Programming the Windows® Runtime by Example

A Comprehensive Guide to WinRT with Examples in C# and XAML

Jeremy Likness
John Garland
Praise for
Programming the Windows Runtime by Example

“This is a great from-the-ground-up, very complete book on building Windows Store Apps. You’ll find it on your desk a year from now all dog-eared and marked up from use.”

Dave Campbell, MVP, WindowsDevNews.com

“Programming with Windows Runtime by Example is a must-have book for any professional developer building apps for WinRT/Win8.1, especially in the LOB space for modern apps on Windows 8.1. For me it is the reference I provide my team building LOB applications for WinRT. Jeremy and John have done a great job putting together a great reference and educational book on professional development for the WinRT platform.”

David J. Kelley, CTO, Microsoft MVP

“Jeremy and John are both very much IT masters from the old guard of software development. With countless years of bending, shaping, and influencing the world of software development behind them both, they continue to do so as they push forward into new and emerging technologies.

“As with everything they do, this book also reflects their ongoing dedication and passion for their quest to bring the reader not only the information he or she requires, but far more beyond that, they build knowledge step-by-step, then deliver it to the reader with cutting-edge, ninja-like precision to deliver exactly what knowledge is needed, when it’s needed, and where it’s needed.

“If you want to learn the Windows Runtime, then I can think of no finer book, and no finer guides to the WinRT landscape. By the end of this book, you’ll have the knowledge, the power, and a hefty dose of passion to go out into the new millennium and create some of the best WinRT apps available.”

Peter “Shawty” Shaw, LinkedIn .NET User Group manager

“This book is an invaluable resource for budding WinRT developers. It covers the basics to more advanced topics like MVVM. Readers will find the chapter entitled ‘Connecting to the Cloud’ especially useful in getting up to speed with Azure and creating cloud connected apps.”

Daniel Vaughan, President of Outcoder, Microsoft MVP, Author of Windows Phone 8 Unleashed

“There are books that provide reference for a development topic, and others that you will read from cover to end. Programming the Windows Runtime by Example by Jeremy Likness and John Garland should be your go-to guide for getting up to speed on WinRT. Jeremy and John wrote this book with the intention of being easy to follow and hard to forget, and they succeeded in both areas. I recommend this book for all developers, whether new to WinRT development, or those like me who just want to fill in the gaps on advanced topics.”

Chris Woodruff, DeepFriedBytes.com, Microsoft MVP
Visit informit.com/mswinseries for a complete list of available publications.

The Windows Development Series grew out of the award-winning Microsoft .NET Development Series established in 2002 to provide professional developers with the most comprehensive and practical coverage of the latest Windows developer technologies. The original series has been expanded to include not just .NET, but all major Windows platform technologies and tools. It is supported and developed by the leaders and experts of Microsoft development technologies, including Microsoft architects, MVPs and RDs, and leading industry luminaries. Titles and resources in this series provide a core resource of information and understanding every developer needs to write effective applications for Windows and related Microsoft developer technologies.

“This is a great resource for developers targeting Microsoft platforms. It covers all bases, from expert perspective to reference and how-to. Books in this series are essential reading for those who want to judiciously expand their knowledge and expertise.”

– JOHN MONTGOMERY, Principal Director of Program Management, Microsoft

“This series is always where I go first for the best way to get up to speed on new technologies. With its expanded charter to go beyond .NET into the entire Windows platform, this series just keeps getting better and more relevant to the modern Windows developer.”

– CHRIS SELLS, Independent Consultant specializing in Windows, devices, and the cloud
For Doreen and all her arrows, Lizzie and all her travels,
and Gordon and all his paint.
—Jeremy Likness

To Karen, Callie, Winnie, and Dude,
for the new adventure that is soon to begin.
—John Garland
Contents at a Glance

1 The New Windows Runtime  1
2 Windows Store Apps and WinRT Components  29
3 Layouts and Controls  81
4 Data and Content  153
5 Web Services and Syndication  199
6 Tiles and Toasts  225
7 Connecting to the Cloud  261
8 Security  323
9 Model-View-ViewModel (MVVM)  349
10 Networking  379
11 Windows Charms Integration  415
12 Additional Windows Integration  451
13 Devices  479
14 Printers and Scanners  531
15 Background Tasks  559
16 Multimedia  589
17 Accessibility  615
18 Globalization and Localization  631
19 Packaging and Deploying  649
20 Debugging and Performance Optimization  685
  A Under the Covers  719
  B Glossary  733
Index  749
Contents

Foreword xix
Preface xxii

1 The New Windows Runtime 1
Windows Runtime Specifics 1
Windows Store Apps 4

Example: Create a Windows Store App 5
.NET and WinRT 9
Fundamental Types 9
Mapped Types 10
Streams and Buffers 14
Desktop Applications 15

Example: Reference WinRT from a Desktop Application 15
Example: Examine Projections in a WinRT Component 20
Asynchronous Functions 24
Summary 27

2 Windows Store Apps and WinRT Components 29
Fundamentals of a Windows Store App 30

Windows Store App Templates 32
Understanding the App Manifest 45
Finding Your Package on Disk 52
Running Your App 54
3 Layouts and Controls  81
The Visual Tree  83
Data-Binding  85
  Dependency Properties  91
  Attached Properties  94
  Value Precedence  95
  Property Change Notification  95
Animations  97
  Example: Dynamically Apply Animations to a Control  97
The Visual State Manager  100
  Example: Visual State Manager  101
  Groups  103
  States  105
  Transitions  106
The Visual State Manager Workflow  107
Programmatic Access to Visual States  109
Custom Visual State Managers  109
Styles  111
Templates  112
  Example: Using Templates  112
Layouts  115
  Panel  115
  Border  115
  Canvas  116
  Grid  116
  StackPanel  117
  VirtualizingPanel and VirtualizingStackPanel  118
12 Additional Windows Integration 451
   Integrating with the File and Contact Pickers 452
      The Example App 453
      File Open Picker 454
      File Save Picker 458
      Contact Picker 460
   Application Activation Integration 462
      The Example App 463
      File Activation 463
      Protocol Activation 467
      Account Picture Provider 470
      AutoPlay 471
   Working with Contacts and Appointments 473
      The Example App 474
      Contacts 474
      Appointments 476
   Summary 478

13 Devices 479
   Working with Input Devices 480
      The Example App 480
      Identifying Connected Input Devices 481
      Pointer, Manipulation, and Gesture Events 484
      Keyboard Input 495
   Sensor Input 498
      The Example App 498
      Geolocation 502
      Geofencing 510
      Motion and Orientation Sensors 517
   Summary 529

14 Printers and Scanners 531
   Working with Printers 532
      The Example App 532
      Getting Started 533
## Contents

- Configuring a Print Task 534
- Providing Printing Content 542

### Working with Scanners 547
- The Example App 547
- Determining Scanner Availability 548
- Working with Scan Sources 549
- Previewing 550
- Scanning 551
- Scanner Settings 552

### Summary 556

## 15 Background Tasks 559

- The Thread Pool 560
- Uploads and Downloads 562
- Audio 563

### Lock Screen Tasks 570
- Lock Screen Capabilities 570
- The Background Task 573
- Listing Background Tasks 576
- Timer 578
- Conditions 578
- Debugging Background Tasks 580

### Raw Push Notifications 581
- Control Channel 585
- System Events 587

### Summary 588

## 16 Multimedia 589

- Playing Multimedia Content 590
  - The Example App 590
  - Getting Started 591
  - Controlling Playback 592
  - Appearance 595
  - Audio Settings 596
  - Media Information 597
  - Markers 597
Acquiring Audio and Video 598
  The Example App 599
  Declaring Application Capabilities 599
  Using CameraCaptureUI 600
  Using MediaCapture 604
Text-to-Speech Support 610
  The Example App 611
  Using the SpeechSynthesizer 611
Summary 613

17 Accessibility 615
  Requested Theme 616
    High Contrast 618
  Keyboard Support 620
  Automation Properties 622
  Testing with Narrator 623
  Automation and Lists 624
  Live Settings 625
  Automation Peers 626
  Accessibility Checker 627
Summary 629

18 Globalization and Localization 631
  Design Considerations 632
  Default Language 633
  Configuring Preferred Languages 635
  Resource Qualification and Matching 637
  Localizing XAML Elements 639
  Formatting Dates, Numbers, and Currencies for Locale 642
  MVVM and Localization 643
  Multilingual Toolkit 644
Summary 648
19 Packaging and Deploying  649
   Packaging Your App  650
      Creating an App Package  650
      App Package and App Bundle Contents  654
      Package Identifier  655
   Deploying Your App  657
      Publishing Your App in the Windows Store  657
      Other Deployment Options  665
   Making Money with Your App in the Windows Store  667
      The Example App  668
      Pricing Your App in the Windows Store  669
      Trial Mode Apps  670
      In-App Purchases  675
      Including Advertisements  678
   Summary  683

20 Debugging and Performance Optimization  685
   Understanding the Debugger  686
      Native, Managed, and Script Debuggers  686
      Just My Code  688
      Edit and Continue  690
      Just in Time Debugging  691
      How to Launch the Debugger  691
      Program Databases  692
      Debug Windows  693
      Managing Exceptions  694
   Logging and Tracing  696
   Profiling and Performance Analysis  702
      Performance Tips  704
      CPU Sampling  706
      XAML UI Responsiveness  709
      Energy Consumption  710
   Code Analysis  712
   Summary  717
A  Under the Covers  719
   Fundamental WinRT Concepts  719
   Namespaces  720
   Base Types  720
   Primitives  720
   Classes and Class Methods  721
   Structures  722
   Generics  722
   Null  723
   Enumerations  723
   Interfaces  723
   Properties  723
   Delegates  724
   Events  724
   Arrays  725
   WinRT Internals  725

B  Glossary  733

Index  749
Foreword

The concept of an app has changed dramatically over time, and more increasingly so in the past eight years. The approachability for the masses to have super computers in their pockets has led to the rapid adoption of mobile apps at the fingertips of every user—not just those in cubicles all day long. You can’t sit in public transit, walk down a street, or even enjoy a nice meal without looking around and seeing the glow from a screen of some sort on someone’s face. Everyone is a part of the app ecosystem now. Whether it is a mobile phone, music device, e-reader, watch, or even glasses, apps are a part of our lives. People desire them to make their lives and jobs more productive or just to have fun. As a software developer, it is hard to ignore this surge in opportunity and the desire to capitalize on this ecosystem.

Microsoft technologies present a large opportunity to software developers to reach a vast ecosystem of traditional users who have used Windows technologies in their personal, educational, and professional lives. These users seek out new ways to accomplish tasks and have fun on their technology devices. Microsoft has computing devices across the various screens presented in our lives in our hands, on our desks, and in our living rooms. All these represent opportunities for you, the developer, to extend your reach and ideas into the world.

As this evolution of mobility, multiple screens, and wearables has increased, so has technology. Microsoft technologies have evolved as well
on the client app areas. Over time Microsoft has delivered various ways to write client applications through standard C++, MFC, Windows Forms, Windows Presentation Foundation (WPF), Silverlight, and HTML. Putting developers on a better path for development, Microsoft introduced the Windows Runtime (WinRT). This technology and principles enable developers to have a single platform to target that extends their potential across the personal, professional, and entertainment endpoints we have in our lives. WinRT enables developers to choose how they can be most productive using their skills in C++, C#, Visual Basic, or JavaScript. Alongside the language of choice, developers have a native UI framework in XAML they can use for the best client app experience on Windows. XAML is everywhere now in Windows, from system shell UI to system apps to key experiences delivered from Microsoft, such as Microsoft Office. When developing an app in C# and XAML, you’ll be joining other successful developers in the world and can tap into that ecosystem of knowledge, experience, and examples.

Software is an art. Just like any art project, approaching software development requires thought into the necessary tools, philosophies, and principles you will use to create your app. I still remember one of my earliest “professional” software development jobs, sitting in a meeting listening to the customer describe all these (what was at the time) high-tech requirements of their app, all needing to be done in Internet Explorer 3. I scribbled notes as fast as I could while my dev lead at the time, all too quickly I thought, was busy nodding his head in acceptance of the requirements. As we walked out of the meeting, I expressed my concern about the requirements and available technology at the time. He smiled and shrugged like it was no problem stating, “No worries Tim, we just need the right tools.”

One of the key tools is a good guide and mentor. In my early days, for me that was books just like this one you have now. To this day I still prefer books on my shelf when learning new technology concepts. I’ve had the pleasure of working with Jeremy Likeness over the years in the XAML ecosystem, and I can attest to his expertise in building real-world apps using these technologies. In Programming the Windows Runtime by Example, Jeremy and John provide these key tools for any software developer to understand the fundamentals of the Windows Runtime and XAML, and be
successful quickly. This book doesn’t try to only focus on singular concepts but also provides an end-to-end perspective on building an app in WinRT. Jeremy and John know that your scenarios are connected ones and deal with web services, data, security, and integration. The book will walk you through understanding how the pieces fit together in WinRT while still providing you the knowledge and tools to be productive at the core concepts of working with C# and XAML in the Windows Runtime. John and Jeremy describe philosophies and different approaches to using WinRT, empowering you with knowledge to make the best decisions for your app. This knowledge will enable you to write the best apps for Windows, Windows Phone, Xbox, and whatever future Microsoft has in store for WinRT areas.

Like any artist, tools are essential. This book is one of those essential tools for Windows developers and will help you complete your software goals sooner than without it! To this day, my bookshelf is filled with books just like this one that I refer to often. Even as your experience grows, you’ll find yourself referring back to this book for knowledge when developing, just like I did.

—**Tim Heuer**, Principal Program Manager Lead, XAML Platform, Microsoft Corporation
In 2011 I heard the first rumors about Windows 8 and knew immediately what my next book would be about. Unlike Designing Silverlight Business Applications that captured years of experience writing Line of Business (LOB) apps in Silverlight, this book would be an introduction to an entirely new platform. My goal was to take what I knew and loved about Silverlight, find its similarities in the new platform, and then highlight what I felt were some amazing developer experiences. It was important to get to market fast, so through several iterations of the Windows 8 releases (including changes to terminology) that required substantial rewrites of content and a rapid release cycle, I managed to release Building Windows 8 Apps with C# and XAML as Windows 8 was revealed to the world.

By necessity, this book introduced developers to the new platform but didn’t dig into best practices (there were none yet) or get very deep (there simply wasn’t time). I vowed to release another book that would fill in the missing pieces and provide a comprehensive overview of the entire Windows Runtime. Because anyone can read the documentation and reference the API, my intent with this book was to make it example-driven and provide thousands of lines of code for you to integrate and use to kick-start your own Windows Store apps.

I was relieved at the thought of not rewriting most of the book three times, as I had to do with the first one, but Microsoft once again proved too fast for me. What sounded at first like a relatively minor release (Windows
managed to integrate enough changes to warrant revisiting every one of the ten chapters I had completed to date. With an eye on //BUILD in 2014, I reached out to Windows Store expert and Wintellect colleague John Garland to help me finish the remaining chapters. John and I have worked on several projects together (and incidentally two of them won awards for their groundbreaking use of XAML for touch and mobile), and he helped write pilot code for several of our customers who were early Windows 8 adopters, so I knew he was the right person to bring a fresh set of example projects and content-rich chapters. As a bonus, he is also well-versed in cloud technology and brought this firsthand knowledge to bear in the chapters that deal with connecting to Azure.

In Windows 8.1 and the Windows Runtime, Microsoft has successfully demonstrated their commitment to the development ecosystem by providing us with a rich, vast array of APIs, SDKs, and tools for building incredible apps that run on a variety of devices. I was absolutely amazed when I discovered how easy it was to connect to a web cam, open a web socket, download files in the background, or profile my app to find “hot spots” that I could target to improve performance using WinRT. I was delighted to find that Portable Class Libraries (PCL), something I evangelized heavy as a solution to target multiple platforms in the Silverlight and WPF days, was evolving to embrace Windows Store apps. The first-class support for mature design patterns like MVVM makes it easier than ever to write stable, reusable code that runs on a variety of target devices.

In Building Windows 8 Apps with C# and XAML, I shared my intent to guide you through the process of learning the new territory quickly to begin building amazing new applications using skills you already had with C# and XAML. In this book, it is our goal to take you beyond that initial exposure and help you dive deep into all the various APIs WinRT makes available. Our goal was to hit virtually any scenario possible using the Windows Runtime—not just provide code snippets, but full projects you can use to experiment, learn, and use as a starting point for your own apps. The most rewarding feedback I received from my first book was hearing from authors sharing with me their excitement having their first Windows 8 apps approved for the Store. I hope this book not only helps take those apps to the next level, nor simply inspires your imagination, but
empowers you to implement solutions you only dreamed possible using this incredible new platform. I know I speak for both John and myself when I say we look forward to hearing back from you about what you were able to achieve with Visual Studio, Windows 8.1, and this reference on your desk.

What This Book Is About

The purpose of this book is to explain how to write applications—mainly Windows Store apps—that are based on the Windows Runtime. The intent is to explore every available API, exposing you to possibilities across all areas and diving deep into major areas that are likely common to most apps that will be built. Instead of a traditional reference guide that shares API details and code snippets, this book includes more than 80 sample projects. These projects provide a “by example” approach to learning the various APIs; and the text either walks through how they were built, or breaks apart the code step-by-step to make it easy to understand and use as a template for your own projects.

This book is not an introduction to Windows 8.1. We assume you have some experience working with C# and XAML and are familiar with Windows Store apps. We also assume that you are at least familiar with the concept of design patterns and the notion of decoupled code. Both of these ideas have been core to the success of the applications we’ve helped build and will be used as foundations for the concepts presented in this book.

Whether you’re a Windows 8.1 developer looking to improve an existing app, or an experienced client technologies developer transitioning to the Windows Runtime for the first time, this book will give you the guidance, proven patterns and practices, and example projects you’ll need to build functional apps that run well across the myriad Windows 8.1 devices.

This version of the book specifically addresses Windows 8.1 using Visual Studio 2013. At this writing, the Windows 8.1 Update was announced at //BUILD, but fortunately the changes did not impact development as much as use of the OS and deployment options. During the course of this book, several changes have occurred that may not be reflected throughout: Visual Studio 2013 Update 2 was released, the name SkyDrive was changed
Where to Access the Source Code

The source code for this book is open source and will be maintained and updated as needed to match any future revisions that may come out. You can download the code samples from the companion website: winrtexamples.codeplex.com.

How to Use This Book

The aim of this book is to enable you to discover the appropriate APIs to build your Windows Store apps. Each chapter is designed to help you discover what features are available in that area of the framework and how they are applied through example projects. Code examples are provided that demonstrate the features for programming them using C# and XAML. Although different chapters may relate to various parts of a comprehensive project, the individual samples are designed to stand on their own.

Each chapter is similarly structured. The chapters begin with an introduction to a topic and an inventory of the capabilities that topic provides. This is followed by explanations of areas of the framework and runtime and a walkthrough of the target APIs. The code samples are explained in detail, either as a walkthrough “lab” or by analyzing the existing sample, and the topic is summarized to highlight the specific information that is most important for you to consider.

I suggest you start by reading the book from start to finish, regardless of your existing situation. Inexperienced developers will find their understanding grows as they read each chapter and concepts are introduced, reinforced, and tied together. Experienced developers will gain insights into areas they might not have considered or had to deal with in the past, or simply didn’t factor into their software lifecycles. Once you’ve read the book in its entirety, you will then be able to keep it as a reference guide and refer to specific chapters any time you require clarification about a particular topic.
Acknowledgments

Jeremy Likness: Although this is my third book through Pearson and fourth full book I’ve authored, writing a good book still depends on a solid team. I continue to be grateful for my superhuman Editor, Joan Murray, who has been patient and understanding, encouraging, and continuously provided her support and guidance throughout the process. Once again, Eleanor Bru braved working with me on this very ambitious project and, like Joan, was very patient and understanding while keeping me honest and on target. I can’t thank Lori Lyons and the production team (including Krista Hansing and Debbie Williams) enough for taking my rambling and helping turn it into coherent prose.

The content of this book was amazingly enriched by our thorough and passionate technical editors. Thank you, Harry Pierson and Christophe Nasarre, for your incredible attention to detail. If anything was missed, I’ll take the blame because Harry and Christophe ran every example, pored over every word, and provided me with volumes of suggestions and feedback that helped shape the book to its present form. It is always a pleasure to work with technical editors who bring strong technical insights to the table and help keep me honest when I want to take a shortcut and leave a thread spinning where it shouldn’t.

Many thanks to my boss and friend, Steve Porter, for letting me devote a large chunk of my time to a project that made me disappear for a few hours every day. Thanks to Barbara Keihm for her support and encouragement,
to Todd Fine for always recognizing our hard work and being one of the first to pre-order copies whenever they are available, and Bethany Vananda and Sara Faatz for working tirelessly to help spread the word and share what we’re doing.

A special note goes to Dave Baskin, Dave Black, Josh Carroll, Aaron Carta, Phil Denoncourt, Dave Frommer, James Katic, Edward Kim, Wes McCammon, and Dan Sloan. This team worked with me on a major project that has lasted longer than the writing of this book and always understood when I had to turn down dinner or other outings so I could get back to my hotel and write. OK, who am I kidding—sometimes I managed to break away.

My wife and daughter have waited patiently through several books now, so they know the routine. Doreen is always quick to remind me when I need to push away from the dinner table and get back to writing, but Lizzie always noticed when I’d been writing too much and was always ready to have a movie date so I could unwind.

Finally, last but certainly not least, thank you! I appreciate my readers—and of course it is for you this was written—so it is my sincere hope you receive tremendous value from these pages.

**John Garland:** Like Jeremy, I’d very much like to thank Joan Murray, Eleanor Bru, and Lori Lyons, as well as everyone else at Pearson for their unwavering help and guidance throughout this project. Many thanks go to Harry Pierson and Christophe Nasarre for their invaluable help and insight throughout the technical review process—especially for helping to me find the right mix of code and prose, which invariably was along the lines of less prose and more code.

I’d like to very much thank my friends and colleagues at Wintellect. It is truly a privilege for me to count myself in your company and your passion for your craft is absolutely contagious. Many thanks to Steve Porter and Todd Fine for the continued opportunity, and to Bethany Vananda for all the help in putting my work in the best possible light. Much gratitude is owed to Jeff Richter, Jeff Prosise, and John Robbins for their insights into the writing process and for providing the Wintellect stage that I am fortunate to be able to stand on.
Families often have to take a back seat when these projects are in high gear, and mine was no exception. My wife Karen has been more than understanding and forgiving of many late nights, lost weekends, and grumpy mornings. My daughter Callie continues to be a walking smile that forces me to keep things in perspective, despite our having had to skip a few of our priceless Daddy-Callie days. Now that the book is done and the snow has melted, we can get back to bike rides, games of tag, and swing-pushes in the backyard.

I owe many thanks to the folks on and involved with the Zumo (Azure Mobile Services) team, including Kirill Gavrylyuk, Yavor Georgiev, Merwan Hade, and Heinrich Nielsen, among several others. Your insights into the Mobile Services inner workings, and prompt and helpful replies to my inquiries, have been invaluable both for the content included in this book as well as in my professional endeavors.

Finally, I’d like to thank Jeremy for asking me to come along not only on this ride as his co-author, but also as a technical editor on two of his previous books. The experiences, insights, and most importantly, the friendship, have been both personally and professionally invaluable.
About the Authors

Jeremy Likness is a multi-year Microsoft MVP for XAML technologies. A Principal Consultant for Wintellect with 20 years of experience developing enterprise applications, he has worked with software in multiple verticals ranging from insurance, health and wellness, supply chain management, and mobility. His primary focus for the past decade has been building highly scalable web-based solutions using the Microsoft technology stack with client stacks ranging from WPF, Silverlight, and Windows 8.1 to HTML5 and JavaScript. Jeremy has been building enterprise line of business applications with Silverlight since version 2.0, and he started writing Windows 8 apps when the Consumer Preview was released in 2011.

Prior to Wintellect, Jeremy was Director of Information Technology and served as development manager and architect for AirWatch, where he helped the company grow and solidify its position as one of the leading wireless technology solution providers in the United States prior to their acquisition by VMware. A fluent Spanish speaker, Jeremy served as Director of Information Technology for HolaDoctor (formerly Dr. Tango), where he architected a multilingual content management system for the company’s Hispanic-focused online diet program. Jeremy accepted his role there after serving as Development Manager for Manhattan Associates, an Atlanta-based software company that provides supply chain management solutions.
John Garland is a Principal Consultant for Wintellect with more than 15 years of experience developing software solutions. Prior to consulting, he spent much of his career working on high-performance video and statistical analysis tools for premier sports teams, with an emphasis on the NFL, the NBA, and Division 1 NCAA football and basketball. His consulting clients range from small businesses to Fortune-500 companies, and his work has been featured at Microsoft conference keynotes and sessions.

John is a Microsoft Client Development MVP, as well as a member of the Windows Azure Insiders and Windows Azure Mobile Services Advisory Board. He lives in New Hampshire with his wife and daughter, where he is an active speaker and participant in the New England software development community. He is a graduate of the University of Florida with a Bachelor’s degree in Computer Engineering and holds Microsoft Certifications spanning Windows, Silverlight, Windows Phone, and Windows Azure. John is the author of the ebook *Windows Store Apps Succinctly* (Syncfusion, 2013).
This page intentionally left blank
Network connectivity is a major feature of most Windows Store apps, as you learned in previous chapters. Although you have learned how to connect to services and keep your content fresh, Windows 8.1 devices are capable of connecting to the Internet and other devices in myriad ways. In this chapter, you learn some of these more advanced methods and how to integrate them into your own apps.

In addition to supporting the HTTP protocols, WinRT provides APIs that make it easy to enumerate resources on your HomeGroup network. You can enumerate network information and obtain the current data plan so that your app can modify its behavior to avoid downloading large amounts of data over a metered connection. The sockets APIs enable low-level communications using traditional UDP and TCP protocols, as well as the newer HTML5 WebSockets protocol. The proximity APIs enable communications between peer devices using Near Field Communications (NFC) and Wi-Fi Direct. Finally, the background transfer API allows your app to effectively manage long-running data transfers even when the app itself is not running.

Web and HTTP

In Chapter 5, “Web Services and Syndication,” you learned how to use the HttpClient class to connect to an HTTP server and retrieve content
using the REST architecture. The `Windows.Web.Http` namespace contains several classes that you can use to connect with HTTP-based services. The `HttpClient` class represents a simple and easy-to-use interface for sending HTTP-related requests and retrieving responses. Other classes provide more advanced features and fine-grained control over interactions.

To provide more control over HTTP requests, use the `HttpRequestMessage` class. For example, the following requests content from my blog:

```csharp
var client = new HttpClient();
var httpResponse = await client.GetAsync(new Uri(
    "http://csharperimage.jeremylikness.com/", UriKind.Absolute));
```

If you want more control over the type of request and process the request immediately after the headers have been read (instead of having to wait for the entire body), you can issue the request like this instead:

```csharp
var client = new HttpClient();
var request = new HttpRequestMessage(
var response = await client.SendRequestAsync(request,
    HttpCompletionOption.ResponseHeadersRead);
```

Using the latter method also gives you more control over the response. You can create a cancellation token and convert the response to a `Task` that uses the token:

```csharp
this.cancellation = new CancellationTokenSource();
var response = await client.SendRequestAsync(
    request, HttpCompletionOption.ResponseHeadersRead)
    .AsTask(cancellation.Token);
```

When the page takes a significant time to load, from either a slow network or a large amount of information, you can cancel the load automatically or through user input by calling the cancel method on the cancellation token. You see an example of this in the `CancelUrl` method of the `ViewModel` class in the `AdvancedHttpExample` project:

```csharp
cts.Cancel();
cts.Dispose();
```
The project enables you to enter a URL and then downloads and displays the content. The initial request ends when the headers are received so that you can stream the content with progress updates. You can cancel longer-running downloads and watch the progress. The content is exposed through the Content property of the HttpResponseMessage that is returned. The LoadUrl method demonstrates creating a progress handler that takes a type ulong and asynchronously downloads the content as a string.

```csharp
this.progress = new Progress<ulong>(ProgressHandler);
var stringContent = await response.Content
    .ReadAsStringAsync().AsTask(cancellation.Token, this.progress);
```

The progress handler is passed the number of bytes received and uses the dispatcher to set them as a property on the viewmodel to show the progress to the user.

```csharp
private void ProgressHandler(ulong progressArgs)
```

If you use the default URL of my blog, the content loads immediately and the progress method never gets called. Using a longer URL, such as the URL to a large book such as Ulysses in HTML format from the Gutenberg project, results in a longer download and progress updates. The URL, listed in the source of the viewmodel, to make it easy for you to copy, is www.gutenberg.org/files/4300/4300-h/4300-h.htm.

You can also use the request message to post content, including streams, to the server. The Content property of the HttpRequestMessage can be assigned any instance that implements IHttpContent. This includes the following content:

- HttpBufferContent—Content that uses an IBuffer instance
- HttpFormUrlEncodedContent—Content that uses name/value pairs for a form post
- HttpJsonContent—Content that is represented using the JSON format
- HttpMultipartContent—Content that uses the multipart MIME type for uploading multiple attachments
- `HttpMultipartFormDataContent`—A special format for forms encoded using the `multipart/form-data` MIME type
- `HttpStreamContent`—Content that uses a stream, such as when uploading files to the server
- `HttpStringContent`—Content that uses a string

The HTTP API also provides the `HttpProgress` class for tracking and handling the progress of long-running HTTP uploads. Simply create an instance of the progress handler and pass it to the extension method that converts the call to a `Task`:

```csharp
var progress = new Progress<HttpProgress>(ProgressHandler);
HttpResponseMessage response = await httpClient.PostAsync(
    resourceAddress, streamContent).AsTask(cts.Token, progress);
```

The signature of the handler is a simple method that takes an instance of `HttpProgress` and can query items such as bytes sent versus total bytes sent, number of retries, and the stage of the process (for example, sending or receiving content).

**HomeGroup**

Microsoft provides a special service named HomeGroup that is designed to make it easier to share folders, files, and devices on home networks. If you are not familiar with HomeGroup, Microsoft provides an online tutorial to help you set one up “from start to finish.”

The Windows shell handles the special network behind the scenes and exposes it as a file system in Explorer.

Figure 10.1 shows an example folder in the HomeGroup. Notice that the initial set of “folders” corresponds to users on the network, followed by the machines they are logged into. These, in turn, expose libraries based on the user’s preferences for sharing pictures, documents, music, or other items. You can browse to the folders you have permissions for and access the items as you normally would.

---

1HomeGroup from start to finish, http://bit.ly/1ak28nC
The **HomeGroupExample** project for Chapter 10 demonstrates access to the HomeGroup. The first step is to declare your capabilities in the package manifest. You must have at least one of the available library capabilities (music, pictures, or videos) checked, or you will receive an access denied exception when you attempt to access the HomeGroup. Otherwise, you will have access only to the folder types that you specified capabilities for.

Use the `KnownFolders.HomeGroup` enumeration to access the HomeGroup network. The first set of folders you receive is mapped to the usernames of users currently participating in the HomeGroup. The following code in the `Initialize` method of the `ViewModel` class fetches the user-level folders:

```csharp
var folders = await Windows.StorageKnownFolders.HomeGroup.GetFoldersAsync();
```

The example project defines the `HomeGroupUser` class for user information and maps the `DisplayName` attribute of the folder to the username displayed.
foreach (var user in folders.Select(
    folder => new HomeGroupUser
    {
        UserName = folder.DisplayName,
        IsHomeGroupUser = true
    }))) { this.Users.Add(user); }

When you have a StorageFolder instance for the user, you can use queries to iterate items within the folder. This query sets up a search for pictures with a known set of filename extensions and ultimately retrieves any shared photos that user is sharing across all devices on the HomeGroup.

var query = new QueryOptions(CommonFileQuery.OrderBySearchRank,
    new() { ".jpg", ".png", ".bmp", ".gif" }
    { UserSearchFilter = "kind:picture" });
var files = await targetFolder.CreateFileQueryWithOptions(query).GetFilesAsync();

The app is designer-friendly and shows a sample image and title in the designer. When you run the app, you see either an error message displayed on a disabled button if the app cannot access a valid HomeGroup, or a list of buttons for each user on the HomeGroup. Tap the button to see the images that user is sharing. You can use similar functionality as covered in Chapter 4, “Data and Content,” to access other folders and content types.

**Connectivity and Data Plans**

Windows Store apps can be connected in a number of ways. Although traditional wired connections (Ethernet LAN) and Wireless Fidelity (WiFi) connections (also known as wireless local area connections, or WLAN) are still popular, many devices offer wireless wide area network (WWAN) connections over cellular technologies such as Global System for Mobile Communications (GSM) and Long Term Evolution (LTE). Many of these data plans have data limits and may charge for bandwidth usage. If users roam outside their regular coverage area, they could incur additional charges.

Windows Store apps should be aware of the type of connection they are using to access information over the Internet so they can implement specific behaviors that are suitable for the type of connection. An app might consider implementing this typical set of behaviors:
• **Offline**—The app cannot connect to the Internet and must rely on local cached data to function.

• **High Cost**—The app is connected to the Internet, but the data plan is either roaming, approaching a fixed data limit, or over the data limit and, therefore, might incur additional charges. The app should limit network activity to only extremely low bandwidth scenarios (such as loading a set of headers but deferring the details).

• **Conservative**—The app is connected to the Internet over a metered connection. Downloading data is fine but should be done only as needed and based on user-configurable preferences (the user must have a way to disable large downloads when the connection is metered). Lower-resolution images and lesser-bandwidth movies should be used when available.

• **Standard**—The app is connected to the Internet, and no charges appear to be associated with data usage; therefore, the application can download or upload data as needed.

The Windows.Networking.Connectivity namespace contains the APIs necessary to determine the types of connections that are available and examine data plans and usage. You interact with the NetworkInformation class to determine the available connections, the connection your app will use to access the Internet, and what type of connection is being used. The example app that demonstrates this API is called **NetworkInfoExample**; you can find it in the Chapter 10 solution folder.

Each network that your device either is currently connected to or has connected to in the past (as long as you did not ask Windows to forget the connection) has a ConnectionProfile instance associated with it. The UpdateNetworkInformation method in the ViewModel class in the Data folder demonstrates how to access this API. A simple call retrieves the full list of available profiles:

```csharp
var profiles = NetworkInformation.GetConnectionProfiles();
```

You can iterate the various profiles and acquire information from each of them, but the most interesting profile is the one used to gain access to
the Internet. You can use the `GetInternetConnectionProfile` call to get the profile associated with the active connection, if one exists. If the result is null, the user is not currently connected. In the example app, this call is used to get the identifier for the network adapter that is being used to connect and then select that connection from the list. If your connection is bridged for any reason (for example, you might be running Hyper-V virtual machines that use virtual adapters to connect to your wireless connection), the bridged connection might show up as the active connection instead of the connection you were expecting.

The `ConnectionProfile` has a name that matches what you see in the various network dialogs (either the list of available connections from the `Control Panel` or the list of networks in the `Networks` flyout accessed from the Charms bar). It indicates whether the network is a WLAN (wireless) or a WWAN (wide area network or cellular) connection. If it is neither, it is likely a wired Ethernet or Bluetooth connection.

You can quickly access information about the connected network adapter, as well as the security settings for the connection. For example, the wireless access point I run in my house uses RSNA-PSK authentication with CCMP encryption. You might have security settings available for both wired and wireless networks. The `FromConnectionProfile` method on the `ConnectionInfo` class demonstrates how these values are obtained.

```csharp
if (profile.NetworkSecuritySettings != null)
{
    connectionInfo.AuthenticationType = profile
    connectionInfo.EncryptionType = profile
}
```

Other information is available through method calls. To get the signal strength from the connection (a value that ranges from 0 for no signal to 5 for maximum signal strength), you call the `GetSignalBars` method. The example app shows only four of five possible bars because it uses the built-in symbol library, and that provides only four bars.

```csharp
connectionInfo.SignalBars = profile.GetSignalBars();
```
The main reason for examining the connection is likely to understand whether costs are associated with it. To find out, call the GetConnectionCost method. This returns a class that contains an enumeration and several flags. The enumeration provides you with details about how the connection is metered.

- **Unrestricted**—No costs are associated with data usage.
- **Fixed**—A data limit exists; until that limit is reached, usage is unrestricted.
- **Variable**—Data usage is charged on a per-byte basis.
- **Unknown**—No cost information is available for the connection.

Additional flags provide further insights into the current plan:

- **Roaming**—This flag is set when the user is outside the normal usage area. You can assume that additional charges will apply.
- **ApproachingDataLimit**—The plan has almost reached its limit; additional costs might be incurred.
- **OverDataLimit**—The plan has exceeded the data limit, and the user is likely being charged for any additional usage.

Use this information to strategize how you will access the Internet from your Windows Store app. When the type is fixed or variable, you should follow a conservative behavior. When the flags indicate that the connection is roaming or over the data limit, you should implement the high-cost behavior and allow the user to opt in to any data usage. Other scenarios can follow the standard or offline behavior, depending on the status of the connection.

If you need to find out more details about the plan, you can call the GetDataPlanStatus method, as shown in the FromProfile method on the DataPlanInfo class in the example app. The result gives you more details when available, including the data limit and how much has been used against the limit, the available speeds of the connection, and even when the next billing cycle begins so you know when the usage is reset.
You can also query for historical usage of any connection. The `GetNetworkUsageAsync` method enables you to specify a time range and a sample frequency (increments in minutes, hours, or days, or a total for the time period). Depending on how you call the method, you can get a list of `NetworkUsage` instances for each data sample. If you requested hourly samples, each instance represents a sample taken for a given hour. The instance contains the duration it represents, along with the bytes received and sent during that period. The `ConnectionInfo` class in the example app retrieves a total for the previous day:

```csharp
var usage =
    await profile.GetNetworkUsageAsync(
        DateTimeOffset.Now.AddDays(-1),
        DateTimeOffset.Now,
        DataUsageGranularity.Total,
        new NetworkUsageStates { Roaming = TriStates.DoNotCare,
            Shared = TriStates.DoNotCare });
```

You might not sample data earlier than 60 days before the current date (about 2 months), and minute granularity is available for only the previous 2 hours. You can also specify what network states you want to sample. You can restrict the data to times when the connection was roaming or part of a shared connection, or indicate that you “do not care,” as in the example code.

The advantage of many Windows 8.1 devices is that they are highly mobile. For this reason, it’s common for the current active connection to change frequently. The user might be using a cellular connection and might come into range of a wireless connection that is lower cost, or the user might travel and switch to different connections. The `NetworkInformation` class raises an event when the current connection status changes. The `ViewModel` class in the example app registers for this event:

```csharp
NetworkInformation.NetworkStatusChanged +=
    this.NetworkInformationNetworkStatusChanged;
```

The event itself does not provide other information. The typical practice is to query for the current Internet connection again to determine whether the app behavior should change. You can prompt the user or restrict data usage when you find that the user has roamed or moved from an
unrestricted connection to a metered one. By default, Windows 8.1 prefers unrestricted networks over metered networks and automatically connects to the fastest available network in its category when multiple choices are available.

**Sockets**

Windows Store apps have the capability to communicate over lower-level networking protocols. The Windows Runtime provides built-in support for User Datagram Protocol (UDP), Transmission Control Protocol (TCP), Bluetooth RFCOMM, and the recent HTML5 WebSocket Protocol. Support for socket-based operations is provided through the types of the Windows.Networking.Sockets namespace. Sockets in general provide low-level network communications and enable real-time network notifications.

**WebSockets**

The WebSocket protocol was designed to be implemented in web browsers and web servers, and it is fully supported from Windows Store apps. Although it is part of the HTML5 group of specifications, it is an independent TCP protocol. Its main advantage is that it provides a way for the browser or Windows Store app to maintain a single connection with a server and send data both ways while keeping that connection open. The standard port for WebSockets is 80, the same one HTTP uses, which means it is less likely to be blocked by firewalls.

The WebSocketsExamples project for Chapter 10 demonstrates two APIs you can use from WinRT to take advantage of the WebSockets protocol. The example app leverages a server supplied by the WebSocket.org website that provides an “echo service.” This service, when connected to,

---


5 WebSocket Protocol, RFC 6455
echos back any data sent to it. WebSockets are accessed using a standard URI, as declared in MainPage.xaml.cs:

```csharp
private readonly Uri echoService =
    new Uri("ws://echo.websocket.org", UriKind.Absolute);
```

The `MessageWebSocket` class is an abstraction of the protocol that focuses on sending simple messages. A message is either read or written in a single operation, instead of being streamed continuously. It is also the class you must use to support UTF8 messages; the stream-based API supports only binary (although you can encode and decode the binary to and from UTF8, the `MessageWebSocket` class provides native support for this). To use any socket type within a Windows Store app, you must enable a networking capability such as **Internet (Client)**.

The `ButtonBase_OnClick` method in the `MainPage.xaml.cs` file demonstrates how to use the `MessageWebSocket` class. After creating an instance of the class, set the type of the message (either binary or UTF8):

```csharp
this.socket.Control.MessageType = SocketMessageType.Utf8;
```

You can also register for events that fire whenever a message is received and when the socket is closed. The socket uses underlying unmanaged resources, and you should dispose of it when you are done using it. The easiest way to do this is to call `Dispose` in the `Closed` event handler.

Initiate the connection by calling and waiting for `ConnectAsync` to complete:

```csharp
await this.socket.ConnectAsync(echoService);
```

The example app accepts any message you type and sends it to the echo service. The message must be sent using the `OutputStream` property exposed by the socket. The easiest way to do this is to create an instance of a `DataWriter` to send the message. The `DataWriter` enables you to write various data types that it buffers until you call `StoreAsync`. This flushes the buffer to the underlying stream.

```csharp
var writer = new DataWriter(this.socket.OutputStream);
writer.WriteString(this.Text.Text);
await writer.StoreAsync();
```
Not all error messages for the socket are mapped to .NET Exception class instances. Instead, you must inspect the HRESULT of the underlying exception to determine what went wrong. Fortunately, the WebSocketError class provides a static method that translates the result to the corresponding WebErrorStatus enumeration. The ToErrorMessage method returns a string with the original message and the enumeration value.

```csharp
private static string ToErrorMessage(Exception ex) {
    var status = WebSocketError.GetStatus(
        ex.GetBaseException().HRESULT);
    return string.Format("{0} ({1})", ex.Message, status);
}
```

The MessageReceived event is raised whenever a message is sent from the server to the client through the socket. In the example app, this should happen any time data is sent because the server echoes back the data. The event provides the socket that the information was received from with event arguments: You can inspect the message type (binary or UTF8) and open a reader or stream to access the message. In this example, the reader is set to use UTF8 encoding; then it obtains the message and displays it in the SocketMessageReceived event handler.

```csharp
using (var reader = args.GetDataReader()) {
    reader.UnicodeEncoding = UnicodeEncoding.Utf8;
    var text = reader.ReadString(reader.UnconsumedBufferLength);
    this.Response.Text = text;
}
```

This is the simplest method for dealing with sockets that are designed to share messages. When you are using the socket to stream real-time information and you don’t necessarily have simple messages, you might want to use the StreamWebSocket implementation instead. It provides a continuous two-way stream for sending and receiving information. The example app uses the same echo service to stream prime numbers and echo them back to the display when you click the Start button.

You create and connect to a StreamWebSocket the same way as with a MessageWebSocket. You can also register for the Closed event. Instead of sending and receiving messages, however, the stream version expects you
to interface directly with the input and output streams provided by the
socket. The app starts a long-running Task encapsulated in the ComputePrimes
method. It is passed the OutputStream of the socket. It iterates through posi-
tive integers and writes out any that are computed to be primes; then it
delays for 1 second:

```csharp
if (IsPrime(x))
{
    var array = Encoding.UTF8.GetBytes(string.Format(" {0} ", x));
    await outputStream.WriteAsync(array.AsBuffer());
    await Task.Delay(TimeSpan.FromSeconds(1));
}
```

If the integer is not a prime, it delays for a millisecond just to prevent
hogging the CPU. Another long-running task receives the echo. It allocates
a buffer, waits for data to arrive in the stream, and then reads and decodes
the data.

```csharp
var bytesRead = await stream.ReadAsync(buffer, 0, buffer.Length);
if (bytesRead > 0)
{
    var text = Encoding.UTF8.GetString(buffer, 0, bytesRead);
    this.DispatchTextToPrimes(text);
}
```

This example also demonstrates that you can have multiple sockets
open to the same server and port at once. You can run the example, click
the button to start generating primes, and then use the message-based
version to send and receive messages without interrupting the stream of
prime numbers. Both methods for communicating with the socket simplify
the amount of code you have to write by not worrying about the details
of the underlying transport (TCP). When you need to manage a raw TCP
connection, you can use the traditional sockets components.

**UDP and TCP Sockets**

UDP and TCP protocols have been around for decades. Many modern pro-
tocols, including HTTP, sit on top of these more low-level protocols (TCP is
the transport used by both HTTP and the WebSocket protocol you learned
to use in the previous section). Two main differences exist between UDP
and TCP: UDP does not require a connection, and UDP does not require
any special ordering of packets or chunks of data. As a result, TCP tends to be more reliable and useful for bidirectional communication, and UDP is used when faster transmission rates are required and the application understands how to deal with unordered data.

Examples of protocols that sit on top of UDP include Domain Name Service (DNS) and Simple Network Management Protocol (SNMP). Protocols that sit on top of TCP include HTTP and Simple Mail Transfer Protocol (SMTP). The UDP classes are all prefixed with `Datagram` and operate similarly to the TCP classes prefixed with `StreamSocket`. The API enables you to “connect” to either protocol and send or receive messages. This provides a consistent interface and approach to using each protocol. The main difference is that no specific “listener” service for the UDP implementation exists because a persistent connection is not needed. Instead, you simply create a socket, register for the event when a message is received, and then send data packets or process incoming data as needed.

The `SocketsGame` example provides a more comprehensive example of using a persistent TCP connection. Although the game starts a server to listen for incoming requests, it should be clear that you cannot use these types of connections for communication between Windows Store apps on the same machine. Network isolation prevents the loopback interface from allowing connections across processes. The only reason this works in the example project is that the client and server are hosted in the same process. The example should show how to spin up a server to listen when necessary (for example, the same type of connection can be used to host a service for a Bluetooth service that allows Bluetooth devices to connect), as well as act as a client for a server hosted on the Internet.

The game itself is a text-based adventure game. It creates a 10x10 matrix of rooms for 100 rooms total and randomly connects rooms and places trophies in the various rooms. The object of the game is to explore the rooms and collect trophies until all have been found. A rudimentary parser accepts commands such as “look,” “get,” “north,” and “inventory.” Instead of playing as a local game, however, the game is hosted on a socket; the app must connect as a client to issue commands and receive updates.

Two sockets are defined in `MainPage.xaml.cs`: a `StreamSocketListener`, which is the server that listens for and establishes connections to clients,
and a StreamSocket, which emulates a client connecting to the server. The server provides several options to bind to a generic service and listen to all incoming connections, to bind to a specific address, or even to bind to a specific network adapter. The service name can be a local service name or a port, or it can remain empty to have a port assigned. If you are using the socket for Bluetooth (RFCOMM), use the Bluetooth service ID. In this example, the name is set to 21212 as a unique port for the game. Binding enables your app to use that specific port to listen for incoming requests. If another app has already bound to the specified service, an exception is thrown.

```csharp
this.serverSocket = new StreamSocketListener();
this.serverSocket.ConnectionReceived +=
    this.ServerSocketConnectionReceived;
await this.serverSocket.BindServiceNameAsync(ServiceName);
```

As with Web Sockets, to understand errors thrown by the sockets API, use the GetStatus static method of the SocketError class, as shown in the GetErrorText method.

```csharp
private static string GetErrorText(Exception ex)
{
    return string.Format("{0} {{1}}", ex.Message,
                   SocketError.GetStatus(ex.GetBaseException().HResult));
}
```

When a connection is received, the server creates a persistent writer and reader for the connection (note that this example uses exactly one client, so only one writer and reader are used—if you are building a server to manage multiple connections, you need to spin up a new reader and writer for each unique connection).

```csharp
if (serverWriter == null)
{
    serverWriter = new DataWriter(args.Socket.OutputStream);
    serverReader = new DataReader(args.Socket.InputStream);
}
```

The listener for the socket goes into an infinite loop waiting for messages. As messages are received, they are passed to the parser to interact with the game world, and the result is written back to the client. To
facilitate communication over the socket, the messages are written with a special format. The size of the string in bytes is sent ahead of the string itself so that the reader can allocate the appropriate buffer size to process the incoming message. The `SendString` method encodes the text and sends it over the socket.

```csharp
writer.WriteUInt32(writer.MeasureString(text));
writer.WriteString(text);
await writer.StoreAsync();
```

Listing 10.1 shows the `GetStringFromReader` method that receives the incoming data. It loads enough data to constitute an unsigned integer, processes the integer, and finally loads enough data to create a string based on the size that was passed in.

**LISTING 10.1  Reading a String from the TCP Socket**

```csharp
private static async Task<string> GetStringFromReader(    
    IDataReader reader)
{
    var sizeFieldCount = await reader.LoadAsync(sizeof(uint));
    if (sizeFieldCount != sizeof(uint))
    {
        return string.Empty;
    }
    var stringLength = reader.ReadUInt32();
    var actualStringLength = await reader.LoadAsync(stringLength);
    if (stringLength != actualStringLength)
    {
        return string.Empty;
    }
    var data = reader.ReadString(actualStringLength);
    return data;
}
```

Just as the server goes into an infinite loop after a connection is received, waits for instructions, and then returns a response, the client also starts a long-running task. On the UI thread, the `Go_OnClick` method is called whenever the user clicks the button to send the next command. The click handler simply sends the command to the socket and then forgets about it. The long-running `ClientListener` method waits to get the data from the server and then writes it for the end user to see.
Figure 10.2 shows a game in progress. At the top, you can see the server messages that involve receiving the incoming connection, receiving commands, and sending responses. The bottom is the client console for game play; it shows all the responses from the server and provides an input box for the user to type and send commands.

The provided example handles both client and server aspects for TCP connections. The RFCOMM for Bluetooth uses the same classes. Although UDP uses a different set of classes, the implementation is similar—the only difference is that you don’t create a persistent listener for managing connections because the protocol is stateless.
Near Field Communications (NFC) is a set of standards based on Radio-Frequency Identification (RFID) standards for smartphones, tablets, smart tags, and other devices to establish communications in extremely close situations (less than a few inches difference). Two main NFC scenarios exist. The first is a tap gesture for a short transmission of information, such as contact information, a URL, or a “smart poster.” The second is a similar gesture used to create a handshake between two devices so they can establish a peer-to-peer connection over wireless to exchange large amounts of information.

NFC not only operates over extremely short distances, but it also has a fairly slow transfer rate, with theoretical speeds between 50 and 100 bytes per second. For this reason, it is useful for exchanging only a small amount of information, unless you use the NFC tap to establish a more persistent connection over a longer range and using faster technology, including Bluetooth, Wi-Fi, and Wi-Fi Direct. The WinRT API fully supports both of these scenarios.

When you exchange information via NFC, you must either send or receive a message encoded in the NFC Data Exchange Format (NDEF). This is a lightweight, platform-independent binary format for exchanging messages. The message allows one or more specific payloads (referred to as NDEF records) to be sent in a single package. Windows provides built-in support for a set of proprietary NDEF records that Windows 8.1 and Windows Phone devices can exchange. You can also format and exchange other types of records that target other platforms or are platform-independent by either building your own payload or using an open source library such as the NDEF Library for Proximity APIs that is available as a NuGet package.7

---


The **ProximityExample** project provides some examples of using the Proximity APIs defined in the `Windows.Networking.Proximity` namespace. The `ProximityDevice` class provides the simplest API to use and focuses specifically on short-range, short-duration NFC scenarios. To see whether the system has a proximity device available, simply call the `GetDefault` static method, shown in the constructor of the `ViewModel` class. Be sure to declare the **Proximity** capability in the application’s manifest.

```csharp
this.proximityDevice = ProximityDevice.GetDefault();
```

The call returns null when a device is not present. If this is the case on your machine, you will not be able to take advantage of NFC exchanges and gestures, but you may still be able to create peer-to-peer connections using Bluetooth, Wi-Fi, or Wi-Fi Direct. You learn more about that in a later section. The proximity device exposes properties for its unique identifier, the maximum number of bytes it can send in a single message, and the bits per second it is capable of transmitting or receiving. You can also register for events that fire when another proximity device comes within range:

```csharp
this.proximityDevice.DeviceArrived +=
    this.ProximityDeviceDeviceArrived;
this.proximityDevice.DeviceDeparted +=
    this.ProximityDeviceDeviceDeparted;
```

The events are purely informational and do not provide any specific information. The `ProximityDevice` parameter of the handler is a reference back to the device that detected the event, which, in most cases, is the default device referenced in the constructor. Other classes exist for enumerating multiple proximity devices, in the rare case that the machine has multiple ones installed. This is a rare scenario because one NFC device is usually sufficient.

An easy way to share information with another NFC device is to use the `PublishMessage` method on the `ProximityDevice` class. This method is useful for sharing simple string data with other Windows or Windows Phone devices. It takes two parameters: the message type and the message itself. The message type is a unique identifier that enables other devices to determine how to handle the message. The message type always starts with a
Protocol, followed by a dot, followed by whatever custom identifier you prefer. In this case, the protocol must always be Windows. (The simple code for publishing and subscribing in this section is shared here for reference purposes but is not part of a specific example project.)

```csharp
var publishedMessageId =
    proximityDevice.PublishMessage("Windows.WinRTByExampleMessage",
"This is a simple message.");

The publication is not a transient event. The message will be available until you explicitly stop publishing, so multiple NFC devices over time can connect and subscribe for that message to receive it. To stop publishing, you call the StopPublishingMethod on the ProximityDevice.

```csharp
proximityDevice.StopPublishingMessage(publishedMessageId);
```

If you want to know when the message has been transmitted, you can pass a MessageTransmittedHandler as a third parameter when you publish. The handler is called with the proximity device and the identifier for the message. You can use this to log that the message was transmitted, or even unsubscribe in the callback to ensure that the message is sent only once.

```csharp
private void MessagePublished(ProximityDevice sender,
    long messageId)
{
    proximityDevice.StopPublishingMessage(messageId);
}
```

To receive a message, you use the SubscribeForMessage method on the ProximityDevice class. You do not have to wait for a device to arrive or depart before you subscribe, and the subscription is valid for any device that publishes that particular message type. The subscription includes a handler that is called whenever the message is received, and it is provided a unique identifier that you can use to unsubscribe when you want to stop receiving the message.

```csharp
var subscribedMessageId =
    proximityDevice.SubscribeForMessage("Windows.
    \nWinRTByExampleMessage",
    MessageReceived);
```
The method to receive the message is passed the `ProximityDevice` and a `ProximityMessage`. The message includes the data as a buffer, the data as a string, and the subscription ID, in case you want to use that to stop subscribing.

```csharp
private void MessageReceived(ProximityDevice device, ProximityMessage message)
{
    var messageText = message.DataAsString;
    device.StopSubscribingForMessage(subscribedMessageId);
}
```

The subscription method enables you to subscribe to any type of message. For messages that use non-Windows protocols, you need to decode the message. For example, the message type `WindowsUri` provides a URI, but you must first decode it from UTF16LE:

```csharp
void messageReceivedHandler(ProximityDevice device, ProximityMessage message)
{
    var buffer = message.Data.ToArray();
    var uri = Encoding.Unicode.GetString(buffer, 0, buffer.Length);
}
```

Note that some devices, such as the Windows Phone, handle URIs at the operating system level. In other words, you cannot override the default behavior. The OS itself intercepts the NFC tag and opens the corresponding program. The program depends on the protocol. HTTP launches the Internet Explorer browser and navigates to the encoded web page, and a `mailto` protocol results in the default mail program being launched.

You can use the NFC API to write to smart tags, or special tags that use induction to store and publish information. Smart tags have varying capacities, depending on the manufacturer. Publishing to a smart tag always overwrites the data, and most smart tags have a lifetime of several hundred thousand writes. To get the capacity of a smart tag, you can subscribe to the `WriteableTag` message. This transmits an `Int32` message that contains the capacity of the tag.
private void MessageReceived(ProximityDevice device,
    ProximityMessage message)
{
    var capacity = System.BitConvert.ToInt32(
        message.Data.ToArray(), 0);
}

Table 10.1 lists the various message types you can subscribe to.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>Consists of raw binary data.</td>
</tr>
<tr>
<td>Windows.*</td>
<td>Provides a custom string type proprietary to Windows, where * represents</td>
</tr>
<tr>
<td></td>
<td>a custom type.</td>
</tr>
<tr>
<td>WindowsUri</td>
<td>Consists of a UTF-16LE encoded URI string. Note that the operating system</td>
</tr>
<tr>
<td></td>
<td>shell intercepts these messages and marshals them to the appropriate protocol</td>
</tr>
<tr>
<td></td>
<td>handler.</td>
</tr>
<tr>
<td>WindowsMime</td>
<td>Contains a specific MIME type–like image/jpeg for a bitmap image.</td>
</tr>
<tr>
<td>WriteableTag</td>
<td>Published by smart tags when they come within range of reading or writing.</td>
</tr>
<tr>
<td></td>
<td>Contains the capacity of the smart tag in bytes.</td>
</tr>
<tr>
<td>NDEF[:*]</td>
<td>Consists of formatted NDEF records. Third-party libraries are available to</td>
</tr>
<tr>
<td></td>
<td>easily encode and decode these record formats.</td>
</tr>
</tbody>
</table>

You also can publish messages for cross-platform compatibility or for the purpose of writing to smart tags. Instead of using the proprietary PublishMessage method, use the PublishBinaryMessage method. You can use this method to publish messages to other NFC devices, but it is also useful for writing messages to smart tags. The following code snippet encodes the URI to launch Skype and calls the echo service on a Windows or Windows Phone device.

```csharp
var uri = new Uri("skype:echo123?call");
var buffer = Encoding.Unicode.GetBytes(uri.ToString());
var publishId = device.PublishBinaryMessage("WindowsUri:WriteTag", buffer.AsBuffer());
```
Table 10.2 lists various protocols you can use when writing messages to tags.

**TABLE 10.2 Message Protocols for Writing to Smart Tags**

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows:WriteTag</td>
<td>Publish binary data to a static smart tag</td>
</tr>
<tr>
<td>WindowsUri:WriteTag</td>
<td>Write a URI to a static smart tag</td>
</tr>
<tr>
<td>LaunchApp:WriteTag</td>
<td>Write a tag that launches an app with specific launch parameters</td>
</tr>
<tr>
<td>NDEF:WriteTag</td>
<td>Write a cross-platform message using the NDEF format</td>
</tr>
</tbody>
</table>

To write a tag that launches an app, use the `LaunchApp:WriteTag` format; then provide a tab-delimited list that starts with the text to pass in as an argument and then includes pairs of platforms and application names. You can find the application name for a Windows 8.1 application in the application manifest. It is in the format of the Package family name (from the Packaging tab) and an exclamation mark. The following tag passes an argument named `id` with a value of 1 to both the Windows 8.1 ProximityExample app and a fictional app on Windows Phone 8 (the application name on Windows Phone is simply the GUID for the application ID).

```javascript
var launchTag = "id=1\tWindows\tWinRTByExampleProximityExample_req6rhny9ggkj! " + "ProximityExample.App\tWindowsPhone\t{063e933a-fc8e-4f0c" + "-8395-ab0e84725f0f}";
```

If the app is present on the target device, it is launched with the arguments passed (the user is always prompted to opt in for the launch whenever this type of tag is encountered). If the app is not present, the device automatically takes the user to the app’s entry in the Windows Store. This makes the tag extremely useful: If you pass out smart tags with the encoding, users can easily discover and install your app, as well as subsequently launch it.
In this section, you learned ways to publish small messages that can be sent to other devices or encoded in smart tags. You also learned how to subscribe to and receive these messages. I mentioned earlier a way to share much more information than permitted by the limited bandwidth and speed of the NFC protocol. In this next section, you explore the tap-to-connect scenario that uses NFC to establish a persistent peer-to-peer connection for exchanging information.

**Tap-to-Connect Scenarios**

The `PeerFinder` class enables you to find and interact with other devices capable of peer-to-peer communications. Although a common use case is through NFC, you can also use Bluetooth and Wi-Fi Direct to locate and communicate with peers. The WinRT API abstracts these decisions from you and enables you to focus on the actual process of locating a peer and establishing a socket so that you can stream data back and forth.

Even if you don’t have a proximity device, chances are good that you can take advantage of the `ProximityExample` sample app to create a peer-to-peer connection. That’s because the WinRT API supports a browse scenario using Wi-Fi Direct, a technology that enables peer-to-peer wireless connections between devices that exists in most modern radios. Using the browsing scenario, you can install the app on two different devices and use them to discover each other.

The proximity APIs support finding peers running the same application. The application is defined by the package family, a unique identifier for your app that is shared across target platforms. For this reason, your app on a machine running Windows 8.1 can easily connect to the same app on a machine running Windows RT. You can also extend the peer to find instances of your app on other platforms, such as Windows Phone and Android. The `PeerFinder` class contains a dictionary named `AlternateIdentities` that hosts a list of platforms and application identifiers. In the previous section, you learned how to create a tag that launches the application and can contain multiple platforms and identities. You can add the same identifier to recognize that app as a peer like this:

```csharp
PeerFinder.AlternateIdentities.Add("WindowsPhone", 
  "{063e933a-fc8e-4f0c-8395-ab0e84725f0f}");
```
You can discover and negotiate the peer connection either through an NFC tap gesture or by browsing Wi-Fi Direct. After the devices recognize each other and initiate the handshake, Windows tries to connect simultaneously using infrastructure (wireless or wired), Wi-Fi Direct, and Bluetooth. It uses whichever connection completes first (most likely, Bluetooth, when available) and passes the connection as an active socket to your app. You can restrict which connection types to allow by setting the static AllowBluetooth, AllowInfrastructure, and AllowWiFiDirect properties on the PeerFinder class.

The PeerSocket class in the example app provides a convenient way to manage a persistent socket connection. It takes a StreamSocket in the constructor and immediately creates a persistent reader and writer to interact with it.

```csharp
public PeerSocket(StreamSocket socket)
{
    this.socket = socket;
    reader = new DataReader(socket.InputStream);
    writer = new DataWriter(socket.OutputStream);
}
```

It exposes a write method that uses the DataWriter to send a message to the socket and starts an infinite loop that runs on a background thread to listen for incoming messages. When it receives an incoming message, it raises an event so the app can register for the event, receive the message, and process it (in the case of the sample app, by marshalling it to the UI thread and showing it on the display). It also raises an error event whenever it encounters an error and disposes of both the reader and the writer when its own Dispose method is called.

To begin the process of connecting with a peer, you must first set your app to advertise. This broadcasts its identity over Wi-Fi Direct and makes it available for tap gestures if a proximity device is present. The Wi-Fi Direct mode is referred to as a browsed connect, and the NFC mode is referred to as a triggered connect. The PeerFinder class is instructed to begin advertising in the StartPeerFinder method on the ViewModel class.

First, the app registers to two events: the TriggeredConnectionStateChanged that is raised when an NFC tap gesture is received, and the ConnectionRequested
event that is raised when another device browses your device and requests a connection.

```csharp
PeerFinder.TriggeredConnectionStateChanged +=
    this.PeerFinderTriggeredConnectionStateChanged;
PeerFinder.ConnectionRequested +=
    this.PeerFinderConnectionRequested;
```

Next, the role is set. Three possible roles exist. In the Peer role (included in the example app), two apps can connect with each other and communicate as peers. In a client/server scenario, one app can serve as the host and must set the Host role; then up to four other apps can connect using the Client role. Note that only Peer roles can browse to each other. The Host role can browse only Client roles, and vice versa.

```csharp
PeerFinder.Role = PeerRole.Peer;
```

Finally, some discovery text is set. This is additional text you can share, such as an application name, an invitation to connect, information about the host system, or any other data up to 240 bytes in length. This data is broadcast and can be displayed when browsing. After the data is set, the PeerFinder starts advertising when you call the Start method.

```csharp
PeerFinder.DiscoveryData = Encoding.UTF8.GetBytes(
    DiscoveryText).AsBuffer();
PeerFinder.Start();
```

When both peers have started advertising, one of two scenarios can take place. The first is the NFC tap-to-connect scenario. When the proximity devices are tapped together, the TriggeredConnectionStateChanged event is raised. This event fires multiple times as the devices come within range and negotiate a connection.

The event handler for the triggered connection receives a State property of the type TriggeredConnectState (an enumeration). The handler on the viewmodel is called PeerFinderTriggeredConnectionStateChanged. The Listening state indicates that the proximity device is waiting for a tap. When the state is PeerFound or Connecting, the connection is being established and the handler simply updates the status for the user. If the connection fails, a
Failed state is passed. The Completed state indicates success, and the arguments contain a Socket property with the active socket between the two devices:

```csharp
case TriggeredConnectState.Completed:
    this.RouteToUiThread(() => { this.IsConnecting = false; });
    this.InitializeSocket(args.Socket);
    break;
```

The InitializeSocket method sets up an instance of the PeerSocket to handle further communications. A state of Canceled means the connection was broken for some reason—for example, the devices moved out of range or a user intervention occurred.

The browse scenario starts when you request a list of available peers. The BrowseCommand method on the viewmodel calls the FindAllPeersAsync method and then loads the results to the list of available peers.

```csharp
var peers = await PeerFinder.FindAllPeersAsync();
```

The user can then select a peer and request a connection. The connection is initiated in the ConnectCommand method.

```csharp
var socket = await PeerFinder.ConnectAsync(
    this.SelectedPeer.Information);
    this.InitializeSocket(socket);
```

Note that the end result is the same as the triggered connection scenario: A socket is obtained and initialized to establish communications. The mode of the connection is transparent to your app, and there is no way to determine whether the connection was made using Bluetooth, infrastructure, or Wi-Fi Direct (unless you have restricted the allowable connection types to a single mode).

If your device is running a version of the app and the connection is requested from another device, a ConnectionRequested event is raised. The viewmodel handles this in the PeerFinderConnectionRequested method. In this scenario, you typically prompt the user to confirm that he or she wants to accept the request, and then either ignore the request or connect. The sample app automatically initiates the connection. The method to connect is identical for the host, client, or peer; the only difference is that, instead of
passing a peer from a list of selections, the peer requesting the connection is passed as arguments to the event.

var socket = await PeerFinder.ConnectAsync(args.PeerInformation);
this.InitializeSocket(socket);

If the call succeeds for both peers, a connection is established and duplex communication can be initiated. You can transmit anything over the binary socket—from images, to streaming videos, to text or documents. The sample app simplifies the connection by transmitting only text. The text you enter is sent to the peer via the output stream of the socket, and any text received raises an event that is marshalled to the UI.

To use the sample program, install it on two Windows 8.1 devices that support Wi-Fi Direct or have proximity devices. The easiest way is to build and deploy the source, but you can also use the Store option on the Project Properties menu to create a side load package. Copy the package to a thumb drive and execute the included PowerShell script to install it on the other device.

Run the app on both devices. You must start advertising on both devices to establish a connection. After you’ve started advertising, either tap the devices or tap Browse to use Wi-Fi Direct. If you browse, select another machine and tap Connect. When the connection is established, via either NFC tap or browsing, you can begin to send messages between the two peers (see Figure 10.3).

Figure 10.3  Example of communicating between peers using the Proximity API
Numerous possibilities exist for taking advantage of the peer connection. You can use it to share documents or pictures between devices, archive data, create a chat session, or even share game state in a multiplayer game. The API handles all the necessary low-level handshakes and connectivity so that you can focus on the implementation of your application without worrying about the underlying NFC protocol or even whether the devices connect over Bluetooth or Wi-Fi Direct. The Proximity API is nearly identical on the Windows Phone, making it possible to build apps that span devices and create a truly continuous user experience among Windows PCs, tablets, and phones.

**Background Transfers**

Many apps must download large amounts of information to present to the user. For example, an app focused on providing instructional videos might need to download new videos from the Internet. These files could be hundreds of megabytes or even gigabytes in size. Although the `HttpClient` class is capable of retrieving files of this size, you must also take into account the application lifecycle.

As you learned in Chapter 2, “Windows Store Apps and WinRT Components,” whenever the user moves your app into the background, your app can be suspended or frozen, essentially stopping any downloads dead in their tracks. In some scenarios, the app might even be terminated, forcing you to create a new instance of the class in an attempt to start the download again. Fortunately, WinRT provides a way to handle this specific scenario using a background task.

You learn more about background tasks in Chapter 15. This chapter introduces a specific API for downloading files that exists in the `Windows.Networking.BackgroundTransfer` namespace. The API is defined for several reasons. The most obvious is to enable your app to download files without interruption. These download tasks should continue even if your app is swapped to the background or terminated. You should also be able to discover any existing downloads when your app is launched again, to either continue to download or cancel them as needed. The extra advantage this API provides is a power-friendly and cost-aware means of transferring
Background Transfers

files. The API is architected to handle the download in a way that maximizes battery life and can pause the transfer when the user switches to a metered network. These features combine to provide the best mobile experience possible for the device user.

The reference project TapAndGoProximityNetworking serves two purposes. As a follow-up to the previous section about the Proximity API, it downloads an excellent video presentation by my colleague Jeff Prosise from Microsoft’s Channel 9 website. His talk, given at TechEd Europe in 2013, covers the Proximity API and provides working examples of encoding tags, reading tags, and tapping to share data between multiple devices. It is a great way to reinforce the information you learned in the previous section. The project downloads a high-fidelity version of the video that is almost 600MB in size. The second purpose is to demonstrate the background transfer capabilities.

To simplify the example, I placed all the code in the code-behind of the main page to simply download a file and then play it using the file launcher. The associated video player should pick up the file and begin playing the presentation after it is downloaded. The app first checks to see whether the movie already exists, based on a specific name in your video library. The Video Library capability must be enabled in the manifest for this to work. If the video exists, you are given the option to delete it to start over or launch it.

To start a background transfer, you need only two pieces of information: the URI of the resource to download and a file to download it to. The example app encodes the URI to the video download and creates a file with the name TapAndGo_Prose.mp4 in your video library in the DownloadOnClick method.

```csharp
var source = new Uri(DownloadUri, UriKind.Absolute);
var destinationFile =
    await KnownFolders.VideosLibrary.CreateFileAsync(
        LocalName, CreationCollisionOption.ReplaceExisting);

An instance of the BackgroundDownloader class is created, and the CreateDownload method is called with the source and destination.

```csharp
var downloader = new BackgroundDownloader();
download = downloader.CreateDownload(source, destinationFile);
```
You can provide a callback to receive updates as the download progresses. This is done by creating an instance of the Progress class of type DownloadOperation and passing the callback handler, as shown in the DownloadProgressAsync method.

```csharp
var progress = new Progress<DownloadOperation>(UpdateProgress);
```

The download is then kicked off and cast to a Task with a cancellation token and the callback for progress.

```csharp
await this.download.StartAsync().AsTask(cts.Token, progress);
```

The download is now kicked off and continues to execute even after your app terminates. If it encounters an error, it updates the error state for your app to query when the app is launched again. While the app is running, it provides progress updates, as shown in the UpdateFromProgress method.

```csharp
BytesReceived.Text = download.Progress.BytesReceived.ToString();
TotalBytes.Text = download.Progress.TotalBytesToReceive.ToString();
```

Table 10.3 lists the possible statuses available via the Progress.Status enumeration. Use this to determine the state of the download and take appropriate action (in the example app, it is used to enable or disable the Pause and Resume buttons).

**Table 10.3 BackgroundTransferStatus Enumeration**

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>The application is idle (the download is still active).</td>
</tr>
<tr>
<td>Running</td>
<td>The transfer is in progress.</td>
</tr>
<tr>
<td>PausedByApplication</td>
<td>The app has paused the download by calling the Pause method on the DownloadOperation.</td>
</tr>
<tr>
<td>PausedCostedNetwork</td>
<td>The user transitioned to a metered network, and the download has been paused to avoid additional cost. It will resume when the user returns to a nonmetered network.</td>
</tr>
</tbody>
</table>
### Status Description

<table>
<thead>
<tr>
<th>Status</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PausedNoNetwork</td>
<td>The user has lost network connectivity. The download will resume when Internet connectivity is restored.</td>
</tr>
<tr>
<td>Completed</td>
<td>The operation successfully completed.</td>
</tr>
<tr>
<td>Canceled</td>
<td>The operation was canceled.</td>
</tr>
<tr>
<td>Error</td>
<td>An error was encountered.</td>
</tr>
</tbody>
</table>

While the download is running, you can perform a number of actions. For example, you can call the Pause method on the DownloadOperation to temporarily pause the download. After it is paused, you can call Resume to continue the download. Calling Pause twice in a row or calling Resume before Pause results in an exception, so always keep track of or check the current status. If you passed a cancellation token to the task, you can also call Cancel on the token source to abort the download.

If the download completes while your app is still running, it returns control after `await` of the StartAsync call. The example app disposes of the cancellation token and then launches the video. If your app is terminated or exits before the download is finished, it will continue in the background. When the app is launched again, you can check for existing transfers, as the CheckState method shows.

```csharp
var downloads = await BackgroundDownloader.GetCurrentDownloadsAsync();
```

An entry for the download exists whether it is still downloading or it completed when your app was not running. Either way, you can obtain the reference to the download, query the status, or attach to receive updates. The sample app always reattaches to update the status. If the download has completed, the call to AttachAsync returns immediately; otherwise, it continues the same way the call to StartAsync worked.

```csharp
await this.download.AttachAsync().AsTask(cts.Token, progress);
```
To test the app, compile, deploy, and run it. Tap the Download button. You then see a status similar to Figure 10.4. You can pause, resume, or cancel the download. After the download has begun, close the app by stopping it if you are running through the debugger or by pressing Alt+F4. You can navigate to the video library and refresh the file list to verify that the download is still running. Start the app again; it should return to the progress display and begin showing you the current progress. If you let the download finish, the app automatically launches the video and closes itself.

![Figure 10.4 The download progress](image)

The transfer API enables you to launch multiple downloads and keep track of each download individually. You can also group downloads and perform various tasks on the group. In addition, you can set a priority for the download and even request that the download run unconstrained so that it happens more quickly. This prompts the user and also can affect battery life and quality of the user experience. You learn more about the various background APIs in Chapter 15.

**Summary**

In this chapter, you learned how to use advanced features of the HttpClient. You used the Windows 8.1 seamless integration of HomeGroup technology to enumerate resources on your home network and then queried network information to determine what type of connection was active and see whether it was a metered plan. You leveraged the Sockets APIs to transfer messages and packets of data between a client and a server. You learned
how to use NFC to transmit short, fast messages; subscribe to messages; and write data to smart tags. The APIs also enable a scenario to tap and create a persistent connection over your wired or wireless infrastructure, Bluetooth, or Wi-Fi Direct. Finally, the background transfer API enabled an app to download a large video resource even when it wasn’t running.

In Chapter 11, “Windows Charms Integration,” you learn more about the special icons that appear on the right side of your monitor when you swipe or hold down Windows+C. These icons, called charms, provide a special way for your app to integrate with the OS and communicate with other apps. Using charms enables scenarios such as streaming media to a projector, using one app to take notes and then sending those notes to another app to post them online, or accessing the specific settings of various apps in a consistent way.
This page intentionally left blank
In earlier chapters, you saw that although the built-in controls you can use in your Windows 8.1 apps include extensive support for touch-based interactions, input from mouse and keyboard input devices continues to be fully supported. The Windows Runtime also features extensive support for gathering information from other inputs, including sensors. The information these sensors provide includes details about a device’s location, as well as knowledge about its position and motion within its immediate environment. Having the capability to incorporate this information into your apps means you can consider giving your users new kinds of interactivity and immersion.

In this chapter, you see how the WinRT APIs provide a common model for working with the various kinds of input pointer devices. This model provides a range of access, allowing you not only to obtain information about raw pointer events, but also to work with higher-level abstract gestures, depending on the needs of your app. You also see how you can access keyboard events from your code and obtain information about the user’s key presses.

In addition, you learn about the WinRT APIs for working with location information, including the capability to set up geographic fences that can result in automatic notifications to your app when your device crosses a fence boundary. Furthermore, you learn how to work with the WinRT APIs that provide access to sensors that can give you information about your
device’s interactions with the physical world around it, including details about its orientation, its heading, the rate and direction of its motion, and even the amount of light currently shining on it.

**Working with Input Devices**

In Chapter 2, “Windows Store Apps and WinRT Components,” you saw how the built-in controls that the Windows Runtime provides are designed to support first-class interactions through touch, as well as keyboard and mouse combinations. Although access to touch input is becoming more common in modern computers and devices, it is not yet available everywhere. Attached keyboards, mouse devices, and pens continue to be important tools for application interaction, not only when touch input is unavailable, but also in addition to touch input when certain interactions are simply easier and more natural using these other input mechanisms.

For touch, mouse, and pen inputs, the Windows Runtime API provides several different kinds of methods and events for working with these devices and responding to user interaction with them. In addition to the APIs for working with these devices, a set of methods and events are available for responding to user interactions with their keyboards.

**The Example App**

The *InputsExample* project illustrates several kinds of input device API integration that you can add to your apps. The app enables the user to add shapes to the application canvas, which are then animated to move around the canvas area. The app also detects what input devices are available and shows information about these connected devices, and it provides options for configuring what device types the app will listen to for input and which of the screen or keyboard events the app will respond to. Shapes can be added through buttons provided on the user interface or by pressing predefined keyboard buttons. The shapes themselves are configured to respond in several ways to interaction with pointer input devices. When a pointer intersects the edge of a shape, the shape is highlighted and stops moving. The shapes can also be manipulated to change position, degree of rotation, and size, with or without inertia. Finally, the shapes respond to
gestures by changing color when tapped, changing direction when double-tapped, and resetting to their initial size, color, and rotation when they are held or right-clicked.

**Identifying Connected Input Devices**

You can determine which touch input devices are connected and what their capabilities are in a couple ways. One approach is to use the information that the `PointerDevice` class provides to obtain detailed information about available touch, mouse, or pen devices. Alternatively, higher-level classes can garner more general information about the current mouse and touch capabilities.

The `PointerDevice` class can obtain detailed information about one or more connected pointer devices. It provides a static `GetPointerDevices` method that returns a list of available devices as `PointerDevice` object instances, as well as a static `GetPointerDevice` method that can retrieve a specific device based on a pointer ID value (the “Pointer Events” section, later in this chapter, explains how to obtain a pointer ID). Properties of particular interest that the `PointerDevice` type exposes include the `PointerDeviceType`, which shows whether the device is a Mouse, Touch, or Pen device, and the `IsIntegrated` flag, to indicate whether the device is considered to be integrated into the current machine or has been connected externally. It also includes a `SupportedUsages` collection that lists Human Interface Device (HID) “usages” as `PointerDeviceUsage` objects. These usages are defined by Usage Page and Usage Id values that are part of the USB HID specification\(^1\) and expose value ranges that the pointer device supports.

Listing 13.1 shows how the example application uses device information to determine whether touch, mouse, or pen devices are available. A list of available devices is obtained depending on whether the list should include only integrated devices. The resulting values are then queried to see if any of the desired device types are present.

---

\(^1\)USB HID information, [www.usb.org/developers/hidpage](http://www.usb.org/developers/hidpage)
LISTING 13.1 Determining Device Availability

```csharp
var devices = PointerDevice.GetPointerDevices();
if (PointerIntegratedDevicesOnly)
{
    devices = devices.Where(x => x.IsIntegrated).ToList();
}
IsTouchAvailable
    = devices.Any(x => x.PointerDeviceType == PointerDeviceType.Touch);
IsMouseAvailable
    = devices.Any(x => x.PointerDeviceType == PointerDeviceType.Mouse);
IsPenAvailable
    = devices.Any(x => x.PointerDeviceType == PointerDeviceType.Pen);
```

The MouseCapabilities and TouchCapabilities classes obtain higher-level system-wide information about the available mouse and touch device support. When an instance of one of these types is created, its properties provide access to information about the respective device availability.

For MouseCapabilities:

• The MousePresent property is set to a value of 1 if one or more mouse devices are currently available.

• The NumberOfButtons value indicates the highest value available for any given device.

• The VerticalWheelPresent or HorizontalWheelPresent properties is set to a value of 1 to indicate whether a device is connected that has each respective feature.

• The SwapButtons property is set to 1 if the mouse buttons have been swapped in the system settings.

For TouchCapabilities:

• The TouchPresent property returns a value of 1 if a touch digitizer is present.

• The Contacts property indicates the highest number of concurrent contacts that are supported.

The example application uses these values to populate the message boxes that display when the user clicks the Details buttons next to the check boxes that it provides to enable or disable mouse and touch input (see Listings 13.2 and 13.3).
LISTING 13.2  Displaying Mouse Capabilities

```javascript
var capabilities = new MouseCapabilities();
String message;
if (capabilities.MousePresent == 1)
{
    var rawMessage =
        "There is a mouse present. " +
        "The connected mice have a max of {0} buttons. " +
        "There {1} a vertical wheel present. " +
        "There {2} a horizontal wheel present. " +
        "Mouse buttons {3} been swapped.";

    message = String.Format(rawMessage
        , capabilities.NumberOfButtons
        , capabilities.VerticalWheelPresent == 1 ? "is" : "is not"
        , capabilities.HorizontalWheelPresent == 1 ? "is" : "is not"
        , capabilities.SwapButtons == 1 ? "have" : "have not"
    );
}
else
{
    message = "There are no mice present.";
}
ShowMessage(message, "Mouse Properties");
```

LISTING 13.3  Displaying Touch Capabilities

```javascript
var capabilities = new TouchCapabilities();
String message;
if (capabilities.TouchPresent == 1)
{
    var rawMessage =
        "Touch support is available. " +
        "Up to {0} touch points are supported.";

    message = String.Format(rawMessage, capabilities.Contacts);
}
else
{
    message = "Touch support is not available.";
}
ShowMessage(message, "Touch Properties");
```
**Pointer, Manipulation, and Gesture Events**

Instead of having a separate set of input events for touch, mouse, and pen inputs, the Windows Runtime API combines input from these devices and provides several distinct tiers of events that can be raised in response to input from any of these devices. At the lowest tier are the pointer events, which are raised for each press, move, release, or other simple interaction. Next are the manipulation events, which track and consolidate actions from one or more pointers into higher-level events related to motion, scale, rotation, and inertia. Finally, the gesture events consolidate pointer actions into even higher-level gesture abstractions, such as tapping, double-tapping, and holding.

In the example application, all the support for working with input device pointer, manipulation, and gesture events has been consolidated into a single `InputEventHandler` class. This class handles the subscriptions to the desired events and provides the event handler implementations for these subscriptions.

**NOTE**

Chapter 2 introduced you to the Visual Studio simulator for Windows Store Apps, which enables you to run and test your Windows 8.1 app within a simulated environment on your development system. Ultimately, testing touch support in an application is best done with a device that actually has touch support. However, if you happen to be using a development environment that does not provide this support, using the simulator’s touch-emulation features is a good start toward exercising this kind of functionality in your app. Ultimately, however, it is a good idea to make sure your app is exercised for some amount of time in an actual touch environment.

**Pointer Events**

The Windows Runtime combines input from touch, mouse, or stylus devices into the abstract concept of a pointer. Each contact point from each device is represented by a unique pointer instance. For example, imagine an app running on a touch-enabled tablet that supports multiple touch points, and imagine that multiple fingers are pressing the screen simultaneously. In
this case, each finger touching the screen is treated as a unique pointer. The same holds true if the touch actions include a combination of several fingers, as well as a click by a mouse or screen contact with a stylus. The mouse and/or stylus inputs are treated as additional unique pointers.

In Windows 8 XAML apps, the most common way to subscribe to pointer events is through events that individual UIElement objects expose. An alternative approach involves subscribing to similar events exposed by an ICoreWindow instance, which can be obtained through the Window.Current.CoreWindow property. This latter approach is primarily used by DirectX WinRT games when UIElement objects aren’t readily available. Table 13.1 summarizes the pointer events that are available when a UIElement is used.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PointerEntered</td>
<td>A pointer has moved into the item’s bounding area. For mouse and stylus input, this does not require a press. For touch input, because there is no “hover” support, an actual touch is required; it results in an immediate subsequent PointerPressed event, unless cancelled in this event’s handler.</td>
</tr>
<tr>
<td>PointerExited</td>
<td>A pointer that was in an element’s bounding area has left that area. For touch input, this event immediately follows a PointerReleased event.</td>
</tr>
<tr>
<td>PointerPressed</td>
<td>A pointer has been pressed while within the bounding area for an item. Note that a PointerPressed is not always terminated by a PointerReleased event, but it can instead be ended by PointerCanceled or PointerCaptureLost events.</td>
</tr>
<tr>
<td>PointerMoved</td>
<td>A pointer that has entered an item’s bounding area is being moved within that area, or a pointer that has been captured by an item is moving, even if its position is beyond the item’s bounding area.</td>
</tr>
<tr>
<td>PointerReleased</td>
<td>A pointer that was pressed has been released, usually within an item’s bounding area. This occurs if the pointer was pressed while inside the item’s bounding area; a corresponding PointerPressed event then has been raised, or if the pointer was already pressed when it moved into the item’s bounding area, the PointerPressed event might have occurred elsewhere. If the pointer is currently captured by an item, this event can also be raised when the pointer is released outside the item’s boundary.</td>
</tr>
</tbody>
</table>
**Event** | **Description**
--- | ---
PointerCanceled | A pointer has lost contact with an item in an unexpected way. This event can fire instead of the PointerReleased event. Potential reasons for unexpected contact loss include changes in an app’s display size, the user logging off, or the depletion of available contact points. Note that this event is only part of the UIElement events, and the ICoreWindow interface does not provide or raise it.

PointerCaptureLost | A pointer capture that the event source item obtained has been released either programmatically or because a corresponding PointerPressed has been released.

Several of the pointer events in Table 13.1 either are directly related to or have side effects that are related to the idea of a pointer being captured. When a pointer is captured, only the element that captured it receives any of the input events related to that pointer until the capture has been released. Typically, a pointer is captured within the handler for a PointerPressed event because a pointer must be pressed to be captured. To capture a pointer, the UIElement class includes a CapturePointer method that takes a Pointer class instance that identifies the pointer to capture. It just so happens that the PointerRoutedEventArgs that are passed to the UIElement pointer event handlers include this pointer object, as the following code illustrates:

```csharp
private void HandlePointerPressed(Object sender, PointerRoutedEventArgs args)
{
    _eventSourceElement.CapturePointer(args.Pointer);
}
```

The Pointer object includes a PointerId, which is simply a unique integer that is assigned to the current pointer and identifies it throughout the various subsequent pointer events. It also includes a PointerDeviceType property that returns a value of the PointerDeviceType enumeration and indicates whether the current pointer is related to input from a touch device, a mouse device, or a pen device. In the example project, this value
is used to ignore processing in the pointer events when a particular device type is deselected in the user interface.

```csharp
    if (!IsValidDevice(args.Pointer.PointerDeviceType)) return;
```

The `Pointer` object also includes a pair of flags to indicate the position of the pointer relative to the touch sensor. `IsInContact` indicates whether the device is actually contacting the sensor, such as whether a stylus is in direct contact with the screen when using a touchscreen tablet. In the case of a mouse device, this is true when one of its buttons is being pressed. `IsInRange` indicates whether the device is within detection range but not touching; it is primarily meant for pen devices because, unlike touch devices, they can usually be detected before they make physical contact. Generally, mouse devices always return `true` for this value, and touch devices return `true` only when a touch is actually occurring.

In addition to the `Pointer` object, the arguments passed to the pointer events include a `KeyModifiers` property that indicates whether one or more of the Control, Menu, Shift, or Windows special keyboard keys was pressed at the time of the event.

Finally, the event arguments include a pair of methods that obtain additional information about the input pointer associated with the current interaction. The `GetCurrentPoint` and `GetIntermediatePoints` methods both accept a `UIElement` to provide a frame of reference for any of the coordinate properties included in the method results. If this value is `null`, the coordinate values that are returned are relative to the app itself. Whereas `GetCurrentPoint` returns a single `PointerPoint` instance, the `GetIntermediatePoints` returns a collection of `PointerPoint` instances from the last pointer event through the current one. In addition to being able to obtain `PointerPoint` information from the pointer event arguments, the `PointerPoint` class itself includes static methods that accept a `PointerId` value and return the current or intermediate `PointerPoint` values, with coordinates relative to the app.

The `PointerPoint` class includes a lot of information about the current interaction. At the root, it includes the `PointerId` value, a `Position` value indicating the point where the pointer event occurred, and a `PointerDevice` property that provides the same `PointerDevice` value discussed in the earlier section “Identifying Connected Input Devices.” It also includes a
Properties value that provides access to significantly more detailed information. Among the properties provided, this value includes touch information, such as the contact rectangle value; mouse information, such as whether the left, middle, right, first extended, or second extended buttons are pressed; and pen information, including several values that describe the physical position of the pen, whether it is inverted, and the amount of pressure being applied to its tip. Furthermore, the HasUsage and GetUsage methods are useful in obtaining HID value information from the device for the current interaction. These are the same HID values that can be enumerated with the SupportedUsages method that PointerDevice class instances mentioned earlier provide. The following code shows how to request the amount of tip pressure (usageId value 0x30) applied to a digitizer stylus device (usagePage value 0x0D).

```csharp
if (pointerDetails.Properties.HasUsage(0x0D, 0x30))
{
    pressure = pointerDetails.Properties.GetUsageValue(0x0D, 0x30);
}
```

Although the amount of detail provided by the pointer events can harness a lot of power, the information provided is at a very low level. For most application needs, this information needs to be synthesized into more abstract concepts. Examples might include recognizing a pair of PointerPressed and PointerReleased events potentially as either a single tap or a hold action, depending on how much time elapses between the two pointer actions, or perhaps tracking multiple pointer actions to determine whether pinch or rotation actions are occurring. Fortunately, you will most likely not need to write and maintain the state-tracking code required to achieve this level of abstraction; these kinds of events are already calculated and provided for you in the form of the manipulation events and gesture events.

**Manipulation Events**

Manipulation events are the result of grouping and translating several pointer events associated to an item that originate from either one or several pointers. During a manipulation, changes to translation (position), scale (size), and rotation are computed, tracked, and made available via the event argument parameters provided by these events. A manipulation
also tracks the velocities with which these changes are occurring and includes the capability to optionally calculate and apply inertia based on these velocities when the pointer events complete.

In Windows 8.1 XAML apps, the most common way you subscribe to manipulation events is through the events that individual UIElement objects expose. For a UIElement to generate manipulation events, the element needs to have its ManipulationMode property set to a value of the ManipulationModes enumeration other than None or System. The default value for most controls is System, and it enables the UIElement to process manipulations internally, whereas a value of None suppresses all manipulations. Other significant values include TranslateX and TranslateY to track movement on the x- and y-axis, Rotate to track rotation, and Scale to track stretching or pinching. Values for TranslateInertia, RotateInertia, and ScaleInertia are also available to indicate that these manipulations should trigger inertia calculations. Table 13.2 summarizes the manipulation events exposed by the UIElement class.

### Table 13.2 Manipulation Events

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ManipulationStarting</td>
<td>A PointerPressed event has occurred, and manipulation processing starts looking for the pointer to move, to actually start tracking a manipulation.</td>
</tr>
<tr>
<td>ManipulationStarted</td>
<td>A pressed pointer has moved. This marks the beginning of the manipulation, which contains some number of Manipulation-Delta events and is concluded with a ManipulationCompleted event.</td>
</tr>
<tr>
<td>ManipulationDelta</td>
<td>One or more of the pressed pointers have moved or inertia is being applied.</td>
</tr>
<tr>
<td>ManipulationInertiaStarting</td>
<td>The manipulation has been configured to support inertia, and the last pointer was released while the manipulation still had a velocity. ManipulationDelta events are raised until velocity falls below the inertia-defined threshold.</td>
</tr>
<tr>
<td>ManipulationCompleted</td>
<td>The last pointer is no longer pressed, and any inertia calculations have completed.</td>
</tr>
</tbody>
</table>
The first event received during a manipulation is the `ManipulationStarting` event. This event includes a `Mode` property that initially matches the `ManipulationMode` value set on the `UIElement` object. It allows the types of manipulations that will be tracked to be modified one last time before the manipulation tracking actually starts. If a pressed pointer is moved, the `ManipulationStarted` event is fired, followed by one or more `ManipulationDelta` events as the pointer continues to move.

The arguments provided to the `ManipulationDelta` event handler provide the information that can be used to react to the manipulation. The arguments contain some general-purpose informational properties that include the `PointerDeviceType`, which is the same as it was for the pointer events (note that this implies that a manipulation cannot span device types, such as a pinch occurring with both a finger and a mouse); a `Container` value that indicates the `UIElement` on which the manipulation is occurring; and an `IsInertial` flag that specifies whether the `ManipulationDelta` event is a result of inertia that occurs after pointers have been released. Of particular interest, however, are the `Delta`, `Cumulative`, and `Velocity` values.

The `Delta` property provides the changes in the values for `Translation`, `Expansion`, `Scale`, and `Rotation` that have occurred since the last `ManipulationDelta` event occurred. `Translation` indicates how much movement occurred on the x- and y-axis. `Expansion` specifies how far the distance grew or shrank between touch contacts. `Scale` is similar to `Expansion`, but it specifies the change in distance as a percentage. Finally, `Rotation` specifies the change in the rotation degrees. The `Cumulative` property returns the same items, except that the values returned are the overall changes that have occurred since the manipulation started instead of since the previous `ManipulationDelta` event. Finally, the `Velocity` provides a `Linear` property that contains the x and y velocities specified in pixels/milliseconds, an `Expansion` property that specifies the scaling change in pixels/milliseconds, and an `Angular` property that specifies the rotational velocity in degrees/milliseconds.

In the example application, the delta values are applied to the shape being manipulated to move it onscreen, resize it, or rotate it (rotation is better seen with the square shape than the circular one). Listing 13.4 shows
the event handler in the `InputEventHandler` class for the `ManipulationDelta` event.

**Listing 13.4 Handling Manipulation Changes**

```csharp
private void HandleManipulationDelta
    (Object sender, ManipulationDeltaRoutedEventArgs args)
{
    // Check to see if this kind of device is being ignored
    if (!IsValidDevice(args.PointerDeviceType)) return;

    // Update the shape display based on the delta values
    var delta = args.Delta;
    _shapeModel.MoveShape(delta.Translation.X, delta.Translation.Y);
    _shapeModel.ResizeShape(delta.Scale);
    _shapeModel.RotateShape(delta.Rotation);
}
```

The processing in the `ShapeModel` class is fairly straightforward. The `MoveShape` method simply makes sure that adding the offset values to the current position doesn’t move the shape beyond the current borders and adjusts the resulting position value accordingly. `ResizeShape` multiplies the current shape scale by the provided percentage and then makes sure the resulting shape size is within the minimum and maximum boundaries established for a shape. `RotateShape` simply adds the degree value to the current `Rotation` property. A `TranslateTransform` is bound to the shape position values. A `RotateTransform` has its `Angle` value bound to the rotation angle, as well as its `CenterX` and `CenterY` values bound to the position of the shape. Finally, a `ScaleTransform` has its `ScaleX` and `ScaleY` values bound to the scale of the shape, with the `CenterX` and `CenterY` values also bound to the shape position.

The final manipulation concept to be discussed is inertia. If one or more of the inertia `ManipulationMode` values is specified, the manipulation processing can include the application of inertia, depending on whether the last pointer involved in the manipulation was removed following an action that had a velocity. In the example app, this occurs when a shape is being dragged from one side of the screen to another and, halfway through, the finger/mouse/pen is suddenly released. In the physical world, the object
would tend to continue to slide along until slowed by friction. With manipulation support for inertia, your app can include similar behavior without any extra work on your part.

When inertia starts, the `ManipulationInertiaStarting` event is raised. The arguments for this event include the arguments that were discussed for the `ManipulationDelta` event, as well as `TranslationBehavior`, `ExpansionBehavior`, and `RotationBehavior` arguments to control the behavior of the inertia effect. Each of these values includes a value called `DesiredDeceleration` that defines the deceleration rate, as well as a value to indicate the final desired value for each property, respectively named `DesiredDisplacement`, `DesiredExpansion`, and `DesiredRotation`. You can either leave the default values in place or replace them with your own value for more control over the inertia behavior. After the handler for this event has completed, the manipulation processor automatically raises `ManipulationDelta` events with values based on the application of inertia to the current state until either the desired value is reached (if specified) or deceleration results in a velocity of zero.

When the last pointer has been released, or when inertia has completed (when specified through the `ManipulationMode` setting), the `ManipulationCompleted` event is raised, signaling that the manipulation is now complete. The arguments to this event include the general-purpose informational properties that were discussed previously, as well as the `Cumulative` and `Velocities` information that was also provided to the `ManipulationDelta` event.

---

**NOTE**

Although the manipulation and gesture events the `UIElement` class provides will take care of most needs, more control or additional gesture types are required in some cases. The Windows Runtime provides the `Windows.UI.Input.GestureRecognizer` class, which can directly process pointer events to generate these high-level events.
Gesture Events

Gesture events are similar to manipulation events, in that they are the result of grouping and interpreting several pointer events. However, a few key differences set them apart. First, gesture events communicate more abstract and discrete concepts than manipulation events. Manipulation events communicate information about the beginning, middle, and end of a manipulation and include arguments that provide information about the different kind of changes that have occurred. Gesture events each relay information about the occurrence of a single, isolated event, such as a tap or a double-tap. Second, manipulation events provide information that synthesizes input from several pointers, whereas gesture events are concerned with the action of only one pointer at a given time.

As with manipulation events, the UIElement class provides the most commonly used access to gesture events and related configuration settings. Table 13.3 summarizes the gesture events made available by UIElement instances.

<table>
<thead>
<tr>
<th>Event</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tapped</td>
<td>A tap has occurred, defined by a quick pointer press and release (where a long press followed by a release results in Holding and RightTapped events). This is equivalent to a mouse Click event.</td>
</tr>
<tr>
<td>DoubleTapped</td>
<td>A second tap has occurred after a first tap event, within a system-setting defined time. This is equivalent to a mouse DoubleClick event.</td>
</tr>
<tr>
<td>Holding</td>
<td>A long-duration press is occurring or has completed. The event is raised when the long-press is initially detected, and once again when the long-press is either completed or cancelled. Mouse devices generally do not raise this event.</td>
</tr>
<tr>
<td>RightTapped</td>
<td>A right-tap has occurred, defined by either the completion of a holding gesture (for touch and pen devices) or a click with the right button (for mouse devices). This is equivalent to a mouse RightClick event.</td>
</tr>
</tbody>
</table>
All the gesture events include a `PointerDeviceType` property that indicates the type of device that generated the event, as well as a `GetPosition` method that returns the coordinates of the action that led to the event, relative to the `UIElement` argument in the method call. If a null value is provided to `GetPosition`, the coordinates returned are relative to the app itself. The `Holding` event also includes a `HoldingState` property that is discussed shortly. Note that the `Tapped` and `Holding` events are mutually exclusive. Also, when a double-tap occurs, a `Tapped` event is raised for the first interaction, but the second one generates only the `DoubleTapped` event.

The `UIElement` class also provides the `IsTapEnabled`, `IsDoubleTapEnabled`, `IsHoldingEnabled`, and `IsRightTapEnabled` properties. By default, they are all set to `true`; setting them to `false` prevents the corresponding event from being raised.

The `Tapped`, `DoubleTapped`, and `RightTapped` events are similar, but the `Holding` event behaves a little differently. As Table 13.3 mentioned, the `Tapped` event is usually generated only by interaction with touch and stylus devices, not by mouse devices. It is also the only event that is raised when the pointer involved in the event is in a pressed state. When a pointer is pressed and held steady, and after the initial hold time interval has passed, the `Holding` event is raised with its `HoldingState` property set to a value of `Started`. After the hold has begun, if the pointer is moved or the same element captures another pointer, the hold is considered to have been cancelled and the `Holding` event is raised once again, with the `HoldingState` property set to a value of `Cancelled`. Otherwise, when the pressed pointer is lifted, the `Holding` event is raised again with a `HoldingState` property set to a value of `Completed`. If the hold was successfully completed, the `RightTapped` event follows.

In the example application, the tap-related gesture events cause different actions to happen to the shapes they occur on. The `Tapped` event changes the shape color to a random value, the `DoubleTapped` event causes the shape to take a new randomly calculated direction, and the `RightTapped` event causes the shape to be reset to its original color, size, and rotation. The code in Listing 13.5 illustrates this interaction for a `Tapped` event.
LISTING 13.5   Processing a Gesture Event

private void HandleTapped(Object sender, TappedRoutedEventArgs args)
{
    // Check to see if this kind of device is being ignored
    if (!IsValidDevice(args.PointerDeviceType)) return;

    // Examine the current position
    var position = args.GetPosition(_eventSourceElement);
    Debug.WriteLine("Tapped at X={0}, Y={1}", position.X, position.Y);

    // Alter the shape based on the gesture performed
    _shapeModel.SetRandomColor();
}

Keyboard Input
In addition to the pointer-based input devices, the Windows Runtime includes support for working with input gathered from keyboards. To obtain information about the available keyboard support, you can use the KeyboardCapabilities class. Similar to the MouseCapabilities and TouchCapabilities counterparts, it includes a KeyboardPresent property that is set to a value of 1 if one or more keyboards are currently available. The example application uses this value to provide the text for a message box that displays when the user clicks the Details button next to the Keyboard header, as in Listing 13.6.

LISTING 13.6   Displaying Keyboard Capabilities

    var keyboardCapabilities = new KeyboardCapabilities();
    var message = keyboardCapabilities.KeyboardPresent == 1
        ? "There is a keyboard present."
        : "There is no keyboard present."
    ShowMessage(message, "Keyboard Properties");

    The UIElement class provides two available keyboard events. The KeyDown event is raised when a key is pressed, and the KeyUp event is raised when a pressed key is released. These events are raised by a control only when the control has the input focus, either when the user taps inside the control or uses the Tab key to rotate focus to that control, or when the control’s Focus method has been called programatically.
As an alternative, the CoreWindow class provides three events related to keyboard interactions. Similar to the UIElement, it provides KeyDown and KeyUp events. However, these events are raised regardless of which control currently has input focus. The CoreWindow class also includes a CharacterReceived event, which is discussed in more detail shortly.

In the case of the UIElement, both the KeyDown and KeyUp events provide KeyRoutedEventArgs arguments; for the CoreWindow class, the KeyDown and KeyUp events provide KeyEventArgs arguments. The most significant difference between these argument types is the naming of the property used to identify the key involved in the action that led to the event being raised. KeyRoutedEventArgs provides a property named Key that returns a value of the VirtualKey enumeration indicating the specific key on the keyboard that was pressed or released. In the KeyEventArgs class, the corresponding property is named VirtualKey.

In either case, the KeyStatus property contains additional information about the key event. For KeyDown events, its WasKeyDown property is particularly interesting because it indicates whether the event is being raised in response to a key being held down. In this case, several KeyDown events usually are raised, followed by a single KeyUp event. The first KeyDown event has its WasKeyDown value set to false, with the subsequent KeyDown events setting the value to true.

The CharacterReceived event of the CoreWindow class was previously mentioned. This event is fired between the KeyDown and KeyUp events and provides access to the actual interpreted character resulting from the current key combination. This value is returned as an unsigned integer in the CharacterReceivedEventArgs KeyCode property. It can be converted to the corresponding Char character using the Convert.ToChar function:

```csharp
var interpretedChar = Convert.ToChar(args.KeyCode);
```

To put this in perspective, with a standard U.S. keyboard, pressing the equals (=) key while the Shift key is also pressed is interpreted to result in the plus (+) character. The KeyDown and KeyUp events understand this key only as VirtualKey 187, regardless of whether the Shift key is pressed. However,
the `KeyCode` value provided in the arguments to the `CharacterReceived` event provides either a value of 61 for the equals key or a value of 43 for the plus key.

To illustrate the use of the keyboard input events, the main page in the example application listens for `KeyUp` events via the `CoreWindow` class to add either a new ball or a square shape whenever the B or S keys are pressed, respectively. The following code illustrates this:

```csharp
if (args.VirtualKey == VirtualKey.B)
    CreateShape(ShapeModel.ShapeType.Ball);
```

Note that if you are interested in key combinations in which a “modifier key,” such as one or more of the Shift, Control, or Alt keys pressed in concert with another key, you have two options. First, you can track the individual key down and key up events to determine which keys are up or down at any given instant. Second, you can actively interrogate the state of a given key by using the `GetKeyState` method that the `CoreWindow` class provides. Because the result of `GetKeyState` returns a flag value, it is a best practice to mask the result value before comparing it with the desired value. Also note that the Alt key corresponds to the `Menu` member of the `VirtualKey` enumeration. Listing 13.7 shows this approach.

**LISTING 13.7  Checking for Modifier Keys**

```csharp
// Check for shift, control, alt (AKA VirtualKey.Menu)
var currentWindow = CoreWindow.GetForCurrentThread();
var ctrlState = currentWindow.GetKeyState(VirtualKey.Control);
var shftState = currentWindow.GetKeyState(VirtualKey.Shift);
var altState = currentWindow.GetKeyState(VirtualKey.Menu);
var isControlKeyPressed =
    (ctrlState & CoreVirtualKeyStates.Down) == CoreVirtualKeyStates.Down;
var isShiftKeyPressed =
    (shftState & CoreVirtualKeyStates.Down) == CoreVirtualKeyStates.Down;
var isAltKeyPressed =
    (altState & CoreVirtualKeyStates.Down) == CoreVirtualKeyStates.Down;
```
Sensor Input

Devices such as touchscreens, mouse devices, styluses, and keyboards provide interactivity by allowing an app to respond to their interactions with components shown on their device displays. Users tap elements drawn to the screen or type characters that will appear inside onscreen text regions. However, a class of input devices known as sensors can give a running app information about the device’s relationship to its physical environment. Examples of the information sensors gather include details about which way the device is facing, its velocity in any particular direction, its position on the globe, and how much light is shining on it at a given moment. Devices might or might not include one or more of these kinds of sensors.

The Windows Runtime API includes support for working with several different kinds of sensors and relaying the information they gather. These APIs not only enable an app to ask for sensor measurements, but they also provide events that can be subscribed to and, in most cases, the capability to throttle how often these events can be raised. Some of the environmental information that can be obtained through these APIs includes information about a device’s physical location, its movement and orientation, and how bright of an environment it is in.

The Example App

The SensorsExample project highlights a few different ways sensors can be used from within an application. The app features an instance of the interactive Bing Maps control surrounded by boxes that show information from and allow interaction with each of the various sensors. The boxes along the left side also allow the app to coordinate the information it receives from the sensors with the display of the Bing Maps control. The Location section allows the map to be centered at the current geolocation coordinates and also offers support for working with geofencing (the upcoming sections explain geofencing). The Compass section enables the app to set the map’s orientation to approximate the current compass heading (although the support offered for setting a specified heading in the Bing Maps control is currently somewhat limited). The Inclinometer section allows the map to be panned in concert with the direction in which the device itself is being tilted.
**Working with the Bing Maps Control**

The Bing Maps control in the example project is part of the Bing Maps platform, which includes the Windows control, controls for other platforms, and several related data services. You can access information about the Bing Maps control and the related services and the tools and resources you need to include in your project through the Bing Maps Platform Portal.² Although the control and the related services offer a tremendous amount of functionality, you need to be aware of some important license-related and technical considerations for this example application and in case you are considering their use in your own app.

From a licensing standpoint, some restrictions govern how this control can be used. The Bing Maps Platform Portal includes a Licensing Options page that explains how the restrictions apply to your app, under what circumstances the tools can be used for free, and when a fee needs to be paid to license the use of the control. As of this writing, you can access this page by clicking the Licensing link from the Bing Maps Platform Portal page. If you will use the Bing Maps control in your Windows Store App, be sure to look over the restrictions and conditions for use in the context of your application needs and ensure that you are abiding by the appropriate terms of use.

From a technical standpoint, before you can build the **SensorsExample** project, you need to download and install the Bing Maps SDK. You can get to the latest SDK installer by following links on the Bing Maps Platform Portal. Alternatively, you can use the Visual Studio Extension Manager to obtain the SDK.

To use the Extension Manager, launch Visual Studio and select **Extensions and Updates** from the **Tools** menu. In the **Extensions and Updates** dialog box, select the **Online** node and then select **Visual Studio Gallery**. Then type **Bing Maps SDK** into the search box in the upper-right corner (see Figure 13.1). In the search results, select the entry for **Bing Maps SDK for Windows 8.1 Store Apps** and click the **Download** button; then click the **Install** button in the ensuing dialog box after you have read and reviewed the included license agreement. After the installation has

---

completed, you will most likely be instructed to restart Visual Studio so that you can use the installed SDK components.

**Figure 13.1  Locating the Bing Maps SDK Visual Studio extension**

Because this additional download and installation is required to build the example project, the build configuration in the **WinRTByExample** solution has been configured to not include the **SensorsExample** project as part of the solution build by default. To build the project, you need to either select the project in the **Solution Explorer** and choose **Build** from the project file’s context menu, or choose **Build SensorsExample** from the **Build** menu. Another option is to open the **Configuration Manager** entry from the **Build** menu and check the **Build** entry next to the **SensorsExample** project in the **Configuration Manager** dialog box that appears; the **SensorsExample** project then is built along with the other projects in the solution.

Another consideration when building a project that includes the Bing Maps control is the selection of a target platform. Most Windows Store apps are built with the target set to Any CPU. However, the Bing Maps control
relies on the Visual C++ Runtime, which requires selecting a specific processor architecture to build a project that references it. You can set this value in the Configuration Manager dialog box. Select the appropriate Platform for your build either for the entire solution or for the SensorsExample project. For example, you need to select a value of ARM to create a version of the resulting app that will run on Windows RT devices. Note that in order to work with the XAML designer in Visual Studio, you need to select the value x86. If you prefer to have the interactive designer available, you can always set the value temporarily to x86 and then set it to your desired target platform when you have finished working in the XAML designer.

To deploy an app that includes the Bing Maps control, you need to specify a value for the map’s Credentials property. If you do not specify a valid map key for the map Credentials property, the map control displays with a banner indicating that invalid credentials are being used, as you can see in the example app screen in Figure 13.2.

![Sensors Example](image)

**FIGURE 13.2** The Bing Maps control displayed without valid credentials

The following markup shows the credentials being set in the example project:

```xml
<maps:Map Credentials="{StaticResource MapKey}"/>
```
In this example, the credentials are located in the resource defined by the value `MapKey`, which is defined in the project’s `App.xaml` file. The map key is a value you obtain from the Bing Maps Account Center.³ Sign into the account center with your Microsoft Account credentials and select **Create or View Keys**. At this point, you can define a new key by specifying information about your application or retrieve a previously defined key. Place this key value into the `MapKey` resource in your project, and build and run your project to make sure that the warning message from Figure 13.2 no longer displays.

³ Bing Maps Account Center, https://www.bingmapsportal.com/

**TIP**

After you deal with the logistics related to licensing for the Bing Maps control and the mechanics related to installing the SDK, configuring the project build, and obtaining and configuring the map key, you will likely find that the Bing Maps control offers a tremendous amount of functionality. The Bing Maps Platform Portal includes both development guides and MSDN API documentation that covers the available functionality. Another helpful resource in the interactive SDK is provided for the Bing Maps AJAX control at www.bingmapsportal.com/isdk/ajaxv7: It provides an interactive map and the JavaScript and related HTML. Many of the concepts and much of the code illustrated in this tool translate readily to the corresponding .NET API.

**Geolocation**

Geolocation refers to information about an item’s geographic location. In the Windows Runtime, one of two data sources provides this location information. The first data source for location information is the Windows Location Provider. The Windows Location Provider obtains its information from a couple different data sources. The first source it attempts to use is Wi-Fi triangulation, in which the proximity to different known Wi-Fi hotspots is used to determine a position. If Wi-Fi data is not available, IP
address resolution is then used. The second data source that the Windows Runtime can use to obtain location information is available if the device optionally includes one or more Global Positioning System (GPS) sensors.

The network-based information that the Windows Location Provider gathers is limited in both accuracy and amount of available detail because only latitude, longitude, and accuracy information are made available. An installed GPS sensor most likely provides more accurate information (different sensors have different resolution capabilities) and also gives more location information than the Windows Location Service, potentially including details about direction, speed, and altitude. Note, however, that the additional detail afforded by GPS sensors tends to come with additional power use and, therefore, reduced battery life.

**Getting Started**

To start working with location information in a Windows 8.1 application, you first need to declare that the app will be accessing this information. Location information is considered to be personally identifiable information (PII), so any app that will access this information needs to explicitly declare its intent to do so. The App Store’s certification process will refuse an app that includes use of the geolocation APIs if it does not provide such a declaration; if the app does provide the declaration, the app’s entry in the store will indicate its intent to access this information. As an additional measure meant to protect users, Windows notifies users the first time an app accesses location information and prompts them to either allow or block access. Windows also provides several places where the user can choose to toggle this same permission on or off, as will be discussed shortly. To declare that an app will attempt to access location information, open the app manifest file, select the **Capabilities** panel, and check the **Location** entry under the **Capabilities** list (see Figure 13.3).
FIGURE 13.3 Setting the location capability in the app manifest

Using the Geolocator

The Geolocator class provides location information in the Windows Runtime. You can obtain the current position value from this class in two ways. The first option is to directly request the current position with the GetGeopositionAsync method. The second option is to provide a handler for the PositionChanged event that is called when a position change is detected, depending on the configuration of the Geolocator instance.

The first time an app calls the GetGeopositionAsync method or registers an event handler for the PositionChanged event, the user is prompted to grant permission for the app to access location information, as Figure 13.4 illustrates. Because this step might display a user interface element, it is important to make sure that this first call takes place on the UI thread; otherwise, an unexpected cross-thread exception might occur whose cause can be difficult to diagnose.
The value selected in the prompt is reflected in the Permissions panel that you can access through the app’s Settings Charm, as well as within the Location panel in the Privacy section located in PC Settings. This system-wide location privacy screen lists all the applications that are registered to access position information and states whether access is currently blocked or enabled. It also includes a system-wide switch to disable access to location information for all apps that request it. However they access it, when users choose to block the app’s access to location information, the LocationStatus property on the Geolocator instance returns a value of PositionStatus.Disabled. The potential consequences of this LocationStatus value and other values that can appear in this property are discussed shortly.

In the example app, interactions with the Geolocator are handled in the GeolocationHelper class. The code in Listing 13.8 shows the Geolocator initialization and subscription to the available events.
LISTING 13.8 Geolocator Initialization and Event Subscription

```csharp
_geolocator = new Geolocator();

// Listen for status change events, but also immediately get the status.
// This is in case it is already at its end-state and therefore
// won't generate a change event.
_geolocator.StatusChanged +=
    (o, e) => SetGeoLocatorReady(e.Status == PositionStatus.Ready);
SetGeoLocatorReady(_geolocator.LocationStatus == PositionStatus.Ready);

// Set the desired accuracy. Alternatively, can use
// DesiredAccuracyInMeters, where < 100 meters ==> high accuracy
_geolocator.DesiredAccuracy = GetDesiredPositionAccuracy();

// Listen for position changed events.
// Set to not report more often than once every 10 seconds
// and only when movement exceeds 50 meters
_geolocator.ReportInterval = 10000; // Value in ms
_geolocator.MovementThreshold = 50; // Value in meters
_geolocator.PositionChanged += GeolocatorOnPositionChanged;
```

The first task in the code in Listing 13.8 is to work with the `LocationStatus` value. The `Disabled` status was previously mentioned, but it is important to note that an attempt to request the current position from a `Disabled` instance results in an `UnauthorizedAccessException`. If location access has not been blocked, the `LocationStatus` property has a value of `NoData` either before the first call to `GetGeopositionAsync` or before the first time an event handler is provided for `PositionChanged`. When either of these happens, the Windows Runtime might trigger a startup sequence that takes a little time to complete. During that time, the `LocationStatus` returns a value of `NotInitialized`. Additionally, if location data is coming from a GPS sensor, the sensor tries to retrieve information from some required minimum number of satellites. Until the device reaches this number, the `LocationStatus` has a value of `Initializing`. When the `Geolocator` instance is ready, the `LocationStatus` returns a value of `Ready`. With all that in mind, when including the `Geolocator` in your project, be sure to account for the fact that, even under ideal circumstances, a lag might occur before it is ready to be used; you need to check to ensure that it has reached the `Ready` status.

In the example code, the `SetGeoLocatorReady` function is called with a value of `true` only when the sensor is in a `Ready` state. It is used to set the
Sensor Settings IsLocationAvailable property, which the application user interface uses to disable access to location retrieval functions. It also sets a local flag that prevents direct calls to get the current position through the GetCoordinate function from actually making the request through the Geolocator until it is in the Ready state.

The next step after working with initialization and status information involves establishing the desired accuracy for the Geolocator instance. The DesiredAccuracy property can be set to either PositionAccuracy.High or PositionAccuracy.Default. A value of High instructs WinRT to always try to use a GPS for its data if one is available, and to otherwise use the Windows Location Provider. A value of Default instructs WinRT to make use of only GPS sensors if it cannot obtain a value from the Windows Location Provider, such as when no Wi-Fi signals exist for triangulation (or the device is either not equipped or not configured to work with Wi-Fi) and when the device does not have an IP address that can be looked up for location information. Ultimately, setting either of these values does not guarantee how the WinRT will make use of GPS devices; it just indicates a preference for how it should behave.

**NOTE**

The Windows Runtime also includes a DesiredAccuracyInMeters property. When this property is set to a non-null value, it resets the DesiredAccuracy property value. A DesiredAccuracyInMeters value of less than 100 meters results in a DesiredAccuracy value of High; a value of 100 meters or higher sets DesiredAccuracy to Default.

The final task in Listing 13.8 is to configure how the Geolocator will go about raising PositionChanged events, which is controlled with the ReportInterval and MovementThreshold properties. Each of these properties limits how often the Geolocator instance can raise the PositionChanged events. Whereas the ReportInterval property specifies the minimum amount of time that must elapse between instances of the Windows Runtime attempting to obtain location information values, the MovementThreshold property indicates how much distance must pass before a subsequent event is
raised. In the example code, the `ReportInterval` property is set to ensure that at least 10 seconds (10,000 milliseconds) pass between event updates. The `MovementThreshold` value is set to ensure that the position has changed by at least 50 meters. (The sensor is checked every 10 seconds, and the class instance raises an event only if the distance between checks exceeds 50 meters.) A value of 0 for `ReportInterval` generates events at whatever the maximum frequency is for the most accurate location source, and it should be used only for apps that require near-real-time position updates. Because it affects how often the location hardware is queried and, therefore, can impact battery life, it is important to set the `ReportInterval` to the maximum value possible for the needs of your app. Also note that not every scenario involving the `Geolocator` needs to subscribe to the `PositionChanged` event; some cases are served just fine by requesting the position directly only when it is needed. Each application has different needs in terms of how frequently to update position information and whether to individually request it with the `GetGeopositionAsync` method or use change events.

**Working with Geocoordinate Values**

An instance of the `Geoposition` class is returned both from a call to `GetGeopositionAsync` and within the `Position` member of the `PositionChangedEventArgs` event arguments that are provided to `PositionChanged` event handlers. Although the `Geoposition` class contains both `Coordinate` and `CivicAddress` properties, the `CivicAddress` values are not populated in Windows 8.1 (the only member that is set is the `Country` property, which is obtained from the country value set in the Windows region settings instead of the location information data sources that were previously mentioned). The `Coordinate` property is an instance of the `Geocoordinate` class and contains several different kinds of position information that are returned either from the Windows Location Provider or from GPS sensors (as you have seen, this depends on how the `Geolocator` is configured).

At its root, the `Geocoordinate` object provides a `PositionSource` property that either indicates how the location information was obtained or includes a value of `Unknown` if information about the source is not available. It also include an `Accuracy` property that indicates how accurate (in meters)
the latitude and longitude position information are believed to be. If the location information is being obtained from a GPS sensor, values for the Speed, Heading, AltitudeAccuracy, and SatelliteData properties might also be included, depending on the sensor’s capabilities.

The actual position information provided is a little buried in the object hierarchy. It is actually returned in the Position property within the Point property of the Geocoordinate instance. Regardless of whether the location information is obtained from a GPS sensor or the Windows Location Provider, values for Latitude and Longitude (measured in degrees) are provided in this Point property. If the information is obtained from a GPS sensor, the Altitude value might be provided as well.

To show how this information looks in practice, the example application includes the capability to display all the fields of the Geocoordinate object for the current location. Clicking the ShowCurrent button in the app’s Location box displays a pop-up that contains these values, as provided by a call to the GetGeoPositionAsync method. The Center Map on Current button also makes a call to the GetGeopositionAsync method and sets a viewmodel property from the previously discussed Point property. The property in the ViewModel is data bound to the Bing Maps control so that when the value changes, the map centers itself at the Latitude and Longitude coordinates specified in the position value.

**Using the Simulator Location Tools**

Chapter 2 introduced you to the Visual Studio simulator for Windows Store Apps (the Simulator), which enables you to run and test your Windows 8.1 app within a simulated environment on your development system. In addition to being able to emulate various screen sizes and resolutions (along with the other functionality it provides), the simulator can be used to provide simulated geolocation values to a running app, which can help you test your location-aware app. To use the simulator’s location functions, several requirements must be met, primarily related to Location Settings enabled on the local system. When you first try to use the location functions, you are prompted and instructed to take corrective action if your system does not meet the necessary requirements for the location simulator to run.
To use the location functions of the simulator with your location-aware app, start debugging in the simulator following the instructions in Chapter 2. When the app is running in the simulator, clicking the icon in the simulator toolbar that resembles a globe displays the location simulation dialog box (see Figure 13.5). When the Use Simulated Location check box is checked, it provides access to text boxes for setting location values such as Latitude, Longitude, and Altitude.

![Sensors Example](image)

**Figure 13.5 Using location tools in the Windows simulator**

When the Set Location button is clicked, the Geolocator API methods and events that are used by the app running in the simulator use those values for their position information. Removing the check from the Use Simulated Location check box returns the simulator to using the host system values for its current position values.

**Geofencing**

Geofencing enables your Windows 8.1 app to define geographic boundaries (known as geofences) and monitor a device’s position relative to those boundaries. Your app then produces notification events when the device enters or exits those boundaries. For applications that need to be alerted
when a device has moved into or beyond one of these boundaries, this provides a much more efficient solution than polling the geolocation APIs and making the determinations programmatically. To support geofencing, the Windows Runtime includes APIs that allow registering and managing geofences, as well subscribing to and processing the related notifications.

**Getting Started**

The geofencing support the Windows Runtime provides is closely related to the geolocation support and includes several of the same restrictions and conditions related to working with personally identifiable information. To use geofencing, you must set the Location capability in the app manifest and set both the app-specific and system-wide permission settings to allow the app to access location information.

The GeofenceMonitor class provides geofencing support in the Windows Runtime. Unlike using the Geolocator, where you create a new instance of the class to access the functionality, a reference to the GeofenceMonitor is accessed through its static Current property:

```javascript
var geofenceMonitor = GeofenceMonitor.Current;
```

The following sections discuss the functionality that the GeofenceMonitor exposes.

NOTE

Unlike with geolocation, accessing the geofencing properties and events does not automatically prompt the user to grant permission to location information. You might have to check to see if location functionality is currently disabled by checking the GeofenceMonitor Status property and instructing the user to access the permissions property in the Settings Charm. If your app also uses the Geolocator to potentially access the user’s current location (perhaps to obtain the center point for a fence), that class access provides the necessary request for permissions.
Defining a Fence

The GeofenceMonitor works with a collection of Geofence instances. Each Geofence object describes the region the fence covers, the types of events to provide, and the conditions under which it indicates that an event has occurred. Table 13.4 describes the settings provided by the Geofence class. Be aware that these values must be set through one of the Geofence constructors and cannot be changed after the geofence has been defined.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id</td>
<td>Specifies the ID for the fence. The ID is a String that must be unique within the scope of the current app, and it must be a maximum of 64 characters long. This value is required.</td>
</tr>
<tr>
<td>Geoshape</td>
<td>Specifies the geofence boundary. Currently supports being set to only a Geocircle instance, which defines the boundary via a center point and a radius. This value is required.</td>
</tr>
<tr>
<td>MonitoredStates</td>
<td>Specifies which events the GeofenceMonitor raises for this fence. Can be set to a combination of the MonitoredGeofenceStates enumeration values, which includes Entered, Exited, and Removed, but must minimally include either Entered or Exited. This value is optional and, by default, is set to the combination of Entered and Exited.</td>
</tr>
<tr>
<td>SingleUse</td>
<td>Specifies whether the fence is automatically removed. If set, when each of the MonitoredStates (with the exception of Removed) is reached at least once, the fence is automatically removed from the GeofenceMonitor collection. This value is optional and, by default, is set to false.</td>
</tr>
<tr>
<td>DwellTime</td>
<td>Specifies the time that must elapse when a geofence condition is met before an event is raised. This value is optional and, by default, is set to 10 seconds.</td>
</tr>
<tr>
<td>StartTime</td>
<td>Specifies the time at which the geofence monitoring begins. This value is optional and, by default, is set to a minimal value of January 1, 1601 (which is the base value for the Windows FILETIME structure).</td>
</tr>
<tr>
<td>Setting</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>Duration</td>
<td>Specifies the amount of time following StartTime during which the fence should be monitored. This value is optional and, by default, is set to TimeSpan.Zero, which indicates an indefinite duration.</td>
</tr>
</tbody>
</table>

After a Geofence instance has been defined, the GeofenceMonitor begins tracking it when it is added to its Geofences collection.

In the example app, the code for working with the GeofenceMonitor has been consolidated into the GeofenceHelper class. This class provides an AddGeofence method used to add new fences (see Listing 13.9). Clicking the Add Fence at Map Center button in the example application produces a flyout that enables the user to define a name for the geofence. The center point is retrieved from the map’s current center point, and the radius is hardcoded to 20KM. Pressing the flyout’s Add Fence button calls this method with the values in the flyout. This method then creates a Geocircle with the specified center and radius, which is provided to the Geofence constructor along with indications that the GeofenceMonitor should listen to Entering, Exited, and Removed events (the next section covers the events) and that the geofence is not configured to be single use. Default values are accepted for the remaining parameters. The resulting Geofence instance is then added to the GeofenceMonitor collection and returned so that it can be used to include a UI entry on the Bing Maps control.

**Listing 13.9  Adding a Geofence**

```csharp
public Geofence AddGeofence(
    String fenceId,
    BasicGeoposition fenceCenter,
    Double radiusInMeters)
{
    var fenceCircle = new Geocircle(fenceCenter, radiusInMeters);

    const MonitoredGeofenceStates states =
        MonitoredGeofenceStates.Entered |
        MonitoredGeofenceStates.Exited |
        MonitoredGeofenceStates.Removed;

    // Create the fence with the desired states and not single-use
```
When defining a geofence boundary, keep in mind the limitations of the accuracy of the various location providers available to the Windows Runtime. Depending on sensor capabilities and network connectivity, extremely small fences might not be all that useful.

**Geofence Events**

You can receive notifications that geofencing events have occurred in two ways. Foreground notifications are configured when an app registers an event handler for the `GeofenceStateChanged` event provided by the `GeofenceMonitor` class. Alternatively, you can set up a background task to process geofence notifications even when the app is not running in the foreground. To configure geofencing background task notifications, the `LocationTrigger` class needs to be provided to a `BackgroundTaskBuilder`, and that builder instance needs to be configured and then registered. Chapter 15, “Background Tasks,” covers this in more detail.

As previously discussed, the `GeofenceMonitor` events can be triggered in response to the device entering or exiting a geofence, depending on the combination of the `Entered` or `Exited` `GeofenceState` enumeration values provided in the `MonitoredStates` value when the `Geofence` instance was defined. Additionally, the event can occur in response to the geofence being automatically removed from the list of monitored fences and depending on whether the `Removed` enumeration value was specified.

Automatic removal of a `Geofence` occurs in response to the values set in its `Duration` and `SingleUse` properties. `Duration` is the easiest to understand. When the time window indicated by the combination of the `StartTime` and `Duration` properties has expired, a geofence event is recorded indicating that this fence is no longer being monitored. In this case, the event includes a `RemovalReason` value that is set to `Expired`.

The other option for automatic removal relates to the `SingleUse` property. When this value is set to `true`, a geofence is removed after all its `MonitoredStates` have occurred. If a `Geofence` instance is defined with only
Entered or Exited specified, then as soon as the corresponding event takes place, the geofence is removed. If both Entered and Exited are specified, the geofence is removed only after both have occurred. In this case, the Removed state is accompanied by a RemovalReason value of Used.

When a geofence notification event is received, the app should call the ReadReports method of the GeofenceMonitor instance, which returns the collection of all notification reports that have accumulated since the last call to ReadReports was made. Each report is actually indicated in an individual GeoStateChangedEventReport, and a single GeofenceMonitor event can encompass multiple reports, especially in the case of background tasks, which run only periodically.

The example app subscribes to geofence notifications only in the foreground. To do so, the GeofenceHelper class registers its HandleGeofenceStateChanged method as a handler for the GeoStateChanged event (see Listing 13.10).

**Listing 13.10 Processing Geofence Events**

```csharp
private void HandleGeofenceStateChanged(GeofenceMonitor monitor, Object o)
{
    // Iterate over and process the accumulated reports
    var reports = monitor.ReadReports();
    foreach (var report in reports)
    {
        switch (report.NewState)
        {
            case GeofenceState.Entered:
            case GeofenceState.Exited:
                var updateArgs = new FenceUpdateEventArgs
                {
                    FenceId = report.Geofence.Id,
                    Reason = report.NewState.ToString(),
                    Timestamp = report.Geoposition.Coordinate.Timestamp,
                };
                OnFenceUpdated(updateArgs);
                break;
            case GeofenceState.Removed:
                var removedArgs = new FenceRemovedEventArgs
                {
                    FenceId = report.Geofence.Id,
                };
                break;
        }
    }
}
```
The event handler retrieves the reports from the provided `GeofenceMonitor` instance and then iterates over the individual report instances. In the case of `Entered` and `Exited` events, information is gathered about which fence caused the event, whether it was triggered on enter or exit, what position caused the event to be triggered, and when exactly the reported event occurred. This information is then used to relay an event out of the `GeofenceHelper` that displays the event’s occurrence in the app UI. In the case of a `Removed` event, the event ID and `Removal` reason are obtained, and a similar event is raised to provide notification as well as remove the geofence entry from the Bing Maps control.

Be aware that because the `GeofenceStateChanged` events are raised from an external entity, the handler will not run on the UI thread. Any reaction to these events that affects the application UI needs to be marshalled to the proper thread using either the `Dispatcher` or a valid `SynchronizationContext`, as discussed in the section “Accessing the UI Thread” in Chapter 9, “Model-View-ViewModel.”

**Managing Geofences**
You can manage geofences by working directly with the `Geofences` collection that the `GeofenceMonitor` instance provides. The example app enumerates these instances in two places. First, on app startup, the existing collection is obtained to put markers on the Bing Maps control for each geofence. Second, clicking the List Fences button shows a flyout that lists all the currently defined geofences. This flyout includes the option to remove the selected `Geofence` instance from the `Geofences` collection. Note that programmatically removing a fence from the collection in this way does not generate the previously discussed `Removed` events. Those occur only when the removal happens automatically in response to the conditions that the `Geofence` instance’s settings identify.
Motion and Orientation Sensors

In addition to using the Geolocator and related APIs to obtain information about a device’s physical location, the Windows Runtime provides APIs for interacting with a class of sensors related to the movement and positioning of the device itself. Table 13.5 lists the kinds of sensors these APIs can interact with and the kind of data they gather.

Table 13.5 Motion and Orientation Sensor Types

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Orientation</td>
<td>Reports the current orientation of the device based on values from the SimpleOrientation enumeration.</td>
</tr>
<tr>
<td>Compass</td>
<td>Provides information about the position of the device in relation to magnetic north. This is actually a composite sensor whose output is based on combined input from magnetometer and gyrometer sensors.</td>
</tr>
<tr>
<td>Inclinometer</td>
<td>Provides information about the pitch, yaw, and roll state of a device. This is a composite sensor whose output is based on combined input from accelerometer, gyrometer, and magnetometer sensors.</td>
</tr>
<tr>
<td>Accelerometer</td>
<td>Provides information about the G-forces affecting the device’s x-, y-, and z-axes.</td>
</tr>
<tr>
<td>Gyrometer</td>
<td>Provides information about the angular velocity along the device’s x-, y-, and z-axes.</td>
</tr>
<tr>
<td>Sensor</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Orientation Sensor</td>
<td>Provides detailed information about how a device is situated in space. This is a composite sensor whose output is based on combined input from accelerometer, gyrometer, and magnetometer sensors.</td>
</tr>
<tr>
<td>Light Sensor</td>
<td>Provides information about the amount of light currently striking the device display.</td>
</tr>
</tbody>
</table>

You might have noticed that several of these sensors’ values are determined in part from a magnetometer, which is itself a sensor whose purpose is to measure the strength of magnetic fields. However, the Windows Runtime does not provide any APIs that allow direct access to output from magnetometers.

For the most part, the API for interacting with sensors is similar across all the different kinds. They all basically offer the capability to obtain a reference to a class instance that provides access to the sensor, as well as methods for obtaining the current sensor value. In addition, they provide events that you can subscribe to for notifications when the value changes. The majority of the sensor APIs define properties that specify the minimum interval with which the sensor can raise these change events, as well as properties that specify the requested interval for reporting value changes.

Most of the code for working with sensors in the example project resides in the SensorHelper class. You might be relieved to know that, unlike the location information, the information these sensors return is not considered to be personally identifiable information. As a result, you do not have to indicate entries in the application manifest, prompt the user for permission, or deal with users blocking access to the sensors if you include code to make use of them in your application.

**Simple Orientation Sensor**

The simple orientation sensor is the simplest of the available sensors. It does not work with the concept of a reporting interval for its change events, and the data values that it reports are simply members of the SimpleOrientation enumeration. When available, the purpose of this sensor is to describe which way the device is facing. The values it can return are NotRotated, for
when the device is sitting in a “natural” landscape orientation; Rotated90, Rotated180, and Rotated270, to indicate that the device has been rotated to stand on one of its other edges; and FaceUp and FaceDown, to indicate that the device is lying flat.

The code in Listing 13.11 shows how the example project is configured to work with the simple orientation sensor, which is exposed via the SimpleOrientationSensor class. The GetDefault static method obtains a reference to the sensor, the value of which is null if the sensor is not available. After that, it simply provides a handler for the OrientationChanged event and then uses the GetCurrentOrientation method to obtain the current sensor value.

LISTING 13.11 Configuring the Simple Orientation Sensor

```csharp
// Get the reference to the sensor and see if it is available
_simpleOrientation = SimpleOrientationSensor.GetDefault();
if (_simpleOrientation == null) return;

_sensorSettings.IsSimpleOrientationAvailable = true;

// NOTE - Simple Orientation does not offer a minimum interval setting
_simpleOrientation.OrientationChanged += SimpleOrientationOnOrientationChanged;

// Read the initial sensor value
_sensorSettings.LatestSimpleOrientationReading = _simpleOrientation.GetCurrentOrientation();
```

The Visual Studio simulator for Windows Store Apps, which the preceding section “Using the Simulator Location Tools” discussed, also includes support for simulating device rotation by providing buttons that rotate the simulator in 90-degree increments clockwise or counterclockwise.

**Compass**

The compass provides information about the current heading of the device relative to magnetic north. When available, this sensor returns readings as instances of the CompassReading type, which includes both HeadingMagneticNorth and HeadingTrueNorth properties, indicating degrees to magnetic north and degrees to true north, respectively. HeadingMagneticNorth always is provided; the availability of HeadingTrueNorth values depends on
the individual capabilities of the actual sensor hardware. HeadingTrueNorth returns a value of null if it is not available.

Listing 13.12 shows how the example project is configured to work with the compass, which is exposed via the Compass class. The GetDefault static method obtains a reference to the sensor, the value of which is null if the sensor is not available. It next proceeds to set the sensor’s reporting interval.

**Listing 13.12 Configuring the Compass**

```csharp
// Get the reference to the sensor and see if it is available
_compass = Compass.GetDefault();
if (_compass == null) return;

_sensorSettings.IsCompassAvailable = true;

// Set the minimum report interval. Care must be taken to ensure
// it is not set to a value smaller than the device minimum
var minInterval = _compass.MinimumReportInterval;
_compass.ReportInterval = Math.Max(_sensorSettings.SensorReportInterval, minInterval);
_compass.ReadingChanged += CompassOnReadingChanged;

// Read the initial sensor value
_sensorSettings.LatestCompassReading = _compass.GetCurrentReading();
```

The **ReportInterval** property is common to most of the available sensors. The purpose of the property is to provide access to the minimum time (in milliseconds) that must elapse between **ReadingChanged** events. Take care when setting this value; setting it to a value below the minimum value that the sensor can support can result in either an exception or unpredictable behavior, depending on the sensor. You can obtain the minimum allowable report interval value through the **MinimumReportInterval** property. Note that the **ReportInterval** setting has some of the characteristics of a request rather than a certain value. Several factors can influence how the actual sensor handles the **ReportInterval** setting. For example, when other apps on the system that make use of the same sensor set their own values for this property, the sensor might simply elect to use whichever is the smallest defined value. Also be aware that the **ReadingChanged** event is raised only when the reading actually changes, regardless of the **ReportInterval** setting. It is important to not confuse the **ReportInterval** value with a frequency
value that somehow guarantees that the ReadingChanged event will be raised repeatedly in a steady cadence. After the ReportInterval is set, the code simply provides a handler for the ReadingChanged event and then uses the GetCurrentReading method to obtain the current sensor value.

Another important note is that the value the compass returns is relative to the device being in a regular landscape orientation, with the device base sitting at the bottom. (If the device is a tablet device built with Portrait as its primary orientation, this sensor landscape condition still applies; the “natural” landscape mode is the one where the hardware Windows button ends up on the right side of the display.) Figure 13.6 shows devices in natural landscape orientation.

![Figure 13.6 Devices in natural landscape orientation](image)

If the device is in a different orientation, the value the sensor returns needs to be adjusted to account for this. The example project includes a CompassOffset extension method for the DisplayOrientations class that you can use to obtain the offset to apply to a compass direction based on a provided orientation value. This method simply returns a value of 0, 90, 180, or 270, depending on what is needed to correct the compass reading for the given orientation. You obtain the DisplayOrientations value to use from the CurrentOrientation property of the DisplayInformation class. After you determine the offset, you can add it to the HeadingMagneticNorth or HeadingTrueNorth values, using modular arithmetic to constrain the resulting value between 0 and 360 degrees, as follows:

\[
\text{(LatestCompassReading.HeadingMagneticNorth + offset)} \mod 360
\]
The example project includes a Sensor Settings flyout that you can bring up using the Settings Charm. The panel includes a slider for updating the minimum reporting interval for the sensors. It also includes a check box that corresponds to a flag that the app uses to decide whether to compensate for orientation changes when using and displaying sensor values. By toggling these values and switching the orientation of the device on which the app is running from a landscape to an inverted landscape orientation, you can see the effect that changing an orientation has on sensor values, as well as how the compensation code will correct them to their expected state.

Another feature present in the example app is the capability for the Bing Maps control to “follow” the compass sensor value. Note that the current version of the Bing Maps control supports rotating its display contents only when viewed at high zoom levels (and to only one of four discrete views), so this behavior is best viewed when the map is set to display and is zoomed in enough to show bird’s-eye imagery. To enable this feature, check the Follow box in the Compass panel in the app, and then point the device in different directions. When the Follow box is checked, the Tick event handler for a timer on the display page periodically polls the SensorSettings class for the LatestCompassReading value, which is set by the ReadingChanged handler for the compass. This value then is set to a view-model property that is data bound to the Bing Maps control. This approach of using a timer to check for the most recent value is used because the ReadingChanged event is fired only when a compass value actually changes, as previously discussed. Listing 13.13 shows the code in the timer event handler that obtains and applies the compass value.

Listing 13.13 Applying the Compass Orientation to the Map Display

```csharp
if (_sensorSettings.IsFollowingCompass)
{
    // Get the latest compass reading
    var compassReading = _sensorSettings.LatestCompassReading;

    // Adjust the reading based on the display orientation, if necessary
    var displayOffset = _sensorSettings.CompensateForDisplayOrientation
                         ? _sensorSettings.DisplayOrientation.CompassOffset() : 0;
    var heading = (compassReading.HeadingMagneticNorth + displayOffset) % 360;
```
// Set the value used by data binding to update the map's heading
DefaultViewModel["Heading"] = heading;

**Inclinometer**
The inclinometer provides information about the current pitch, yaw, and roll of the device. Pitch represents the degrees of rotation around the x-axis, yaw represents degrees of rotation around the z-axis, and roll represents degrees of rotation around the y-axis. Figure 13.7 illustrates how these values map to the physical position of a tablet device. When available, this sensor returns readings as instances of the `InclinometerReading` type, which provides its results in `PitchDegrees`, `RollDegrees`, and `YawDegrees` properties.

**FIGURE 13.7 Pitch, roll, and yaw relative to a tablet device**

Listing 13.14 shows how the example project is configured to work with the inclinometer, which is exposed via the `Inclinometer` class. The steps involved in configuring the inclinometer are basically identical to those shown in Listing 13.12 for configuring the compass.
Listing 13.14 Configuring the Inclinometer

```csharp
// Get the reference to the sensor and see if it is available
_inclinometer = Inclinometer.GetDefault();
if (_inclinometer == null) return;

_sensorSettings.IsInclinometerAvailable = true;

// Set the minimum report interval. Care must be taken to ensure
// it is not set to a value smaller than the device minimum
var minInterval = _inclinometer.MinimumReportInterval;
_inclinometer.ReportInterval
    = Math.Max(_sensorSettings.SensorReportInterval, minInterval);
_inclinometer.ReadingChanged += InclinometerOnReadingChanged;

// Read the initial sensor value
_sensorSettings.LatestInclinometerReading = GetInclinometerReading();
```

Much like the compass, the values the inclinometer returns are relative to the device being in a regular landscape orientation, and the resulting values also need to be normalized if the device is being used from any other orientation. The example project includes an `AxisAdjustmentFactor` extension method for the `DisplayOrientations` class that you can use to obtain the factors to apply to the x-, y-, and z-axis results, based on the current device orientation.

The example app includes a fun feature that you can enable by checking the Follow box in the Inclinometer panel in the app. When this box is checked, the content of Bing Maps control slides based on the Inclinometer readings, allowing you to navigate the map simply by tilting your device back and forth or left and right.

---

**NOTE**

If you find that tilting your device is causing your screen orientation to be toggled, you can disable the automatic screen rotation feature that Windows provides by bringing up the Settings Charm, selecting Screen, and tapping the rectangular icon above the brightness adjustment slider. If that icon has a pair of arrows next to it, automatic rotation is enabled. If it has a small padlock next to it, the current screen orientation is locked and will not automatically adjust as you tilt your device.
As with the Follow feature discussed previously for the compass sensor, implementation for this feature simply polls the SensorSettings class in response to the same timer Tick event. In this case, the value used to obtain the current device orientation is the LatestInclinometerReading value, which the inclinometer’s ReadingChanged handler sets. The displayAdjustment value used to compensate for device orientation changes returns per-axis values of +1 or –1 that are multiplied to the sensor result to normalize the value.

Listing 13.15 shows the calculations that move the map. First, the inclinometer reading is obtained and normalized, depending on the value of the compensation setting and the device orientation. Next, a rate of one full screen per timer tick was found to be a good maximum rate of traversal, so the number of x- and y-axis pixels to move are obtained from the map control. Then trigonometric functions convert the adjusted pitch and roll values to percentage values so that the traversal is nearly nothing when the device is lying flat and is full-value when it is held vertically. This percentage determines the actual number of x and y pixels to move in the current tick, which is applied to the center point to determine the equivalent destination point. From here, the Bing Maps TryPixelToLocation utility function converts a pixel onscreen to equivalent latitude and longitude values, which then set the new map position.

**Listing 13.15 Applying the Inclinometer Reading to the Map Display**

```csharp
if (_sensorSettings.FollowInclinometer)
{
    var inclinometerReading = _sensorSettings.LatestInclinometerReading;

    // Optionally normalize the sensor reading values
    var displayAdjustment
        = _sensorSettings.CompensateForDisplayOrientation
          ? _sensorSettings.DisplayOrientation.AxisAdjustmentFactor()
            : SensorExtensions.AxisOffset.Default;
    var adjustedPitchDegrees
        = inclinometerReading.PitchDegrees * displayAdjustment.X;
    var adjustedRollDegrees
        = inclinometerReading.RollDegrees * displayAdjustment.Y;

    // At full speed/inclination, move 100% map size per tick
    const Double maxScreensPerTick = 1.00;
    var mapWidth = ExampleMap.ActualWidth;
    var xFullRateTraversalPerTick = mapWidth * maxScreensPerTick;
    var mapHeight = ExampleMap.ActualHeight;
    var yFullRateTraversalPerTick = mapHeight * maxScreensPerTick;
```
// Turn rotation angles into percentages
var xTraversalPercentage
    = Math.Sin(adjustedRollDegrees*Math.PI/180);
var yTraversalPercentage
    = Math.Sin(adjustedPitchDegrees*Math.PI/180);

// Compute the final traversal amounts based on the percentages
// and compute the new destination center point
var xTraversalAmount
    = xTraversalPercentage*xFullRateTraversalPerTick;
var yTraversalAmount
    = yTraversalPercentage*yFullRateTraversalPerTick;
var destinationPoint = new Point(
    mapWidth/2 + xTraversalAmount,
    mapHeight/2 + yTraversalAmount);

// Use the Bing Maps methods to convert pixel pos to Lat/Lon
// rather than trying to figure out Mercator map math
Location location;
if (ExampleMap.TryPixelToLocation(destinationPoint, out location))
{
    // Obtain the current map position (for altitude)
    var position = (BasicGeoposition)DefaultViewModel["Position"];

    var newPosition = new BasicGeoposition
    {
        Altitude = position.Altitude,
        Latitude = location.Latitude,
        Longitude = location.Longitude
    };

    DefaultViewModel["Position"] = newPosition;
}

**Accelerometer**
The accelerometer provides information about the current G-forces acting on the device in the x, y, and z directions. At rest, the most significant G-force affecting a device is the force of gravity, which pulls down along whichever axis corresponds to the bottom edge of the device with a value of –1.0. For example, if a device is standing up on its bottom edge in a landscape profile, the y value has a value of approximately –1.0. When available, this sensor returns readings as instances of the `AccelerometerReading` type,
which provides its results in AccelerationX, AccelerationY, and AccelerationZ properties.

Listing 13.16 shows how the example project is configured to work with the accelerometer, which is exposed via the Accelerometer class. The steps involved in configuring the accelerometer are otherwise identical to those shown previously for configuring the other sensors, with one notable exception. The accelerometer sensor includes an additional Shaken event that is raised when the sensor detects that the device is being subjected to several quick back-and-forth motions.

**Listing 13.16 Configuring the Accelerometer**

```csharp
// Get the reference to the sensor and see if it is available
_accelerometer = Accelerometer.GetDefault();
if (_accelerometer == null) return;

_sensorSettings.IsAccelerometerAvailable = true;

// Set the minimum report interval. Care must be taken to ensure
// it is not set to a value smaller than the device minimum
var minInterval = _accelerometer.MinimumReportInterval;
_accelerometer.ReportInterval
    = Math.Max(_sensorSettings.SensorReportInterval, minInterval);
_accelerometer.ReadingChanged += AccelerometerOnReadingChanged;
_accelerometer.Shaken += AccelerometerOnShaken;

// Read the initial sensor value
_sensorSettings.LatestAccelerometerReading = GetAccelerometerReading();
```

**Gyrometer**

The gyrometer provides information about the device’s current rate of rotation around the x-, y-, and z-axes, measured in degrees per second. When available, this sensor returns readings as instances of the GyrometerReading type, which provides its results in AngularVelocityX, AngularVelocityY, and AngularVelocityZ properties.

Listing 13.17 shows how the example project is configured to work with the gyrometer, which is exposed via the Gyrometer class. The steps involved in configuring the gyrometer are otherwise identical to those shown previously for configuring the other sensors.
LISTING 13.17 Configuring the Gyrometer

// Get the reference to the sensor and see if it is available
_gyrometer = Gyrometer.GetDefault();
if (_gyrometer == null) return;

_sensorSettings.IsGyrometerAvailable = true;

// Set the minimum report interval. Care must be taken to ensure
// it is not set to a value smaller than the device minimum
var minInterval = _gyrometer.MinimumReportInterval;
_gyrometer.ReportInterval
    = Math.Max(_sensorSettings.SensorReportInterval, minInterval);
_gyrometer.ReadingChanged += GyrometerOnReadingChanged;

// Read the initial sensor value
_sensorSettings.LatestGyrometerReading = GetGyrometerReading();

Orientation Sensor
The last sensor directly related to motion and/or orientation to be discussed
is the orientation sensor. As Table 13.4 described, the orientation sensor is
a composite sensor whose output consists of information gathered from
accelerometer, gyrometer, and magnetometer data. As you can see in Listing
13.18, the orientation sensor is configured using the OrientationSensor class
in the same way the rest of the sensors have been in this section. Its results
are returned in an instance of the OrientationSensorReading class, which
contains properties for Quaternion and RotationMatrix values, structures
that 3D and gaming apps often use.

LISTING 13.18 Configuring the Orientation Sensor

// Get the reference to the sensor and see if it is available
_orientationSensor = OrientationSensor.GetDefault();
if (_orientationSensor == null) return;

_sensorSettings.IsOrientationSensorAvailable = true;

// Set the minimum report interval. Care must be taken to ensure
// it is not set to a value smaller than the device minimum
var minInterval = _orientationSensor.MinimumReportInterval;
_orientationSensor.ReportInterval
    = Math.Max(_sensorSettings.SensorReportInterval, minInterval);
_orientationSensor.ReadingChanged += OrientationSensorOnReadingChanged;
// Read the initial sensor value
_sensorSettings.LatestOrientationSensorReading = GetOrientationSensorReading();

**Light Sensor**

The light sensor isn’t actually a motion-/orientation-related sensor, but it is included as an honorable mention with these sensors because the APIs for working with this sensor are closely related to the rest of the APIs in this section. The light sensor reports the intensity of the light shining on the current device display in units of lux, is accessed through the `LightSensor` class, and returns its values in a `LightSensorReading` instance (which contains the property `IlluminanceInLux`). Listing 13.19 shows how the example project is configured to work with the light sensor.

**LISTING 13.19 Configuring the Light Sensor**

```csharp
// Get the reference to the sensor and see if it is available
_lightSensor = LightSensor.GetDefault();
if (_lightSensor == null) return;

_sensorSettings.IsLightSensorAvailable = true;

// Set the minimum report interval. Care must be taken to ensure
// it is not set to a value smaller than the device minimum
var minInterval = _lightSensor.MinimumReportInterval;
_lightSensor.ReportInterval = Math.Max(_sensorSettings.SensorReportInterval, minInterval);
_lightSensor.ReadingChanged += LightSensorOnReadingChanged;

// Read the initial sensor value
_sensorSettings.LatestLightSensorReading = GetLightSensorReading();
```

**Summary**

In this chapter, you learned how to work with several different user input devices, including pointer-based devices such as touch inputs, mouse devices, stylus devices, and keyboards. You saw how the Windows Runtime provides the capability to determine which devices are connected, as well as how adding the capability to interact with the various different kinds of
pointer devices has coalesced into a set of APIs that are differentiated more by the level of abstraction than the characteristics of a specific device type.

You also saw how the Windows Runtime provides the capability to work with sensors that supply information about how the device is interacting with its physical environment. This includes working with the geolocation APIs to obtain device position information. It also includes the related geofencing APIs for defining geographic boundaries that can result in app notifications when a device either enters or exits those boundaries. You also worked with the motion and orientation sensor APIs that provide insight into the device’s physical position and movement.

In the next chapter, you learn about the support the Windows Runtime offers for working with these peripheral devices. This includes a discussion about how you can add the capability to scan from your Windows Store apps. You also see how you can print from your app, including how to generate content and layouts specifically for printing, as well as how to customize and interact with the Print Settings and Print Preview experiences.
Index

Numbers
3DES (Triple Data Encryption Algorithm), 338

A
accelerometer, 517, 526-527
Accelerometer class, 527
AcceptAllSetting property, 420-421
accessibility, 615
  accessibility checker, 627-628
  automation and lists, 624-625
  automation peers, 626-627
  automation properties, 622-623
  keyboard support, 620-622
  live settings, 625
Narrator, 623
themes
  high contrast, 618-619
  requested theme, 616-618
AccessibilityTestProject, 626-627
AccessibleItemConverter, 624
accessing
  badges, 239
  SkyDrive content, 316
  UI thread, 369-370
Account Picture Provider, 50, 470-471
accounts (developer), 657-658
acquiring audio/video, 598-599
CameraCaptureUI, 600-603
declaring application capabilities, 599-600
MediaCapture, 604-610
  audio/video capture process, 608-610
  camera settings, 607-608
  capture preview, 607
  creating and initializing, 604-607
  image capture process, 610
ActionCommand class, 156
Activate method, 435
AdControl components, 680-683
Add a Language dialog box, 635
Add Service Reference dialog, 205
Add Translation Languages option (Multilingual App Toolkit), 646
AddPages event, 546-547
AdDuplex, 679
Advanced Encryption Standard (AES), 338
advertisements, 678-683
    AdControl components, 680-683
    pubCenter configuration, 679-680
Advertising SDK for Windows 8.1, 678
AES (Advanced Encryption Standard), 338
Alerts setting (AudioCategory), 564
algorithms
    asymmetric algorithms, 345-347
    compression algorithms, 190-191
AllowCropping property (CameraCaptureUI PhotoSettings), 602
AllowedMayUseActiveRealTimeConnectivity, 574
AllowedWithAlwaysOnRealTimeConnectivity, 574
AllowTrimming property (CameraCaptureUI VideoSettings), 601
AllTasks collection, 576
animations, applying to controls, 97-100
AnimationsExample project, 97
APIs
    proximity, 403
    sockets APIs, 379
    Windows Azure Mobile Services, 289-291
APM (Asynchronous Programming Model), 218
APNS (Apple Push Notification Service), 292
App Manifest entry, 458
app manifest (Windows Store apps), 45-46
    application UI, 46-48
    capabilities, 48-50
    content URIs, 52
    declarations, 50-51
    packaging, 52
    in-app purchases, 675-678
AppBar control, 130
AppBarButton control, 130
AppBarSeparator control, 130
AppBarToggleButton control, 130
AppCredentialStorage class, 331
AppendLinesAsync method, 174
AppendTextAsync method, 174
Apple Push Notification Service. See APNS (Apple Push Notification Service)
application activation, integrating, 462-463
    account picture providers, 470-471
    AutoPlay, 471-473
    example app, 463
    file activation, 463-467
    protocol activation, 467-470
Application class, 86, 90
application lifecycle
    NavigationHelper class, 67-69
    SuspensionManager class, 67-69
    Windows Store apps, 61-67
application sharing, 431-433
application storage
    app storage hierarchy, 159-160
    composite values, 165-166
    containers, 162
    roaming data, 161-162
    settings, 163-165
application UI, 46-48
ApplicationData class, 163
ApplicationDataCompositeValue class, 165
ApplyDeviceSettings method, 605-606
appointments, integrating, 473-474, 476-478
ApproachingDataLimit, 387
apps
    accessibility, 615
        accessibility checker, 627-628
        automation and lists, 624-625
        automation peers, 626-627
        automation properties, 622-623
        keyboard support, 620-622
        live settings, 625
        Narrator, 623
        themes, 616-619
    in-app purchases, 675-678
application activation, integrating, 462-463
  account picture providers, 470-471
AutoPlay, 471-473
example app, 463
file activation, 463-467
protocol activation, 467-470
application lifecycle
  NavigationHelper class, 67-69
  SuspensionManager class, 67-69
  Windows Store apps, 61-67
application storage
  app storage hierarchy, 159-160
  composite values, 165-166
  containers, 162
  roaming data, 161-162
  settings, 163-165
debugging
  code analysis, 712-716
  debug windows, 693-694
  Edit and Continue, 690
  exceptions, 694-696
  Just In Time debugging, 691
  Just My Code, 688-689
  launching debugger, 691-692
  logging and tracing, 696-702
  native, managed, and script debuggers, 686-688
  overview, 685-686
  program databases, 692-693
deployment
  other deployment options, 665-667
  overview, 649-650
  publishing to Windows Store, 657-665
globalization and localization
  default language, 633-634
  design considerations, 632-633
  formatting dates, numbers, and currencies for locale, 642-643
  Multilingual App Toolkit, 644-648
  MVVM (Model-View-ViewModel), 643-644
  overview, 631-632
  preferred languages, 635-637
  resource qualification and matching, 637-639
  XAML elements, 639-642
making money from, 667-668
  advertisements, 678-683
  in-app purchases, 675-678
  PackageAndDeployExample project, 668-669
  pricing apps, 669-670
  trial mode apps, 670-675
packaging
  app package and app bundle contents, 654-655
  Create App Packages Wizard, 650-654
  overview, 649-650
  package identifiers, 655-656
performance optimization
  code analysis, 712-716
  logging and tracing, 696-702
  profiling and performance analysis, 702-712
Play To Source, 444-446
Play To Target, 446-448
pricing, 669-670
publishing to Windows Store, 657-665
  application certification, 661-665
  developer accounts, 657-658
  steps, 658-661
Share Source, 423-433
  application sharing, 431-433
  bitmap sharing, 429
  DataRequestEventArgs class, 424
  DataTransferManager class, 424-425
  file sharing, 430
  HTML sharing, 428
  predefined metaproperties available in share data package, 426
  share data types, 427
Share Target, 433-442
  creating, 434-441
  debugging, 441-442
sharing, 431-433
Skrape, 154
trial mode apps, 670-675
.appx file extension, 650, 654
AppxBundleManifest.xml, 654
archives (zip), 187-191
AreTransportControlsEnabled property, 564, 595
ARM-based chip, 2
arrays, 725
AsBuffer method, 171
AsForecastEntry method, 207
AspectRatioHeight property
(MediaElement), 597
AspectRatioWidth property
(MediaElement), 597
AsWeatherForecast method, 208
asymmetric algorithm encryption, 345-347
asynchronous functions, 24-26
asynchronous printer actions, 536
Asynchronous Programming Model (APM), 218
Atom, 2, 199
Atom Publishing Protocol (AtomPub), 219
AtomPub (Atom Publishing Protocol), 219
AttachAsync() method, 411, 563
attached properties, 94-95
attributes. See specific attributes
audio
acquiring, 598-599
CameraCaptureUI, 600-603
declaring application capabilities, 599-600
MediaCapture, 604-610
capturing, 598-599
looping in toasts, 248
playing, 590
audio settings, 596-597
in background, 563-569
controlling playback, 592-595
media information, 597
media markers, 597-598
MediaElement control appearance, 595-596
MediaElement control instantiation, 591-592
MultimediaExample project framework, 590-591
text-to-speech support, 610-612
AudioBackgroundExample project, 563-569
AudioCategory settings, 564
AudioDeviceController property
(MediaCapture class), 607-608
AudioDeviceId (MediaCapture class), 604
AuthenticateAsync method, 327
authentication, 324-329
documentation, 326
Facebook, 329
Live Connect, 304-307
multistep (Google), 330-331
options, 328
symmetrical encryption, 342-343
unlocking password vault, 331-333
Windows Azure Mobile Services, 269-270
authenticating users, 272-274
registration and configuration for
Microsoft account logins, 270-272
authorization, Windows Azure Mobile Services, 287-289
automation
automation peers, 626-627
automation properties, 622-623
lists, 624-625
AutomationProperties, 622-623
AutoPlay, 471-473
AutoPlay Content, 50
AutoPlay Device, 50
AutoPlay property (MediaElement), 595
Autos window (debugger), 693
Autoscale values, 300
availability
devices, 482
for scanners, 548-549
await keyword, 148
AxisAdjustmentFactor extension
method, 524
Azure. See Windows Azure
B
background tasks, 50, 559
audio, 563-569
control channel, 585-586
downloads, 562-563
lock screen tasks, 570
  conditions, 578-580
  creating background tasks, 573-576
  debugging background tasks, 580-581
  listing background tasks, 576-577
  lock screen capabilities, 570-573
timer, 578
overview, 559-560
raw push notifications, 581-585
system events, 587-588
thread pool, 560-562
uploads, 562-563
background transfers, 408-412
BackgroundAccessStatus enumeration, 574
BackgroundCapableMedia setting
  (AudioCategory), 564
BackgroundDownloadAsync method, 319
BackgroundTaskBuilder, 577
BackgroundTaskRegistration class, 576
BackgroundTransferStatus, 410
BackgroundUploader class, 562-563
BackgroundWorkCostChange event, 587
BackgroundWorkCostNotHigh condition, 579
BadgeHelper class, 241
badges, 239-241
  accessing, 239
  periodic notifications, 242
  XML, 241
Balance property (MediaElement), 597
Base Class Library (BCL), 187
base types, 720
BaseCrypto class, 339
BaseTile class, 235
Basic tier (Windows Azure Mobile
  Services), 299
BCL (Base Class Library), 187
BCP-47 language tag, 633
binding. See data-binding
Bing Maps control, 499-502
Bing Maps Platform, 499
Bitmap data type, 427
BitmapIcon control, 130
bitmaps, sharing, 429
Blank app, 32, 45
block ciphers, 337-341
Bluetooth RFCOMM, 389, 396
Border control, 115
branding live tiles, 232
Buffer class, 172
buffers, 171-174
  .NET Framework, 14-15
  Buffer class, 172
BuildPicturePage function, 545
bundles (app), 654-655
Button control, 103-106, 130
C
C#, 9, 722
Cached File Updater, 50
cached files, 180-187
CalculatePi method, 560
calendars, Live Connect, 311-314
Call Stack window, 689, 694
calling managed WinRT components,
  78-79
camera settings, 50, 607-608
CameraCaptureUI, 600-603
cancel method, 380
Canvas control, 116
CaptureAsync method, 603
CaptureFileAsync method, 600
CaptureMode (MediaCapture class), 604
CapturePhotoToStorageFileAsync target
  type, 610
CapturePhotoToStreamAsync target type,
  610
capturing
  audio/video. See acquiring audio/
  video
  pointers, 486
CBC (cipher-block chaining), 338
CDYNE Weather, 201
certificates, 50, 347
ChannelPushNotificationReceived
  method, 583
Charms
Devices Charm
  overview, 442-443
  Play To Source app, 444-446
  Play To Target app, 446-448
  PlayToExample project, 443-444
overview, 415-417
setting entries, adding, 418-421
Settings Charm
  overview, 417
  ShareTargetExample project, 418-421
Share Charm
  overview, 421-423
  share process, 422
  Share Source app, 423-433
  Share Target app, 433-442
CheckBox control, 130
CIL (Common Intermediate Language), 30
ciphers, 337
cipher-block chaining, 338
stream ciphers, 341
class ID (CLSID), 726
class library templates, 35
class methods, 721-722
classes, 721-722. See also specific classes
ClearAll method, 567
client app data storage, 278-282
ClientListener method, 395
clients
  OData (Open Data), 217-219
  SyndicationClient, 219-222
Clipboard class, 154-159
copying content to, 158-159
data formats, 158
pasting content from, 157-158
Close method, 13
clouds, connecting to
  with Live Connect. See Live Connect
  Windows Azure Mobile Services. See Windows Azure Mobile Services
CLR (Common Language Runtime), 1
CLSID (class ID), 726, 728
cmdlets, Import-Module, 666
code analysis, 712-716
code reuse, MVVM (Model-View-ViewModel), 366
coded UI tests, 43
collections, AllTasks, 576
COM (Component Object Model), 726
CompoSource app, 624-625
CommandBar control, 130
commands
  MVVM (Model-View-ViewModel), 371
  ViewModel, 361
Common Intermediate Language. See CIL
  (Common Intermediate Language)
Common Language Runtime (CLR), 1
CommonAssemblyInfo.cs, 6
CommonFileQuery enumeration values, 176
Communications setting
  (AudioCategory), 564
company developer accounts, 657-658
Compare method, 334
compass, 517, 519-523
CompassOffset extension method, 521
compile errors when referencing
  WinRT, 18
Component Object Model (COM), 726
components. See also classes; controls
  examining projections in WinRT
    components, 20-24
  managed WinRT components, 75-76
    calling in any language, 78-79
    creating, 76-77
templates, 42
composite values, 165-166
compression, 187-191
conditions for lock screen tasks, 578-580
ConfigureBackgroundTask method, 583
ConfigureScanner method, 554
configuring
  Microsoft account logins, 270-272
  print tasks, 534-537
  QuickLinks, 439-441
ConnectCommand method, 406
Connected Services Manager, 268
connecting apps to Mobile Services
  instances, 267-268
connecting to cloud with Live Connect.
  See Live Connect
ConnectionInfo class, 386
ConnectionProfile, 386
connectivity, data plans and, 384-389
Contact and Appointment plans integration, 473-474
appointments, 476-478
contacts, 474-476
text added for example app, 474
Contact Cards, 474
contact pickers, 51, 452-453, 460-462
ContactManager class, 475
contacts, 473-474
integrating, 474-476
Live Connect, 310-311
containers, 162
creating, 162
nesting, 162
content URLs, 52
ContentControl, 120-121
contents of app packages, 654-655
ContentSourceApplicationLink property (Share DataPackage), 426
ContentSourceWebLink property (Share DataPackage), 426
control channel, 585-586
control independence, UI design patterns, 351
ControlChannelReset event, 587
ControlChannelTrigger, 585-586
controlling multimedia playback, 592-595
controls. See also classes
AdControl, 680-683
animations, applying, 97
AppBar, 130
AppBarButton, 130
AppBarSeparator, 130
AppBarToggleButton, 130
BitmapIcon, 130
Border, 115
Button, 130
groups and states, 103
states, 103-106
Canvas, 116
CheckBox, 130
ComboBox, 130, 624-625
CommandBar, 130
ContentControl, 120-121
custom controls, 135-140
DatePicker, 130
FlipView, 124-125
flyout, 133-135
FontIcon, 130
Grid, 116-117
GridView, 123
Hub, 131
HyperLink, 131
HyperlinkButton, 131
Image, 131
ItemsControl, 121
ListBox, 123-124
ListView, 124
MediaPlayer, 131
Panel, 115
PasswordBox, 131
PathIcon, 131
PopupMenu, 131
ProgressBar, 131
ProgressRing, 131
RadioButton, 131
RepeatButton, 131
RichEditBox, 131
RichTextBlock, 131, 196-197
RichTextBlockOverflow, 132
ScrollBar, 132
ScrollViewer, 122
SearchBox, 132
SearchBox control, 73-75
SemanticZoom, 132
Slider, 86, 89, 96, 132
StackPanel, 117-118
SymbolIcon, 132
table of, 130-132
templates
ControlTemplate class, 112
DataTemplate class, 112
TemplatesExample project, 112-115
TextBlock, 132
TextBox, 132
TimePicker, 132
ToggleButton, 132
ToggleButtonSwitch, 132
ToolTip, 132
VariableSizedWrapGrid, 119-120
ViewBox
   LayoutsExample project, 125-129
   modes, 122
VirtualizingPanel, 118
VirtualizingStackPanel, 118
WebView, 132, 143-150
Windows Charms
   Devices Charm, 442-448
   overview, 415-417
   Settings Charm, 417-421
   Share Charm, 421-442
WrapGrid control, 119
ControlTemplate class, 112
ConvertBinaryToString, 334
Converter attribute, 88
ConverterLanguage attribute, 88
ConverterParameter attribute, 88
ConvertStringToBinary, 334
CroppedAspectRatio property
   (CameraCaptureUI PhotoSettings), 602
CroppedSizeInPixels property
   (CameraCaptureUI PhotoSettings), 602
CryptographicBuffer class, 334
Culture folder, 639
Cumulative property, 490
currencies, formatting for locale, 642-643
CurrencyFormatter class, 642-643
CurrentAppSimulator class, 671, 672
CurrentStateChanged event, 447-448
custom API scripting objects, 290
custom APIs, 289-291
custom controls, 135-140
creating, 135-140
custom data types, 427
custom print options, 538-542
custom VSMs (Visual State Managers), 109-110
CustomVisualStateManager class, 110
cycling tile notifications, 234-236

D
Dark theme, 616
data access scripts, 282-287
data-binding
   attached properties, 94-95
   attributes, 88
   classes, 86-87
dependency properties, 91-94
   explained, 85-86
   MVVM (Model-View-ViewModel), 366
   Portable ViewModel, 41
   property change notification, 95-97
   value precedence, 95
Data Encryption Standard (DES), 338
data formats, 191-192
DataFormatsExample project, 192-194
XSLT transformation, 195
data manipulation. See also application storage
Clipboard class, 154-159
  copying content to, 158-159
data formats, 158
  pasting content from, 157-158
document data, 196-197
files. See files
folders. See folders
Skrape app, 154
data plans, connectivity and, 384-389
data protection providers, 333-337
data storage, Windows Azure Mobile Services, 274
  authorization, 287-289
data access scripts, 282-287
data in client apps, 278-282
  managing data tables, 274-278
data tables, managing, 274-278
data types, share, 427
databases (program), 692-693
DataBindingExample project, 83-84, 89-91
DataFormatsExample project, 192-194
DataPackage method, 159
DataPackage object, 426
DataPackageView class, 436
DataReader class, 172
DataReaderLoadOperation class, 172
DataRequested event handler, 424
DataRequestEventArgs class, 424
DataTemplate class, 87, 112
DataTransferManager class, 424-425
DataWriter class, 172
DataWriterStoreOperation class, 172
DatePicker control, 130
dates, 12
  formatting for locale, 642-643
DateTimeFormatter class, 643
debug windows, 693-694
debugging
  background tasks, 580-581
code analysis, 712-716
debug windows, 693-694
Edit and Continue, 690
exceptions, 694-696
Fiddler, 211-213
Just In Time debugging, 691
Just My Code, 688-689
launching debugger, 691-692
logging and tracing, 696-702
native, managed, and script debuggers, 686-688
overview, 685-686
program databases, 692-693
Share Target app, 441-442
DecimalFormatter class, 642-643
declarations, Windows Store apps, 50-51
DecodeFromBase64String, 335
DecodeFromHexString, 335
decryption, 341, 342
default language, 633-634
default tiles, 227-229
DefaultPlaybackRate property
  (MediaElement), 594
DefaultQuery value (CommonFileQuery), 176
DefaultScanSource property, 549
deferrals, 64
DelayedBitmapRequestCallback method, 432
delegates, 724
DELETE request (REST), 209
DeleteAsync method, 169, 170
deleting SkyDrive items, 318
Delta property, 490
dependency properties, 91-94
DependencyObject class, 87
DependencyProperty class, 87
deployment
  deployment tiers (Windows Azure Mobile Services), 298-300
  other deployment options, 665-667
overview, 649-650
publishing apps to Windows Store, 657-665
  application certification, 661-665
developer accounts, 657-658
steps, 658-661
DES (Data Encryption Standard), 338
Description property (ShareDataPackage), 426
design, globalization and localization, 632-633
design-time data, MVVM (Model-View-ViewModel), 367-369
Desired View option, 465
DesiredAccuracy property, 507
desktop applications, 15
creating, 15
referencing WinRT, 15-20
toasts, 248
developer accounts (Windows Store), 657-658
Developer Command Prompt, 728
Developer Preview version (Windows 8), 29
devices. See input devices; sensor input
Devices Charm
overview, 442-443
Play To Target app, 446-448
dialogs, MVVM (Model-View-ViewModel), 371
dictionaries, resource, 90-91
DirectX WinRT, 485
Disabled state, 103
DisplayedOptions collection, 541
DisplayOrientations class, 521
document data, 196-197
Document Object Model (DOM), 143
documentation
authentication, 326
WebRequest object, 36
DocumentDataExample project, 196-197
DocumentsLibrary folder, 167
DOM (Document Object Model), 143
DoubleAnimation class, 98
DoubleTapped, 493
downloading from OneDrive (SkyDrive), 319-320
downloads, 562-563
DownloadsFolder class, 166, 167
DPAPI (Windows Data Protection API), 333
Duration, 513
DwellTime, 512

E
earning money from apps. See income, earning from apps
Edit and Continue option (debugging), 690
ElementName attribute, 88
elements (XAML), 81
localization, 639-642
embedded HTML, 146-150
EncodeToBase64String, 335
EncodeToHexString, 335
crypt parameter, 359
encryption, 333
asymmetric algorithms, 345-347
data protection providers, 333-337
symmetrical encryption, 337
authentication, 342-343
block ciphers, 337-341
stream ciphers, 341
verification, 343
hash algorithms, 343-344
MAC (Message Authentication Code), 344-345
EncryptSymmetric method, 340
development kits, deploying apps to, 666-667
energy consumption, 710-712
EnsureLength method, 340
enterprise authentication, 49
enumerations, 723
en-US folder, 637
errors, HTTP status codes and REST correlation, 211
es-ES folder, 637
Ethernet LAN, 384
ETW (Event Tracing for Windows), 696-702
event handlers, 4
Event Tracing for Windows (ETW), 696-702
EventListener class, 698
events, 85, 724
application lifecycle, 61-62
CurrentStateChanged, 447-448
geofencing, 514-516
<table>
<thead>
<tr>
<th>Event</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Live Connect</td>
<td>311-314</td>
</tr>
<tr>
<td>LogFileGenerated</td>
<td>701</td>
</tr>
<tr>
<td>LoggingEnabled</td>
<td>701</td>
</tr>
<tr>
<td>NavigationCompleted</td>
<td>144</td>
</tr>
<tr>
<td>PlayRequested</td>
<td>447-448</td>
</tr>
<tr>
<td>PropertyChanged</td>
<td>94</td>
</tr>
<tr>
<td>RateChanged</td>
<td>594</td>
</tr>
<tr>
<td>routed events</td>
<td>85</td>
</tr>
<tr>
<td>SourceChangeRequested</td>
<td>447</td>
</tr>
<tr>
<td>SourceRequested</td>
<td>444-445</td>
</tr>
<tr>
<td>SourceSelected</td>
<td>445</td>
</tr>
<tr>
<td>system events</td>
<td>587-588</td>
</tr>
<tr>
<td>EventSource class</td>
<td>697-698</td>
</tr>
<tr>
<td>examine projections in WinRT components</td>
<td>20-24</td>
</tr>
<tr>
<td>Exception class instances</td>
<td>391</td>
</tr>
<tr>
<td>exceptions</td>
<td>13, 391, 694-696</td>
</tr>
<tr>
<td>Execute method</td>
<td>246</td>
</tr>
<tr>
<td>Expansion</td>
<td>490</td>
</tr>
<tr>
<td>expiration, toasts</td>
<td>245</td>
</tr>
</tbody>
</table>

**Extensible Application Markup Language.** See XAML (Extensible Application Markup Language)

**Extensible Markup Language (XML),** 199

**Extensible Stylesheet Language Transformations (XSLT),** transformation, 195

**extension methods,** 15, 25

---

**F**

**Facebook**
- authentication sessions, 329
  - SSO (single sign-on), 326
- FacebookAuthenticator class, 327
- FacebookIdentity class, 329
- FadeOutThemeType class, 99
- FailIfExists option, 169
- FailWithDisplayText method, 424
- fences, geofencing, 512-514
- Fiddler, 211-213
- file activation, 463-467
- file pickers
  - file open pickers, 51, 454-458
  - file save pickers, 51, 458-460
- integrating, 452-453

**files**
- file type associations, 51
- FileInputStream class, 173
- FileIO class, 174-175
- FileLoggingSession, 701
- FileOpenPickerUI, 454-457
- FileOutputStream class, 173
- FilePickerExample project, 183-187
- FileRandomAccessStream class, 173
  - AppxBundleManifest.xml, 654
  - buffers and streams, 171-174
  - compression, 187-191
  - flag.lang-en.jpg, 637
  - flag.lang-es.jpg, 637
  - path and file helper classes, 174-175
  - pickers and cached files, 180-187
  - sharing, 430
  - storage files, 170-171
  - storage query operations, 176-180
  - .vspx extension, 706
  - .xlf (Localization Interchange File Format) files, 645
- FileSavePicker class, 181
- FileSavePickerUI, 458
- FileTypeFilter property, 454
- Fill mode (ViewBox), 122
- filled arrays, 725
- filtered lists, MVVM (Model-View-ViewModel), 373-374
- finding packages on disk, 52-54
- firewalls, push notifications, 249
- flag.lang-en.jpg file, 637
- flag.lang-es.jpg file, 637
- FlipView control, 124-125
- flyouts, 133-135
- FlyoutsExample project, 133-135

**folders**
- creating in OneDrive (SkyDrive), 318
- Culture, 639
- DocumentsLibrary, 167
- DownloadsFolder class, 167
- en-US, 637
- es-ES, 637
- HomeGroup, 167
- MediaServerDevices, 167
- MultilingualResources, 645
Index

MusicLibrary, 167
PicturesLibrary, 167
Playlists, 167
RemovableDevices, 167
SavedPictures, 168
storage folders, 167-170
storage query operations, 176-180
table of, 166-167
VideosLibrary, 168
WindowsStoreProxy, 672
FontIcon control, 130
ForegroundOnlyMedia setting
(AudioCategory), 564
Format method, 698
Format property
CameraCaptureUI PhotoSettings, 602
CameraCaptureUI VideoSettings, 601
formats (data), 191-192
DataFormatsExample project, 192-194
XSLT transformation, 195
formatting dates, numbers, and
currencies for locale, 642-643
frame rate counters, 702-704
Framework design guidelines, 712
FrameworkElement class, 87
FrameworkElementAutomation-
Peer class, 626
Free tier (Windows Azure Mobile
Services), 299
free_busy, 313
FreeNetworkAvailable condition, 579
FromConnectionProfile method, 386
functions. See specific functions
fundamental types (.NET Framework),
9-10
FutureAccessList, 187

G
GameEffects setting (AudioCategory),
564
GameMedia setting (AudioCategory), 565
GCM (Google Cloud Messaging
service), 292
Generate Machine Translations option
(Multilingual App Toolkit), 646
GenerateRandom, 335
GenerateUniqueName option, 169
generating print preview content, 543-545
generics, 722
Geocoordinate, 508-509
GeofenceMonitor, 513
gEOFences, managing, 516-517
gEOFencing, 510-511
defining fences, 512-514
events, 514-516
managing geofences, 516-517
gEOlocation, sensor input, 502-503
simulator location tools, 509-510
Geolocator, 504-508
Geoshape, 512
gesture events, 484, 493-495
GET request (REST), 209
GetBasicPropertiesAsync method, 170
GetConnectionCost method, 387
GetCurrentPoint method, 487
GetCurrentUploadsAsync() method, 562
GetDeferral method, 430, 432
GetFileAsync method, 169
GetFileFromApplicationUriAsync
method, 170
GetFileFromPathAsync method, 170
GetFilesAsync method, 169
GetFolderAsync method, 169
GetFolderFromPathAsync method, 169
GetFoldersAsync method, 169
GetForCurrentView method, 424-425
GetGeopositionAsync method, 504
GetImageUriForType method, 207
GetIntermediatePoints method, 487
GetInternetConnectionProfile method,
386
GetItemsAsync method, 169
GetKeyState method, 497
GetNetworkUsageAsync method, 388
GetPointerDevice method, 481
GetPreviewPage event, 545
GetScannersAsync method, 548
GetTemplateChild method, 139
GetTemplateContent method, 230
GetThumbnailAsync method, 169, 170
GetToken method, 254
GetWeatherForZipCode method, 208
GetXmlDocument method, 222
Git source control integration, 286
Global System for Mobile
Communications (GSM), 384
globalization
default language, 633-634
design considerations, 632-633
formatting dates, numbers, and
currencies for locale, 642-643
Multilingual App Toolkit, 644-648
MVVM (Model-View-ViewModel), 643-644
overview, 631-632
preferred languages, 635-637
resource qualification and matching, 637-639
XAML elements, 639-642
GlobalizationExample project. See
globalization
globally unique identifier (GUID), 726
GlobalViewModel class, 167
Google
Google Cloud Messaging service
(GCM), 292
multistep authentication, 330-331
GoogleIdentity class, 331
GoToVisualState method, 127
graphs, rich object, 82
Grid app, 33
Grid App template, 5
Grid control, 116-117
grids
Grid control, 116-117
GridView control, 123
VariableSizedWrapGrid control, 119-120
WrapGrid control, 119
GridView control, 123
GroupDetailPage, 7
GroupItemsPage, 7
groups (VSM), 103-105
GSM (Global System for Mobile
Communications), 384
GUID (globally unique identifier), 726
gyrometer, 517, 527-528

H
HandleOrientation property, 101
HandleSwitchToggled event handler, 420
handling exceptions, 694-696
hash algorithms, 343-344
HelpText property, 622
high contrast, 618-619
HighPriority setting, 163
Holding, 493
HomeGroup, 167, 382-384
HTML (Hypertext Markup Language)
   pages, 143-150
   sharing, 428
HTML data type, 427
HTML5 WebSocket Protocol, 389
HtmlFormatHelper class, 159, 428
HTTP (Hypertext Transfer Protocol),
   379-382
   HTTP API, 382
   status codes and REST correlation, 211
HttpBufferContent, 381
HttpClient, 216
HttpFormUrlEncodedContent, 381
HttpJsonContent, 381
HttpMultipartContent, 381
HttpMultipartFormDataContent, 382
HttpRequestMessage, 380
HttpStreamContent, 382
HttpStringContent, 382
Hub app, 33-34
Hub control, 131
HyperLink control, 131
HyperlinkButton control, 131
Hypertext Transfer Protocol. See HTTP
   (Hypertext Transfer Protocol)
I
IAsyncAction interface, 561
IAsyncOperation interface, 26
IBackgroundTask interface, 574
IBuffer interface, 171, 335
IClose interface, 13
ICommand interface, 39, 361
ICoreWindow instance, 485
Id geofencing settings, 512
Index

IDataReader interface, 172
IDataWriter interface, 172
identifiers (package), 655-656
identifying connected input devices, 481-483
IID (interface ID), 726
IInputStream interface, 172
Image control, 131
ImageEncodingProperties class, 610
images, capturing with MediaCapture, 610
ImageScanner class, 549, 551
ImageScannerFormat, 553
Immediate window (debugger), 693
Import-Module cmdlet, 666
inclinometer, 517, 523-526
income, earning from apps advertisements, 678-683
   AdControl components, 680-683
   pubCenter configuration, 679-680
in-app purchases, 675-678
PackageAndDeployExample project, 668-669
pricing, 669-670
trial mode apps, 670-675
individual developer accounts, 657-658
initialization vector (IV), 338
InitializeAsync method, 334, 604, 606
InitializeComponent method, 141
InitializeSocket method, 406
InitializeWithServiceData method, 217
initializing MediaCapture, 604-607
InMemoryRandomAccessStream class, 173, 551
INotifyCollectionChanged interface, 97
INotifyPropertyChanged interface, 94, 96
input devices, 480
   example app, 480-481
   gesture events, 484, 493-495
   identifying connected, 481-483
   keyboard input, 495-497
   manipulation events, 484, 488-492
   pointer events, 484-488
InputEventHandler class, 484
InputStreamOverStream class, 173
installation ID, 294
integrated push notification support (Windows Azure Mobile Services), 291-297
integrating, 451-452
application activation, 462-463
   account picture providers, 470-471
   AutoPlay, 471-473
   example app, 463
   file activation, 463-467
   protocol activation, 467-470
with contacts and appointments, 473-474
   appointments, 476-478
   contacts, 474-476
with file and contact pickers, 452-453
   contact pickers, 460-462
   example app, 453
   file open pickers, 454-458
   file save pickers, 458-460
interface ID (IID), 726
interfaces, 723, 731
IAsyncAction, 561
IBackgroundTask, 574
IBuffer, 171
IDataReader, 172
IDataWriter, 172
IInputStream, 172
INotifyCollectionChanged, 97
INotifyPropertyChanged, 94
IOutputStream, 172
IRandomAccessStream, 172, 173
IValueConverter, 89-90
internal code, 719
internals (WinRT), 725-731
Internet (Client), 49
Internet (Client & Server), 49
InternetAvailable condition, 579
InternetAvailable event, 587
InternetNotAvailable condition, 579
Invalid condition, 579
Invalid event, 587
IOutputStream interface, 172
IRandomAccessStream instance, 551
IRandomAccessStream interface, 172, 173
IReference, 13
IsAudioOnly property (MediaElement), 597
IsFullWindow property (MediaElement), 596
IsIntegrated flag, 481
IsLooping property (MediaElement), 595
IsMuted property (MediaElement), 596
isolated storage, 159
IsScreenCaptureEnabled property (ApplicationView class), 433
IStorageItem data type, 427
ItemPage class, 700
items (SkyDrive), renaming, 318
ItemsControl, 121
IV (initialization vector), 338
IValueConverter interface, 89-90
J
JavaScript, invoking through WebView control, 148
journals, 8
JSON data values, 277
Json.NET, 216
Just In Time debugging, 691
Just My Code feature, 688-689
K
keyboard input
   accessibility, 620-622
   input devices, 495-497
KeyboardCapabilities class, 495
KeyDown, 496
keys, modifier, 497
keyUp, 496
L
language options (WinRT), 3
language projections, 731
languages
   default language, 633-634
   Multilingual App Toolkit, 644-648
   preferred languages, 635-637
LastOrientation property, 101
LaunchActivatedEventArgs, 65
LaunchFileAsync method, 464
launching debugger, 691-692
layouts, 115
   Border control, 115
   Canvas control, 116
   ContentControl, 120-121
   FlipView control, 124-125
   Grid control, 116-117
   GridView control, 123
   ItemsControl, 121
   LayoutsExample project, 125-129
   ListBox control, 123-124
   ListView control, 124
   Panel control, 115
   ScrollViewer control, 122
   StackPanel control, 117-118
   VariableSizedWrapGrid control, 119-120
   VBox control
      LayoutsExample project, 125-129
   modes, 122
   VirtualizingPanel control, 118
   VirtualizingStackPanel control, 118
   WrapGrid control, 119
LayoutsExample project, 125-129
LicenseInformation class, 670-671
LicenseInformation element, 677
LicenseInformation property, 671
licensing Bing Maps Platform, 499
lifecycle, application, 61
lifecycle events, print tasks, 535
light sensors, 518, 529
Light theme, 616
limited_details, 313
Link data type, 427
ListBox control, 123-124
listing background tasks, 576-577
lists
   automation, 624-625
   filtered lists, 373-374
ListView control, 124
Live Connect, 301-302
   authentication, 304-307
   calendars and events, 311-314
   contacts, 310-311
   Example app, 304
Live Connect Developer Center, 302-303
Live Connect SDK, 302-303, 309
profile information, 308-309
OneDrive (SkyDrive), 315-321
live settings, 625
live files, 229-234
branding, 232
templates, 229
LiveAuthClient class, 305
LiveSetting property, 625
LoadListingInformationAsync method, 671-673
LoadStoreProxyFile method, 672
LoadSyndicatedContent method, 220
local deployment, running, 55-56
localization
default language, 633-634
design considerations, 632-633
formatting dates, numbers, and currencies for locale, 642-643
Multilingual App Toolkit, 644-648
MVVM (Model-View-ViewModel), 643-644
overview, 631-632
preferred languages, 635-637
resource qualification and matching, 637-639
XAML elements, 639-642
Localization Interchange File Format (.xlf) files, 645
Locals window (debugger), 693
location, 49
simulator location tools, 509-510
lock screen tasks, 570
conditions, 578-580
creating background tasks, 573-576
debugging background tasks, 580-581
listing background tasks, 576-577
lock screen capabilities, 570-573
timer, 578
lockers, unlocking password vault, 331-333
LockScreen Example project. See lock screen tasks
LockScreenApplicationAdded event, 587
LockScreenApplicationRemoved event, 587
LogEventListener class, 698-699
LogFileGenerated event, 701
logging, 696-702
LoggingEnabled event, 701
LoggingHelper project, 697-702
LoggingSession, 701
LogoBackgroundColor property (Share DataPackage), 426
Long Term Evolution (LTE), 384
long-duration data load data type, 427
long-duration toasts, 248
looping audio, 248
LTE (Long Term Evolution), 384
M
MAC (Message Authentication Code), 344-345
MainPage.xml, 83-84
managed debugger, 686-688
managed WinRT components, 75-76
calling in any language, 78-79
creating, 76-77
managing exceptions, 694-696
manipulation events, 484, 488-492
ManipulationCompleted, 489
ManipulationDelta, 489, 490
ManipulationInertiaStarting, 489, 492
ManipulationModes, 489, 490, 492
ManipulationStarted, 489
MarkerReached event, 598
markers (media), 597-598
MaxDurationInSeconds property
(CameraCaptureUI VideoSettings), 601
MaxResolution property
CameraCaptureUI PhotoSettings, 602
CameraCaptureUI VideoSettings, 601
MD5 (Message Digest 5), 343
media markers, 597-598
MediaCapture, 604-610
audio/video capture process, 608-610
camera settings, 607-608
capture preview, 607
creating and initializing, 604-607
image capture process, 610
MediaCaptureInitializationSettings class, 604
MediaCaptureJob class, 608
MediaElement control, 131
appearance, 595-596
audio settings, 596-597
instantiating, 591-592
media information, 597
media markers, 597-598
MediaEncodingProfile parameter (MediaCapture), 608
MediaPlayer control, 131
MediaServerDevices folder, 167
Message Authentication Code (MAC), 344-345
Message Digest 5 (MD5), 343
message protocols, smart tags, 402
MessageReceived event, 391
MessageWebSocket class, 390
methods. See specific methods
Metro, 29
microphones, 49
Microsoft account logins, authenticating, 270-272
Microsoft Advertising SDK for Windows 8.1, 678
Microsoft design language, 29
Microsoft Push Notification Service. See MPNS (Microsoft Push Notification Service)
Microsoft System Center, 666
Mobile Services. See Windows Azure Mobile Services
mobile Services instance
adding, 264
connecting apps to, 267-268
Mode attribute, 88
models, UI design patterns, 351-352
Model-View-Controller. See MVC (Model-View-Controller)
Model-View-ViewModel. See MVVM (Model-View-ViewModel)
Modem UT, 29
modifier keys, 497
money, making from apps. See income, earning from apps
MonitoredStates, 512
motion sensors, 517-518
accelerometer, 526-527
gyrometer, 527-528
inclinometer, 523-526
light sensors, 529
MouseCapabilities class, 482
MoveAndReplaceAsync method, 170
MoveAsync method, 170
MovementThreshold, 508
MPNS (Microsoft Push Notification Service), 292
ms-appdata:/// protocol, 175
ms-appx:/// protocol, 175
ms-appx-web protocol, 146
Multilingual App Toolkit, 644-648
MultilingualResources folder, 645
multimedia
acquiring, 598-599
CameraCaptureUI, 600-603
declaring application capabilities, 599-600
MediaCapture, 604-610
overview, 589-590
playing, 590
audio settings, 596-597
controlling playback, 592-595
media information, 597
media markers, 597-598
MediaElement control appearance, 595-596
MediaElement control instantiation, 591-592
MultimediaExample project framework, 590-591
text-to-speech support, 610-612
MultimediaExample project. See multimedia
multipart uploads, 562
multiple files, uploading, 562
MultipleAnimationType class, 99
multistep authentication, 330-331
Music Library, 49
MusicLibrary folder, 167
MVC (Model-View-Controller), 351, 353-354
MVP (Model-View-Presenter), 354-355
MVVM (Model-View-ViewModel), 96-97, 349
accessing, 369-370
benefits of, 364-366
code reuse, 366
commands, 371
common misperceptions, 362-364
data-binding, 366
design-time data, 367-369
dialogs, 371
filtered lists, 373-374
localization, 643-644
refactoring, 366
selection lists, 371-373
tooling support, 366
UI design patterns, 350-351, 355-356
models, 351-352
MVC (Model-View-Controller), 353-354
MVP (Model-View-Presenter), 354-355
views, 352-353
unit testing, 365
validation, 375-377
viewmodel, 356-362
Windows Store apps, 17

N
namespaces, 720
Narrator, 623
native debugger, 686-688
NaturalVideoHeight property
(MediaElement), 597
NaturalVideoWidth property
(MediaElement), 597
navigation, state management and, 69-73
NavigationCompleted event, 144
NavigationHelper class, 7, 44, 67-69
NCP (Notification Client Platform), 249
NDEF (NFC Data Exchange Format), 397
Near Field Communications. See NFC (Near Field Communications)
negotiation, REST (Representational State Transfer), 211-213
nesting containers, 162
.NET Base Class Library (BCL), 187
.NET Framework, 9
extension methods, 25
WinRT
buffers, 14-15
fundamental types, 9-10
mapped types, 10-13
streams, 14-15
networking
background transfers, 408-412
connectivity and data plans, 384-389
HomeGroup, 382-384
HTTP, 379-382
proximity and NFC (Near Field Communications), 397
sockets. See sockets
web, 379-382
NetworkStateChanged event, 587
NFC (Near Field Communications), 379
common message protocols, 401
proximity, 397
scenarios, 397-403
NFC API, 400
NFC Data Exchange Format (NDEF), 397
None mode, 122, 328
Normal state, 103
Notification Client Platform. See NCP (Notification Client Platform)
notifications, 226
integrated push notification support, 291-297
periodic notifications, 242
push notifications. See push notifications
raw push notifications, 581-585
tile notifications, cycling, 234-236
toast notifications, 225
notify method, 149
notifyOwner function, 296
null values, 723
numbers, formatting for locale, 642-643
O
OAuth, 324
ObservableDictionary, 44
OData (Open Data), 200, 213, 217-219
ODataServiceExample project, 217-219
OneDrive, 315-321
accessing, 316
downloading from, 319-320
folders, creating, 318
items
deleting, 318
renaming, 318
uploading to, 320-321
OneDrive.com, 161-162
OnEventWritten method, 699
OnHandleOrientationChanged method, 101
OnLaunched method, 699
OnNavigatedTo method, 8, 533, 582
OnNavigationFailed method, 700
OnShareTargetActivated method, 434, 442
Open Data (OData), 200, 213, 217-219
OpenAsync method, 171
OpenID, 324
OpenIfExists option, 169
OpenReadAsync method, 171
OpenSequentialReadAsync method, 171
OpenTransactedWriteAsync method, 171
optimizing performance. See performance optimization
OrderByDate value
(CommonFileQuery), 177
OrderByMusicProperties value
(CommonFileQuery), 177
OrderByName value
(CommonFileQuery), 176
OrderBySearchRank value
(CommonFileQuery), 177
OrderByTitle value
(CommonFileQuery), 176
orientation sensors, 518, 528-529
compass, 519-523
light sensors, 529
simple orientation sensors, 518-519
OrientationHandler class, 101
Other setting (AudioCategory), 565
OutputStreamOverStream class, 173
OverDataLimit, 387
owner, 313
P
package identifiers, 655-656
PackageAndDeployExample project, 668-669
packages on disk, finding, 52-54
packaging
app package and app bundle contents, 654-655
Create App Packages Wizard, 650-654
overview, 649-650
package identifiers, 655-656
package information, 729
Windows Store apps, 52
PageAndGroupManager class, 162, 165
pages (HTML), 143-150
Paginate events, 544
Panel control, 115
panels
StackPanel control, 117-118
VirtualizingPanel control, 118
VirtualizingStackPanel control, 118
parallel threads, 560
parallel workflows, 351
parameters
encrypt parameter, 359
query parameters, 254
request parameters, 256
response parameters, 258
parsing XAML (Extensible Application Markup Language), 140-142
passed arrays, 725
PasswordBox control, 131
Paste method, 157-158
pasting content from Clipboard, 157-158
path and file helper classes, 174-175
path attribute, 88
PathIcon control, 131
PathIO class, 175
PCL (Portable Class Library), 35-41
PeerFinder class, 403
peers (automation), 626-627
PercentageConverter class, 89
PercentFormatter class, 642-643
Performance and Diagnostics dialog box, 705
performance optimization
code analysis, 712-716
logging and tracing, 696-702
overview, 685-686
profiling and performance analysis, 702-712
   CPU sampling, 706-709
   energy consumption, 710-712
   frame rate counters, 702-704
   performance tips, 704-706
   XAML UI responsiveness, 709-710
performance tips, 704-706
periodic notifications, 242
Permissions page, table data operations, 287
PhotoSettings properties
   (CameraCaptureUI), 601
pi, calculating, 560
pickers and cached files, 180-187
PickSingleFileAsync method, 184
Pictures Library, 49
PicturesLibrary folder, 167
Plain Old CLR Object (POCO), 86
Play To Source app, 444-446
Play To Target app, 446-448
PlaybackRate property
   (MediaElement), 594
playing multimedia, 590
   audio settings, 596-597
   background audio, 563-569
   controlling playback, 592-595
   media information, 597
   media markers, 597-598
   MediaElement control appearance, 595-596
   MediaElement control instantiation, 591-592
   MultimediaExample project framework, 590-591
Playlists folder, 167
PlayRequested event, 447-448
PlayToExample project, 443-444
PlayToManager class, 444
PlayToReceiver class, 446-448
PlayToReceiver event handler, 447
POCO (Plain Old CLR Object), 86
pointer events, 484-488
Pointer object, 487
PointerCanceled, 486
PointerCaptureLost, 486
PointerDevice, 481
PointerDeviceType, 481, 494
PointerEntered, 485
PointerExited, 485
PointerMoved, 485
PointerOver state, 103
PointerPoint, 487-488
PointerPressed, 485
PointerReleased, 485
pointers, capturing, 486
pools (thread), 560-562
PopupMenu controls, 131
Portable Class Library (PCL), 35-41
Portable ViewModel, 41
PositionChanged, 506
POST method, 291
POST request (REST), 209
PosterSource property (MediaElement), 595
PowerShell cmdlets, Import-Module, 666
precedence, value, 95
preferred languages, 635-637
PrepareContainerForItemOverride method, 624
presentation logic testing, 351
Pressed state, 103
previewing scanners, 550-551
pricing apps, 669-670
PrimeCheckerExample project
debug windows, 693-694
Edit and Continue, 690
exceptions, 694-696
Just In Time debugging, 691
Just My Code, 688-689
launching debugger, 691-692
native, managed, and script debuggers, 686-688
profiling and performance analysis, 702-712
  CPU sampling, 706-709
  energy consumption, 710-712
  performance tips, 704-706
  XAML UI responsiveness, 709-710
program databases, 692-693
primitives, 720-721
print options, customizing, 538-542
print preview content, generating, 543-545
print task settings, 51
print tasks
  configuring, 534-537
lifecycle events, 535
PrintDocument, 544
printers, 532-534
  asynchronous actions, 536
customizing, 538-542
example app, 532-533
generating content to send to a printer, 546-547
print tasks, configuring, 534-537
PrintTaskSourceRequested-Handler callback, 537-538
  providing content, 542-545
PrintManager, 534
PrintTaskRequested, 533
PrintTaskSourceRequestedHandler callback, 535, 537-538
Private Networks (Client & Server), 49
Product element, 677
ProductLicenses property, 677
profile information (Live Connect), 308-309
profiling and performance analysis, 702-712
  CPU sampling, 706-709
  energy consumption, 710-712
  performance tips, 704-706
  XAML UI responsiveness, 709-710
profiting from apps
  advertisements, 678-683
    AdControl components, 680-683
    pubCenter configuration, 679-680
in-app purchases, 675-678
PackageAndDeployExample project, 668-669
pricing, 669-670
trial mode apps, 670-675
program databases, 692-693
programmatic access to visual states, 109
progress handlers, 381
ProgressBar control, 131
ProgressRing control, 131
projections, examining in WinRT
  components, 20-24
projects
  AccessibilityExample. See
    AccessibilityExample project
  AccessibilityTestProject, 626-627
  AnimationsExample, 97
  AudioBackgroundExample, 563-569
  DataBindingExample, 192-194
  MainPage.xml, 83-84
  DataFormatsExample, 192-194
  DocumentDataExample, 196-197
  FilePickerExample, 183-187
  FlyoutsExample, 133-135
  GlobalizationExample. See globalization
  LayoutsExample, 125-129
  LockScreen Example. See lock screen
tasks
  LoggingHelper, 697-702
  MultimediaExample. See multimedia
  ODataServiceExample, 217-219
  PackageAndDeployExample, 668-669
  PlayToExample, 443-444
  PrimeCheckerExample. See
    PrimeCheckerExample
  QueryPicturesLibrary, 176-180
  RawNotificationExample, 582-585
  ShareTargetExample, 418-421
  SoapServiceExample. See
    SoapServiceExample project
  SocketsGame, 713-716
  TemplatesExample, 112-115
  ThreadPoolExample, 560-562
  VisualStateExample, 101-102
  XamlParsingExample, 140-142
properties, 723-724. See also specific properties
attached properties, 94-95
automation properties, 622-623
data-binding attributes, 87
dependency properties, 91-94
property change notification, 95-97
property change notification, 95-97
PropertyChanged event, 94
ProtectDataAsync method, 334
protocol activation, 463, 467-470
protocols. See specific protocols
providing print content, 542-543
generating content to send to a printer, 546-547
generating print preview content, 543-545
proxies, SoapServiceExample project, 206-207
proximity, 49
NFC (Near Field Communications), 397
scenarios, 397-403
proximity APIs, 403
tap-to-connect scenarios, 403-408
pubCenter configuration, 679-680
PublishBinaryMessage method, 401
publishing apps to Windows Store, 657-665
application certification, 661-665
developer accounts, 657-658
steps, 658-661
tools, 666
PublishMessage method, 401
purchases, in-app, 675-678
push notifications, 249-251
firewalls, 249
raw push notifications, 581-585
registering to receive, 251-253
request parameters, 257
response parameters, 258
sending, 253-259
PushNotificationChannel class, 253
PushNotificationReceived event, 583
PushNotificationTrigger, 582
PUT request (REST), 209

Q
query parameters, 254
QueryInterface, 731
QueryPicturesLibrary project, 176-180
QuickLinks, 439-441

R
RadioButton control, 131
RandomAccessStream class, 173
RandomAccessStreamOverStream class, 173
RandomAccessStreamReference class, 173
RateChanged event, 594
raw push notifications, 581-585
RawNotificationExample project, 582-585
RawNotificationTask, 582
RC2 (Ron’s Code), 338
RCW (Runtime Callable Wrapper), 76
read, 313
read_write, 313
ReadBufferAsync method, 174
ReadBufferAsync method, 174
ReadingChanged event, 521
ReadLinesAsync method, 174
ReadOnlyArray attribute, 26
ReadTextAsync method, 174
Really Simple Syndication (RSS), 199, 219-222
received arrays, 725
refactoring, 366
referencing WinRT, 15-20
RefreshTiles method, 573
Register method, 92, 577
RegisterAttached method, 94
registering
attached properties, 94
dependency properties, 92
Microsoft account logins, 270-272
to receive push notifications, 251-253
Registry, 727-730
Relative Source attribute, 88
RelayCommand, 44
ReloadSimulatorAsync method, 672
remote deployment, running, 56-58
Removable Storage, 49
RemovableDevices folder, 167
Index

RenameAsync method, 171
renaming SkyDrive items, 318
RepeatButton control, 131
ReplaceExisting option, 169
ReplaceWithStreamedFileFromUriAsync method, 171
ReplaceWithStreamFileAsync method, 171
ReportCompleted method, 439
ReportDataRetrieved method, 438-439
ReportError method, 439
ReportStarted method, 438-439
Representational State Transfer. See REST (Representational State Transfer)
request.body, 290
requested theme, 616-618
request.headers, 290
RequestProductPurchaseAsync method, 677
request.query, 290
requests
push notification request parameters, 256
REST (Representational State Transfer), 215-216
request.service.mssql, 290
request.service.push, 290
request.service.tables, 290
request.users, 290
resource dictionaries, 90-91
resource qualification and matching, 637-639
ResourceDictionary class, 87
ResourceLoader, 641
responses
push notification response parameters, 258
response status, 328
response.send, 290
REST (Representational State Transfer), 209-216
compared to SOAP, 210
HTTP status codes and REST correlation, 211
JSON for single category, 214-215
negotiation, 211-213
requests, 215-216
sample result from REST service call, 214
RestorePageData method, 189
.resw extension, 631
RetrievePassword method, 332
rich object graphs, 82
Rich Site Summary. See RSS (Really Simple Syndication)
Rich Text data type, 427
RichTextBox control, 131
RichTextBox control, 131, 196-197
RichTextBoxOverflow control, 132
RightTapped, 493
Rivest, Ron, 338
roaming, 161-162, 387
RoamingSettings, 163-165
Ron’s Code (RC2), 338
RoResolveNamespace, 731
Rotation, 490
routed events, 85
RSS (Really Simple Syndication), 199, 219-222
running Windows Store apps, 54
local deployment, 55-56
remote deployment, 56-58
simulators, 59-61
Runtime Callable Wrapper. See RCW (Runtime Callable Wrapper)

S
SafeNotes project, 334
sampling CPUs, 706-709
Save option (Multilingual App Toolkit), 647
SavedPictures folder, 168
SaveNoteAsync method, 335
SavePageData method, 187
Scalable Vector Graphics (SVG), 143
Scale, 490
scan sources, 549-550
scanners, 547
determining availability, 548-549
example app, 547-548
previewing, 550-551
scan sources, 549-550
scanning, 551-552
settings, 552-556
scanning, 551-552
ScanPicturesAsync method, 552
scheduled tasks (Windows Azure Mobile Services), 297-298
scheduling toasts, 245
schemas, 230
script debugger, 686-688
scripting objects, 290
scripts
data access scripts, 282-287
Update script, 296
ScrollBar control, 132
ScrollViewer control, 122
SDDL (security descriptor definition language), 334
SDK Live Connect, 302-303
search declarations, 51
SearchBox control, 73-75, 132
SearchBoxControlOnQuerySubmitted method, 73
secondary tiles, 236-239
SecondaryTile class, 237
Secure Hash Algorithm (SHA), 343
security, 323-324
authentication, 324-329
documentation, 326
Facebook, 329
multistep (Google), 330-331
options, 328
unlocking password vault, 331-333
certificates, 347
encryption, 333
asymmetric algorithms, 345-347
data protection providers, 333-337
symmetrical encryption. See symmetrical encryption verification, 343
SSO (single sign-on), 326
security descriptor definition language (SDDL), 334
security descriptor (SID), 334
Security Identifier. See SID (Security Identifier)
SelectedScanRegion property, 553
SelectFileOnClick method, 565
selection lists, MVVM (Model-View-ViewModel), 371-373
selling apps
in-app purchases, 675-678
PackageAndDeployExample project, 668-669
pricing, 669-670
trial mode apps, 670-675
SemanticZoom control, 132
SemaphoreSlim class, 699
sending
content to printers, 546-547
push notifications, 253-259
SendNotificationHelper class, 256
sendNotifications function, 297
sensor input, 498
example app, 498-502
geofencing, 510-511
defining fences, 512-514
events, 514-516
managing geofences, 516-517
geolocation, 502-503
Geocoordinate, 508-509
Geolocator, 504-508
simulator location tools, 509-510
motion sensors, 517-518
accelerometer, 526-527
gyrometer, 527-528
inclinometer, 523-526
orientation sensors, 517-518
compass, 519-523
light sensors, 529
orientation sensors, 528-529
simple orientation sensors, 518-519
services (web). See web services
ServicingComplete event, 587
SessionConnected condition, 579
SessionConnected event, 588
SessionDisconnected method, 579
SetApplicationLink function, 430
SetBitmap function, 428
SetButtonStates method, 564
SetData function, 431
SetDataProvider method, 432
SetGeoLocatorReady function, 507
SetMediaStreamSource method, 592
SetMostRecentlyUsedFile method, 186
SetPosition method, 593
SetProperty method, 357
SetSource method, 447, 591
SetStorageItems function, 430
SetText function, 428
SetThumbnailPrefetch method, 177
settings
   RoamingSettings, 163-165
   setting entries, adding, 418-421
   Settings Charm, 417-421
Settings Charm
   overview, 417
   setting entries, adding, 418-421
ShareTargetExample project, 418-421
Settings Flyout template, 417-421
SetTransportControlStates method, 567, 568
SetTrigger method, 578
SetupTroubleshooting method, 109
SetWebLink function, 430
SHA (Secure Hash Algorithm), 343
ShapeConverter class, 126
ShapeModel class, 491
Share Charm
   overview, 421-423
   share process, 422
Share Source app, 423-433
   application sharing, 431-433
   bitmap sharing, 429
   DataRequestEventArgs class, 424
   DataTransferManager class, 424-425
   file sharing, 430
   HTML sharing, 428
   predefined metaproperties available in share data package, 426
   share data types, 427
Share Target app, 433-442
   creating, 434-441
   debugging, 441-442
Share Target app, 433-442
   creating, 434-441
   debugging, 441-442
   share target declarations, 51
ShareTargetActivatedEventArgs class, 436
ShareTargetExample project, 418-421
sharing. See Share Charm
Show method, 608
SID (Security Identifier), 253, 334
sideloading Windows Store apps, 665-666
signatures, verifying, 344
signing up for developer accounts, 657-658
silencing toasts, 247
SilentMode, 328
Simple Object Access Protocol. See SOAP
simple orientation sensors, 517-519
simulator location tools, 509-510
simulators, running, 59-61
single sign-on. See SSO (single sign-on)
SingleUse, 512
sizesof tiles, 226-227
Skrape app, 154
SkrapeDataManager class, 164
SkyDrive. See OneDrive
Slider control, 86, 89, 96, 132
smart tags, 402
SmsReceived event, 588
SOAP, 200-209
  compared to REST, 209
  explained, 200-201
SoapServiceExample project, 200-209
  Add Service Reference dialog, 205
  AsForecastEntry method, 207
  AsWeatherForecast method, 208
design-time data for forecast entry, 202-203
generated proxies, 206-207
GetWeatherForZipCode method, 208
WeatherHelperService class, 207
XAML for forecast entry, 203-204
sockets, 389
  sockets APIs, 379
TCP (Transmission Control Protocol), 392-396
UDP (User Datagram Protocol), 392-396
WebSocket protocol, 389-392
SocketsGame project, 713-716
SoundEffects setting (AudioCategory), 565
Source attribute, 88
SourceChangeRequested event, 447
SourceRequested event, 444-445
SourceSelected event, 445
SpeakContentAsync method, 611-612
SpeechSynthesizer class, 611-612
Storyboard class, 97, 109
stream ciphers, 341
streams, 14-15, 171-174
StreamSocket, 393
StopRecordAsync method, 609-610
storage
  application storage
    app storage hierarchy, 159-160
    composite values, 165-166
    containers, 162
    roaming data, 161-162
    settings, 163-165
  isolated storage, 159
  storage files, 170-171
  storage folders, 167-170
  storage query operations, 176-180
StorageFile class, 452
StorageFile objects, 169
StorageFileQueryResult class, 178
Store Logo, 227
Storyboard class, 97, 109
stream ciphers, 341
streams, 14-15, 171-174
StreamSocket, 393
Stretched state, 107
structures, 722
Style class, 87, 111-112
styles, 111-112
subscription method, 400
subscriptions, 263, 267
support for Windows RT, 3
SupportUsages, 481
SuspensionManager class, 6, 44, 67-69
SVG (Scalable Vector Graphics), 143
SymbolIcon control, 132
symmetrical encryption, 337
  authentication, 342-343
  block ciphers, 337-341
  stream ciphers, 341
syndication, 219-222
SyndicationClient, 219-222
StartRecordToStreamAsync target type, 608
StartTime, 512
states
  state management, navigation and, 69-73
VSM (Visual State Manager)
  explained, 105-106
  programmatic access to visual states, 109
StopRecordAsync method, 609-610
SynthesizeSsmIToStreamAsync method, 612
SynthesizeTextToStreamAsync method, 612
system events, 587-588
SystemCondition class, 578
SystemTrigger, 587-588

T

table access methods, 282
table operation scripting objects, 284
table proxy, 281
tables
  table access methods, 282
  table data operations, 287
  table operation scripting objects, 284
  table proxy, 281
Windows Azure Mobile Services, 274-278
Tapped, 493
tap-to-connect scenarios, 403-408
TargetFileRequested, 459-460
Task class, 560
TaskRunCompleted handler, 577
tasks
  background. See background tasks
  scheduled (Windows Azure Mobile Services), 297-298
TCP (Transmission Control Protocol), 389, 392-396
template assets, 44-45
TemplateBinding keyword, 138
templates
  adding text to (tiles), 231
  ControlTemplate class, 112
  DataTemplate class, 112
  explained, 112
  Grid App template, 5
  live tiles, 229
  TemplatesExample project, 112-115
toasts, 244
Windows Store apps, 32
  Blank app, 32
  class library, 35
  coded UI tests, 43

Grid app, 33
Hub app, 33-34
PCL (Portable Class Library), 35-41
split app, 34
template assets, 44-45
unit test library, 42-43
WinRT components, 42
TemplatesExample project, 112-115
testing with WACK (Windows App Certification Kit), 662-664
text
  adding to templates (tiles), 231
text-to-speech support
    Narrator, 623
    SpeechSynthesizer class, 610-612
Text data type, 427
TextBlock control, 132
TextBox control, 132
text-to-speech support
  Narrator, 623
  SpeechSynthesizer class, 610-612
themes
  high contrast, 618-619
  requested theme, 616-618
thread pool, 560-562
ThreadPoolExample project, 560-562
threads
  parallel threads, 560
  thread pool, 560-562
tile helper, 231
tile notifications, cycling, 234-236
TileExplorer project, 233
tiles, 225-227
  badges, 239-241
  cycling tile notifications, 234-236
  default tiles, 227-229
  live tiles, 229-234
  periodic notifications, 242
  schemas, 230
  secondary tiles, 236-239
  sizes, 226-227
TileTemplateType, 230
TileUpdateManager, 230, 232
TileUpdater method, 235
time, formatting for locale, 642-643
Index

TimelineMarker objects, 597-598
TimePicker control, 132
timer, 578
TimeTrigger, 578
TimeZoneChange event, 588
Title property (Share DataPackage), 426
toast notifications, 225
toasts, 242-248
desktop applications, 248
expiration, 245
long-duration toasts, 248
scheduling, 245
silencing, 247
standard toast, 243
templates, 244
ToggleButton control, 132
toggle requests, 254
tooling support, MVVM (Model-View-ViewModel), 366
ToolTip control, 132
TouchCapabilities class, 482
tracing, 696-702
TransformToString method, 195
transitions, VSM (Visual State Manager), 106-107
Translate option (Multilingual App Toolkit), 646
TranslateTransform, 491
Translation, 490
Translation Languages dialog box, 647
Transmission Control Protocol. See TCP (Transmission Control Protocol)
TransportControlsButtonPressed handler, 568
trial mode apps, 670-675
triggers
ControlChannelTrigger, 585-586
PushNotificationTrigger, 582
SystemTrigger, 587-588
TimeTrigger, 578
Triple Data Encryption Algorithm (3DES), 338
troubleshooting HTTP status codes and REST correlation, 211
TryGetClickablePoint method, 626
T-SQL data types, 277
TypeConverter class, 82

U
UDP (User Datagram Protocol), 389, 392-396
UI design patterns, 350-351
models, 351-352
MVC (Model-View-Controller), 353-354
MVP (Model-View-Presenter), 354-355
MVVM (Model-View-ViewModel), 355-356
views, 352-353
UI thread, accessing, 369-370
UIElement class, 87, 485
UnauthorizedAccessException, 506, 606
Uniform mode (ViewBox), 122
UniformToFill mode (ViewBox), 122
unit testing
MVVM (Model-View-ViewModel), 365
unit test library templates, 42-43
unlocking password vault, 331-333
Update script, 296
uploading to OneDrive (SkyDrive), 320-321
uploads, 562-563
URLs, 13
UriToString method, 23
URLs, 381
USB HIB, 481
UseCorporateNetwork, 328
UseHttpPost, 328
User Datagram Protocol. See UDP (User Datagram Protocol)
UserAway event, 588
UserInformation.SetAccountPictureAsync method, 471
UserNotPresent method, 579
UserPresent event, 588
UserPresent method, 579
users, authenticating, 272-274
UseTitle, 328
UsingTransport method, 586
UTC (Coordinated Universal Time), 12
utilities. See specific utilities
V
validation, MVVM (Model-View-ViewModel), 375-377
value precedence, 95
values
    composite values, 165-166
    value precedence, 95
VariableSizedWrapGrid control, 119-120
Verb entry, 471
verification
    encryption, 343
        hash algorithms, 343-344
        MAC (Message Authentication Code), 344-345
    signatures, 344
video
    acquiring, 598-599
        CameraCaptureUI, 600-603
declaring application capabilities, 599-600
    MediaCapture, 604-610
    playing, 590
        audio settings, 596-597
        controlling playback, 592-595
        media information, 597
        media markers, 597-598
    MediaElement control appearance, 595-596
    MediaElement control instantiation, 591-592
    MultimediaExample project framework, 590-591
VideoDeviceController property
    (MediaCapture class), 607
VideoDeviceId (MediaCapture class), 604
Videos Library, 49
VideoSettings properties
    (CameraCaptureUI), 601
VideosLibrary folder, 168
ViewBox control
    LayoutsExample project, 125-129
    modes, 122
ViewModel, 126, 246, 643-644
    commands, 361
MVVM (Model-View-ViewModel), 356-362

ViewModel locators, 360
views
    FlipView control, 124-125
    GridView control, 123
    ListView control, 124
    ScrollViewer control, 122
    UI design patterns, 352-353
ViewBox control
    LayoutsExample project, 125-129
    modes, 122
WebView control, 130, 143-150
VirtualizingPanel control, 118
VirtualizingStackPanel control, 118
VirtualKey enumeration, 497
Visual State Manager. See VSM (Visual State Manager)
Visual Studio 2012, 29
visual tree (XAML), 83-85
VisualStateExample project, 101-102
VoiceInformation class, 611
Volume property (MediaElement), 596
VSM (Visual State Manager)
custom VSMs, 109-110
explained, 100
groups, 103-105
states
    explained, 105-106
    programmatic access to visual states, 109
transitions, 106-107
VisualStateExample project, 101-102
workflow, 107-108
.vspx extension, 706

W
WACK (Windows App Certification Kit), 662-664
WasKeyDown property, 496
Watch windows (debugger), 694
WeatherHelperService class, 207
web authentication brokers, 326
Web Graphics Library (WebGL), 143
web links, sharing, 431
web services
    OData (Open Data), 217-219
overview, 199-200
REST (Representational State Transfer), 209-216
   compared to SOAP, 209
HTTP status codes and REST correlation, 211
JSON for single category, 214-215
negotiation, 211-213
requests, 215-216
sample result from REST service call, 214
SOAP, 200-209
   compared to REST, 209
   explained, 200-201
SoapServiceExample project, 201-209
syndication, 219-222
Webcam, 49
WebGL (Web Graphics Library), 143
WebRequest object, 36
WebSocket protocol, 389-392
WebSocketKeepAlive task, 585
WebView control, 132, 143-150
Wide 310x150 Logo, 227
Wi-Fi (Wireless Fidelity), 384
Windows 8, Developer Preview version, 29
Windows App Certification Kit (WACK), 662-664
Windows Azure Mobile Services, 262-267
   authentication, 269-270
      authenticating users, 272-274
      registration and configuration for Microsoft account logins, 270-272
connecting apps to, 267-268
custom APIs, 289-291
data storage, 274
   authorization, 287-289
data access scripts, 282-287
data in client apps, 278-282
      managing data tables, 274-278
deployment tiers, 298-300
features, 262-263
Git source control integration, 286
integration push notification support, 291-297
scheduled tasks, 297-298
subscriptions, 263, 267
Windows Charms
   Devices Charm
      overview, 442-443
      Play To Source app, 444-446
      Play To Target app, 446-448
      Play To Example project, 442-448
   overview, 415-417
   Settings Charm
      overview, 417
      setting entries, adding, 418-421
      ShareTargetExample project, 418-421
   Share Charm
      overview, 421-423
      share process, 422
      Share Source app, 423-433
      Share Target app, 433-442
Windows Data Protection API (DPAPI), 333
Windows Intune, 666
Windows Media Player, 444
Windows Phone 7, 29
Windows print functionality, 533
Windows Push Notification Service. See WNS (Windows Push Notification Service)
Windows RT, 2-3
Windows Store apps, 4-5, 30-31
   app manifest, 45-46
      application UI, 46-48
      capabilities, 48-50
      content URLs, 52
      declarations, 50-51
      packaging, 52
   application lifecycle, 61-67
collections, 384-385
creating, 5-8
finding your package on disk, 52-54
making money from, 667-668
   advertisements, 678-683
   in-app purchases, 675-678
   PackageAndDeployExample project, 668-669
pricing apps, 669-670
trial mode apps, 670-675
package information, 729
publishing, 657-665
  application certification, 661-665
developer accounts, 657-658
steps, 658-661
running, 54
  local deployment, 55-56
  remote deployment, 56-58
  simulators, 59-61
sideloading, 665-666
templates, 32
  Blank app, 32
class library, 35
coded UI tests, 43
Grid app, 33
Hub app, 33-34
PCL (Portable Class Library), 35-41
split app, 34
template assets, 44-45
unit test library, 42-43
WinRT components, 42
WinRT RuntimeBufferExtension class, 171
WindowsStoreProxy folder, 672
winmd extension, 719
WinRT, 1-4
  adding references, 17
asynchronous functions, 24-26
components, 20-24
language options, 3
.NET Framework
  buffers, 14-15
  fundamental types, 9-10
  mapped types, 10-13
  strings, 14-15
referencing from desktop applications, 15-20
templates, 42
WinRT internals, 725-731
wireless wide area network (WWAN), 384
WithExpiration method, 236
wizards, Create App Packages Wizard, 650-654
WLAN (wireless local area connections), 384
WNS (Windows Push Notification Service), 249, 292, 293
workflow, VSM (Visual State Manager), 107-108
WrapGrid control, 119
WriteBufferAsync method, 174
WriteBytesAsync method, 175
WriteEvent method, 698
WriteLinesAsync method, 175
WriteTextAsync method, 175
WWAN (wireless wide area network), 384

X
x86 chips, 2
XAML (Extensible Application Markup Language)
animations, applying to controls, 97-100
data-binding
  attached properties, 94-95
  attributes, 87
classes, 86-87
  DataBindingExample, 89-91
dependency properties, 91-94
explained, 85-86
  property change notification, 95-97
  value precedence, 95
events, 85
explained, 81-83
localization, 639-642
parsing, 140-142
rich object graphs, 82
TypeConverter class, 82
visual tree, 83-85
VSM (Visual State Manager)
explained, 100
groups, 103-105
states, 105-106
transitions, 106-107
VisualStateExample project, 101-102
XAML UI responsiveness, 709-710
XamlParsingExample project, 140-142
.xlf (Localization Interchange File Format) files, 645
XML (Extensible Markup Language), 81
XML badges, 241
XmlDocument, 230
XOR (exclusive or), 338
XSL Formatting Objects, 195
XSLT transformation, 195

Z
zip archives, 187-191
ZipArchive class, 188
ZipFileExtensions class, 189