as CPU time, disk I/O, and network I/O.

But today, users want mobile computer systems that do run on battery power and they want the battery to last as long as possible between charges. For Windows to meet user demands, Windows Store apps are allowed to consume system resources (and power) only when the user is interacting with the app; when the user switches away from a Windows Store app, the OS suspends all threads in the process, preventing the app from executing any more of its code, and this prevents consumption of power.

In addition, the original version of Windows was designed for keyboard and mouse input only. But nowadays, users demand systems that use more intuitive and natural touch-based input. When using a mouse as an input device, users are more likely to tolerate a lag. For example, when paging down in a document, the user can click the mouse on a scroll bar and then, after releasing the mouse button, the document scrolls. The user clicks and then the document scrolls. But, with touch input, the document needs to scroll as the user swipes his finger. With touch, users won’t tolerate a lag between swiping and the document scrolling. When apps are allowed to run and consume resources when the user is not interacting with them, these apps can take resources away from the app the user is interacting with, negatively affecting the performance and introducing lag for the user. This is another reason why Windows Store apps have all their threads suspended when the user is not interacting with them.

Furthermore, Windows puts a lot of time restrictions on Windows Store apps. If your app does not meet a time restriction, the OS terminates your app, bringing the user back to the Start screen where he can relaunch your app or run another app that performs more satisfactorily.

Figure 3-7 shows the lifetime of a Windows Store app. When your app is activated, Windows immediately shows your app’s splash screen (as specified in your app’s manifest file). This gives the user immediate feedback that your app is initializing. While the splash screen is visible, Windows invokes your app’s Main method and runs through all the activation steps as described at the beginning of this chapter. One of the last things your app does after initializing is activate its window (drawing surface) by calling Windows.UI.Xaml.Window’s Activate method. If your app does not call this method within 15 seconds, the OS terminates your app and returns the user to the Start screen.\(^3\) While the OS gives your app 15 seconds to activate its window, your app must actually activate its window within 5 seconds in order to pass Windows Store certification. So you really should design your app to complete its initialization and activate its window within 5 seconds, not 15 seconds.

\(^3\) Actually, Windows terminates your app only if the user navigates away from its splash screen. If the user leaves the splash screen in the foreground, the app is not terminated.
**FIGURE 3-7** Lifetime of a Windows Store app.

If your app needs more than 5 seconds to initialize, you can implement an *extended splash screen* as shown in the Process Model app available with the downloadable code that accompanies this book. This means that your app is activating a window that looks similar to the splash screen during its initialization. But, because you activated a window, the OS believes that your app is alive and well and it will not terminate your app now. Because you are in control of this window, you can show the user a progress ring or use other UI affordances to indicate to the user that your app requires more time to initialize. For an example of an app that shows an extended splash screen, see the Skype app that comes with Windows.

If your app displays content such as news articles, your app can bring up an empty wireframe or grid that gets populated as data flows in from the network. In this scenario, your app does not require an extended splash screen; the user can start interacting with it immediately.

**Windows Store app suspension**

When the user switches away from your app, the OS suspends all the threads in your process. You can see this for yourself in Task Manager (shown in Figure 3–8). First, in Task Manager, select the View menu’s Status Values option and make sure that Show Suspended Status is selected. Then launch multiple Windows Store apps. After a few seconds, Task Manager shows a status of Suspended for any apps whose threads are suspended. For suspended apps, you’ll also notice that their CPU, Disk, and Network consumption all go to 0. Of course, memory is not impacted because these apps are still resident in memory.
When the user switches back to a suspended app, the system simply resumes the app's threads and allows the app to interact with the user again. This is great, but what if your app shows real-time data like temperature, stock prices, or sports scores? Your app could have been suspended for weeks or maybe months. In this case, you wouldn’t want your app to simply resume and show the user stale data. So WinRT's Application base class offers a Resuming event (which really just wraps CoreApplication's Resuming event). When an app is resumed, this event is raised and your app can refresh its data to show the user current information. To know how long your app was suspended, query the time in the Suspending event and subtract this value from the time obtained in the Resuming event; there might be no need to refresh data if only a small amount of time passed. There is no time restriction placed on your Resuming event's callback method. Many apps do not show real-time data, so many apps have no need to register with the Resuming event.

Important If Windows suspends your app and subsequently activates it with a hosted view activation (such as Share), Windows does not resume all the threads in your app; the main view thread remains suspended. This can lead to blocking threads if you attempt to perform any kind of cross-thread communication.

Windows Store app termination
In this chapter, we’ve talked a lot about how to efficiently manage memory used by your app. This is critically important because many mobile PCs do not have the amount of memory that desktop computers traditionally have. But, even if all Windows Store apps manage their memory as described in this chapter, there is still a chance that the user could start many Windows Store apps and the system
will still run out of memory. At this point, a user has to close some currently running app in order to run some new app. But which apps should the user close? A good choice is the one using the most amount of memory, but how does the user know which app this is? There is no good answer to this question, and even if there was, it puts a big burden on the user to figure this stuff out and to manage it.

So, for Windows Store apps, Microsoft has taken this problem away from the user and has instead solved the problem in the OS itself—although you, as a software developer, must also contribute effort to solving the problem. When available memory is running low, Windows automatically terminates a Windows Store app that the user is not currently interacting with. Of course, the user is not aware that this has happened because the user is not interacting with the app. The system remembers that the app was running and allows the user to switch back to the app via the Windows Store apps task list (Windows key+Tab). When the user switches back to the app, the OS automatically relaunches the app so that the user can interact with the app again.

**Note** The less memory your app uses, the less likely the OS is to terminate it.

Of course, an app uses its memory to maintain state on behalf of the user. And, when the OS terminates an app, the memory is freed up and therefore the state is discarded. This is where you, as a developer, come in. Before your app is terminated, it must save its state to disk and, when your app is relaunched, it must restore its state. If your app does this correctly, it gives the illusion to the user that your app was never terminated and remained in memory the whole time (although your app's splash screen is shown while your app re-initializes). The result is that users do not have to manage an app’s lifetime; instead, the OS works with your app to manage it, resulting in a better end-user experience. Again, this is especially useful with mobile PCs, which have limited amounts of memory.

Earlier, we talked about the Resuming event and how it is raised when the OS resumes your app’s threads. Well, the WinRT Application base class also offers a Suspending event (which really just wraps CoreApplication’s Suspending event). Just before an app’s threads are suspended, this event is raised so that your app can persist its state out to a file on the user’s disk. Windows gives your app 5 seconds to complete its suspension; if you take longer than this, Windows just terminates your app. Although Windows gives you 5 seconds, your suspension must actually complete within 2 seconds to be certified for the Window Store. If you follow the model described in the “XAML page navigation” section of this chapter, you are in great shape because all you have to do in your suspension code is create a file on disk and serialize the list of dictionaries into it. You’ll also need to call your Frame object’s GetNavigationState method, which returns a String that has encoded in it the

---

4 When an app goes to the background, Windows waits a few seconds before raising the Suspending event. This gives the user a few seconds to switch back to the app in case the user switched away from it by accident.

5 If you need more time than 2 seconds to complete your suspension, you could look at Window’s VisibilityChanged event. This event is raised whenever a window becomes visible or invisible. A window always becomes invisible first before the app is suspending and its Suspending event is raised.
collection of pages the user built up while navigating through your app; serialize this string out to the file as well.\(^6\)

While your app is suspended, the OS might terminate it to free up memory for other apps. If the OS chooses to terminate your app, your app is given no additional notification; it is simply killed. The reason is obvious: if the system allowed your app to execute code before termination, your app could allocate more memory, making the situation worse. The main point to take away from this is that your app must save its state when it receives the Suspending event because your app will not be given a chance to execute more code if the OS decides to terminate it.

Even if the OS terminates your app, it gives the illusion to the user that your app is still running and allows the user to switch back to your terminated app. Figure 3-9 shows the system’s task list and Task Manager after the App1 app has been terminated. Notice that the task list shows the App1 app, allowing the user to switch to it.\(^7\) However, Task Manager does not show any entry for the App1 app at all because it is no longer resident in memory.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{task_manager.png}
\caption{The Windows task list showing running, suspended, and terminated apps while Task Manager shows only running and suspended apps.}
\end{figure}

When the user switches back to a terminated app, the OS performs a main view activation of the app (showing its splash screen). The app must now initialize itself and restore its state back to what it

---

\(^6\) GetNavigationState internally calls the page’s OnNavigateFrom method so that it can store any state in its dictionary before GetNavigationState returns its encoded String. The format of the string is undocumented; do not write code that parses or interprets the string in any way.

\(^7\) The task list shows the contents of the app’s view if the app is still running and shows the default logo for the app if Windows terminated it.
was before the app was terminated. The fact that the app got terminated should be transparent to the user. This is an important point. As far as the user is concerned, your app never stopped running: whether it is running, suspended, or even terminated, your app is available to the user.

When your app is activated, your app’s Main method runs, the main view thread is created, your App’s constructor executes, and then Application’s virtual OnWindowCreated method is called, followed by one of the other virtual OnXxx methods (depending on why your app is being re-activated). If your app is being activated with a hosted view activation, there is no need to restore your app’s state to what it was when it was suspended. But, when your app starts due to a main view activation, you’ll need to find out if your app is being re-activated because the OS terminated it.

All the virtual OnXxx methods are passed a parameter whose type implements the IActivatedEventArgs interface. This interface has a PreviousExecutionState property that returns an ApplicationExecutionState value. This type is an enumerated type, and if the PreviousExecutionState property returns ApplicationExecutionState.Terminated, your app knows that it’s being relaunched because the OS terminated it. At this point, your code should open the file on the user’s disk where you previously serialized the app’s state, deserialize the list of dictionaries, and then grab the String with the encoded frame pages in it and pass it to your Frame object’s SetNavigationState method. When you call SetNavigationState, it resets the state of the Frame object back to what it was when your app was suspended so that the user will be looking at the exact same thing she was looking at when the app got suspended. To the user, it looks like your app never terminated.

Note that memory pressure is not the only reason your app can terminate. The user can close your app by typing Alt+F4, dragging your app’s window from the top of the screen to the bottom and holding for a few seconds, or right-clicking your app in the task list and selecting Close. In addition, the OS closes all apps when the user logs off or shuts down the machine. In all the scenarios just given, the OS does raise the Window’s VisibilityChanged event, followed by the App’s Suspended event, giving your app a chance to save its state. However, in the future, when your app is launched, you should not restore your app’s state because the user has explicitly taken action to close your app as opposed to the OS implicitly terminating your app. If you check the PreviousExecutionState property, you’ll see that in all these scenarios, it returns ApplicationExecutionState.ClosedByUser.

Users can also forcibly kill an app using the Task Manager and, of course, an app can kill itself by throwing an unhandled exception. In addition, Windows will automatically kill an app if it’s running when the user uninstalls it or if the system updates the app to a newer version. In all these scenarios, when the app relaunches in the future, it should just initialize itself and not restore any previous state because state might have gotten corrupted, which is what might have caused the unhandled exception in the first place. If you check the PreviousExecutionState property, you’ll see that in these scenarios, it returns ApplicationExecutionState.NotRunning.

---

8 This does not always make sense for every app. For some apps, if they are suspended for a long time, the user might not remember or care about what she was last doing with the app. In this case, your app can just initialize itself and not restore any previous user state. You might take this approach for a newsreader app where the article might be stale or a weather app where the data is stale.

9 SetNavigationState internally calls the page’s OnNavigatedTo method so that the page can load state from its dictionary back into its UI.
**Note** Windows Store apps are not supposed to close themselves or offer any kind of UI that allows the user to close the app. If your app violates this rule, it will not pass Windows Store certification. The CoreApplication class offers an Exit method and an Exiting event. These members are for use in debug scenarios during app development only, such as memory-leak detection, unit testing, and so on. When you submit your app to the Windows Store for certification, your app must not use these members. To discourage the use of these members, the Windows.UI.Xaml.Application does not wrap these members; therefore, they are not easily available to your App class.

**How to best structure your app class' code**

I know that all the information presented in this chapter can be difficult to take in, memorize, and turn into correctly implemented code. So, to simplify things, I’ve created an AppAid class that encapsulates a lot of this knowledge and makes building new Windows Store apps easier. Here is what this class looks like:

```csharp
namespace Wintellect.WinRT.AppAids {
    public enum ViewType { None, Main, Hosted, Auxiliary }
    public enum LaunchReason { PrimaryTile, SecondaryTile, Toast, Proximity }

    public static class AppAid {
        private static ApplicationInitializationCallback m_appInitCallback;
        private static Func<Frame, IActivatedEventArgs, Task<bool>> s_deserializeFramePageStateAsync;

        /// <summary>Call this method from Main instead of calling Application.Start</summary>
        /// <param name="callback">The callback that constructs the App singleton object.</param>
        /// <param name="deserializeFramePageStateAsync">A callback that restores the user's session state. Called during 1st main view activation if the app was previously terminated.</param>
        public static void Start(ApplicationInitializationCallback callback,
            Func<Frame, IActivatedEventArgs, Task<bool>> deserializeFramePageStateAsync = null) {
            // Invoked via process' primary thread each time the process initializes
            s_deserializeFramePageStateAsync = deserializeFramePageStateAsync;
            m_appInitCallback = callback;
            Application.Start(AppInitialization);
        }

        private static void AppInitialization(ApplicationInitializationCallbackParams p) {
            // Invoked via main view thread
            // But the main view's CoreWindow & CoreDispatcher do NOT exist yet;
            // they are created by Application.Start after this method returns
            m_appInitCallback(p); // Creates a singleton App object that never gets GC'd
            // because the base class (Application) holds a reference to it
            m_appInitCallback = null; // Allow delegate to be GC'd
        }
    }
}
```
/// <summary>Call this method from inside App's OnWindowCreated method to determine
/// what kind of window is being created.</summary>
/// <returns>The view type (main or hosted) for this kind of activation.</returns>
public static ViewType OnWindowCreated(this WindowCreatedEventArgs args) {
    // Invoked once via main view thread and once for each hosted view/auxiliary thread
    // NOTE: You can't tell what kind of activation (Share, Protocol, etc.) is occurring.
    return ViewType;
}

/// <summary>This method returns the kind of view for a given activation kind</summary>
/// <param name="args">Indicates what kind of activation is occurring.</param>
/// <returns>The view type (main or hosted) for this kind of activation.</returns>
public static ViewType GetViewType(this IActivatedEventArgs args) {
    switch (args.Kind) {
    case ActivationKind.AppointmentsProvider:
        String verb = ((IAppointmentsProviderActivatedEventArgs)args).Verb;
        if (verb == AppointmentsProviderLaunchActionVerbs.AddAppointment) return ViewType.Hosted;
        if (verb == AppointmentsProviderLaunchActionVerbs.ReplaceAppointment) return ViewType.Hosted;
        if (verb == AppointmentsProviderLaunchActionVerbs.RemoveAppointment) return ViewType.Hosted;
        if (verb == AppointmentsProviderLaunchActionVerbs.ShowTimeFrame) return ViewType.Main;
        break;
    case ActivationKind.Contact:
        verb = ((IContactsProviderActivatedEventArgs)args).Verb;
        if (verb == ContactLaunchActionVerbs.Call) return ViewType.Main;
        if (verb == ContactLaunchActionVerbs.Map) return ViewType.Main;
        if (verb == ContactLaunchActionVerbs.Message) return ViewType.Main;
        if (verb == ContactLaunchActionVerbs.Post) return ViewType.Main;
        if (verb == ContactLaunchActionVerbs.VideoCall) return ViewType.Main;
        break;
    case ActivationKind.Launch:
        case ActivationKind.Search:
        case ActivationKind.File:
        case ActivationKind.Protocol:
        case ActivationKind.Device:
        case ActivationKind.LockScreenCall:
            return ViewType.Main;
        break;
    case ActivationKind.ShareTarget:
        case ActivationKind.FileOpenPicker:
        case ActivationKind.FileSavePicker:
        case ActivationKind.CachedFileUpdater:
        case ActivationKind.ContactPicker:
        case ActivationKind.PrintTaskSettings:
        case ActivationKind.CameraSettings:
            return ViewType.Hosted;
    }
    throw new ArgumentException("Unrecognized activation kind");
}
public static ViewType ViewType {
    get {
        try {
           CoreApplicationView cav = CoreApplication.GetCurrentView();
            return cav.IsMain ? ViewType.Main :
                (cav.IsHosted ? ViewType.Hosted : ViewType.Auxiliary);
        } catch { return ViewType.None; }
    }
}

/// <summary>Whenever you override one of App's virtual activation methods
/// (eg: OnLaunched, OnFileActivated, OnShareTargetActivated), call this method.
/// If called for the 1st Main view activation, sets Window's Frame,
/// restores user session state (if app was previously terminated), and activates window.
/// If called for a Hosted view activation, sets Window's Frame & activates window.
/// </summary>
/// <param name="args">The reason for app activation</param>
/// <returns>True if previous state was restored; false if starting fresh.</returns>
public static async Task<Boolean> ActivateViewAsync(this IActivatedEventArgs args) {
    Window currentWindow = Window.Current;
    Boolean previousStateRestored = false; // Assume previous state is not being restored
    if (args.GetViewType() == ViewType.Main) {
        if (currentWindow.Content == null) {
            currentWindow.Content = new Frame();
        }
        // The UI is set; this is the 1st main view activation or a secondary activation
        // If not 1st activation,
        // PreviousExecutionState == ApplicationExecutionState.Running
        if (args.PreviousExecutionState == ApplicationExecutionState.Terminated
            && s_deserializeFramePageStateAsync != null) {
            // Restore user session state because app relaunched after OS termination
            previousStateRestored =
                await s_deserializeFramePageStateAsync(CurrentFrame, args);
            s_deserializeFramePageStateAsync = null;   // Allow delegate to be GC'd
        } else {
            currentWindow.Content = new Frame();
        }
    }
    currentWindow.Activate();  // Activate the MainView window
    return previousStateRestored;
}

/// <summary>Returns the Frame in the calling thread's window.</summary>
public static Frame CurrentFrame { get { return (Frame)Window.Current.Content; } }

public static LaunchReason GetLaunchReason(this LaunchActivatedEventArgs args) {
    if (args.Arguments == ProximityLaunchArg) return LaunchReason.Proximity;
    if (args_TILE_ID == Windows.ApplicationModel.Core.CoreApplication.Id) {
        return (args.Arguments == String.Empty)
            ? LaunchReason.PrimaryTile : LaunchReason.Toast;
    }
    return LaunchReason.SecondaryTile;
}
The code that accompanies this book has a souped-up version of the AppAid class. The souped-up version supports extended splash screens and thread logging, and it has some navigation helpers. Here is some code for a sample App class that uses my AppAid class. The code calls some additional methods that I provide in the code that accompanies this book to simplify saving and restoring user session state in case of app termination. The most important part of the following code is the comments.

```
// Our singleton App class; store all app-wide data in this class object
public sealed partial class App : Application {
    // Invoked because DISABLE_XAML_GENERATED_MAIN is defined:
    public static void Main(String[] args) {
        // Invoked via process' primary thread each time the process initializes
        AppAid.Start(AppInitialization,
            (f, a) => f.DeserializePageStateAsync(c_FramePageStateFileName, a));
    }

    private static void AppInitialization(ApplicationInitializationCallbackParams p) {
        // Invoked via main view thread
        // But the main view's CoreWindow & CoreDispatcher do NOT exist yet;
        // they are created by Application.Start after this method returns

        // Create a singleton App object. It never gets GC'd because the base class (Application)
        // holds a reference to it obtainable via Application.Current
        var app = new App();
    }

    private App() {
        // Invoked via main view thread; CoreWindow & CoreDispatcher do NOT exist yet
        this.InitializeComponent();
        this.Resuming += OnResuming;   // Raised when main view thread resumes from suspend
        this.Suspending += OnSuspending; // Raised when main view thread is being suspended
        // TODO: Add any additional app initialization
    }

    private void OnResuming(Object sender, Object e) {
        // Invoked via main view thread when it resumes from suspend
        // TODO: Update any stale state in the UI (news, weather, scores, etc.)
    }

    private void OnSuspending(Object sender, SuspendingEventArgs e) {
        // Invoked via main view thread when app is being suspended or closed by user

        // Windows gives 5 seconds for app to suspend or OS kills the app
        // Windows Store certification requires suspend to complete in 2 seconds

        // TODO: Save session state in case app is terminated
        // (see ApplicationData.Current.LocalFolder)
        // NOTE: I perform this operation synchronously instead of using a deferral
        this.GetCurrentFrame().SerializePageStateAsync(c_FramePageStateFileName)
            .GetAwaiter().GetResult();
    }
```
protected override void OnWindowCreated(WindowCreatedEventArgs args) {
    // Invoked once via the main view thread and once for each hosted view thread
    // NOTE: In here, you do not know the activation kind (Launch, Share, Protocol, etc.)
    switch (args.OnWindowCreated()) {
        case ViewType.Main:
            // TODO: Put code here you want to execute for the main view thread/window
            break;
        case ViewType.Hosted:
            // TODO: Put code here you want to execute for a hosted view thread/window
            break;
        case ViewType.Auxiliary:
            // TODO: Put code here you want to execute for an auxiliary view thread/window
            break;
    }

    // Optional: register handlers with these events
    Window w = args.Window; // Refers to the view's window (drawing surface)
    w.Activated += Window_Activated;
    w.VisibilityChanged += Window_VisibilityChanged;
}

private void Window_Activated(Object sender, WindowActivatedEventArgs e) {
    // Invoked via view thread each time its window changes activation state
    CoreWindowActivationState activateState = e.WindowActivationState;
}

private void Window_VisibilityChanged(Object sender, VisibilityChangedEventArgs e) {
    // Invoked via view thread each time its window changes visibility
    // A window becomes not-visible whenever the app is suspending or closing
    if (!e.Visible) return;
}

protected override async void OnLaunched(LaunchActivatedEventArgs args) {
    Boolean previousStateRestored = await args.ActivateViewAsync();
    switch (args.GetLaunchReason()) {
        case LaunchReason.PrimaryTile:
            if (previousStateRestored) {
                // Previous state restored back to what it was
                // before app was terminated; nothing else to do
            } else {
                // Previous state not restored; navigate to app's first page
                // TODO: Navigate to desired page
            }
            break;

        case LaunchReason.SecondaryTile:
            // TODO: Navigate to desired page
            break;

        case LaunchReason.Toast:
            // TODO: Navigate to desired page
            break;
    }
}
case LaunchReason.Proximity:
    // TODO: Navigate to desired page
    break;
}

Debugging process lifetime management

When debugging, Windows will not suspend or terminate a Windows Store app because this would lead to a poor debugging experience. This makes it impossible for you to debug and step through your app’s Resuming and Suspending event handlers. So, to allow you to debug these event handlers, Visual Studio offers a way to force suspending, resuming, and terminating your app. While your app is running, go to Visual Studio’s Debug Location toolbar and select the operation you want to force, as shown in Figure 3-10. You might also want to use the PLMDebug tool, which you can download with the Debugging Tools for Windows. This tool allows you to turn off PLM for your app so that you can attach a debugger and debug the app without the OS suspending it.

![FIGURE 3-10 Forcing an app to suspend, resume, or terminate using Visual Studio’s Debug Location toolbar.](image)

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using Windows.ApplicationModel;
```
Index

Numbers and symbols
~ (tilde), 39

A
Account Picture Provider app declaration, 33
Activated event, 57, 197–199
activating Windows Store apps
    app declarations, 32–34
    hosted view, 53
    launch activation, 44, 50
    main view, 53, 55
    process overview, 49–55
    share target apps, 242
    time considerations, 64–65
    toast notifications, 50, 197–198
ActivationKind enumeration, 50, 53, 57
Add-AppxPackage PowerShell script, 40–41
Advanced Query Syntax (AQS), 117
age rating for apps, 250, 254
Alarm app declaration, 34
AlarmApplicationManager class, 33
AllowAllTrustedApps Group Policy setting, 42
animating tile contents, 190–191
app activation. See activating Windows Store apps
app containers, 3, 271–273
app declarations
    about, 32–33
    activating apps, 44, 49–55
    adding, 213–219
    listed, 33–34
app development, structuring class code, 70–75
app licenses, 262–266
app logos, 184–186
app packages
    about, 25
    accessing files or folders, 97
    associating with reserved package name, 43
    building, 34–40
    debugging, 46–48
    deploying, 6, 40–44
    manifest file. See manifest file
    package data. See package data
    Package Explorer desktop app, 45–46
    package files. See package files
    package identity. See package identity
    privacy policy, 147, 254
    project files, 25–27
    size considerations, 250
    staging and registration, 44–45
App singleton object, 52–53, 57, 61
AppCert.exe tool, 252
App.g.i.cs file, 52
Application class
    about, 56–57, 70
    Current property, 53
    OnActivated method, 54, 107, 238
    OnFileActivated method, 104, 106, 238
    OnLaunched method, 104, 193, 197–199, 238
    OnSearchActivated method, 238
    OnShareTargetActivated method, 241–242, 244
    OnWindowCreated method, 54–55, 58, 69, 238
    Resuming event, 66, 75
    Start method, 52
    Suspending event, 67–68, 75
    virtual methods, 53–54, 56–57
Application Display Name package property, 29, 43
application models. See desktop apps; Windows Store apps
Application Security ID (SID), 272
ApplicationData class

about, 12, 79, 81, 95
ClearAsync method, 84, 88
Current property, 80
DataChanged event, 88–89, 206
LocalFolder property, 96
RoamingFolder property, 86, 96
RoamingSettings property, 86
RoamingStorageQuota property, 86
SetVersionAsync method, 84, 85n
SignalDataChanged method, 89, 206, 225
TemporaryFolder property, 96
Version property, 84
ApplicationDataCompositeValue class, 81–82, 88
ApplicationDataContainer class, 88
ApplicationDataContainerSettings class, 81
ApplicationDataLocality enumeration, 88
ApplicationDataManager class, 46
ApplicationExecutionState enumeration, 69
ApplicationManager class, 82
Applications And Services Logs, 88
ApplicationView class, 58, 237n
Appointment Provider app declaration, 34
AppointmentManager class, 33
.appx package file, 37–39
App.xaml.cs file, 27
AppxBlockMap.xml file, 38
.appxbundle package file, 39
AppxManifest.xml file, 38
.appxsym file, 37
.appxupload file, 43
AQS (Advanced Query Syntax), 117
AssemblyInfo.cs file, 26
Assets folder, 26
async keyword, 18, 20, 85n
asynchronous operations
  blocking threads, 21, 92–93
  calling asynchronous methods, 18–19
  calling from .NET code, 16–17
  cancellation and progress, 19–21
  WinRT deferrals, 21–23
AtomPubClient class, 123
authentication, 168, 202, 248
automatic updates, 247n
AutoPlay Content app declaration, 33
AutoPlay Device app declaration, 33
await keyword, 18

B
background tasks
about, 191, 205
accessing user files, 97
adding manifest declarations, 213–219
architectural overview, 205–207
canceling, 227–228
debugging, 222–223
deploying new versions of apps, 225
determining code triggers, 208–213
hanging, 227
implementing code, 207–208
language considerations, 214
latched, 221
lock screen and, 184–185, 214–216
polite reader issues, 132
raw notifications, 200, 209
registering, 207, 219–222
resource quotas, 223–224
single app packages, 27
time limits, 23
toast notifications, 198
tracking progress and completion, 225–227
usage considerations, 205
versioning package data, 84
Background Tasks app declaration, 33
background transfer feature, 154–160
BackgroundAccessStatus enumeration, 218–219
BackgroundDownloader class
about, 155–156
Cost property, 156
CreateDownload method, 156
encrypting data traversing networks, 181
GetCurrentDownloadsAsync method, 159
GetCurrentDownloadsForTransferGroupAsync method, 156
Method property, 156
ProxyCredential property, 156
RequestUnconstrainedDownloadsAsync method, 156
ServerCredential property, 156
TransferGroup property, 156
BackgroundExecutionManager class
GetAccessStatus method, 218
RemoveAccess method, 219
RequestAsyncAccess method, 218–219
BackgroundTaskBuilder class
about, 33
CancelOnConditionLoss property, 222
commerce engine

bundle package files, 39–40
bytes, transferring buffer’s, 124–126

C
C# compiler, 36
C++ Component eXtensions (C++/CX), 13
CA (Certificate Authority), 41–42
.cab file extension, 87
Cached File Updater app declaration, 33
CachedFileUpdater class, 140
Calendar app, 214
Camera Settings app declaration, 33
Canceled event, 227–228
canceling background tasks, 227–228
capabilities, device, 31–32, 110–111, 148
Capability SID, 272
C++/CX (C++ Component eXtensions), 13
Certificate Authority (CA), 41–42
Certificates package declaration, 34
certification, Windows Store. See Windows Store
certification
CertUtil.exe utility, 40
collection of channel URLs, 201–202, 211–212
Char data type, 81
Char.ToString() method, 17
CheckNetIsolation.exe tool, 149–150
Choose Default Apps By File Type pane, 102
classes. See also specific classes
class member, 56
static, 56
WinRT type system, 10–11
CleanMgr.exe utility, 80
CleanupTemporaryState task, 95
client/server architecture
client-side HTTP(S) communication, 161–168
client-side TCP communication, 170–172
client-side WebSocket communication, 173–177
class, 10
server-side TCP communication, 172–173
Windows Store apps and, 150
Clipboard class, 232–233
clipboard, sharing via, 231–234
Closed event, 57, 175
closing Windows Store apps, 69–70, 159
CLR (Common Runtime Language), 10
CLR projections, 13
COM APIs, 7–8
collection of commerce engine (Windows Store)
about, 248, 256–257
app trials, 262–263
payment percentages, 256
Common Runtime Language

commerce engine, continued
purchasing app licenses, 262–263
purchasing consumable in-app product offers, 269–270
purchasing consumable in-app products, 266–269
purchasing durable in-app product licenses, 264–266
WinRT APIs, 257–261
Common Runtime Language (CLR), 10
CommonFileQuery enumeration, 117
CommonFolderQuery enumeration, 117
Completed event, 226
compressing data, 134–136
Compressor class
about, 123, 134
sockets and, 168
WriteAsync method, 136
connected standby, 207, 227
ConnectionCost class
ApproachingDataLimit property, 151
NetworkCostType property, 151
OverDataLimit property, 151
Roaming property, 152
ConnectionProfile class
GetConnectionCost method, 151
GetDataPlanStatus method, 152
GetNetworkConnectivityLevel method, 151
NetworkAdapter property, 151
ConnectionReceived event, 172
consumable in-app product offers, 269–270
consumable in-app products, 266–269
Contact app declaration, 34
Contact Picker app declaration, 33
ContactPicker class, 33
content indexer, 108, 141
ContentControl class, 60
ContentIndexer class
CreateQuery method, 142
GetIndexer method, 141
ContentIndexerQuery class, 142
ContentPrefetcher class, 166
Content-Type header, 212
control channel triggers, 208, 212–213
ControlChannelReset trigger type, 210
ControlChannelTrigger class, 208, 212–214
cookies, 166
CoreApplication class
about, 51, 56
CreateNewView method, 58
Exit method, 70
Exiting event, 70
Id property, 193
Resuming event, 66–67, 75
Suspending event, 67–68, 75
CoreApplicationView class
about, 51, 54–55, 57–58
IsHosted property, 58
IsMain property, 58
CoreDispatcher class
about, 19, 51, 57–58, 89
RunAsync method, 225
CoreWindow class
about, 51, 57–58
Activate method, 57
Activated event, 57
Close method, 57
Closed event, 57
GetAsyncKeyState method, 57
GetForCurrentThread method, 57
GetKeyState method, 57
ReleasePointerCapture method, 57
SetPointerCapture method, 57
SizeChanged event, 57
VisibilityChanged event, 57, 67n, 69
Create App Packages wizard, 34–36, 39
CreationCollisionOption enumeration, 96
Credential Manager applet, 87, 137
CurrentApp class
about, 257–259
AppId property, 260
GetAppReceiptAsync method, 263
GetUnfulfilledConsumablesAsync method, 269
LicenseInformation property, 261–262, 266
LinkUri property, 260
LoadListingInformationAsync method, 259–260
ReportConsumableFulfillmentAsync method, 268–269
RequestAppPurchaseAsync method, 262
RequestProductPurchaseAsync method, 265, 267–268
CurrentAppSimulator class
about, 257–259
AppId property, 260
GetAppReceiptAsync method, 263
GetUnfulfilledConsumablesAsync method, 269
LicenseInformation property, 261, 266
LinkUri property, 260
LoadListingInformationAsync method, 258, 260
ReloadSimulatorAsync method, 258, 261, 264, 267
ReportConsumableFulfillmentAsync method, 268–269
RequestAppPurchaseAsync method, 262–263
RequestProductPurchaseAsync method, 265, 267–270

D
Dashboard (Windows Store)
about, 248–249
Age Rating And Rating Certificates page, 250
App Name page, 249
Cryptography page, 250
Description page, 250–251, 264
link to app's privacy policy, 254
monitoring apps, 254–255
Notes To Testers page, 251
Packages page, 250
purchasing consumable in-app product offers, 269
purchasing consumable in-app products, 267
purchasing durable in-app products, 264–266
Selling Details page, 250, 257
Services page, 250, 267
submitting apps, 248–251
testing apps, 252–254
updating apps, 255–256
data types
package data settings, 81
writing and reading, 127–130
DataChanged event, 88–89, 206
DataContractJsonSerializer class, 130
DataContractSerializer class, 80, 130
DatagramSocket class
about, 168, 172, 177–181
ConnectAsync method, 179–180
GetOutputStreamAsync method, 179–180
DatagramSocketMessageReceivedEventArgs class
RemoteAddress property, 180
RemotePort property, 180
DatagramWebSocket class, 181
DataPackage class
about, 229–233, 239–240
SetApplicationLink method, 231
SetBitmap method, 231
SetData method, 230–231
SetDataProvider method, 139, 239
SetHtmlFormat method, 231
SetRtf method, 231
SetStorageItems method, 231
SetText method, 231
SetWebLink method, 231
sharing via Share charm, 235–237
DataPackageOperation enumeration, 230
DataPackagePropertySet class
  ApplicationName property, 230
  Description property, 230
  Title property, 230
DataPackageView class
about, 233
AvailableFormats property, 231
implementing share target apps, 240, 242
sharing via Share charm, 235
DataPlanStatus class, 152
DataProtectionProvider class
about, 123, 128–130
sockets and, 168
DataReader class
about, 123, 128–130
client-side TCP communication, 171
LoadAsync method, 171
ReadAsync method, 171
sockets and, 168
DataRequest class, 238
DataRequested event, 237–238
DataRequestedEventArgs class, 238
DataTransferManager class
about, 33
DataRequested event, 237–238
GetDataForCurrentView method, 237
DataWriter class
about, 123, 127–128, 130
sockets and, 168
StoreAsync method, 128
DateTimeOffset data type, 81
deadlocking threads, 21
debugging
  background tasks, 222–223
  background transfers, 160
  package directory location, 94
  process lifetime management, 75
  share target apps, 245
  storage folders, 83
  Windows Store apps, 46–48, 70, 255
declarations, app
declarations, app, 32–34
decompressing data, 134–136
Decompressor class
about, 123, 135
ReadAsync method, 136
decrypting data, 136–137
deep linking, 193
deferrals (WinRT), 21–23
deploying Windows Store package, 40–43
Deployment Image Servicing and Management, 43
desktop apps. See also specific apps
about, 3
accessing package data, 82
deploying new versions, 225
launching files, 106
Share charm and, 236–237
toast notifications, 194n
Windows Certification and, 247n
Windows RT PC, 5
WinRT APIs and, 3
Desktop.ini file, 115
Developer License dialog box, 25–26
developer licenses, 25–26, 41, 43
developing apps, structuring class code, 70–75
device capabilities (package manifest), 31–32, 110–111, 148
dictionaries, 61, 80
differential download, 38
digital signatures, 263
direct invoke feature, 106
Direct2D library, 8
Direct3D library, 8
DirectX APIs, 8
Disable-AppBackgroundTaskDiagnosticLog
PowerShell command, 223
Disk Cleanup utility, 80
Dismissed event, 197–199
Dispatcher class, 58
DNS names, 170
Documents Library capability, 31, 110–111, 113–114, 116
dots per inch (DPI), 95, 186, 216
Double data type, 81
DownloadOperation class
about, 157
GetResultStreamAt method, 156
Progress property, 158
StartAsync method, 157, 159
Downloads folder, 114–116
DownloadsFolder class, 115
DPI (dots per inch), 95, 186, 216
durable in-app products, 264–266
E
ECMA-335 format, 11
Enable-AppBackgroundTaskDiagnosticLog
PowerShell command, 223
encrypting data, 136–137, 181–182, 250
EndpointPair class, 170, 180
endpoints, 170, 180
Enterprise Authentication capability, 32, 114, 137, 148
temporary deployments, 40–43
Enterprise Sideload keys, 42
enumerations, 11. See also specific enumerations
ERROR_OPLOCK_HANDLE_CLOSED Windows error, 133
event logs
BackgroundTaskInfrastructure, 223–224
locations for, 45, 204, 223
Microsoft-Windows-TWinUI-Operational, 188
PackageStateRoaming, 88
SettingSync, 88
Websocket-Protocol-Component, 175
Event Viewer, 160
events. See specific events
Exception class
HResult property, 133, 172
Message property, 133
exception handling, 69, 132–133, 157
Exiting event, 70
extended splash screen, 65, 73
F
Failed event, 197, 199
fat package, 39
FIFO (first-in, first-out) algorithm, 191
File Open Picker app declaration, 33, 55
file pickers, 97–101, 103, 145
File Save Picker app declaration, 33
File Type Associations app declaration, 33
FileAccessMode enumeration, 131
FileActivatedEventArgs class
about, 104
Files property, 105
NeighboringFilesQuery property, 105
Verb property, 105
FILE_ID_DESCRIPTOR parameter for
OpenFileById function, 101n
FileIO class
about, 12, 119–120
ReadTextAsync method, 83
WriteTextAsync method, 83
FileOpenPicker class
about, 33, 138
CommitButtonText property, 99
FileTypeFilter property, 99
PickMultipleFilesAsync method, 99
PickSingleFileAsync method, 99
SettingsIdentifier property, 99
SuggestedStartLocation property, 99
ViewMode property, 99
FileSavePicker class
about, 33, 138
CommitButtonText property, 99
DefaultFileExtension property, 99
FileTypeChoices property, 99
PickSaveFileAsync method, 99
SettingsIdentifier property, 99
SuggestedFileName property, 99
SuggestedSaveFile property, 99
SuggestedStartLocation property, 99
file-type associations
about, 101, 104–107, 138
declaring, 103
Documents library and, 111
editing, 102
forbidden, 103
properties supported, 104
viewing, 102
filters, networking, 164–168
firewalls, 148–150, 173–174, 201
first-in, first-out (FIFO) algorithm, 191
FlushFileBuffers function, 122
FolderPicker class
about, 33
CommitButtonText property, 99
FileTypeFilter property, 99
PickSingleFileAsync method, 99
SettingsIdentifier property, 99
SuggestedStartLocation property, 99
ViewMode property, 99
Frame class
BackStackDepth property, 61
GetNavigationState method, 67, 68n
Navigate method, 60–61
HttpBaseProtocolFilter class
SetNavigationState method, 69
XAML page navigation, 60–63
framework packages, 27n
Framework projections, 16
FreeNetworkAvailable system condition, 221
G
GameExplorer package declaration, 34
GCs (garbage collectors), 22, 61
Generate App Bundle package property, 29
Geofence class, 210–211
GeofenceMonitor class, 210–211
GET method (HTTP), 156
Get-AppBackgroundTask PowerShell command, 223–224
Get-WindowsDeveloperLicense PowerShell command, 26
GIF image format, 188
Guid data type, 81
H
.hdmp file extension, 255
high integrity level, 271–272
Home Or Work Networking capability, 113
HomeGroup feature, 113–114, 116
host names, 170
hosted view (Windows Store apps)
activating, 53
shared content, 235–236
window for, 53–54
HostName class
CanonicalName property, 169
Compare method, 169
DisplayName property, 169
IsEqual method, 169
Type property, 169
Hosts text file, 170
HTTP(S) communication
HttpBaseProtocolFilter class, 164–168
HttpClient class, 161–163
WebSocket protocol and, 173–174
HTTP methods, 156, 159, 162, 202–203
HttpBaseProtocolFilter class
about, 164–168
CacheControl property, 165
CookieControl property, 165
CookieManager property, 166
HttpClient class

HttpClient class
core features, 161–163
encrypting data traversing networks, 181
HttpBaseProtocolFilter class and, 164–168
SendRequestAsync method, 162–163

HttpResponseMessage class
Content property, 163
Source property, 165

HttpClient class

IActivatedEventArgs interface
Kind property, 57
PreviousExecutionState property, 69

IANA website, 106
IApplicationActivationManager interface, 46
IAsyncAction interface, 18
IAsyncActionWithProgress interface, 18
IAsyncInfo interface, 17
IAsyncOperation interface, 18–19, 93
IAsyncOperationWithProgress interface, 18, 157
IBackgroundTask interface
about, 220
Run method, 208, 224–226

IBackgroundTaskInstance interface
about, 208
Canceled event, 227–228
GetThrottleCount property, 224
Progress property, 226
SuspendedCount property, 224
TriggerDetails property, 154

IBuffer interface, 119, 124, 126
IBufferByteAccess interface, 124–125
IClosable interface, 13n, 120–122, 175n
IDisposable interface, 13n, 120–121, 175n
If-Modified-Since header, 165n

IHttpContent interface
about, 163
ReadAsBufferAsync method, 163
ReadAsInputStreamAsync method, 163
ReadAsStringAsync method, 163

IHttpFilter interface, 167
IInputStream interface
about, 120–123
background transfers, 156
ReadAsync method, 121–122, 124, 126, 133, 171
sockets and, 168

IndexableContent class
Id property, 142
Stream property, 142
StreamContentType property, 142
Indexing Options dialog box, 96
IndexOutOfRangeException class, 133
IHttpBaseProtocolFilter class
Content property, 163
Source property, 165

HttpClient class

IOutputStream interface
about, 120–123
FlushAsync method, 121–122
WriteAsync method, 121–122, 124

IRandomAccessStream interface
about, 122–123, 130
CloneStream method, 121
GetInputStreamAt method, 121–122
GetOutputStreamAt method, 121–122
Position property, 121–122
Seek method, 121–122
Size property, 121–122
isolated storage, 79

IStorageFile interface
about, 91–92
background transfers, 156–157
ContentType property, 107
FileExtent property, 107
OpenAsync method, 121–122, 130–131
OpenTransactedWriteAsync method, 121–122, 130

IStorageFolder interface, 91–92
IStorageFolderQueryOperations interface, 92
IStorageItem interface
about, 91–92
accessing user files, 100
Attributes property, 91, 107
manifest file

DateCreated property, 91, 107
GetBasicPropertiesAsync method, 108
Name property, 91, 107
Path property, 91, 107
IStorageItemAccessList interface, 100–101
IStorageItemProperties interface
about, 92, 107
DisplayName property, 107
DisplayType property, 107
FolderRelativeId property, 107
GetThumbnailAsync method, 108
Properties property, 108

J
JavaScript technology stack, 8–9, 12
JPEG image format, 188
JSON format, 130, 203

K
KnownFolders class
about, 112–113
HomeGroup property, 113
MediaServerDevices property, 114
RemovableDevices property, 114

L
latched background tasks, 221
launch activation, 44, 50
LaunchActivatedEventArgs class, 193
Launcher class, 33, 106, 139
LauncherOptions class, 106
libraries (virtual folders), 110, 116
LicenseChanged event, 263
LicenseInformation class
about, 261
ExpirationDate property, 262
IsActive property, 263
LicenseChanged event, 263
licenses
app, 262–266
developer, 25–26, 41, 43
lifetime management, process. See process lifetime management
LINQ to XML, 123
listing information, 257

ListingInformation class
FormattedPrice property, 262
ProductListings property, 260
Live Connect, 250
Load Hive dialog box, 82
local package data, 80–81, 83
LocalState directory, 93, 95
Location capability, 32
location triggers, 208, 210–211
LocationTrigger class, 208, 210–211
lock screen
about, 214–219
background tasks, 184–185, 214–216
triggers and, 209–210, 214
Lock Screen Call app declaration, 34
LockScreenApplicationAdded trigger type, 209, 219
LockScreenApplicationRemoved trigger type, 210, 219
logos, app, 184–186
loopback exempt list, 149
low integrity level, 271–272

M
Mail app, 105, 244
Main method, 52, 85n
main view (Windows Store apps)
activating, 53, 55
threads for, 52
window for, 53–54
maintenance triggers, 208–209
MaintenanceTrigger class, 208
MakeAppx.exe utility, 37–38
MakePRI.exe utility, 36
managing process model, 55–59
mandatory integrity control, 271–272
mandatory label, 271
manifest designer
about, 26
Application tab, 29, 216
Capabilities tab, 31
Declarations tab, 32–34, 213
Packaging tab, 28–29
Visual Assets tab, 185
manifest file
about, 6, 27–28, 148–149
adding capabilities, 31–32
app declarations, 32–34, 213–219
MapChanged event

manifest file, continued
   capabilities, 31–32, 110–111
   Capability SID and, 272
   enabling periodic tile updates, 192
   file-type associations, 103
   package identity, 28–30
   Share Target declaration, 240–241
   toast notifications, 196
MapChanged event, 81
MediaServerDevices virtual folder, 116
medium integrity level, 271–272
memory management
   memory-leak detection, 70
   suspended apps and, 65, 68
MessageBeep API (Win32), 7
MessageDialog class, 12
MessageWebSocket class
   about, 168, 174, 176–177
   encrypting data traversing networks, 181
metadata, 11–12, 100
metered networks, 146, 152, 167–168
Microphone capability, 32
Microsoft account, 44, 85, 257
Microsoft .NET Framework. See .NET Framework
Microsoft Skype app, 214
Microsoft Visual Studio. See Visual Studio
Microsoft-Windows-TWinUI-Operational event log, 188
monitoring Windows Store apps, 254–255
MSBuild, 36–37
ms-windows-store protocol, 248
multicast IP addresses, 180–181
multicast UDP communication, 180–181
Music Library capability, 31, 110–113, 116

N
Native C/C++ technology stack, 7–8, 12
navigating XAML pages, 59–63
NavigationViewEventArgs class, 61
.NET APIs, 8
.NET Framework
   interoperating between WinRT streams and, 123–124
   isolated storage, 79
   metadata, 11
   WinRT types and, 14–15
.NET technology stack, 8, 12
network connections
   background transfer, 154–160
   change notifications, 153–154
   encrypting data with certificates, 181–182
   HttpClient class, 161–168
   network information, 145–147
   network isolation, 147–150
   profile information, 150–154
   Windows Runtime sockets, 168–181
NetworkInformation class
   GetConnectionProfile method, 154
   GetInternetConnectionProfile method, 150
   NetworkStatusChanged event, 153–154
   NetworkStateChange trigger type, 154, 209
   NetworkStateChangedEventDetails class, 154
   NetworkStatusChanged event, 153–154
   Notification Extension Library (Wintellect), 199
   NotificationsExtension library, 199n
   notify.windows.com, 201
   NuGet package, 231
   NullReferenceException class, 157

O
OAuth tokens, 202–203
Object class
   about, 13
   Equals method, 10
   GetHashCode method, 10
   GetType method, 10
   toast notifications, 198n
   ToString method, 10
object models
   storage, 91–93
   streams, 120–123
On MulticastListenerMessageReceived event handler, 180
OnLineIdConnectedStateChanged trigger type, 209
OpenFileById function, 101n
OutOfProcessServer package declaration, 34
Package class
- Current property, 30
- Id property, 30

package data
- about, 27, 79–81
- change notifications, 89
- data localities, 80–81
- file size considerations, 83
- local, 80–81, 83
- passwords and, 87
- roaming, 80, 83, 85–88
- settings for, 81–82
- storage folders, 83
- temporary, 80, 83
- upgrades and, 45–46
- versioning, 83–85

package declarations, 34

Package Display Name package property, 29, 43
Package Explorer desktop app (Wintellect), 41, 45–46, 82

Package Family Name package property, 29, 43–44

package files
- about, 6
- contents of, 37–39
- creating bundle, 39–40
- manifest file. See manifest file
- read-only, 93–95
- read-write, 93–97
- signing certificates, 43

Package Full Name package property, 29–30, 43–44

package identity
- Application Display Name, 29, 43
- Generate App Bundle, 29
- Package Display Name, 29, 43
- Package Family Name, 29, 43–44
- Package Full Name, 29–30, 43–44
- Package Name, 28–29, 43
- Publisher, 29, 43
- Publisher Display Name, 29, 43
- Publisher ID, 29–30, 43
- Version, 29, 44

Package Name package property, 28–29, 43

Package Resource Index (.pri) file, 36–37

PackageSecurity ID (SID), 202

Package.appxmanifest file, 26–28

Package.Current.Id.Version, 255

PackageManager class, 46

PackageRoot registry value, 93n

process lifetime management

PackageStateRoaming event log, 88

Page class
- NavigationCacheMode property, 61n
- OnNavigatedFrom method, 61–62
- OnNavigatedTo method, 61–62, 69n, 238
- OnNavigatingFrom method, 238

XAML page navigation, 60–63

page navigation, XAML, 59–63

PasswordCredential class, 87, 137

passwords, storing, 87

PasswordVault class, 87, 137

PathIO class, 120

PayPal commerce engine, 256

PDB files, 255

peek templates, 190

peer-to-peer UDP communication, 177–180

Permissions pane, 217


pinning
- secondary tiles to Start screen, 193–194
- websites to Start screen, 184

/platform:anycpu32bitpreferred compiler switch, 36

Playlists virtual folder, 116

PLM (Process Lifetime Management), 75, 244, 268–269

PLMDebug tool, 75

PNG image format, 188

Point data type, 81

polite reader data access, 131–134

downloading web servers, 192

port numbers, 170

POST method (HTTP), 156, 159, 162, 202–203

PowerCfg.exe tool, 207

PowerShell commands, 26, 40–41, 223

.pri (Package Resource Index) file, 36–37

primary thread, 52–53

primitive data types, 127–130

Print Task Settings app declaration, 33

privacy policy, app packages, 147, 254

Private Networks (Client & Server) capability, 31, 114, 148

Process Explorer, 272–273

process lifetime management
- about, 64–65
- debugging, 75
- file management and, 134
- structuring app class code, 70–75
- Windows Store app suspension, 65–69
- Windows Store app termination, 66–70

285
Process Lifetime Management (PLM)

Process Lifetime Management (PLM), 75, 244, 268–269
process model (Windows Store apps)
about, 49
additional resources, 63
app activation, 49–55
background transfer, 154–160
managing, 55–59
process lifetime management, 64–75
XAML page navigation, 59–63
ProductLicense class, 261
ProductListing class, 260
ProductPurchaseStatus enumeration, 266
Progress event, 226
project files (Windows Store app), 25–27
Properties folder, 26
Protocol app declaration, 33
Proximity capability, 32, 148
ProxyStub package declaration, 34
Publisher Display Name package property, 29, 43
Publisher ID, 29–30, 43
Publisher package property, 29, 43
PurchaseResults class
about, 265–266
ReceiptXml property, 270
TransactionId property, 266
purging roaming package data, 87–88
push notification triggers, 208, 211–212
push notifications, 88, 200–204, 213. See also WNS
(Windows Push Notification Service)
PushNotificationChannel class
Close method, 201
ExpirationTime property, 201
PushNotificationReceived event, 201, 212
PushNotificationChannelManager class
CreatePushNotificationChannelForApplicationAsync method, 201
CreatePushNotificationChannelForSecondaryTileAsync method, 201
PushNotificationReceived event, 201, 212
PushNotificationReceivedEventArgs class
Cancel property, 201, 212
Notification Type property, 212
Raw Notification property, 212
PushNotificationTrigger class, 208, 211–212, 214
PushNotificationType enumeration, 212
PUT method (HTTP), 159

Q
queries, file and folder, 97, 116–118
QueryOptions class
ApplicationSearchFilter property, 117
DateStackOption property, 117
FileTypeFilter property, 117
FolderDepth property, 117
GroupPropertyName property, 117
IndexerOption property, 117
Language property, 117
SetPropertyPrefetch method, 118
SetThumbnailPrefetch method, 118
SortOrder property, 117
UserSearchFilter property, 117
quick links, 244
QuickLink class, 244–245

R
raw notifications, 200, 209, 211–212
RCWs (Runtime Callable Wrappers), 10, 22, 124
read operations
polite reader data access, 131–134
primitive data types, 127–130
Reading List app, 240
read-only package files, 93–95
read-write package files, 93–97
real-time communication (RTC), 211–212, 214
receipts, validating, 263
Rect data type, 81
References folder, 26
registering
app package, 44–45
background tasks, 219–222
Remote Tools for Visual Studio, 48
Removable Storage capability, 31, 114, 116
resource quotas, background tasks, 223–224
restricted deployments, 40–41
Resume trigger, 222
Resuming event, 66–67, 75, 198n
resuming Windows Store apps, 66–67, 75
RFC 2616, 165
Roaming Monitor Tool, 88
roaming package data
about, 80, 85–88
directory locations, 83
purging, 87–88
synchronizing, 87
RoamingState directory, 93, 95
RTC (real-time communication), 211–212, 214
Runtime Callable Wrappers (RCWs), 10, 22, 124
RuntimeBroker.exe process, 101

S

scheduled times
  showing toast notifications at, 198–199
  updating tiles at, 192
ScheduledTileNotification class, 192
Search app declaration, 33
SearchBox class, 141
SearchPane class, 33
Secondary Tile Approval dialog box, 194
secondary tiles, 192–194
SecondaryTile class
  Arguments property, 193
  LockScreenBadgeLogo property, 215n
  LockScreenDisplayBadgeAndTileText property, 215n
  RequestCreateAsync method, 194
  RequestDeleteAsync method, 194
  RoamingEnabled property, 193
  TileId property, 193
Secure Sockets Layer (SSL), 181, 250
security, app containers, 271–272
semantic zoom mode, 184
serialization technologies, 123, 128, 130
server-side communications. See client/server architecture
service names, 170
Services text file, 170
ServicingComplete trigger type, 209, 225
SessionConnected system condition, 221
SessionConnected trigger type, 210
SessionNotConnected system condition, 221
Set-AppBackgroundTaskResourcePolicy PowerShell command, 223–224
Settings dictionary, 80
Settings.dat hive file, 82
SettingSync event log, 88
Share charm
  hosted view activation, 53
  networking and, 145
  share target apps and, 240, 244
  sharing via, 234–238
  transferring files via, 138
  workflow process, 234–237
  share source apps, 237–240
Share Target app declaration, 33
share target apps, 240–245
Shared User Certificates capability, 32
ShareOperation class
  about, 241
  Data property, 242
  DismissUI method, 243
  QuickLinkId property, 244
  RemoveThisQuickLink method, 245
  ReportCompleted method, 243
  ReportDataRetrieved method, 244
  ReportError method, 243
  ReportStarted method, 243
  ReportSubmittedBackgroundTask method, 244
ShareTargetActivatedEventArgs class, 241, 244
sharing contract, 235
sharing data between apps
  about, 229
  debugging share target apps, 245
  implementing share source app, 237–240
  implementing share target app, 240–245
  via clipboard, 231–234
  via DataPackage class, 229–231
  via Share charm, 234–237
Show-WindowsDeveloperLicense-Registration PowerShell command, 26, 40
SID (Application Security ID), 272
SID (Package Security ID), 202
sideloading technique, 40–42
SignTool.exe utility, 37
Silverlight, 58
simulator, 47–48, 254
Single data type, 81
Size data type, 81
SizeChanged event, 57
SkyDrive account, 86n, 88, 108
Skype app, 214
SmsReceived trigger type, 209
SoC (System on Chip) devices, 207
socket addressing, 169–170
SocketProtectionLevel enumeration, 181
sockets
  client-side TCP communication, 170–172
  client-side WebSocket communication, 173–177
  identifying remote systems to WinRT, 169–170
  multicast UDP communication, 180–181
  peer-to-peer UDP communication, 177–180
  server-side TCP communication, 172–173
  types supported, 168
Software Assurance for Windows

Software Assurance for Windows, 42
Software Publisher Certificate (SPC), 41–42
source apps, 229, 234–240
SPC (Software Publisher Certificate), 41–42
splash screens, 52, 64–65, 67–68, 73, 85
SRV records, 170
SSL (Secure Sockets Layer), 181, 250
staging app package, 44–45
StandardDataFormats class, 230–231
Start screen
about, 183
activating apps, 50
app bar, 184
cycling through notifications, 191
pinning secondary tiles to, 193–194
pinning websites to, 184
polling web servers, 192
terminated apps and, 64
tiles and badges, 184–185
Start-AppBackgroundTask PowerShell command, 223
static classes, 56
storage files and folders
file-type associations, 101–107
package data, 80–81, 83
package files, 93–97
performing queries, 116–118
storage item properties, 107–109
storage object model, 91–93
user files, 93–94, 97–101, 109–116
StorageApplicationPermissions class
FutureAccessList property, 100, 116
MostRecentlyUsedList property, 100
StorageFile class
about, 12, 91–93, 123
accessing read-only package files, 94
accessing read-write package files, 96
accessing user files, 100
CreateStreamedFileAsync method, 138–140
CreateStreamedFileFromUriAsync method, 140
FolderRelativeId property, 106
GetFileAsync method, 16, 19
GetThumbnailAsync method, 139
IStorageItemProperties2 interface and, 108
Path property, 140
RenameAsync method, 131
ReplaceWithStreamedFileAsync method, 140
StorageFileQueryResult class
GetFilesAsync method, 105
OnOptionsChanged method, 118
StorageFolder class
about, 12, 91–93
accessing read-only package files, 94
accessing read-write package files, 95–96
accessing user files, 99–100, 112
IStorageItemProperties2 interface and, 108
Path property, 92, 116
StorageFolderQueryResult class
GetFolderAsync method, 118
OnOptionsChanged method, 118
StorageItemContentProperties class, 108–109
StorageLibrary class, 110
StorageStreamTransaction class
about, 122, 130
CommitAsync method, 121, 131
Stream property, 130
Stream class, 123
stream input and output
compressing and decompressing data, 134–136
encrypting and decrypting data, 136–137
interoperating between WinRT and .NET streams, 123–124
performing transacted write operations, 130–131
polite reader data access, 131–134
populating streams on demand, 138–140
searching over stream content, 140–144
simple file I/O, 119–120
streams object model, 120–123
transferring byte buffers, 124–126
writing and reading primitive data types, 127–130
StreamSocket class
about, 168, 170–172, 181
ConnectAsync method, 181
UpgradeToSslAsync method, 181
StreamSocketListener class
about, 168, 172–173
ConnectionReceived event, 172
encrypting data traversing networks, 181
StreamSocketListenerConnectionReceivedEventArgs class, 172
StreamWebSocket class
about, 168–169, 173–175
Close method, 175
Closed event, 175
Dispose method, 175
ToastActivatedEventArgs class

client-side, 170–172
server-side, 172–173
technology stacks, 6–9
templates (XML), 186–187, 190–192, 198–199, 211
temporary package data, 80, 83
_TemporaryKey.pfx file, 27
TempState directory, 93, 95
terminating Windows Store apps, 64–70, 75, 159, 191
testing Windows Store apps, 43, 70, 251–254, 264
threads
blocking, 21, 92–93
deadlocking, 21
hosted view, 54
main view, 52, 54
primary, 52–53
suspended, 65–69, 205, 214
updating user interface, 58
tilde (~), 39
tile notifications, 190–191, 209
TileNotification class, 186–187, 189
tiles
activating apps, 50
animating contents, 190–191
placing badges on, 188–189
secondary, 192–194
updating at scheduled times, 191
updating periodically, 192
updating techniques, 183
updating when app in foreground, 186–188
URL prefixes for images, 188
usage considerations, 184–186
TileSquarePeekImageAndText01 template, 190
TileUpdateManager class
about, 187
Clear method, 189
EnableNotificationQueue method, 190
StopPeriodicUpdate method, 192
time triggers, 208–209, 220–221
TimeSpan data type, 81
TimeTrigger class, 208, 220
timeZoneChange trigger type, 209
toast notifications
about, 194–196
activating apps, 50, 197–198
creating, 196–197
maintenance triggers, 209
showing at scheduled times, 198–199
sound capabilities, 198
updating, 183
ToastActivatedEventArgs class, 198

String data type, 81
structuring class code, 70–75
submitting Windows Store apps
monitoring apps, 254–255
process overview, 248–251
testing apps, 252–254
updating apps, 255–256
Suspend trigger, 222
Suspending event, 67–68, 75
suspending Windows Store apps, 65–69, 75, 168, 191, 205
SuspensionManager class, 63
SynchronizationContext class, 19
synchronizing roaming package data, 87
System Center Configuration Manager, 43
System namespace, 10, 13, 60, 120, 133
System on Chip (SoC) devices, 207
system triggers, 208–210
System.IO namespace, 123, 127
System.IO.Compression namespace, 136
System.Keywords property, 142
System.Media.Duration property, 142
System.Net.WebSockets namespace, 174
SystemProperties class, 109
System.Runtime.InteropServices namespace, 126
System.Threading.Tasks namespace, 18–19
SystemTrigger class, 154, 208
SystemTriggerType enumeration, 154
System.Xml.Linq namespace, 123

T
/target:appcontainerexe compiler switch, 36
target apps, 229, 235–236, 240–245
Task class
ConfigureAwait method, 20
Result property, 20
Task Manager
about, 146
App History tab, 146–147
App1 Download/Upload Host process, 160
killing apps, 69
suspending apps, 65–66, 68
Tile Updates column, 204
Task Scheduler, 80, 95
TaskAwaiter type, 21
TaskCompletionSource class, 19
TCP communication
ToastDismissedEventArgs class

ToastDismissedEventArgs class, 198
ToastNotification class, 197
ToastNotifier class, 198–199
tokens, 100, 202–203
transacted write operations, 130–131
transfer, background, 154–160
transferring byte buffers, 124–126
trial period for apps, 247, 250, 256–257, 262–263
triggers (background tasks)
  about, 205–207
  adding system conditions, 221
  choosing, 208–213
  lock screen and, 209–210, 214
Trusted People certificate store, 41
try/catch blocks, 132
Type class, 60

U

UAC (User Account Control), 252, 271
UDP communication
  multicast, 180–181
  peer-to-peer, 177–180
UIElement class, 60
UInt8 data type, 81
UInt16 data type, 81
UInt32 data type, 81
UInt64 data type, 81
UnfulfilledConsumables class, 269
Unregister-AppBackgroundTask PowerShell command, 223
Unregister-WindowsDeveloperLicense PowerShell command, 26
updating
  badges, 183
  tiles and tile notifications, 183, 186–188, 191–192
  toast notifications, 183
  user interface threads, 58
Windows Store apps, 255–256
UploadOperation class
  about, 157
  Progress property, 158
  StartAsync method, 157, 159
URI technique
  accessing read-only package files, 94–95
  accessing read-write package files, 96
  accessing user files, 106–107
  encrypting data traversing networks, 181
User Account Control (UAC), 252, 271
user files
  about, 93–94
  accessing via explicit user consent, 97–101
  accessing with implicit user consent, 109–116
UserAway trigger type, 210
UserInformation class, 33
UserNotPresent system condition, 221
UserPresent system condition, 221
UserPresent trigger type, 210
UTF-8 encoding, 83, 119
UTF-16 encoding, 119

V

validating receipts, 263
VDA (Virtual Desktop Access), 42
Version package property, 29, 44
versioning package data and apps, 83–85, 255–256
Videos Library capability, 31, 110–111, 113, 116
Virtual Desktop Access (VDA), 42
to virtual folders (libraries), 110, 116
VisibilityChanged event, 57, 67n, 69
Visual Studio. See also manifest designer
  Allow Local Network Loopback debug setting, 148–150
  app tile settings, 185
  Debug Location toolbar, 75, 222
debugging Windows Store apps, 46–48
destroying app operations, 159
  Page-derived classes, 240
  Roaming Monitor Tool, 88
VLSC (Volume Licensing Service Center), 42
Volume Licensing programs, 42
Volume Licensing Service Center (VLSC), 42
VS Wizard, 252

W

WACK (Windows App Certification Kit), 252–253
web service
  pushing notifications to user PCs, 202–204
  receipts and, 263
  securing network traffic, 182
  sending channel URI to, 202, 211
  verifying client authorization, 263
  WNS support, 200–201, 212
Webcam capability, 32
websites, pinning to Start screen, 184
WebSocket protocol
  messaging client-side communication, 176–177
  streaming client-side communication, 173–175
WebSocket-Protocol-Component event log, 175
Win32 APIs, 7–8
Win32 MessageBeep API, 7
Window class, 57, 65
WindowClass class, 54
Windows Advanced Query Syntax, 117
Windows App Certification Kit (WACK), 252–253
Windows Azure Mobile Services, 250
Windows Calendar app, 214
Windows Credential Manager applet, 87, 137
Windows Debugger, 255
Windows Disk Cleanup utility, 80
Windows Event Viewer, 160
Windows InTune, 43
Windows Mail app, 105, 244
Windows PowerShell commands, 26, 40–41, 223
Windows Presentation Foundation (WPF), 58
Windows Push Notification Service. See WNS
Windows RT PC, 5
Windows Runtime APIs. See WinRT APIs
Windows Store
  commerce engine, 248, 256–270
  Dashboard, 248–256
  refund, 261
  submitting apps to, 248–256
Windows Store apps
  about, 3, 247–248
  accessing user files, 110
  activating. See activating Windows Store apps
  additional information, 43
  app containers and, 3
  app package. See app package
  closing, 69–70, 159
  debugging, 46–48, 70, 255
  deploying new versions, 225
  file management, 134
  installing, 148
  isolation of, 148
  monitoring, 254–255
  principles of, 4–6
  process model. See process model
  project files, 25–27
  resuming, 66, 75
  securing network traffic, 182
Windows Store certification
  activation requirements, 64–65
  closing apps, 70
  connectivity profile information, 152–153
  desktop apps, 247
  desktop data and, 137
  encrypting data traversing networks, 181–182
  encryption requirements, 67
  tracking status, 251–252
Windows Task Scheduler, 80, 95
Windows.ApplicationModel namespace, 50
Windows.ApplicationModel.Appointments namespace, 33
Windows.ApplicationModel.Background namespace, 33
Windows.ApplicationModel.Contacts namespace, 33
Windows.ApplicationModel.Core namespace, 56, 193
Windows.ApplicationModel.Search namespace, 33
Windows.ApplicationModel.Store namespace, 257
WindowsApps directory, 41, 43, 46, 93
Windows.Foundation namespace, 120
Windows.Management.Core namespace, 82
Windows.Networking namespace, 169
Windows.Networking.BackgroundTransfer namespace, 155, 166
WindowsRuntimeBuffer class, 126
WindowsRuntimeBufferExtensions class
  AsBuffer method, 125–126
  AsStream method, 125
  ToArray method, 125
WindowsRuntimeStorageExtensions class, 123
WindowsRuntimeSystemExtensions class

WindowsRuntimeSystemExtensions class
AsTask extension method, 19
GetAwaiter extension method, 18–19
Windows.Security.Credentials namespace, 87
Windows.Storage namespace, 109–110
Windows.Storage.Compression namespace, 134–135
Windows.Storage.Pickers namespace, 33, 98
Windows.Storage.Streams namespace, 120, 126
Windows.System namespace, 33, 109
Windows.System.UserProfile namespace, 33
Windows.UI.Controls namespace, 60
Windows.UI.Core namespace, 58
Windows.UI.ViewManagement namespace, 59
Windows.UI.Xaml namespace, 53, 56–57, 64, 70, 107, 141
Windows.Web.Http namespace, 161
WinJS library, 8
WinMD file
about, 10–12
creating, 207
including in app package, 220
loading, 208, 213
location of, 12
WinRT APIs
about, 3
asynchronous, 16–23
commerce engine, 257–261
deferrals, 21–23
desktop apps and, 3
interoperating between .NET streams and, 123–124
storage object model, 91–93
streams object model, 120–123
toast notifications, 194n
Windows Store apps and, 3
WinRT type system
about, 10–11
classes, 10–11
core base types, 10
core data types, 10
corresponding .NET type projection, 14–15
enumerations, 11
interfaces, 11
structures, 11
system projections, 11–16
Wintellect Notification Extension Library, 199
Wintellect Package Explorer desktop app, 41, 45–46, 82
Wired Equivalent Privacy (WEP), 151
WNS (Windows Push Notification Service)
about, 145, 191, 199–200
integrating with apps, 250
maintenance triggers, 209
push notification triggers, 211–212
registering apps with, 200–201
registering PCs with, 200–201
secondary tiles and, 192
WPA (Wi-Fi Protected Access), 151
WPF (Windows Presentation Foundation), 58
write operations
primitive data types, 127–130
transacted, 130–131

X

XAML development
about, 8, 57
implementing share target apps, 240–242
page navigation, 59–63
XDocument class, 123
XML digital signatures, 263
XML documents, 186–187
XML manifest file. See manifest file
XML schema, 188, 196
XML templates, 186–187, 190–192, 198–199, 211
XmlDocument class, 188
X-WNS-Debug-Trace header, 204
X-WNS-DeviceConnectionStatus header, 204
X-WNS-Msg-ID header, 204
X-WNS-NotificationStatus header, 204
X-WNS-RequestsForStatus header, 204
X-WNS-Tag header, 203
X-WNS-TTL header, 203
X-WNS-Type header, 203, 212

Z

ZIP files, 6, 37–38, 87
ZipArchive class, 136
zoom mode, semantic, 184