



Advanced Windows Store App Development Using HTML5 and JavaScript



Exam Ref 70-482

Roberto Brunetti
Vanni Boncinelli

Exam Ref 70-482



Prepare for Microsoft Exam 70-482—and help demonstrate your real-world mastery of building Windows Store apps with HTML5 and JavaScript. Designed for experienced developers ready to advance their status, *Exam Ref* focuses on the critical-thinking and decision-making acumen needed for success at the MCSD level.

Focus on the expertise measured by these objectives:

- Develop Windows Store apps
- Discover and interact with devices
- Program user interaction
- Enhance the user interface
- Manage data and security
- Prepare for a solution deployment

This Microsoft *Exam Ref*

- Organizes its coverage by exam objectives.
- Features strategic, what-if scenarios to challenge you.
- Includes a 15% exam discount from Microsoft.
Offer expires 12/31/2015. Details inside.

microsoft.com/mspress

ISBN: 978-0-7356-7680-0



U.S.A. \$39.99

Canada \$41.99

[Recommended]

Certification/Windows Store Apps

www.it-ebooks.info

Advanced Windows Store App Development Using HTML5 and JavaScript

About the Exam

Exam 70-482 is one of three Microsoft exams focused on the skills and knowledge necessary to develop Windows Store apps with HTML5 and JavaScript.

About Microsoft Certification

Passing this exam earns you credit toward a **Microsoft Certified Solutions Developer (MCSD)** certification that demonstrates your expertise in designing and developing fast and fluid Windows 8 apps.

Exams 70-480, 70-481, and 70-482 are required for MCSD: Windows Store Apps Using HTML5 certification.

See full details at:
microsoft.com/learning/certification

About the Authors

Roberto Brunetti is a consultant, trainer, and author with experience in enterprise applications. He co-founded a company that provides high-value content and consulting services to professional developers.

Vanni Boncinelli is a consultant and author on Microsoft .NET technologies. He works with Roberto Brunetti's team to provide high-value content and consulting services to professional developers.





Exam Ref 70-482: Advanced Windows Store App Development Using HTML5 and JavaScript

**Roberto Brunetti
Vanni Boncinelli**

Published with the authorization of Microsoft Corporation by:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, California 95472

Copyright © 2013 by Roberto Brunetti and Vanni Boncinelli

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

ISBN: 978-0-7356-7680-0

1 2 3 4 5 6 7 8 9 QG 8 7 6 5 4 3

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at mspinput@microsoft.com. Please tell us what you think of this book at <http://www.microsoft.com/learning/booksurvey>.

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

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

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

Acquisitions Editor: Jeff Riley

Developmental Editor: Kim Lindros

Production Editor: Melanie Yarbrough

Editorial Production: Box Twelve Communications

Technical Reviewer: Luca Regnicoli

Copyeditor: Susan Hobbs

Indexer: Angie Martin

Cover Design: Twist Creative • Seattle

Cover Composition: Ellie Volckhausen

Illustrator: Rebecca Demarest

This book is dedicated to my parents.

— ROBERTO BRUNETTI

This book is dedicated to my family.

— VANNI BONCINELLI

Contents at a glance

<i>Introduction</i>	xv
<i>Preparing for the exam</i>	xvii
CHAPTER 1 Develop Windows Store apps	1
CHAPTER 2 Discover and interact with devices	57
CHAPTER 3 Program user interaction	125
CHAPTER 4 Enhance the user interface	181
CHAPTER 5 Manage data and security	247
CHAPTER 6 Prepare for a solution deployment	307
<i>Index</i>	389

Contents

Introduction	xv
<i>Microsoft certifications</i>	xv
<i>Acknowledgments</i>	xv
<i>Errata & book support</i>	xvi
<i>We want to hear from you</i>	xvi
<i>Stay in touch</i>	xvi
Preparing for the exam	xvii
Chapter 1 Develop Windows Store apps	1
Objective 1.1: Create background tasks	1
Creating a background task	2
Declaring background task usage	5
Enumerating registered tasks	7
Using deferrals with tasks	8
Objective summary	9
Objective review	9
Objective 1.2: Consume background tasks	10
Understanding task triggers and conditions	10
Progressing through and completing background tasks	12
Understanding task constraints	15
Cancelling a task	16
Updating a background task	19
Debugging tasks	20
Understanding task usage	22
Transferring data in the background	22

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Keeping communication channels open	27
Objective summary	37
Objective review	37
Objective 1.3: Integrate WinMD components into a solution	38
Understanding the Windows Runtime and WinMD	38
Consuming a native WinMD library	40
Creating a WinMD library	47
Objective summary	50
Objective review	51
Chapter summary	51
Answers.....	52
Objective 1.1: Thought experiment	52
Objective 1.1: Review	52
Objective 1.2: Thought experiment	53
Objective 1.2: Review	53
Objective 1.3: Thought experiment	54
Objective 1.3: Review	54
Chapter 2 Discover and interact with devices	57
Objective 2.1: Capture media with the camera and microphone.....	57
Using <i>CameraCaptureUI</i> to capture pictures or video	58
Using <i>MediaCapture</i> to capture pictures, video, or audio	67
Objective summary	78
Objective review	78
Objective 2.2: Get data from sensors	79
Understanding sensors and location data in the Windows Runtime	79
Accessing sensors from a Windows Store app	80
Determining the user's location	96
Objective summary	104
Objective review	105
Objective 2.3: Enumerate and discover device capabilities.....	105
Enumerating devices	106
Using the <i>DeviceWatcher</i> class to be notified of changes to the device collection	112

Enumerating Plug and Play (PnP) devices	116
Objective summary	118
Objective review	119
Chapter summary	119
Answers.....	121
Objective 2.1: Thought experiment	121
Objective 2.1: Review	121
Objective 2.2: Thought experiment	122
Objective 2.2: Review	122
Objective 2.3: Thought experiment	123
Objective 2.3: Review	124
Chapter 3 Program user interaction	125
Objective 3.1: Implement printing by using contracts and charms.....	125
Registering a Windows Store app for the Print contract	126
Handling <i>PrintTask</i> events	131
Creating the user interface	132
Creating a custom print template	133
Understanding the print task options	136
Choosing options to display in the preview window	139
Reacting to print option changes	140
Implementing in-app printing	142
Objective summary	143
Objective review	143
Objective 3.2: Implement Play To by using contracts and charms	144
Introducing the Play To contract	144
Testing sample code using Windows Media Player on a different machine	147
Implementing a Play To source application	149
Registering your app as a Play To receiver	155
Objective summary	161
Objective review	162
Objective 3.3: Notify users by using Windows Push Notification Service (WNS)	163
Requesting and creating a notification channel	163

Sending a notification to the client	165
Objective summary	173
Objective review	173
Chapter summary	174
Answers.....	175
Objective 3.1: Thought experiment	175
Objective 3.1: Review	175
Objective 3.2: Thought experiment	176
Objective 3.2: Review	177
Objective 3.3: Thought experiment	178
Objective 3.3: Review	178
Chapter 4 Enhance the user interface	181
Objective 4.1: Design for and implement UI responsiveness	181
Choosing an asynchronous strategy	182
Implementing promises and handling errors	183
Cancelling promises	187
Creating your own promises	188
Using web workers	190
Objective summary	194
Objective review	195
Objective 4.2: Implement animations and transitions	195
Using CSS3 transitions	196
Creating and customizing animations	203
Using the animation library	206
Animating with the HTML5 <i>canvas</i> element	211
Objective summary	212
Objective review	213
Objective 4.3: Create custom controls.	213
Understanding how existing controls work	214
Creating a custom control	218
Extending controls	222
Objective summary	226
Objective review	227

Objective 4.4: Design apps for globalization and localization	228
Planning for globalization	228
Localizing your app	231
Localizing your manifest	236
Using the Multilingual App Toolkit	238
Objective summary	239
Objective review	239
Chapter summary	240
Answers	241
Objective 4.1: Thought experiment	241
Objective 4.1: Review	241
Objective 4.2: Thought experiment	242
Objective 4.2: Review	243
Objective 4.3: Thought experiment	244
Objective 4.3: Review	244
Objective 4.4: Thought experiment	245
Objective 4.4: Review	245
Chapter 5 Manage data and security	247
Objective 5.1: Design and implement data caching	247
Understanding application and user data	247
Caching application data	248
Understanding Microsoft rules for using roaming profiles with Windows Store apps	259
Caching user data	260
Objective summary	262
Objective review	263
Objective 5.2: Save and retrieve files from the file system	263
Using file pickers to save and retrieve files	264
Accessing files and data programmatically	270
Working with files, folders, and streams	272
Setting file extensions and associations	274
Compressing files to save space	276
Objective summary	277
Objective review	278

Objective 5.3: Secure application data	278
Introducing the <i>Windows.Security.Cryptography</i> namespaces	279
Using hash algorithms	279
Generating random numbers and data	283
Encrypting messages with MAC algorithms	284
Using digital signatures	288
Enrolling and requesting certificates	290
Protecting your data with the <i>DataProtectionProvider</i> class	296
Objective summary	300
Objective review	300
Chapter summary	301
Answers.....	302
Objective 5.1: Thought experiment	302
Objective 5.1: Review	302
Objective 5.2: Thought experiment	303
Objective 5.2: Review	303
Objective 5.3: Thought experiment	304
Objective 5.3: Review	304
Chapter 6 Prepare for a solution deployment	307
Objective 6.1: Design and implement trial functionality in an app	307
Choosing the right business model for your app	308
Exploring the licensing state of your app	310
Using custom license information	316
Purchasing an app	318
Handling errors	320
Setting up in-app purchases	322
Retrieving and validating the receipts for your purchases	327
Objective summary	329
Objective review	329
Objective 6.2: Design for error handling	330
Designing the app so that errors and exceptions never reach the user	331
Handling promise errors	335
Handling device capability errors	339

Objective summary	343
Objective review	344
Objective 6.3: Design and implement a test strategy	344
Understanding functional testing vs. unit testing	345
Implementing a test project for a Windows Store app	348
Objective summary	355
Objective review	356
Objective 6.4: Design a diagnostics and monitoring strategy	357
Profiling a Windows Store app and collecting performance counters	357
Using JavaScript analysis tools	365
Logging events in a Windows Store app written in JavaScript	371
Using the Windows Store reports to improve the quality of your app	377
Objective summary	380
Objective review	381
Chapter summary	382
Answers	383
Objective 6.1: Thought experiment	383
Objective 6.1: Review	383
Objective 6.2: Thought experiment	384
Objective 6.2: Review	384
Objective 6.3: Thought experiment	385
Objective 6.3: Review	385
Objective 6.4: Thought experiment	386
Objective 6.4: Review	387
<i>Index</i>	389

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Introduction

The Microsoft 70-482 certification exam tests your knowledge of Windows Store application development using HTML5 and JavaScript. Readers are assumed to be Windows Store app developers with deep knowledge of the Windows Runtime architecture, the application life cycle managed by the system (including suspend, termination, resume, and launch), the Visual Studio 2012 project structure, the application manifest, app deployment, and Windows Store requirements. The reader must have also a strong background in HTML5 and JavaScript.

This book covers every exam objective, but it does not cover every exam question. Only the Microsoft exam team has access to the exam questions themselves and Microsoft regularly adds new questions to the exam, making it impossible to cover specific questions. You should consider this book a supplement to your relevant real-world experience and other study materials. If you encounter a topic in this book that you do not feel completely comfortable with, use the links you'll find in text to find more information and take the time to research and study the topic. Great information is available on MSDN, TechNet, and in blogs and forums.

Microsoft certifications

Microsoft certifications distinguish you by proving your command of a broad set of skills and experience with current Microsoft products and technologies. The exams and corresponding certifications are developed to validate your mastery of critical competencies as you design and develop, or implement and support, solutions with Microsoft products and technologies both on-premise and in the cloud. Certification brings a variety of benefits to the individual and to employers and organizations.

MORE INFO ALL MICROSOFT CERTIFICATIONS

For information about Microsoft certifications, including a full list of available certifications, go to <http://www.microsoft.com/learning/en/us/certification/cert-default.aspx>.

Acknowledgments

I'd like to thank Vanni for his side-by-side work. He has shared with me all the intricacies of writing a book with this level of detail.

— Roberto

I'd like to thank Roberto, for teaching me everything I know today about software development, and Marika, for her support and infinite patience during the writing of this book.

— Vanni

Roberto and Vanni want to thank all the people who made this book possible. In particular, we thank Kim Lindros, for her exceptional support throughout the editing process of this book; Jeff Riley, for giving us this opportunity; and Russell Jones, for introducing our team to Jeff.

Special thanks to Wouter de Kort for providing the Chapter 4 content.

Errata & book support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at oreilly.com:

<http://aka.ms/ER70-482/errata>

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

If you need additional support, email Microsoft Press Book Support at mspinput@microsoft.com.

Please note that product support for Microsoft software is not offered through the addresses above.

We want to hear from you

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

<http://www.microsoft.com/learning/booksurvey>

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

Stay in touch

Let's keep the conversation going! We're on Twitter: <http://twitter.com/MicrosoftPress>.

Preparing for the exam

Microsoft certification exams are a great way to build your resume and let the world know about your level of expertise. Certification exams validate your on-the-job experience and product knowledge. While there is no substitution for on-the-job experience, preparation through study and hands-on practice can help you prepare for the exam. We recommend that you round out your exam preparation plan by using a combination of available study materials and courses. For example, you might use this *Exam Ref* and another study guide for your "at home" preparation, and take a Microsoft Official Curriculum course for the classroom experience. Choose the combination that you think works best for you.

Note that this *Exam Ref* is based on publically available information about the exam and the author's experience. To safeguard the integrity of the exam, authors do not have access to the live exam.

CHAPTER 1

Develop Windows Store apps

In this chapter, you learn how to create background tasks, implement the appropriate interfaces, and consume tasks using timing and system triggers. You also find out how to request lock screen access and create download and upload operations using background transferring for Windows Store applications written in Hypertext Markup Language (HTML)/JavaScript (formerly called Windows Store apps using JavaScript). The last part of the chapter is dedicated to creating and consuming Windows Metadata (WinMD) components.

IMPORTANT METHODS CAPITALIZATION

Throughout the code samples in this book, the syntax of the Windows Runtime (WinRT) library methods and events follow the traditional JavaScript syntax, while in the text, the same methods and events are initial capped. This is because the method definitions in the library are initial capped (*SetTrigger*, for example), but thanks to the WinRT language projection feature, developers can use the syntax of their chosen language to invoke them (*setTrigger*, for example). Language projection is discussed in Objective 1.3, "Integrate WinMD components into a solution," later in this chapter. Classes and namespaces are always initial capped.

Objectives in this chapter:

- Objective 1.1: Create background tasks
- Objective 1.2: Consume background tasks
- Objective 1.3: Integrate WinMD components into a solution

Objective 1.1: Create background tasks

Microsoft Windows 8 changes the way applications run. Windows Store application life-cycle management of the Windows Runtime is different from previous versions of Windows: only one application (or two in snapped view) can run in the foreground at a time. The system can suspend or terminate other applications from the Windows Runtime. This behavior forces the developer to use different techniques to implement some form of background work—for example, to download a file or perform tile updates.

This section covers how to implement a background task using the provided classes and interfaces, and how to code a simple task.

This objective covers how to:

- Implement the *Windows.applicationmodel.background* classes
- Implement *WebUIBackgroundTaskInstance*
- Create a background task to manage and preserve resources
- Create a background task to get notifications for an app
- Register the background task by using the *BackgroundTaskBuilder* class

Creating a background task

In Windows Store apps, when users work on an app in the foreground, background apps cannot interact directly with them. In fact, due to the architecture of Windows 8 and because of the application life-cycle management of Windows Store apps, only the foreground app has the focus and is in the Running state; the user can choose two applications in the foreground using the snapped view.

All the other background apps can be suspended, and even terminated, by the Windows Runtime. A suspended app cannot execute code, consume CPU cycles or network resources, or perform any disk activity such as reading or writing files.

You can define a task that runs in the background, however, even in a separate process from the owner app, and you can define background actions. When these actions need to alert users about their outcomes, they can use a toast.

A background task can execute code even when the corresponding app is suspended, but it runs in an environment that is restricted and resource-managed. Moreover, background tasks receive only a limited amount of system resources.

You should use a background task to execute small pieces of code that require no user interaction. You can also use a background task to communicate with other apps via instant messaging, email, or Voice over Internet Protocol (VoIP). Avoid using a background task to execute complex business logic or calculations because the amount of system resources available to background apps is limited. Complex background workloads consume battery power as well, reducing the overall efficiency and responsiveness of the system.

To create a background task, you have to create a new JavaScript file with a function that runs in the background when the task is triggered. The name of the file is used to launch the background task.

The function uses the *current* property of the *WebUIBackgroundTaskInstance* object to get a reference to the background task, and it contains the *doWork* function that represents the code to be run when the task is triggered. See Listing 1-1.

LISTING 1-1 JavaScript function skeleton for a background task

```
(function () {
    "use strict";

    //
    // Get a reference to the task instance.
    //
    var bgTaskInstance = Windows.UI.WebUI.WebUIBackgroundTaskInstance.current;

    //
    // Real work.
    //
    function doWork() {
        // Call the close function when you have done.
        close();
    }

    doWork();
})();
```

Remember to call the *close* function at the end of the worker function. If the background task does not call this method, the task continues to run and consume battery, CPU, and memory, even if the code has reached its end.

Then you have to assign the event that will fire the task. When the event occurs, the operating system calls the defined *doWork* function. You can associate the event, called a *trigger*, via the *SystemTrigger* or the *MaintenanceTrigger* class.

The code is straightforward. Using an instance of the *BackgroundTaskBuilder* object, associate the name of the task and its entry point by using the path to the JavaScript file. The entry point represents the relative path to the JavaScript file, as shown in the following code excerpt:

Sample of JavaScript code

```
var builder = new Windows.ApplicationModel.Background.BackgroundTaskBuilder();
builder.name = "BikeGPS";
builder.taskEntryPoint = "js\\BikeBackgroundTask.js";
```

Then you must create the trigger to let the system know when to start the background task:

```
var trigger = new Windows.ApplicationModel.Background.SystemTrigger(
    Windows.ApplicationModel.Background.SystemTriggerType.timeZoneChange, false);
builder.setTrigger(trigger);
```

 **EXAM TIP**

The *SystemTrigger* object accepts two parameters in its constructor. The first parameter of the trigger is the type of system event associated with the background task; the second, called *oneShot*, tells the Windows Runtime to start the task only once or every time the event occurs.

The complete enumeration, which is defined by the *SystemTriggerType* enum, is shown in Listing 1-2.

LISTING 1-2 Types of system triggers

```
// Summary:  
// Specifies the system events that can be used to trigger a background task.  
[Version(100794368)]  
public enum SystemTriggerType  
{  
    // Summary:  
    //     Not a valid trigger type.  
    Invalid = 0,  
    //  
    // Summary:  
    // The background task is triggered when a new SMS message is received by an  
    // installed mobile broadband device.  
    SmsReceived = 1,  
    //  
    // Summary:  
    // The background task is triggered when the user becomes present. An app must  
    // be placed on the lock screen before it can successfully register background  
    // tasks using this trigger type.  
    UserPresent = 2,  
    //  
    // Summary:  
    // The background task is triggered when the user becomes absent. An app must  
    // be placed on the lock screen before it can successfully register background  
    // tasks using this trigger type.  
    UserAway = 3,  
    //  
    // Summary:  
    // The background task is triggered when a network change occurs, such as a  
    // change in cost or connectivity.  
    NetworkStateChange = 4,  
    //  
    // Summary:  
    // The background task is triggered when a control channel is reset. An app must  
    // be placed on the lock screen before it can successfully register background  
    // tasks using this trigger type.  
    ControlChannelReset = 5,  
    //  
    // Summary:  
    // The background task is triggered when the Internet becomes available.  
    InternetAvailable = 6,  
    //  
    // Summary:  
    // The background task is triggered when the session is connected. An app must  
    // be placed on the lock screen before it can successfully register background  
    // tasks using this trigger type.  
    SessionConnected = 7,  
    //
```

```

// Summary:
// The background task is triggered when the system has finished updating an
// app.
ServicingComplete = 8,
//
// Summary:
// The background task is triggered when a tile is added to the lock screen.
LockScreenApplicationAdded = 9,
//
// Summary:
// The background task is triggered when a tile is removed from the lock screen.
LockScreenApplicationRemoved = 10,
//
// Summary:
// The background task is triggered when the time zone changes on the device
// (for example, when the system adjusts the clock for daylight saving time).
TimeZoneChange = 11,
//
// Summary:
// The background task is triggered when the Microsoft account connected to
// the account changes.
OnlineIdConnectedStateChange = 12,
}

```

You can also add conditions that are verified by the system before starting the background task. The *BackgroundTaskBuilder* object exposes the *AddCondition* function to add a single condition, as shown in the following code sample. You can call it multiple times to add different conditions.

```
builder.addCondition(new Windows.ApplicationModel.Background.SystemCondition(
    Windows.ApplicationModel.Background.SystemConditionType.internetAvailable))
```

The last line of code needed is the registration of the defined task:

```
var task = builder.register();
```

Declaring background task usage

An application that registers a background task needs to declare the feature in the application manifest as an extension, as well as the events that will trigger the task. If you forget these steps, the registration will fail. There is no *<Extensions>* section in the application manifest of the standard template by default, so you need to insert it as a child of the *Application* tag.

Listing 1-3 shows the application manifest for the sample task implemented by Listing 1-2. The *<Extensions>* section is shown in bold.

LISTING 1-3 Application manifest

```
<?xml version="1.0" encoding="utf-8"?>
<Package xmlns="http://schemas.microsoft.com/appx/2010/manifest">
    <Identity Name="e00b2bde-0697-4e6b-876b-1d611365485f"
        Publisher="CN=Roberto"
        Version="1.0.0.0" />
    <Properties>
        <DisplayName>BikeApp</DisplayName>
        <PublisherDisplayName>Roberto</PublisherDisplayName>
        <Logo>Assets\StoreLogo.png</Logo>
    </Properties>
    <Prerequisites>
        <OSMinVersion>6.2.1</OSMinVersion>
        <OSMaxVersionTested>6.2.1</OSMaxVersionTested>
    </Prerequisites>
    <Resources>
        <Resource Language="x-generate"/>
    </Resources>
    <Applications>
        <Application Id="App"
            Executable="$targetnametoken$.exe"
            EntryPoint="BikeApp.App">
            <VisualElements
                DisplayName="BikeApp"
                Logo="Assets\Logo.png"
                SmallLogo="Assets\SmallLogo.png"
                Description="BikeApp"
                ForegroundText="light"
                BackgroundColor="#464646">
                <DefaultTile ShowName="allLogos" />
                <SplashScreen Image="Assets\SplashScreen.png" />
            </VisualElements>
            <Extensions>
                <Extension Category="windows.backgroundTasks"
                    EntryPoint="js\BikeBackgroundTask.js">
                    <BackgroundTasks>
                        <Task Type="systemEvent" />
                    </BackgroundTasks>
                </Extension>
            </Extensions>
        </Application>
    </Applications>
    <Capabilities>
        <Capability Name="internetClient" />
    </Capabilities>
</Package>
```

You have to add as many task elements as needed by the application. For example, if the application uses a system event and a push notification event, you have to add the following XML node to the *BackgroundTasks* element:

```
<BackgroundTasks>
    <Task Type="systemEvent" />
    <Task Type="pushNotification" />
</BackgroundTasks>
```

You can also use the Microsoft Visual Studio App Manifest Designer to add (or remove) a background task declaration. Figure 1-1 shows the same declaration in the designer.

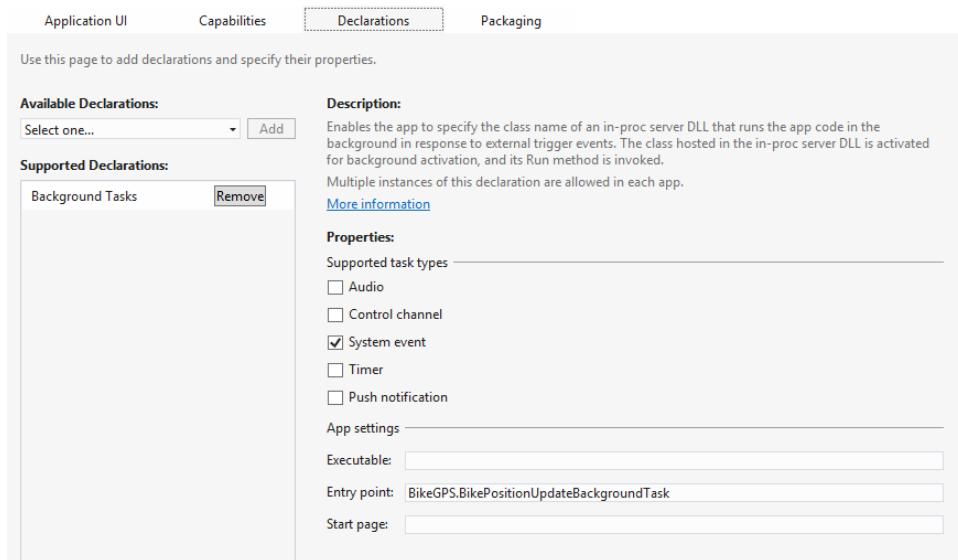


FIGURE 1-1 Background task declaration in Visual Studio App Manifest Designer

Enumerating registered tasks

Be sure to register the task just once in your application. If you forget to check the presence of the task, you risk registering and executing the same task many times.

To check whether a task is registered, you can iterate all the registered tasks using the *BackgroundTaskRegistration* object and checking for the *Value* property that represents the task that, in turns, exposes the *Name* property, as follows:

Sample of JavaScript code

```
var taskName = "bikePositionUpdate";
var taskRegistered = false;

var background = Windows.ApplicationModel.Background;
var iter = background.BackgroundTaskRegistration.allTasks.first();

while (iter.hasCurrent) {
    var task = iter.current.value;

    if (task.name === taskName) {
        taskRegistered = true;
        break;
    }
    iter.moveToNext();
}
```

Using deferrals with tasks

If the code for the *doWork* function is asynchronous, the background task needs to use a deferral (the same techniques as the suspend method). In this case, use the *GetDeferral* method, as follows:

```
(function () {
    "use strict";
    //
    // Get a reference to the task instance.
    //
    var bgTaskInstance = Windows.UI.WebUI.WebUIBackgroundTaskInstance.current;

    //
    // Real work.
    //
    function doWork() {

        var backgroundTaskDeferral = bgTaskInstance.getDeferral();

        // Do work

        backgroundTaskDeferral.complete();

        // Call the close function when you have done.
        close();
    }

    doWork();
});
```

After requesting the deferral using the *GetDeferral* method, use the async pattern to perform the asynchronous work and, at the end, call the *Complete* method on the deferral. Be sure to perform all the work after requesting the deferral and before calling the *Complete* and the *Close* method. Otherwise, the system thinks that your job is already done and can shut down the main thread.

MORE INFO THREADS

Chapter 4, “Enhance the user interface,” discusses threads and CPUs.



Thought experiment

Implementing background tasks

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

Your application needs to perform some lengthy cleaning operations on temporary data. To avoid wasting system resources during application use, you want to perform these operations in the background. You implement the code in a background thread but notice that your application sometimes does not clean all the data when the user switches to another application.

1. Why does the application not clean the data all the time?
2. How can you solve this problem?

Objective summary

- A background task can execute lightweight action invoked by the associated event.
- A task needs to be registered using WinRT classes and defined in the application manifest.
- There are many system events you can use to trigger a background task.
- You have to register a task just once.
- You can enumerate tasks that are already registered.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. How can an application fire a background task to respond to a network state modification?
 - A. By using a time trigger, polling the network state every minute, and checking for changes to this value
 - B. By using a *SystemTrigger* for the *InternetAvailable* event and checking whether the network is present or not
 - C. By using a *SystemTrigger* for the *NetworkStateChange* event and using *false* as the second constructor parameter (called *oneShot*)
 - D. By using a *SystemTrigger* for the *NetworkStateChange* event and using *true* as the second constructor parameter

2. Which steps do you need to perform to enable a background task? (Choose all that apply.)
 - A. Register the task in the Package.appxmanifest file.
 - B. Use the *BackgroundTaskBuilder* to create the task.
 - C. Set the trigger that will fire the task code.
 - D. Use a toast to show information to the user.
3. Is it possible to schedule a background task just once?
 - A. Yes, using a specific task.
 - B. No, only system tasks can run once.
 - C. Yes, using a parameter at trigger level.
 - D. No, only a time-triggered task can run once at a certain time.

Objective 1.2: Consume background tasks

The Windows Runtime exposes many ways to interact with the system in a background task and many ways to activate a task. System triggers, time triggers, and conditions can modify the way a task is started and consumed. Moreover, a task can keep a communication channel open to send data to or receive data from remote endpoints. An application may need to download or upload a large resource, even if the user is not using it. The application can also request lock screen permission from the user to enhance other background capabilities.

This objective covers how to:

- Use timing triggers and system triggers
- Keep communication channels open
- Request lock screen access
- Use the *BackgroundTransfer* class to finish downloads

Understanding task triggers and conditions

Many types of background tasks are available, and they respond to different kind of triggers for any kind of application, which can be:

- **MaintenanceTrigger** Raised when it is time to execute system maintenance tasks
- **SystemEventTrigger** Raised when a specific system event occurs

A maintenance trigger is represented by the *MaintenanceTrigger* class. To create a new instance of a trigger you can use the following code:

```
var trigger = new Windows.ApplicationModel.Background.MaintenanceTrigger(60, false);
```

The first parameter is the *freshnessTime* expressed in minutes, and the second parameter, called *oneShot*, is a Boolean indicating whether the trigger should be fired only one time or every *freshnessTime* occurrence.

Whenever a system event occurs, you can check a set of conditions to determine whether your background task should execute. When a trigger is fired, the background task will not run until all of its conditions are met, which means the code for the *doWork* method is not executed if a condition is not met.

All the conditions are enumerated in the *SystemConditionType* enum:

- **UserNotPresent** The user must be away.
- **UserPresent** The user must be present.
- **InternetAvailable** An Internet connection must be available.
- **InternetNotAvailable** An Internet connection must be unavailable.
- **SessionConnected** The session must be connected.
- **SessionDisconnected** The session must be disconnected.

The maintenance trigger can schedule a background task as frequently as every 15 minutes if the device is plugged in to an AC power source. It is not fired if the device is running on batteries.

System and maintenance triggers run for every application that registers them (and declares them in the application manifest). In addition, an application that leverages the lock screen–capable feature of the Windows Runtime can also register background tasks for other events.

An application can be placed on the lock screen to show important information to the user: the user can choose the application he or she wants on the lock screen (up to seven in the first release of Windows 8).

You can use the following triggers to run code for an application on the lock screen:

- **PushNotificationTrigger** Raised when a notification arrives on the Windows Push Notifications Service (WNS) channel.
- **TimeTrigger** Raised at the scheduled interval. The app can schedule a task to run as frequently as every 15 minutes.
- **ControlChannelTrigger** Raised when there are incoming messages on the control channel for apps that keep connections alive.

The user must place the application on the lock screen before the application can use triggers. The application can ask the user to access the lock screen by calling the *RequestAccessAsync* method. The system presents a dialog to the user asking for her or his permission to use the lock screen.

The following triggers are usable only by lock screen–capable applications:

- **ControlChannelReset** The control channel is reset.
- **SessionConnected** The session is connected.

- **UserAway** The user must be away.
- **UserPresent** The user must be present.

In addition, when a lock screen–capable application is placed on the lock screen or removed from it, the following system events are triggered:

- **LockScreenApplicationAdded** The application is added to the lock screen.
- **LockScreenApplicationRemoved** The application is removed from the lock.

A time-triggered task can be scheduled to run either once or periodically. Usually, this kind of task is useful for updating the application tile or badge with some kind of information. For example, a weather app updates the temperature to show the most recent one in the application tile, whereas a finance application refreshes the quote for the preferred stock.

The code to define a time trigger is similar to the code for a maintenance trigger:

```
var taskTrigger = new Windows.ApplicationModel.Background.TimeTrigger(60, true);
```

The first parameter (*freshnessTime*) is expressed in minutes, and the second parameter (called *oneShot*) is a Boolean that indicates whether the trigger will fire only once or at every *freshnessTime* occurrence.

The Windows Runtime has an internal timer that runs tasks every 15 minutes. If the *freshnessTime* is set to 15 minutes and *oneShot* is set to *false*, the task will run every 15 minutes starting between the time the task is registered and the 15 minutes ahead. If the *freshnessTime* is set to 15 minutes and *oneShot* is set to *true*, the task will run in 15 minutes from the registration time.



EXAM TIP

You cannot set the *freshnessTime* to a value less than 15 minutes. An exception will occur if you try to do this.

Time trigger supports all the conditions in the *SystemConditionType* enum presented earlier in this section.

Progressing through and completing background tasks

If an application needs to know the result of the task execution, the code can provide a callback for the *onCompleted* event.

This is the code to create a task and register an event handler for the completion event:

```
var builder = new Windows.ApplicationModel.Background.BackgroundTaskBuilder();
builder.name = taskName;
builder.taskEntryPoint = "js\\BikeBackgroundTask.js";

var trigger = new Windows.ApplicationModel.Background.SystemTrigger(
    Windows.ApplicationModel.Background.SystemTriggerType.timeZoneChange, false);
```

```
builder.addCondition(new Windows.ApplicationModel.Background.SystemCondition(
    Windows.ApplicationModel.Background.SystemConditionType.internetAvailable))

var task = builder.register();
backgroundTaskRegistration.addEventListener("completed", onCompleted);
```

A simple event handler, receiving the *BackgroundCompletedEventArgs*, can show something to the user, as in the following code, or it can update the application tile with some information.

```
function onCompleted(args)
{
    backgroundTaskName = this.name;

    // Update the user interface
}
```



EXAM TIP

A background task can be executed when the application is suspended or even terminated. The *onCompleted* event callback will be fired when the application is resumed from the operating system or the user launches it again. If the application is in the foreground, the event callback is fired immediately.

A well-written application needs to check errors in the task execution. Because the task is already completed when the app receives the callback, you need to check whether the result is available or if something went wrong. To do that, the code can call the *CheckResult* method of the received *BackgroundTaskCompletedEventArgs*. This method throws the exception occurred during the task execution, if any; otherwise it simply returns a void.

Listing 1-4 shows the correct way to handle an exception inside a single task.

LISTING 1-4 Completed event with exception handling

```
function onCompleted(args)
{
    backgroundTaskName = this.name;

    try
    {
        args.checkResult();
        // Update the user interface with OK
    }
    catch (ex)
    {
        // Update the user interface with some errors
    }
}
```

Use a *try/catch* block to intercept the exception fired by the *CheckResult* method, if any. In Listing 1-4, we simply updated the user interface (UI) to show the correct completion or the exception thrown by the background task execution.

Another useful event a background task exposes is the *onProgress* event that, as the name implies, can track the progress of an activity. The event handler can update the UI that is displayed when the application is resumed, or update the tile or the badge with the progress (such as the percent completed) of the job.

The following code is an example of a progress event handler that updates the application titles manually:

```
function onProgress(task, args)
{
    var notifications = Windows.UI.Notifications;
    var template = notifications.TileTemplateType.tileSquareText01;
    var tileXml = notifications.ToastNotificationManager.getTemplateContent(template);
    var tileTextElements = tileXml.getElementsByTagName("text");
    tileTextElements[0].appendChild(tileXml.createTextNode(args.Progress + "%"));
    var tileNotification = new notifications.TileNotification(tileXml);
    notifications.TileUpdateManager.createTileUpdaterForApplication()
        .update(tileNotification);
}
```

The code builds the XML document using the provided template and creates a *tileNotification* with a single value representing the process percentage. Then the code uses the *CreateTileUpdaterForApplication* method of the *TileUpdateManager* class to update the live tile.

The progress value can be assigned in the *doWork* function of the task using the *Progress* property of the instance that represents the task.

Listing 1-5 shows a simple example of progress assignment.

LISTING 1-5 Progress assignment

```
(function () {
    "use strict";

    //
    // Get a reference to the task instance.
    //
    var bgTaskInstance = Windows.UI.WebUI.WebUIBackgroundTaskInstance.current;

    //
    // Real work.
    //
    function doWork() {

        var backgroundTaskDeferral = bgTaskInstance.getDeferral();

        // Do some work

        bgTaskInstance.progress = 10;

        // Do some work

        bgTaskInstance.progress = 20;
```

```

backgroundTaskDeferral.complete();

// Call the close function when you have done.
close();
}

doWork();
});

```

Understanding task constraints

Background tasks have to be lightweight so they can provide the best user experience with foreground apps and battery life. The runtime enforces this behavior by applying resource constraints to tasks:

- **CPU for application not on the lock screen** The CPU is limited to 1 second. A task can run every 2 hours at a minimum. For an application on the lock screen, the system will execute a task for 2 seconds with a 15-minute maximum interval.
- **Network access** When running on batteries, tasks have network usage limits calculated based on the amount of energy used by the network card. This number can be very different from device to device based on their hardware. For example, with a throughput of 10 megabits per second (Mbps), an app on the lock screen can consume about 450 megabytes (MB) per day, whereas an app that is not on the lock screen can consume about 75 MB per day.

MORE INFO TASK CONSTRAINTS

Refer to the MSDN documentation at <http://msdn.microsoft.com/en-us/library/windows/apps/xaml/hh977056.aspx> for updated information on background task resource constraints.

To prevent resource quotas from interfering with real-time communication apps, tasks using the *ControlChannelTrigger* and the *PushNotificationTrigger* receive a guaranteed resource quota (CPU/network) for every running task. The resource quotas and network data usage constraints remain constant for these background tasks rather than varying according to the power usage of the network interface.

Because the system handles constraints automatically, your app does not have to request resource quotas for the *ControlChannelTrigger* and the *PushNotificationTrigger* background tasks. The Windows Runtime treats these tasks as “critical” background tasks.

If a task exceeds these quotas, it is suspended by the runtime. You can check for suspension by inspecting the *suspendedCount* property of the task instance in the *doWork* function, choosing to stop or abort the task if the counter is too high. Listing 1-6 shows how to check for suspension.

LISTING 1-6 Checking for suspension

```
(function () {
    "use strict";

    //
    // Get a reference to the task instance.
    //
    var bgTaskInstance = Windows.UI.WebUI.WebUIBackgroundTaskInstance.current;

    //
    // Real work.
    //
    function doWork() {

        var backgroundTaskDeferral = bgTaskInstance.getDeferral();

        // Do some work

        bgTaskInstance.progress = 10;

        if (bgTaskInstance.suspendedCount > 5) {
            return;
        }

        backgroundTaskDeferral.complete();

        // Call the close function when you have done.
        close();
    }

    doWork();
})();
```

Cancelling a task

When a task is executing, it cannot be stopped unless the task recognizes a cancellation request. A task can also report cancellation to the application using persistent storage.

The *doWork* method has to check for cancellation requests. The easiest way is to declare a Boolean variable in the class and set it to *true* if the system has cancelled the task. This variable will be set to *true* in the *onCanceled* event handler and checked during the execution of the *doWork* method to exit it.

The code in Listing 1-7 shows the simplest complete class to check for cancellation.

LISTING 1-7 Task cancellation check

```
(function () {
    "use strict";

    //
    // Get a reference to the task instance.
    //
    var bgTaskInstance = Windows.UI.WebUI.WebUIBackgroundTaskInstance.current;

    var _cancelRequested = false;

    function onCanceled(cancelSender, cancelReason)
    {
        cancel = true;
    }

    //
    // Real work.
    //
    function doWork() {

        // Add Listener before doing any work
        bgTaskInstance.addEventListener("canceled", onCanceled);

        var backgroundTaskDeferral = bgTaskInstance.getDeferral();

        // Do some work

        bgTaskInstance.progress = 10;

        if (!_cancelRequested) {
            // Do some work
        }
        else
        {
            // Cancel
            bgTaskInstance.succeeded = false;
        }

        backgroundTaskDeferral.complete();

        // Call the close function when you have done.
        close();
    }

    doWork();
});
```

In the *doWork* method, the first line of code sets the event handler for the *canceled* event to the *onCanceled* method. Then it does its job setting the progress and testing the value of the variable to stop working (return or break in case of a loop). The *onCanceled* method sets the *_cancelRequested* variable to *true*.

To recap, the system will call the *Canceled* event handler (*onCanceled*) during a cancellation. The code sets the variable tested in the *doWork* method to stop working on the task.

If the task wants to communicate some data to the application, it can use the local persistent storage as a place to store some data the application can interpret. For example, the *doWork* method can save the status in a *localSettings* key to let the application know whether the task has been successfully completed or cancelled. The application can then check this information in the *Completed* event for the task.

Listing 1-8 shows the revised *doWork* method.

LISTING 1-8 Task cancellation using local settings to communicate information to the app

```
var localSettings = applicationData.localSettings;
function doWork() {

    // Add Listener before doing any work
    bgTaskInstance.addEventListener("canceled", onCanceled);

    var backgroundTaskDeferral = bgTaskInstance.getDeferral();

    // Do some work

    bgTaskInstance.progress = 10;

    if (!cancelRequested) {
        // Do some work
    }
    else {
        // Cancel
        backgroundTaskInstance.succeeded = false;
        settings["status"] = "canceled";
    }

    backgroundTaskDeferral.complete();

    settings["status"] = "success";

    // Call the close function when you have done.
    close();
}
```

Before “stopping” the code in the *doWork* method, the code sets the *status* value in the *localSettings* (that is, the persistent storage dedicated to the application) to *canceled*. If the task completes its work, the value will be *completed*.

The code in Listing 1-9 inspects the *localSettings* value to determine the task outcome. This is a revised version of the *onCompleted* event handler used in a previous sample.

LISTING 1-9 Task completed event handler with task outcome check

```
function OnCompleted(args)
{
    backgroundTaskName = this.name;
    try
    {
        args.checkResult();
        var status = settings["status"];
        if (status == "completed") {
            // Update the user interface with OK
        }
        else {
        }
    }
    catch (Exception ex)
    {
        // Update the user interface with some errors
    }
}
```

The registered background task persists in the local system and is independent from the application version.

Updating a background task

Tasks “survive” application updates because they are external processes triggered by the Windows Runtime. If a newer version of the application needs to update a task or modify its behavior, it can register the background task with the *ServicingComplete* trigger: This way, the app is notified when the application is updated, and unregisters tasks that are no longer valid.

Listing 1-10 shows a system task that unregisters the previous version and registers the new one.

LISTING 1-10 System task that registers a newer version of a task

```
(function () {
    "use strict";

    function doWork() {
        var unregisterTask = "TaskToBeUnregistered";
        var taskRegistered = false;

        var background = Windows.ApplicationModel.Background;
        var iter = background.BackgroundTaskRegistration.allTasks.first();

        var current = iter.hasCurrent;

        while (current) {
            var current = iter.current.value;

            if (current.name === taskName) {
                return current;
            }
        }
    }
})()
```

```

        current = iter.MoveNext();
    }
    if (current != null) {
        current.unregister(true);
    }

    var builder = new Windows.ApplicationModel.Background.BackgroundTaskBuilder();
    builder.name = " BikeGPS"
    builder.taskEntryPoint = "js\\NewBikeBackgroundTask.js";

    var trigger = new Windows.ApplicationModel.Background.SystemTrigger(
        Windows.ApplicationModel.Background.SystemTriggerType.timeZoneChange, false);

    builder.setTrigger(trigger);

    builder.addCondition(new Windows.ApplicationModel.Background.SystemCondition(
        Windows.ApplicationModel.Background.SystemConditionType.internetAvailable))

    var task = builder.register();

    //
    // A JavaScript background task must call close when it is done.
    //
    close();
}

```

The parameter of the *unregister* method set to *true* forces task cancellation, if implemented, for the background task.

The last thing to do is use a *ServicingComplete* task in the application code to register this system task as other tasks using the *ServicingComplete* system trigger type:

```

var background = Windows.ApplicationModel.Background;
var servicingCompleteTrigger = new background.SystemTrigger(
    background.SystemTriggerType.servicingComplete, false);

```

Debugging tasks

Debugging a background task can be a challenging job if you try to use a manual tracing method. In addition, because a timer or a maintenance-triggered task can be executed in the next 15 minutes based on the internal interval, debugging manually is not so effective. To ease this job, Visual Studio background task integrated debugger simplifies the activation of the task.

Place a breakpoint in the *doWork* method or use the *Debug* class to write some values in the output window. Start the project at least one time to register the task in the system, and then use the Debug Location toolbar in Visual Studio to activate the background task. The toolbar can show only registered tasks waiting for the trigger. You can activate the toolbar using the View | Toolbars menu.

Figure 1-2 shows the background registration code and the Debug Location toolbar.

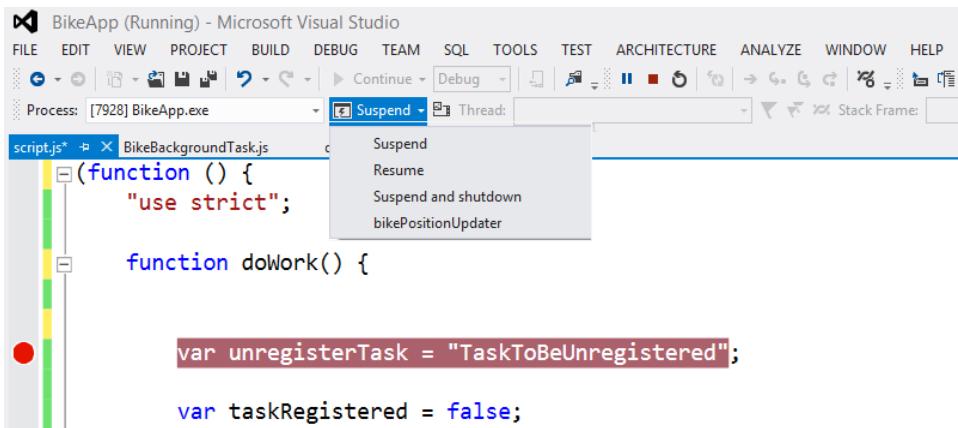


FIGURE 1-2 Visual Studio “hidden” Debug Location toolbar to start tasks

Figure 1-3 shows the debugger inside the *doWork* method.

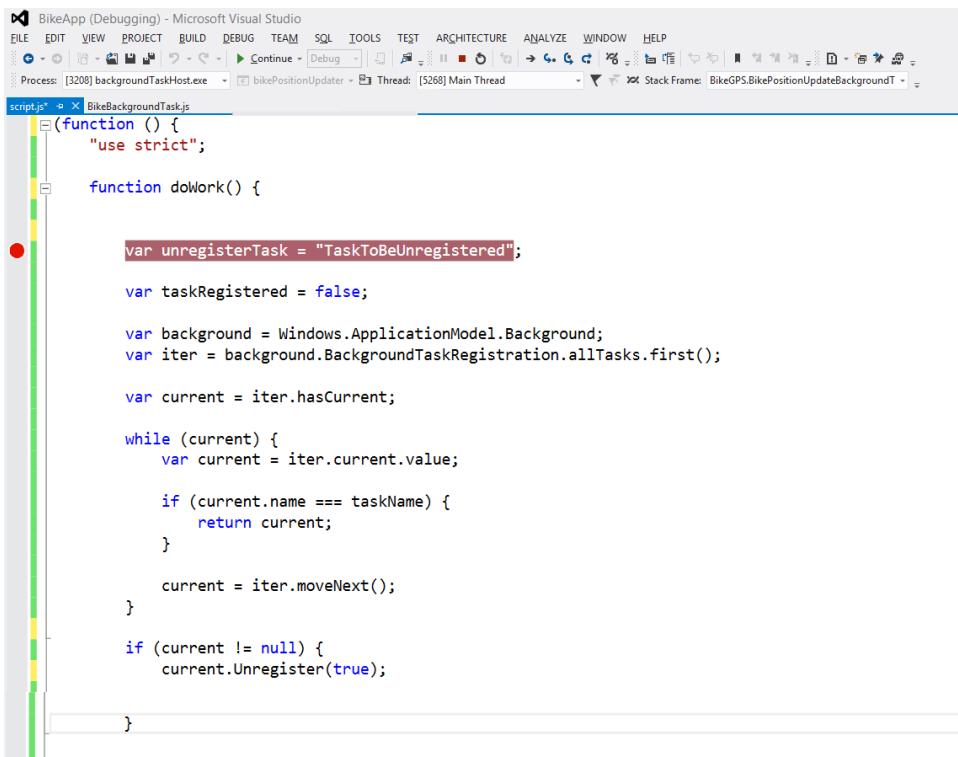


FIGURE 1-3 Debugging tasks activated directly from Visual Studio

Understanding task usage

Every application has to pass the verification process during application submission to the Windows Store. Be sure to double-check the code for background tasks using the following points as guidance:

- Do not exceed the CPU and network quotas in your background tasks. Tasks have to be lightweight to save battery power and to provide a better user experience for the application in the foreground.
- The application should get the list of registered background tasks, register for progress and completion handlers, and handle these events in the correct manner. The classes should also report progress, cancellation, and completion.
- If the *doWork* method uses asynchronous code, make sure the code uses deferrals to avoid premature termination of the method before completion. Without a deferral, the Windows Runtime thinks your code has finished its work and can terminate the thread. Request a deferral, use the *async* pattern to complete the asynchronous call, and close the deferral after the operation completes.
- Declare each background task in the application manifest and every trigger associated with it. Otherwise, the app cannot register the task at runtime.
- Use the *ServicingComplete* trigger to prepare your application to be updated.
- If you use the lock screen–capable feature, remember that only seven apps can be placed on the lock screen, and the user can choose the application she wants at any time. Furthermore, only one app can have a wide tile. The application can provide a good user experience by requesting lock screen access using the *RequestAccessAsync* method. Be sure the application can work without the permission to use the lock screen because the user can deny access to it or remove the permission later.
- Use tiles and badges to provide visual clues to the user, and use the notification mechanism in the task to notify third parties. Do not use any other UI elements in the *Run* method.
- Use persistent storage as *ApplicationData* to share data between the background task and the application. Never rely on user interaction in the task.
- Write background tasks that are short-lived.

Transferring data in the background

Some applications need to download or upload a resource from the web. Because of the application life-cycle management of the Windows Runtime, if you begin to download a file and then the user switches to another application, the first app can be suspended. The file cannot be downloaded during suspension because the system gives no thread and no input/output (I/O) slot to a suspended app. If the user switches back to the application, the download operation can continue, but the download operation will take more time to complete in this

case. Moreover, if the system needs resources, it can terminate the application. The download is then terminated together with the app.

The *BackgroundTransfer* application programming interfaces (APIs) provide classes to avoid these problems. They can be used to enhance the application with file upload and download features that run in the background during suspension. The APIs support HTTP and HTTPS for download and upload operations, and File Transfer Protocol (FTP) for download-only operations. These classes are aimed at large file uploads and downloads.

The process started by one of these classes runs separate from the Windows Store app and can be used to work with resources like files, music, large images, and videos. During the operation, if the runtime chooses to put the application in the Suspended state, the capability provided by the Background Transfer APIs continues to work in the background.

NOTE BACKGROUND TRANSFER APIs

The Background Transfer APIs work for small resources (a few kilobytes), but Microsoft suggests to use the traditional *HttpClient* class for these kinds of files.

The process to create a file download operation involves the *BackgroundDownloader* class: the settings and initialization parameters provide different ways to customize and start the download operation. The same applies for upload operations using the *BackgroundUploader* class. You can call multiple download/upload operations using these classes from the same application because the Windows Runtime handles each operation individually.

During the operation, the application can receive events to update the user interface (if the application is still in the foreground), and you can provide a way to stop, pause, resume, or cancel the operation. You can also read the data during the transfer operation.

These operations support credentials, cookies, and the use of HTTP headers so you can use them to download files from a secured location or provide a custom header to a custom server side HTTP handler.

The operations are managed by the Windows Runtime, promoting smart usage of power and bandwidth. They are also independent from sudden network status changes because they intelligently leverage connectivity and carry data-plan status information provided by the *Connectivity* namespace.

The application can provide a cost-based policy for each operations using the *BackgroundTransferCostPolicy*. For example, you can provide a cost policy to pause the task automatically when the machine is using a metered network and resume it if the user comes back to an “open” connection. The application does nothing to manage these situations; it is sufficient to provide the policy to the background operation.

The first thing to do to enable a transfer operation in the background is enable the network in the Package.appxmanifest file using one of the provided options in the App Manifest Designer. You must use one of the following capabilities:

- **Internet (Client)** The app can provide outbound access to the Internet and networks in public areas, such as coffee shops, airports, and parks.
- **Internet (Client & Server)** The app can receive inbound requests and make outbound requests in public areas.
- **Private Networks (Client & Server)** The app can receive inbound requests and make outbound requests in trusted places, such as home and work.

Figure 1-4 shows the designer with the application capabilities needed for background data transferring.

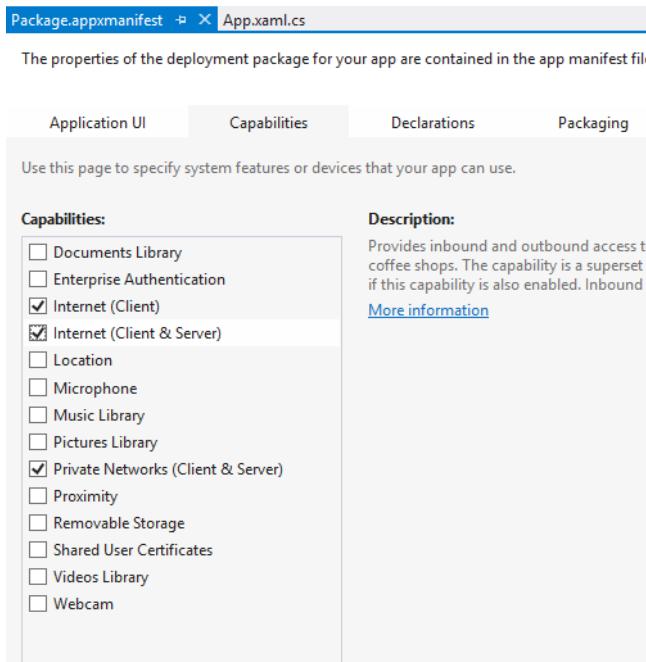


FIGURE 1-4 Capabilities for background transferring

Then you can start writing code to download a file in the local storage folder. The code excerpt in Listing 1-11 starts downloading a file in the Pictures library folder.

LISTING 1-11 Code to activate a background transfer

```
var promise = null;
function downloadInTheBackground (uriToDownload, fileName) {
    try {
        Windows.Storage.KnownFolders.picturesLibrary.createFileAsync(fileName,
            Windows.Storage.CreationCollisionOption.generateUniqueName)
            .done(function (newFile) {
                var uri = Windows.Foundation.Uri(uriToDownload);
                var downloader = new Windows.Networking.BackgroundTransfer
                    .BackgroundDownloader();
```

```

    // Create the operation.
    downloadOp = downloader.createDownload(uri, newFile);

    // Start the download and persist the promise to be able to cancel
    // the download.
    promise = downloadOp.startAsync().then(complete, error);
}, error);
} catch (ex) {
    LogException(ex);
}
};


```

The first line of code sets a local variable representing the file name to download and uses it to create the uniform resource identifier (URI) for the source file. Then the *createFileAsync* method creates a file in the user's Pictures library represented by the *KnownFolders.picturesLibrary* storage folder using the *async* pattern.

The *BackgroundDownloader* class exposes a *createDownload* method to begin the download operation. It returns a *DownloadOperation* class representing the current operation. This *BackgroundDownloader* class exposes the *startAsync* method to start the operation.

The main properties of this class are:

- The *guid* property, which represents the autogenerated unique id for the download operation you can use in the code to create a log for every download operation
- The read-only *requestedUri* property, which represents the URI from which to download the file
- The *resultFile* property, which returns the *IStorageFile* object provided by the caller when creating the download operation

The *BackgroundDownloader* class also exposes the *Pause* and the *Resume* methods, as well as the *CostPolicy* property to use during the background operation.

To track the progress of the download operation, you can use the provided *startAsync* function with a promise that contains the callback function for the progress.

Listing 1-12 shows the revised sample.

LISTING 1-12 Code to activate a background transfer and log progress information

```

var download = null;
var promise = null;

function downloadInTheBackground (uriToDownload, fileName) {
    try {
        Windows.Storage.KnownFolders.picturesLibrary.createFileAsync(fileName,
            Windows.Storage.CreationCollisionOption.generateUniqueName)
            .done(function (newFile) {
                var uri = Windows.Foundation.Uri(uriToDownload);
                var downloader = new Windows.Networking.BackgroundTransfer
                    .BackgroundDownloader();

                // Create the operation.
                downloadOp = downloader.createDownload(uri, newFile);

```

```

        // Start the download and persist the promise to
        // be able to cancel the download.
        promise = downloadOp.startAsync().then(complete, error, progress);
    }, error);
} catch (ex) {
    LogException(ex);
}
};

function progress() {
    // Output all attributes of the progress parameter.
    LogOperation(download.guid + " - progress: ");
}

```

In Listing 1-12, right after the creation of the download operation, the *StartAsync* method returns the *IAsyncOperationWithProgress<DownloadOperation, DownloadOperation>* interface that is transformed in a *Task* using the classic promise *then*.

This way, the callback can use the *DownloadOperation* properties to track the progress or to log (or display if the application is in the foreground) them as appropriate for the application.

The *BackgroundDownloader* tracks and manages all the current download operations; you can enumerate them using the *getCurrentDownloadAsync* method.

Because the system can terminate the application, it is important to reattach the progress and completion event handler during the next launch operation performed by the user. Use the following code as a reference in the application launch:

```

Windows.Networking.BackgroundTransfer.BackgroundDownloader.getCurrentDownloadsAsync()
.done(function (downloads) {
    // If downloads from previous application state exist, reassign callback
    promise = downloads[0].attachAsync().then(complete, error, progress);
})

```

This method gets all the current download operations and reattaches the progress callback to the first one using the *attachAsync* method as a sample. The method returns an asynchronous operation that can be used to monitor progress and completion of the attached download. Calling this method allows an app to reattach download operations that were started in a previous app instance.

If the application can start multiple operations, you have to define an array of downloads and reattach all of the callbacks.

One last thing to address are the timeouts enforced by the system. When establishing a new connection for a transfer, the connection request is aborted if it is not established within five minutes. Then, after establishing a connection, an HTTP request message that has not received a response within two minutes is aborted.

The same concepts apply to resource upload. The *BackgroundUploader* class works in a similar way as the *BackgroundDownloader* class. It is designed for long-term operations on resources like files, images, music, and videos. As mentioned for download operations, small resources can be uploaded using the traditional *HttpClient* class.

You can use the *CreateUploadAsync* to create an asynchronous operation that, on completion, returns an *UploadOperation*. There are three overloads for this method. The MSDN official documentation provides these descriptions:

- ***createUploadAsync(Uri, IIterable(BackgroundTransferContentPart))*** Returns an asynchronous operation that, on completion, returns an *UploadOperation* with the specified URI and one or more *BackgroundTransferContentPart* objects
- ***createUploadAsync(Uri, IIterable(BackgroundTransferContentPart), String)*** Returns an asynchronous operation that, on completion, returns an *UploadOperation* with the specified URI, one or more *BackgroundTransferContentPart* objects, and the multipart subtype
- ***createUploadAsync(Uri, IIterable(BackgroundTransferContentPart), String, String)*** Returns an asynchronous operation that, on completion, returns an *UploadOperation* with the specified URI, multipart subtype, one or more *BackgroundTransferContentPart* objects, and the delimiter boundary value used to separate each part

Alternatively, you can use the more specific *CreateUploadFromStreamAsync* that returns an asynchronous operation that, on completion, returns the *UploadOperation* with the specified URL and the source stream.

This is the method definition:

```
backgroundUploader.createUploadFromStreamAsync(uri, sourceStream)
    .done(
        /* Code for success and error handlers */ );
```

As for the downloader classes, this class exposes the *proxyCredential* property to provide authentication to the proxy and *serverCredential* to authenticate the operation with the target server. You can use the *setRequestHeader* method to specify HTTP header key/value pair.

Keeping communication channels open

For applications that need to work in the background, such as Voice over Internet Protocol (VoIP), instant messaging (IM), and email, the new Windows Store application model provides an always-connected experience for the end user. In practice, an application that depends on a long-running network connection to a remote server can still work, even when the Windows Runtime suspends the application. As you learned, a background task allows an application to perform some kind of work in the background when the application is suspended.

Keeping a communication channel open is required for applications that send data to or receive data from a remote endpoint. Communication channels are also required for long-running server processes to receive and process any incoming requests from the outside.

Typically, this kind of application sits behind a proxy, a firewall, or a Network Address Translation (NAT) device. This hardware component preserves the connection if the endpoints continue to exchange data. If there is no traffic for some time (which can be a few seconds or minutes), these devices close the connection.

To ensure that the connection is not lost and remains open between server and client endpoints, you can configure the application to use some kind of keep-alive connection. A keep-alive connection is a message sent on the network at periodic intervals so that the connection lifetime is prolonged.

These messages can be easily sent in previous versions of Windows because the application stays in the Running state until the user decides to close (or terminate) it. In this scenario, keep-alive messages can be sent without any problems. The new Windows 8 application life-cycle management, on the contrary, does not guarantee that packets are delivered to a suspended app. Moreover, incoming network connections can be dropped and no new traffic is sent to a suspended app. These behaviors have an impact on the network devices that close the connection between apps because they become “idle” from a network perspective.

To be always connected, a Windows Store app needs to be a lock screen-capable application. Only applications that use one or more background tasks can be lock screen apps.

An app on the lock screen can:

- Run code when a time trigger occurs.
- Run code when a new user session is started.
- Receive a raw push notification from WNS and run code when a notification is received.
- Maintain a persistent transport connection in the background to remote services or endpoints, and run code when a new packet is received or a keep-alive packet needs to be sent using a network trigger.

Remember, a user can have a maximum of seven lock screen apps at any given time. A user can also add or remove an app from the lock screen at any time.

WNS is a cloud service hosted by Microsoft for Windows 8. Windows Store apps can use WNS to receive notifications that can run code, update a live tile, or raise an on-screen notification. To use WNS, the local computer must be connected to the Internet so that the WNS service can communicate with it. A Windows Store app in the foreground can use WNS to update live tiles, raise notifications to the user, or update badges. Apps do not need to be on the lock screen to use WNS. You should consider using WNS in your app if it must run code in response to a push notification.

MORE INFO WINDOWS PUSH NOTIFICATION SERVICE (WNS)

For a complete discussion of WNS, refer to Chapter 3, “Program user interaction.”

The *ControlChannelTrigger* class in the *System.Net.Sockets* namespace implements the trigger for applications that must maintain a persistent connection to a remote endpoint. Use this feature if the application cannot use WNS. For example, an email application that uses some POP3 servers cannot be modified to use a push notification because the server does not implement WNS and does not send messages to POP3 clients.

 **EXAM TIP**

The MSDN documentation states "The *ControlChannelTrigger* class and related classes are not supported in a Windows Store app using JavaScript. A foreground app using JavaScript with an in-process C# or C++ binary is also not supported." Therefore, for a JavaScript application, you must implement a WinMD library in C#, Visual Basic (VB), or C++ to define the background task and register it in the main application. We will analyze the C# implementation of the background task because this is an important aspect for the exam.

The *ControlChannelTrigger* can be used by instances of one of the following classes: *MessageWebSocket*, *StreamWebSocket*, *StreamSocket*, *HttpClient*, *HttpClientHandler*, or related classes in the *System.Net.Http* namespace in .NET Framework 4.5. The *IXMLHttpRequest2*, an extension of the classic *XMLHttpRequest*, can also be used to activate a *ControlChannelTrigger*.

The main benefits of using a network trigger are compatibility with existing client/server protocols and the guarantee of message delivery. The drawbacks are a little more complex in respect to WNS and the maximum number of triggers an app can use (which is five in the current version of the Windows Runtime).

 **EXAM TIP**

An application written in JavaScript cannot use a *ControlChannelTrigger* if it uses other background tasks.

An application that uses a network trigger needs to request the lock screen permission. This feature supports two different resources for a trigger:

- **Hardware slot** The application can use background notification even when the device is in low-power mode or standby (connected to plug).
- **Software slot** The application cannot use network notification when not in low-power mode or standby (connected to plug).

This resource capability provides a way for your app to be triggered by an incoming notification, even if the device is in low-power mode. By default, a software slot is selected if the developer does not specify an option. A software slot allows your app to be triggered when the system is not in connected standby. This is the default on most computers.

There are two trigger types:

- **Push notification network trigger** This trigger lets a Windows Store app process incoming network packets on an already established Transmission Control Protocol (TCP) socket, even if the application is suspended. This socket represents the control channel that exists between the application and a remote endpoint, and it is created by the application to trigger a background task when a network packet is received by the application itself. In practice, the control channel is a persistent Transmission Control Protocol/Internet Protocol (TCP/IP) connection maintained alive by the Windows Runtime, even if the application is sent in the background and suspended.
- **Keep-alive network trigger** This trigger provides the capability for a suspended application to send keep-alive packets to a remote service or endpoint. The keep-alive packets tell the network device that a connection is still in use, to avoid closing a connection.

Before using a network trigger, the application has to be a lock screen app. You need to declare application capability and then call the appropriate method to ask the user the permission to place the application on the lock screen.

NOTE LOCK SCREEN REMOVAL

You have also to handle the situation where the user removes the application from the lock screen.

To register an application for the lock screen, ensure that the application has a *WideLogo* definition in the application manifest on the *DefaultTile* element:

```
<DefaultTile ShowName="allLogos" WideLogo="Assets\wideLogo.png" />
```

Add a *LockScreen* element that represents the application icon on the lock screen inside the *VisualElements* node in the application manifest:

```
<LockScreen Notification="badge" BadgeLogo="Assets\badgeLogo.png" />
```

You can use the App Manifest Designer, as shown in Figure 1-5, to set these properties. The Wide logo and the Badge logo options reference the relative images, and the Lock screen notifications option is set to Badge.

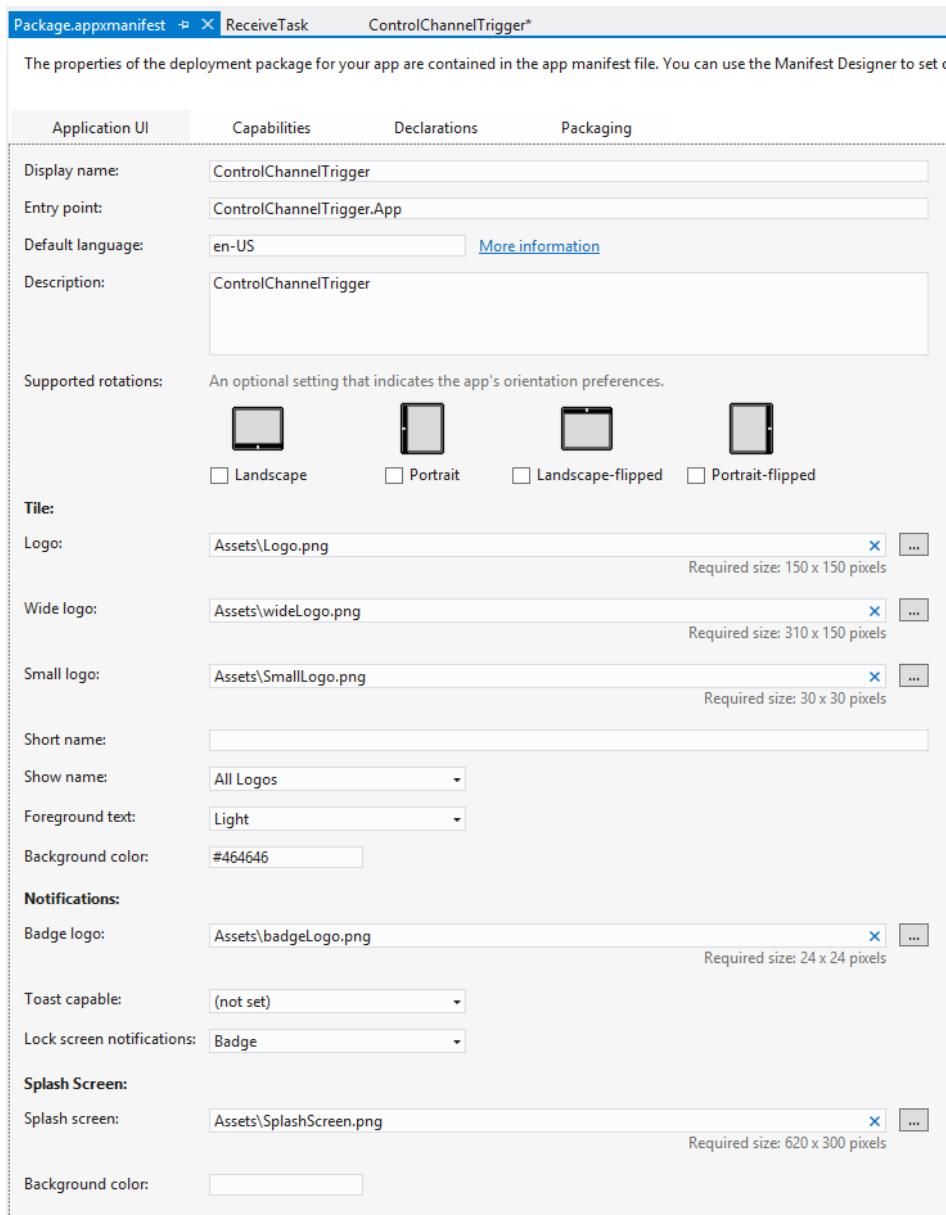


FIGURE 1-5 Badge and wide logo definition

Declare the extensions to use a background task, and define the executable file that contains the task and the name of the class implementing the entry point for the task. The task has to be a *controlChannel* background task type. For this kind of task, the executable file is the application itself. Apps using the *ControlChannelTrigger* rely on in-process activation for the background task.

The dynamic-link library (DLL) or the executable file that implements the task for keep-alive or push notifications must be linked as Windows Runtime Component (WinMD library). The following XML fragment declares a background task:

Sample of XML code

```
<Extensions>
  <Extension Category="windows.backgroundTasks"
    Executable="$targetnametoken$.exe"
    EntryPoint="ControlChannelTriggerTask.ReceiveTask">
    <BackgroundTasks>
      <Task Type="controlChannel" />
    </BackgroundTasks>
  </Extension>
</Extensions>
```

You can also use the App Manifest Designer to set these extensions in an easier way, as shown in Figure 1-6.

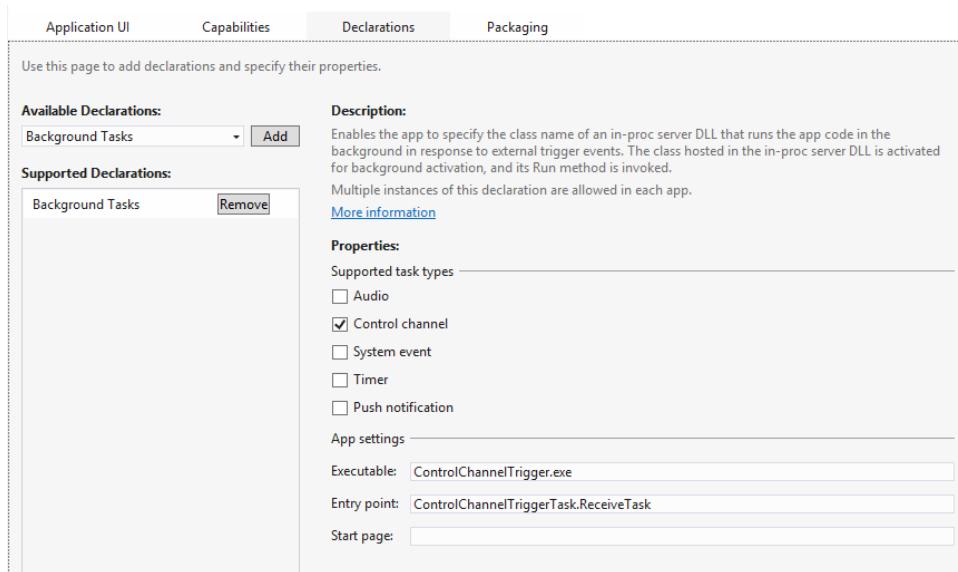


FIGURE 1-6 Background task app settings

The next step is to ask the user for the permission to become a lock screen application using the *RequestAccessAsync* method of the *BackgroundExecutionManager* class of the *Windows.ApplicationModel.Background* namespace. The call to this method presents a dialog to the user to approve the request. See Listing 1-13.

LISTING 1-13 Code to request use of the lock screen

```

var lockScreenEnabled = false;

function ClientInit() {

    if (lockScreenEnabled == false) {
        BackgroundExecutionManager.requestAccessAsync().done(function (result) {
            switch (result) {
                case BackgroundAccessStatus.AllowedWithAlwaysOnRealTimeConnectivity:
                    //
                    // App is allowed to use RealTimeConnection broker
                    // functionality even in low-power mode.
                    //
                    lockScreenEnabled = true;
                    break;
                case BackgroundAccessStatus.AllowedMayUseActiveRealTimeConnectivity:
                    //
                    // App is allowed to use RealTimeConnection broker
                    // functionality but not in low-power mode.
                    //
                    lockScreenEnabled = true;
                    break;
                case BackgroundAccessStatus.Denied:
                    //
                    // App should switch to polling
                    //
                    WinJS.log && WinJS.log("Lock screen access is not permitted",
                        "devleap", "status");
                    break;
            }
        }, function (e) {
            WinJS.log && WinJS.log("Error requesting lock screen permission.",
                "devleap", "error");
        });
    }
}

```



EXAM TIP

The lock screen consent dialog prompts the user just one time. If the user denies permission for the lock screen, you will not be able to prompt the user again. The user can decide later to bring the application on the lock screen from the system permission dialog, but the application has no possibility to ask for the permission again.

The *BackgroundAccessStatus* enumeration lets you know the user's choice. See the comments in Listing 1-13 that explain the various states.

After your app is added to the lock screen, it should be visible in the Personalize section of the PC settings. Remember to handle the removal of the application's lock screen permission by the user. The user can deny the permission to use the lock screen at any time, so you must ensure the app is always functional.

When the application is ready for the lock screen, you have to implement the WinMD library to perform the network operations. Remember you cannot implement a WinMD

library in a Windows Store app using JavaScript. You have to create a C#, VB, or C++ WinMD component and call it from the main application.

The component code has to:

- Create a control channel.
- Open a connection.
- Associate the connection with the control channel.
- Connect the socket to the endpoint server.
- Establish a transport connection to your remote endpoint server.

You have to create the channel to be associated with the connection so that the connection will be kept open until you close the control channel.

After a successful connection to the server, synchronize the transport created by your app with the lower layers of the operating system by using a specific API, as shown in the C# code in Listing 1-14.

LISTING 1-14 Control channel creation and connection opening

```
private Windows.Networking.Sockets.ControlChannelTrigger channel;
private void CreateControlChannel_Click(object sender, RoutedEventArgs e)
{
    ControlChannelTriggerStatus status;

    //
    // 1: Create the instance.
    //

    this.channel = new Windows.Networking.Sockets.ControlChannelTrigger(
        "ch01", // Channel ID to identify a control channel.
        20,      // Server-side keep-alive in minutes.
        ControlChannelTriggerResourceType.RequestHardwareSlot); // Request hardware slot.

    //
    // Create the trigger.
    //
    BackgroundTaskBuilder controlChannelBuilder = new BackgroundTaskBuilder();
    controlChannelBuilder.Name = "ReceivePacketTaskChannelOne";
    controlChannelBuilder.TaskEntryPoint =
        "ControlChannelTriggerTask.ReceiveTask";
    controlChannelBuilder.SetTrigger(channel.PushNotificationTrigger);
    controlChannelBuilder.Register();

    //
    // Step 2: Open a socket connection (omitted for brevity).
    //

    //
    // Step 3: Tie the transport object to the notification channel object.
    //
    channel.UsingTransport(sock);
```

```

// Step 4: Connect the socket (omitted for brevity).
// Connect or Open

//
// Step 5: Synchronize with the lower layer
//
status = channel.WaitForPushEnabled();
}

```

Despite its name, the *WaitForPushEnabled* method is not related in any way to the WNS. This API allows the hardware or software slot to be registered with all the underlying layers of the stack that will handle an incoming data packet, including the network device driver.

There are several types of keep-alive intervals that may relate to network apps:

- **TCP keep-alive** Defined by the TCP protocol
- **Server keep-alive** Used by *ControlChannelTrigger*
- **Network keep-alive** Used by *ControlChannelTrigger*

The keep-alive option for TCP lets an application send packets from the client to the server endpoint automatically to keep the connection open, even when the connection is not used by the application itself. This way, the connection is not cut from the underlying systems.

The application can use the *KeepAlive* property of the *StreamSocketControl* class to enable or disable this feature on a *StreamSocket*. The default is disabled.

Other socket-related classes that do not expose the *KeepAlive* property, such as *MessageWebSocket*, *StreamSocketListener*, and *StreamWebSocket*, have the keep-alive options disabled by default. In addition, the *HttpClient* class and the *IXMLHttpRequest2* interface do not have an option to enable TCP keep-alive.

When using the *ControlChannelTrigger* class, take into consideration these two types of keep-alive intervals:

- **Server keep-alive interval** Represents how often the application is woken up by the system during suspension. The interval is expressed in minutes in the *ServerKeepAliveIntervalInMinutes* property of the *ControlChannelTrigger* class. You can provide the value as a class constructor parameter. It is called server keep-alive because the application sets its value based on the server time-out for cutting an idle connection. For example, if you know the server has a keep-alive of 20 minutes, you can set this property to 18 minutes to avoid the server cutting the connection.
- **Network keep-alive interval** Represents the value, in minutes, that the lower-level TCP stack uses to maintain a connection open. In practice, this value is used by the network intermediaries (proxy, gateway, NAT, and so on) to maintain an idle connection open. The application cannot set this value because it is determined automatically by lower-level network components of the TCP stack.

The last thing to do is to implement the background task and perform some operations, such as updating a tile or sending a toast, when something arrives from the network. The following code implements the *Run* method imposed by the interface:

```
public sealed class ReceiveTask : IBackgroundTask
{
    public void Run(Windows.AppModel.Background.IBackgroundTaskInstance taskInstance)
    {
        var channelEventArgs =
            (IControlChannelTriggerEventArgs)taskInstance.TriggerDetails;
        var channel = channelEventArgs.ControlChannelTrigger;
        string channelId = channel.ControlChannelTriggerId;

        // Send Toast – Update Tile...

        channel.FlushTransport();
    }
}
```

The *TriggerDetails* property provides the information needed to access the raw notification and exposes the *ControlChannelTriggerId* of the *ControlChannelTrigger* class the app can use to identify the various instances of the channel.

The *FlushTransport* method is required if the application sends data.

Remember that an application can receive background task triggers when the application is also in the foreground. You can provide some visual clues to the user in the current page if the application is up and running.



Thought experiment

Transferring data

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

Your application needs to upload photos to a remote storage location in the cloud. Because photos can be greater than 10 MB, you implement a background task that performs this operation. The process works fine, but you discover a slowdown in the process in respect to the same code executed in an application thread (up to 10 times).

1. What is the cause of the slowdown?
2. How can you solve the problem?

Objective summary

- An application can use system and maintenance triggers to start a background task without the need to register the application in the lock screen.
- Lock screen applications can use many other triggers, such as *TimeTrigger* and *ControlChannelTrigger*.
- Background tasks can provide progress indicators to the calling application using events and can support cancellation requests.
- If an app needs to upload or download resources, you can use the *BackgroundTransfer* classes to start the operation and let the system manage its completion.
- Background tasks have resource constraints imposed by the system. Use them for short and lightweight operations. Remember also that scheduled triggers are fired by the internal clock at regular intervals.
- Applications that need to receive information from the network or send information to a remote endpoint can leverage network triggers to avoid connection closing by intermediate devices.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the “Answers” section at the end of this chapter.

1. Which is the lowest frequency at which an app can schedule a maintenance trigger?
 - A. 2 hours.
 - B. 15 minutes every hour.
 - C. 7 minutes if the app is in the lock screen.
 - D. None; there is no frequency for maintenance triggers.
2. How many conditions need to be met for a background task to start?
 - A. All the set conditions.
 - B. Only one.
 - C. At least 50 percent of the total conditions.
 - D. All the set conditions if the app is running on DC power.
3. How can a task be cancelled or aborted?
 - A. Abort the corresponding thread.
 - B. Implement the *OnCanceled* event.
 - C. Catch an exception.
 - D. A background task cannot be aborted.

4. An application that needs to download a file can use which of the following? (Choose all that apply.)
- A. The *BackgroundTask* class
 - B. The *HttpClient* class if the file is very small
 - C. The *BackgroundTransfer* class
 - D. The *BackgroundDownloader* class
 - E. The *BackgroundUploader* class

Objective 1.3: Integrate WinMD components into a solution

The Windows Runtime exposes a simple way to create components that can be used by all the supported languages without any complex data marshalling. A WinMD library, called Windows Runtime Component, is a component written in one of the WinRT languages (C#, VB, or C++, but not JavaScript) that can be used by any supported languages.

This objective covers how to:

- Consume a WinMD component in JavaScript
- Handle WinMD reference types
- Reference a WinMD component

NOTE REFERENCE

The content in this section is excerpted from *Build Windows 8 Apps with Microsoft Visual C# and Visual Basic Step by Step*, written by Paolo Pialorsi, Roberto Brunetti, and Luca Renzicoli (Microsoft Press, 2013).

Understanding the Windows Runtime and WinMD

Windows, since its earliest version, has provided developers with libraries and APIs to interact with the operating system. However, before the release of Windows 8, those APIs and libraries were often complex and challenging to use. Moreover, while working in .NET Framework using C# or VB.NET, you often had to rely on Component Object Model (COM) Interop, and Win32 interoperability via P/Invoke (Platform Invoke) to directly leverage the operating system. For example, the following code sample imports a native Win32 DLL and declares the function *capCreateCaptureWindows* to be able to call it from .NET code:

Sample of C# code

```
[DllImport("avicap32.dll", EntryPoint="capCreateCaptureWindow")]
static extern int capCreateCaptureWindow(
    string lpszWindowName, int dwStyle,
    int X, int Y, int nWidth, int nHeight,
    int hwndParent, int nID);

[DllImport("avicap32.dll")]
static extern bool capGetDriverDescription(
    int wDriverIndex,
    [MarshalAs(UnmanagedType.LPTStr)] ref string lpszName,
    int cbName,
    [MarshalAs(UnmanagedType.LPTStr)] ref string lpszVer,
    int cbVer);
```

Microsoft acknowledged the complexity of the previously existing scenario and invested in Windows 8 and the Windows Runtime to simplify the interaction with the native operating system. In fact, the Windows Runtime is a set of completely new APIs that were reimagined from the developer perspective to make it easier to call to the underlying APIs without the complexity of P/Invoke and Interop. Moreover, the Windows Runtime is built so that it supports the Windows 8 application development with many of the available programming languages/environments, such as HTML5/Windows Library for JavaScript (WinJS), common runtime language (CLR), and C++.

The following code illustrates how the syntax is clearer and easier to write, which makes it easier to read and maintain in the future, when leveraging the Windows Runtime. In this example, *Photo* is an Extensible Application Markup Language (XAML) image control.

Sample of C# code

```
using Windows.Media.Capture;

var camera = new CameraCaptureUI();
camera.PhotoSettings.CroppedAspectRatio = new Size(4, 3);

var file = await camera.CaptureFileAsync(CameraCaptureUIMode.Photo);

if (file != null)
{
    var bitmap = new BitmapImage();
    bitmap.SetSource(await file.OpenAsync(FileAccessMode.Read));
    Photo.Source = bitmap;
}
```

The code for WinJS and HTML5 is similar to the C# version, as follows:

Sample of JavaScript code

```
var camera = new capture.CameraCaptureUI();

camera.captureFileAsync(capture.CameraCaptureUIMode.photo)
    .then(function (file) {
        if (file != null) {
            media.shareFile = file;
        }
    });
});
```

Basically, the Windows Runtime is a set of APIs built upon the Windows 8 operating system (see Figure 1-7) that provides direct access to all the main primitives, devices, and capabilities for any language available for developing Windows 8 apps. The Windows Runtime is available only for building Windows 8 apps. Its main goal is to unify the development experience of building a Windows 8 app, regardless of which programming language you choose.

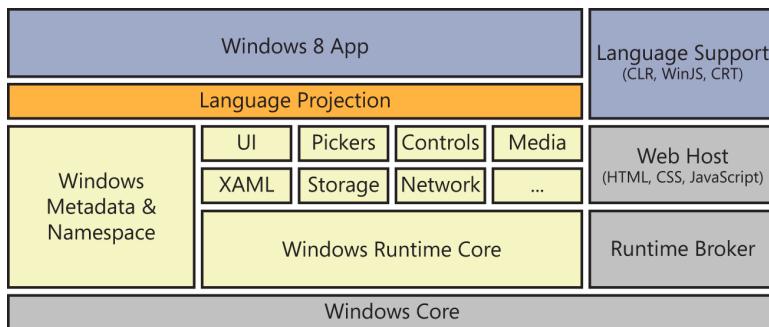


FIGURE 1-7 The Windows Runtime architecture

The Windows Runtime sits on top of the WinRT core engine, which is a set of C++ libraries that bridge the Windows Runtime with the underlying operating system. On top of the WinRT core is a set of specific libraries and types that interact with the various tools and devices available in any Windows 8 app. For example, there is a library that works with the network, and another that reads and writes from storage (local or remote). There is a set of pickers to pick up items (such as files and pictures), and there are several classes to leverage media services, and so on. All these types and libraries are defined in a structured set of namespaces and are described by a set of metadata called Windows Metadata (WinMD). All metadata information is based on a new file format, which is built upon the common language interface (CLI) metadata definition language (ECMA-335).

Consuming a native WinMD library

The WinRT core engine is written in C++ and internally leverages a proprietary set of data types. For example, the *HSTRING* data type represents a text value in the Windows Runtime. In addition, there are numeric types like *INT32* and *UINT64*, enumerable collections represented by *IVector<T>* interface, enums, structures, runtime classes, and many more.

To be able to consume all these sets of data types from any supported programming language, the Windows Runtime provides a projection layer that shuttles types and data between the Windows Runtime and the target language. For example, the WinRT *HSTRING* type will be translated into a *System.String* of .NET for a CLR app, or to a *Platform::String* for a C++ app.

Next to this layered architecture is a Runtime Broker, which acts as a bridge between the operating system and the hosts executing Windows 8 apps, whether those are CLR, HTML5/WinJS, or C++ apps.

Using the Windows Runtime from a CLR Windows 8 app

To better understand the architecture and philosophy behind the Windows Runtime, the example in this section consumes the Windows Runtime from a CLR Windows 8 app.

You can test the use of the native WinMD library by creating a new project in Visual Studio 2012 and using the XAML code in Listing 1-15 for the main page.

LISTING 1-15 Main page with a button control

```
<Page x:Class="WinRTFromCS.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:WinRTFromCS"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">
    <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">
        <StackPanel>
            <Button Click="UseCamera_Click" Content="Use Camera" />
        </StackPanel>
    </Grid>
</Page>
```

In the event handler for the *UserCamera_Click* event, use the following code:

```
private async void UseCamera_Click(object sender, RoutedEventArgs e)
{
    var camera = new Windows.Media.Capture.CameraCaptureUI();
    var photo = await camera.CaptureFileAsync(
        Windows.Media.Capture.CameraCaptureUIMode.Photo);
}
```

Notice the *async* keyword and the two lines of code inside the event handler that instantiate an object of type *CameraCaptureUI* and invoke its *CaptureFileAsync* method.

You can debug this simple code by inserting a breakpoint at the first line of code (the one starting with *var camera* =). Figure 1-8 shows that when the breakpoint is reached, the call stack window reveals that the app is called by external code, which is native code.

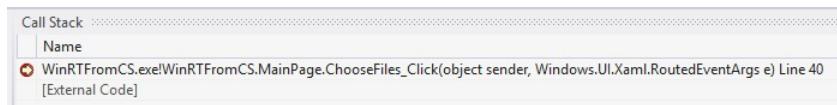


FIGURE 1-8 Call stack showing external code

If you try to step into the code of the *CameraCaptureUI* type constructor, you will see that it is not possible in managed code, because the type is defined in the Windows Runtime, which is unmanaged.

Using the Windows Runtime from a C++ Windows 8 app

The example in this section uses the WinRT Camera APIs to capture an image from a C++ Windows 8 app. First, you need to create a fresh app, using C++ this time.

Assuming you are using the same XAML code as in Listing 1-15, the event handler for the `UseCamera_Click` event instantiates the same classes and calls the same methods you saw in C# using a C++ syntax (and the C++ compiler). See Listing 1-16.

LISTING 1-16 Using the `CameraCaptureUI` class from C++

```
void WinRTFromCPP::MainPage::UseCamera_Click(
    Platform::Object^ sender, Windows::UI::Xaml::RoutedEventArgs^ e) {
    auto camera = ref new Windows::Media::Capture::CameraCaptureUI();
    camera->CaptureFileAsync(Windows::Media::Capture::CameraCaptureUIMode::Photo);
}
```

If you debug this code as in the previous section, the outcome will be very different because you will be able to step into the native code of the `CameraCaptureUI` constructor, as well as into the code of the `CaptureFileAsync` method.

The names of the types, as well as the names of the methods and enums, are almost the same in C# and in C++. Nevertheless, each individual language has its own syntax, code casing, and style. However, through this procedure, you can gain hands-on experience with the real nature of the Windows Runtime: a multilanguage API that adapts its syntax and style to the host language and maintains a common set of behavior capabilities under the covers. What you have just seen is the result of the language projection layer defined in the architecture of the Windows Runtime.

To take this sample one step further, you can create the same example you did in C# and C++ using HTML5/WinJS. If you do that, you will see that the code casing will adapt to the JavaScript syntax.

The following HTML5 represents the user interface for the Windows Store app using JavaScript version:

```
<!DOCTYPE html>
<html>
<head>
    <title>DevLeap WebCam</title>
    <!-- WinJS references -->
    <link rel="stylesheet" href="/winjs/css/ui-dark.css" />
    <script src="/winjs/js/base.js"></script>
    <script src="/winjs/js/wwaapp.js"></script>
    <!-- DevLeapWebcam references -->
    <link rel="stylesheet" href="/css/default.css" />

    <script type="text/javascript">
        function takePicture() {
            var captureUI = new Windows.Media.Capture.CameraCaptureUI();
            captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
                .then(function (photo) {
                    if (photo) {
                        document.getElementById("msg ").innerHTML = "Photo taken."
                    }
                })
        }
    </script>

```

```

        }
    else {
        document.getElementById("msg").innerHTML = "No photo captured."
    }
});
}
</script>
</head>
<body>
<input type="button" onclick="takePicture()" value="Click to shoot" /><br />
<span id="msg"></span>
</body>
</html>

```

The language projection of the Windows Runtime is based on a set of new metadata files, called WinMD. By default, those files are stored under the path <OS Root Path>\System32\WinMetadata, where <OS Root Path> should be replaced with the Windows 8 root installation folder (normally C:\Windows). Here's a list of the default contents of the WinMD folder:

- Windows.ApplicationModel.winmd
- Windows.Data.winmd
- Windows.Devices.winmd
- Windows.Foundation.winmd
- Windows.Globalization.winmd
- Windows.Graphics.winmd
- Windows.Management.winmd
- Windows.Media.winmd
- Windows.Networking.winmd
- Windows.Security.winmd
- Windows.Storage.winmd
- Windows.System.winmd
- Windows.UI.winmd
- Windows.UI.Xaml.winmd
- Windows.Web.winmd

Note that the folder includes a Windows.Media.winmd file, which contains the definition of the *CameraCaptureUI* type used in Listing 1-16.

You can inspect any WinMD file using the Intermediate Language Disassembler (ILDASM) tool available in the Microsoft .NET Software Development Kit (SDK), which ships with Microsoft Visual Studio 2012 and that you can also download as part of the Microsoft .NET Framework SDK. For example, Figure 1-9 shows the ILDASM tool displaying the content outline of the Windows.Media.winmd file, which contains the definition of the *CameraCaptureUI* type from Listing 1-16.

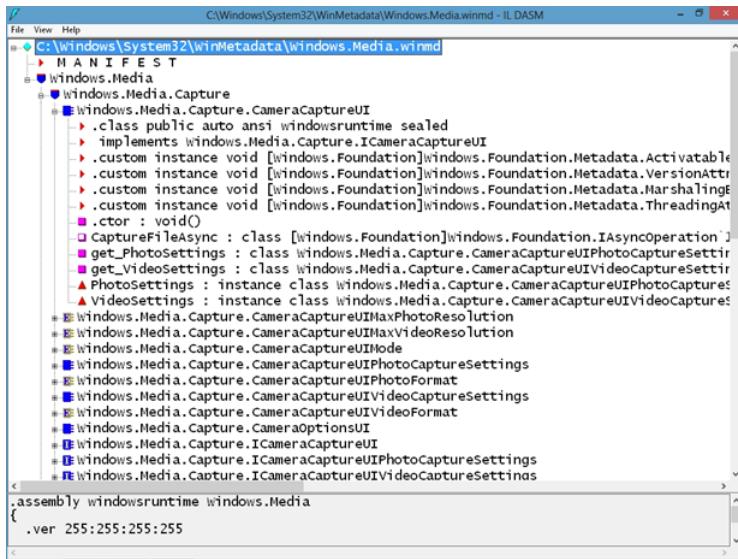


FIGURE 1-9 ILDASM displaying the outline of the Windows.Media.winmd file

The MANIFEST file listed at the top of the window defines the name, version, signature, and dependencies of the current WinMD file. Moreover, there is a hierarchy of namespaces grouping various types. Each single type defines a class from the WinRT perspective. In Figure 1-9, you can clearly identify the *CaptureFileAsync* method you used in the previous example. By double-clicking on the method in the outline, you can see its definition, which is not the source code of the method but rather the metadata mapping it to the native library that will be leveraged under the cover. In the following code excerpt, you can see the metadata definition of the *CaptureFileAsync* method defined for the *CameraCaptureUI* type:

```
method public hidebysig newslot virtual final
    instance class [Windows.Foundation]Windows.Foundation.IAsyncOperation`1
    <class[Windows.Storage]Windows.Storage.StorageFile>
        CaptureFileAsync([in] valuetype Windows.Media.Capture.CameraCaptureUIMode mode)

    runtime managed {
        .override Windows.Media.Capture.ICameraCaptureUI::CaptureFileAsync
    }
// end of method CameraCaptureUI::CaptureFileAsync
```

The language projection infrastructure will translate this neutral definition into the proper format for the target language.

Whenever a language needs to access a WinRT type, it will inspect its definition through the corresponding WinMD file and will use the *IInspectable* interface, which is implemented by any single WinRT type. The *IInspectable* interface is an evolution of the already well-known *IUnknown* interface declared many years ago in the COM world.

First, there is a type declaration inside the registry of the operating system. All the WinRT types are registered under the path HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsRuntime\ActivatableClassId.

For example, the *CameraCaptureUI* type is defined under the following path:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\WindowsRuntime\ActivatableClassId\  
Windows.Media.Capture.CameraCaptureUI
```

The registry key contains some pertinent information, including the activation type (in process or out of process), as well as the full path of the native DLL file containing the implementation of the target type.

The type implements the *IInspectable* interface, which provides the following three methods:

- **GetIds** Gets the interfaces that are implemented by the current WinRT class
- **GetRuntimeClassName** Gets the fully qualified name of the current WinRT object
- **GetTrustLevel** Gets the trust level of the current WinRT object

By querying the *IInspectable* interface, the language projection infrastructure of the Windows Runtime will translate the type from its original declaration into the target language that is going to consume the type.

As illustrated in Figure 1-10, the projection occurs at compile time for a C++ app consuming the Windows Runtime, and it will produce native code that will not need any more access to the metadata. In the case of a CLR app (C#/VB), it happens during compilation into IL code, as well as at runtime through a runtime-callable wrapper. However, the cost of communication between CLR and the WinRT metadata is not so different from the cost of talking with the CLR metadata in general. Lastly, in the case of an HTML5/WinJS app, it will occur at runtime through the Chakra engine.

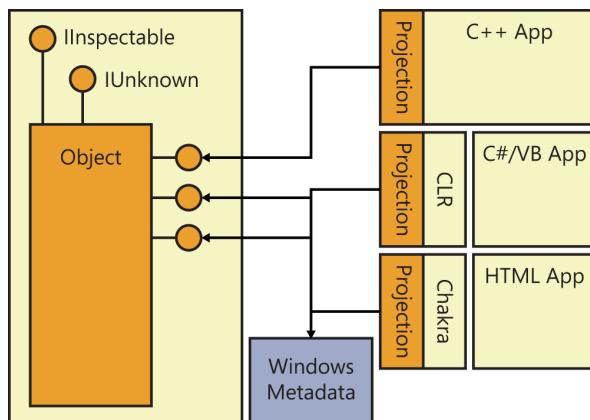


FIGURE 1-10 Projection schema

The overall architecture of the Windows Runtime is also versioning compliant. In fact, every WinRT type will be capable of supporting a future version of the operating system and/or of the Windows Runtime engine by simply extending the available interfaces implemented and providing the information about the new extensions through the *IInspectable* interface.

To support the architecture of the WinRT and the language projection infrastructure, every Windows 8 app—regardless of the programming language used to write it—runs in a standard code execution profile that is based on a limited set of capabilities. To accomplish this goal, the Windows Runtime product team defined the minimum set of APIs needed to implement a Windows 8 app. For example, the Windows 8 app profile has been deprived of the entire set of console APIs, which are not needed in a Windows 8 app. The same happened to ASP.NET, for example—the list of .NET types removed is quite long. Moreover, the Windows Runtime product team decided to remove all the old-style, complex, and/or dangerous APIs and instead provide developers with a safer and easier working environment. As an example, to access XML nodes from a classic .NET application, you have a rich set of APIs to choose from, such as XML Document Object Model (DOM), Simple API for XML, LINQ to XML in .NET, and so on. The set also depends on which programming language you are using. In contrast, in a Windows 8 app written in CLR (C#/VB) you have only the LINQ to XML support, while the XML DOM has been removed.

Furthermore, considering a Windows 8 app is an application that can execute on multiple devices (desktop PCs, tablets, ARM-based devices, and Windows Phone 8 mobile phones), all the APIs specific to a particular operating system or hardware platform have been removed.

The final result is a set of APIs that are clear, simple, well-designed, and portable across multiple devices. From a .NET developer perspective, the Windows 8 app profile is a .NET 4.5 profile with a limited set of types and capabilities, which are the minimum set useful for implementing a real Windows 8 app.

Consider this: The standard .NET 4.5 profile includes more than 120 assemblies, containing more than 400 namespaces that group more than 14,000 types. In contrast, the Windows 8 app profile includes about 15 assemblies and 70 namespaces that group only about 1,000 types.

The main goals in this profile design were to do the following:

- Avoid duplication of types and/or functionalities.
- Remove APIs not applicable to Windows 8 apps.
- Remove badly designed or legacy APIs.
- Make it easy to port existing .NET applications to Windows 8 apps.
- Keep .NET developers comfortable with the Windows 8 app profile.

For example, the Windows Communication Foundation (WCF) APIs exist, but you can use WCF only to consume services, therefore leveraging a reduced set of communication bindings. You cannot use WCF in a Windows 8 app to host a service—for security reasons and for portability reasons.

Creating a WinMD library

The previous sections contained some information about the WinRT architecture and the WinMD infrastructure, which enables the language projection of the Windows Runtime to make a set of APIs available to multiple programming languages. In this section, you will learn how to create a library of APIs of your own, making that library available to all other Windows 8 apps through the same projection environment used by the Windows Runtime.

Internally, the WinRT types in your component can use any .NET Framework functionality that's allowed in a Windows 8 app. Externally, however, your types must adhere to a simple and strict set of requirements:

- The fields, parameters, and return values of all the public types and members in your component must be WinRT types.
- Public structures may not have any members other than public fields, and those fields must be value types or strings.
- Public classes must be sealed (*NotInheritable* in Visual Basic). If your programming model requires polymorphism, you can create a public interface and implement that interface on the classes that must be polymorphic. The only exceptions are XAML controls.
- All public types must have a root namespace that matches the assembly name, and the assembly name must not begin with "Windows."

To verify this behavior, you need to create a new WinMD file.

To create a WinMD library, create a new project choosing the Windows Runtime Component-provided template. The project will output not only a DLL, but also a WinMD file for sharing the library with any Windows 8 app written with any language.

You must also rename the Class1.cs file in SampleUtility.cs and rename the contained class. Then, add this method to the class and the corresponding *using* statement for the *System.Text.RegularExpressions* namespace.

Sample of C# code

```
public Boolean IsEmailAddress(String email)
{
    Regex regexMail = new Regex(@"^[\w\.-]+@[^\w\.-]+\.\w{2,4}$");
    return(regexMail.IsMatch(email));
}
```

Build the project and check the output directory. You will find the classic bin/debug (or Release) subfolder containing a .winmd file for the project you create. You can open it with ILDASM to verify its content.

Add a new project to the same solution using the Blank App (XAML) template from the Visual C++ group to create a new C++ Windows Store Application.

Add a reference to the WinMD library in the Project section of the Add Reference dialog box, and then add the following XAML code in the *Grid* control:

Sample of XAML code

```
<StackPanel>
    <Button Click="ConsumeWinMD_Click" Content="Consume WinMD Library" />
</StackPanel>
```

Create the event handler for the click event in the code-behind file using the following code:

Sample of C++ code

```
void WinMDCPPConsumer::MainPage::ConsumeWinMD_Click(Platform::Object^ sender,
    Windows::UI::Xaml::RoutedEventArgs^ e) {
    auto utility = ref new WinMDCSLibrary::SampleUtility();
    bool result = utility->IsEmailAddress("paolo@devleap.com");
}
```

Build the solution and place a breakpoint in the *IsEmailAddress* method of the WinMD library, and then start the C++ project in debug mode. You might need to select Mixed (Managed and Native) in the debugging properties of the consumer project, as shown in Figure 1-11.

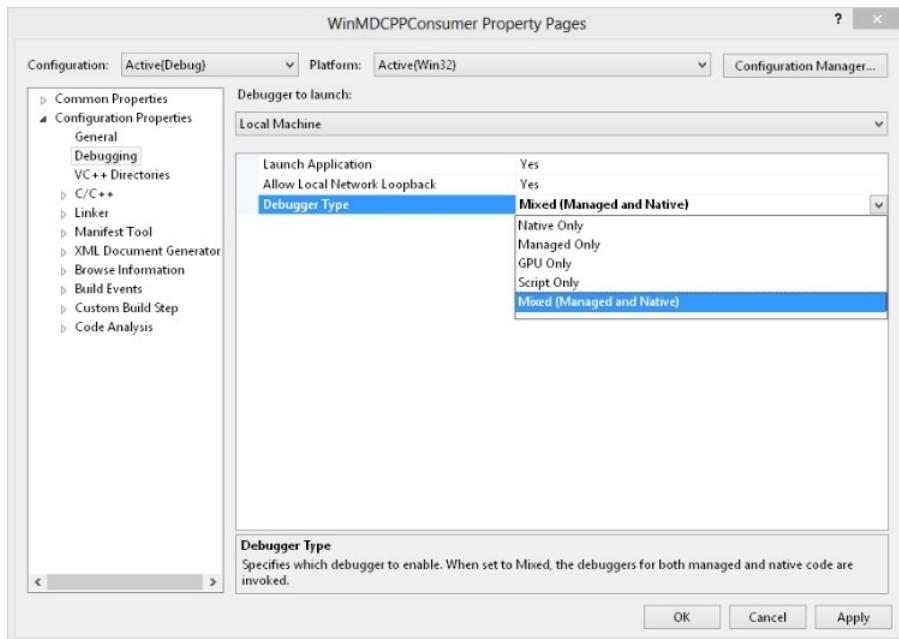


FIGURE 1-11 Debugger settings to debug mixed code

As you can verify, the debugger can step into the WinMD library from a C++ Windows Store application.

You can also verify compatibility with HTML/WinJS project creating a new project based on the Windows Store templates for JavaScript (Blank App).

Reference the WinMD library as you did in the C++ section and add an HTML button that will call the code using JavaScript:

Sample of HTML code

```
<body>
    <p><button id="consumeWinMDLibrary">Consume WinMD Library</button></p>
</body>
```

Open the default.js file, which is in the js folder of the project, and place the following event handler inside the file, just before the `app.start()` method invocation.

```
function consumeWinMD(eventInfo) {
    var utility = new WinMDCSLibrary.SampleUtility();
    var result = utility.isEmailAddress("paolo@devleap.com");
}
```

Notice that the case of the `IsEmailAddress` method, defined in C#, has been translated into `isEmailAddress` in JavaScript thanks to the language projection infrastructure provided by the Windows Runtime.

You can insert the following lines of code into the function associated with the `app.onactivated` event, just before the end of the `if` statement.

```
// Retrieve the button and register the event handler.
var consumeWinMDLibrary = document.getElementById("consumeWinMDLibrary");
consumeWinMDLibrary.addEventListener("click", consumeWinMD, false);
```

Listing 1-17 shows the complete code of the default.js file after you have made the edits.

LISTING 1-17 Complete code for the default.js file

```
// For an introduction to the Blank template, see the following documentation:
// http://go.microsoft.com/fwlink/?LinkId=232509
(function () {
    "use strict";

    WinJS.Binding.optimizeBindingReferences = true;

    var app = WinJS.Application;
    var activation = Windows.ApplicationModel.Activation;

    app.onactivated = function (args) {
        if (args.detail.kind === activation.ActivationKind.launch) {
            if (args.detail.previousExecutionState !==
                activation.ApplicationExecutionState.terminated) {
                // TODO: This application has been newly launched. Initialize
                // your application here.
            } else {
                // TODO: This application has been reactivated from suspension.
                // Restore application state here.
            }
            args.setPromise(WinJS.UI.processAll());
        }
    }

    // Retrieve the button and register our event handler.
    var consumeWinMDLibrary = document.getElementById("consumeWinMDLibrary");
```

```

        consumeWinMDLibrary.addEventListener("click", consumeWinMD, false);
    }
};

app.oncheckpoint = function (args) {
    // TODO: This application is about to be suspended. Save any state
    // that needs to persist across suspensions here. You might use the
    // WinJS.Application.sessionState object, which is automatically
    // saved and restored across suspension. If you need to complete an
    // asynchronous operation before your application is suspended, call
    // args.setPromise().
};

function consumeWinMD(eventInfo) {
    var utility = new WinMDCSLibrary.SampleUtility();
    var result = utility.isEmailAddress("paolo@devleap.com");
}

app.start();
}();

```

Place a breakpoint in the *IsEmailAddress* method or method call and start debugging, configuring Mixed (Managed and Native) for the consumer project and verifying you can step into the WinMD library.



Thought experiment

Using libraries

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

In one of your applications, you create classes that leverage some WinRT features such as a webcam, pickers, and other device-related features. You decide to create a library to let other applications use this reusable functionality.

1. Should you create a JavaScript library or a WinMD library, and why?
2. What are at least three requirements for creating a WinMD library?

Objective summary

- Visual Studio provides a template for building a WinMD library for all supported languages.
- Language projection enables you to use the syntax of the application language to use a WinMD library.
- The field, parameters, and return type of all the public types of a WinMD library must be WinRT types.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the “Answers” section at the end of this chapter.

1. What do public classes of a WinMD library have to be?
 - A. Sealed
 - B. Marked as *abstract*
 - C. Implementing the correct interface
 - D. None of the above
2. Portable classes in a WinMD library can use which of the following?
 - A. All the .NET 4.5 Framework classes
 - B. Only a subset of the C++ classes
 - C. Only WinRT classes
 - D. Only classes written in C++
3. What are possible call paths? (Choose all that apply.)
 - A. A WinJS application can instantiate a C++ WinMD class.
 - B. A C++ application can instantiate a C# WinMD class.
 - C. A C# application can instantiate a WinJS WinMD.
 - D. All of the above

Chapter summary

- Background tasks can run when the application is not in the foreground.
- Background tasks can be triggered by system, maintenance, time, network, and user events.
- A task can be executed based on multiple conditions.
- Lengthy download and upload operations can be done using transfer classes.
- The Windows Runtime lets applications written in different languages share functionality.

Answers

This section contains the solutions to the thought experiments and the answers to lesson review questions in this chapter.

Objective 1.1: Thought experiment

1. Your application, when not used by the user, is put in a suspended state by the Windows Runtime and, if the system needs resources, can be terminated. This is the most common problem that might explain why the app sometimes does not clean all data. The application cannot rely on background threads to perform operations because the application can be terminated if not in the foreground.
2. To solve the problem, you need to implement a background task using the provided classes and register the task during application launch. Because the operations are lengthy, it is important to use a deferral. Do not forget to define the declaration in the application manifest.

Objective 1.1: Review

1. **Correct answer:** C
 - A. **Incorrect:** You cannot schedule a time trigger every minute. The minimum frequency is 15 minutes. Moreover, polling is not a good technique when an event-based technique is available.
 - B. **Incorrect:** The *InternetAvailable* event fires when an Internet connection becomes available. It does not tell the application about changes in the network state.
 - C. **Correct:** *NetworkStateChange* is the correct event. The *false* value for the *oneShot* parameter enables the application to be informed every time the state of the network changes.
 - D. **Incorrect:** *NetworkStateChange* is the correct event, but the value of *true* for the *oneShot* parameter fires the event just one time.
2. **Correct answers:** A, B, C
 - A. **Correct:** The task has to be declared in the application manifest.
 - B. **Correct:** The task can be created by the *BackgroundTaskBuilder* class.
 - C. **Correct:** A trigger must be set to inform the system on the events that will fire the task.
 - D. **Incorrect:** There is no need to use a toast to enable a background task. You can use it, but this is optional.

3. Correct answer: C

- A. **Incorrect:** There is no specific task in the Windows Runtime library to fire a task just once.
- B. **Incorrect:** Every task can be scheduled to run just once.
- C. **Correct:** Many triggers offer a second parameter in the constructor to enable this feature.
- D. **Incorrect:** You can create different tasks to be run once. For example, a task based on network changes can run just once.

Objective 1.2: Thought experiment

1. A background task has network and CPU quotas. Tasks have to be lightweight and short-lived, and they cannot be scheduled to run continuously. You have to rely on the specific class of the *BackgroundTransfer* namespace to upload and download files in the background.
2. To solve the problem, you need to use the *BackgroundUploader* class and declare the use of the network in the application manifest.

Objective 1.2: Review

1. Correct answer: B

- A. **Incorrect:** You can schedule a task to run every 15 minutes.
- B. **Correct:** Fifteen minutes is the internal clock frequency. You cannot schedule a task to run at a lower interval.
- C. **Incorrect:** You can schedule a task to run every 15 minutes.
- D. **Incorrect:** You can schedule a task to run every 15 minutes.

2. Correct answer: A

- A. **Correct:** All assigned conditions need to be met to start a task.
- B. **Incorrect:** All assigned conditions must return *true*.
- C. **Incorrect:** All assigned conditions need to be met to start a task.
- D. **Incorrect:** There is no difference on condition evaluation whether the device is on AC or DC power.

3. Correct answer: B

- A. **Incorrect:** The application has no reference to background task threads.
- B. **Correct:** You need to implement the *OnCanceled* event that represents the cancellation request from the system.
- C. **Incorrect:** There is no request to abort the thread during cancellation request.
- D. **Incorrect:** The system can make a cancellation request.

4. Correct answers: B, D

- A. Incorrect:** This class has no methods to download files.
- B. Correct:** To download small resources, the *HttpClient* is the preferred class.
- C. Incorrect:** The *BackgroundTransfer* is a namespace.
- D. Correct:** The *BackgroundDownloader* is the class to request a lengthy download operation.
- E. Incorrect:** The *BackgroundUploader* class cannot download files.

Objective 1.3: Thought experiment

You can choose a traditional C# or Visual Basic library that is perfectly suited to the need to be reused by other applications. But a traditional library cannot be reused by applications written in other languages.

If you opt for a WinMD library, you can reuse the exposed features in applications written in other languages. Moreover, you can give the library functionalities that work in the background using background tasks.

Creating a WinMD library has no drawbacks. You just have to follow some simple set of requirements:

- The fields, parameters, and return values of all the public types and members in your component must be WinRT types.
- Public structures may not have any members other than public fields, and those fields must be value types or strings.
- Public classes must be sealed. If your programming model requires polymorphism, you can create a public interface and implement that interface on the classes that must be polymorphic. The only exceptions are XAML controls.
- All public types must have a root namespace that matches the assembly name, and the assembly name must not begin with "Windows."

Objective 1.3: Review

1. Correct answer: A

- A. Correct:** A class must be sealed.
- B. Incorrect:** There is no need to mark the class as abstract.
- C. Incorrect:** There is no interface required.
- D. Incorrect:** Answer choice A is correct.

2. Correct answer: C

- A. **Incorrect:** You do not have access to all the .NET 4.5 classes since they simply are not available for a Windows Store app.
- B. **Incorrect:** You cannot access C++ classes directly.
- C. **Correct:** You can access WinRT classes.
- D. **Incorrect:** WinMD library can be written in C# and Visual Basic, not only in C++.

3. Correct answers: A, B, D

- A. **Correct:** A WinJS app can access C++ classes because they are wrapped in a WinMD library.
- B. **Correct:** A C++ app can access C# classes because they are wrapped in a WinMD library.
- C. **Incorrect:** A WinMD library cannot be written in JavaScript.
- D. **Correct:** Answer choice C is incorrect.

CHAPTER 2

Discover and interact with devices

In this chapter, you learn how to capture media and audio using the provided standard user interface (UI) and from code. You also learn how to get data from sensors, such as global positioning system (GPS), and how to discover device capabilities.

Objectives in this chapter:

- Objective 2.1: Capture media with the camera and microphone
- Objective 2.2: Get data from sensors
- Objective 2.3: Enumerate and discover device capabilities

Objective 2.1: Capture media with the camera and microphone

The Windows Runtime (WinRT) provides simple application programming interfaces (APIs) to interact with a device's camera and microphone from .NET, C++, or JavaScript code. As with other WinRT APIs, you do not need references to class libraries. Microsoft Visual Studio 2012 automatically adds the .NET for Windows Store app reference when you create a new Windows Store app project.

This objective covers how to:

- Use *CameraCaptureUI* to take pictures or video, and configure camera settings
- Use *MediaCapture* to capture pictures, video, and audio
- Configure camera settings
- Set media formats
- Handle media capture events

Using *CameraCaptureUI* to capture pictures or video

This section is dedicated to the most simple and effective way to use the WinRT Webcam APIs to capture a photo or video. You need just a few lines of code to configure some settings, and you can let the system manage all the details. In the next section, you will learn how to access the webcam at a lower level.

There are two ways your application can interact with a camera:

- **By code** Leverages the WinRT APIs to gain complete control over the entire flow of operations
- **By using the provided UI** Enables the runtime to use the standard camera control and manages many of the capturing details

In other words, if you need complete control over the capturing operation, want to manage the stream directly, want to perform some operations during the recording, or simply want to create a custom UI, you need some lines of codes that interact with the *MediaCapture* APIs. On the contrary, if you just want to take a picture or a video from the webcam, you can use the provided *CameraCaptureUI* API and let the system manage all the details.

Using a standard UI from your application is important. The user will be presented with the common UI that every Windows Store app uses for capturing photos and video. Remember that the WinRT APIs can be used by any Windows Store app written in any language; the *CameraCaptureUI* class provides a consistent UI for all of them.

Add the Hypertext Markup Language (HTML) code shown in Listing 2-1 to your default.html page as a reference, which creates a button that fires the code to show the standard camera UI and a *div* tag to display the user's choice.

LISTING 2-1 Default.html with a button to capture a photo

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Webcam Test</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- Webcam Test references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <button id="btnTakePicture">Take Photo</button>
    <div id="photoMessage" />
</body>
</html>
```

The page contains a *button* control that fires the *takePicture_click* function, as shown in Listing 2-2. In addition, a *div* tag named *photoMessage* displays a message if the user decides to take a photo from the standard camera UI.

LISTING 2-2 Code to show the camera user interface

```
app.onload = function ()  
{  
    btnTakePicture.addEventListener("click", takePicture_click);  
  
}  
  
function takePicture_click(args)  
{  
    var captureUI = new Windows.Media.Capture.CameraCaptureUI();  
  
    captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)  
        .done(function (capturedItem) {  
            if (capturedItem) {  
                photoMessage.innerHTML = "User shot a photo."  
            }  
            else {  
                photoMessage.innerHTML = "User canceled the operation."  
            }  
        }, function (err) {  
            photoMessage.innerHTML = "Something went wrong";  
        });  
}
```

The first line of the *takePicture_click* function creates an instance of the *CameraCaptureUI* class. This class is responsible for managing the UI for capturing photos and videos in the Windows Runtime.

The second line calls the *CaptureFileAsync* method, which captures the stream asynchronously. This method prevents blocking the UI thread when the user is in front of the camera UI. The method accepts the *CameraCaptureUIMode* parameter, which can assume the value of *Photo*, *Video*, or *PhotoOrVideo*.

NOTE WEBCAM

In the scenario presented in this section, the webcam will be activated to take a photo.

The *CaptureFileAsync* method returns a promise that is handled by the relative *done* function. The promise receives the stream captured by the webcam represented in an instance of the *Windows.Storage.StorageFile* class.

NOTE ACCESSING FILES, STREAMS, AND ASYNCHRONOUS CALLS

You can learn more about accessing files and streams in Chapter 5, “Manage data and security,” and about async calls in Chapter 4, “Enhance the user interface.”

You can display the image taken from the webcam using a standard image tag (*img*) as shown in the following code snippet for the body of the page:

```
<body>
    <button id="btnTakePicture">Take Photo</button>
    <div>
        
        <div id="photoMessage" />
    </div>
</body>
```

The stream, represented by the file, can be opened using the *OpenAsync* method that returns an instance of the *IRandomAccessStream* interface and is then assigned to variables. A simple method binds an *IRandomAccessStream* to an *img* tag. You can transform the object in a uniform resource locator (URL) and assign the result directly to the *src* property of the image tag:

```
function takePicture_click(args) {
    var captureUI = new Windows.Media.Capture.CameraCaptureUI();
    captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo).
        done(function (capturedItem) {
            if (capturedItem) {
                var objectUrl = URL.createObjectURL(capturedItem,
                    { oneTimeOnly: true });
                document.getElementById("photo").src = objectUrl;
                photoMessage.innerHTML = "User shot a photo."
            }
            else {
                photoMessage.innerHTML = "User canceled the operation."
            }
        }, function (err) {
            photoMessage.innerHTML = "Something went wrong";
    });
}
```

If you run the application at this point and click the Capture Photo button, the webcam screen occupies the entire screen, but you cannot take a photo. The result is shown in Figure 2-1.



FIGURE 2-1 *CameraCaptureUI* used without capability declaration

The message that displays informs you that the application needs the user's permission to use the webcam. The reason is simple: you cannot use the Webcam API without declaring the Webcam capability in the application manifest. If you do not have a camera attached to your PC, the application will first ask you to connect the device. This message is part of the standard UI and is displayed in the user's language.

You need to define the use of the webcam in the application manifest by using the Visual Studio App Manifest Designer, shown in Figure 2-2. Note that Visual Studio 2012 automatically adds the Internet (Client) capability.

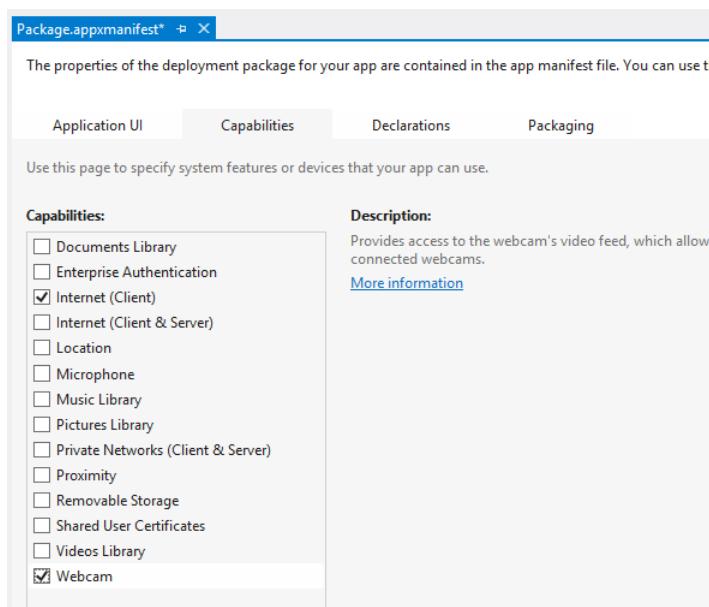


FIGURE 2-2 Selecting the Webcam capability in the Visual Studio App Manifest Designer

You can also add capabilities in the Package.appxmanifest XML file, as shown in Listing 2-3.

LISTING 2-3 The Package.appxmanifest XML file

```
<?xml version="1.0" encoding="utf-8"?>
<Package xmlns="http://schemas.microsoft.com/appx/2010/manifest">
    <Identity Name="70e52984-9c4f-4cb1-acbb-c5b1d9f17618" Publisher="CN=Roberto"
        Version="1.0.0.0" />
    <Properties>
        <DisplayName>Camera</DisplayName>
        <PublisherDisplayName>Roberto</PublisherDisplayName>
        <Logo>Assets\StoreLogo.png</Logo>
    </Properties>
    <Prerequisites>
        <OSMinVersion>6.2.1</OSMinVersion>
        <OSMaxVersionTested>6.2.1</OSMaxVersionTested>
    </Prerequisites>
    <Resources>
```

```

<Resource Language="x-generate" />
</Resources>
<Applications>
    <Application Id="App" Executable="$targetnametoken$.exe" EntryPoint="Camera.App">
        <VisualElements DisplayName="Camera" Logo="Assets\Logo.png"
            SmallLogo="Assets\SmallLogo.png" Description="Camera" ForegroundText="light"
            BackgroundColor="#464646">
            <DefaultTile ShowName="allLogos" />
            <SplashScreen Image="Assets\SplashScreen.png" />
        </VisualElements>
    </Application>
</Applications>
<Capabilities>
    <Capability Name="internetClient" />
    <DeviceCapability Name="webcam" />
</Capabilities>
</Package>

```

The file declares the Webcam device capability in the `<DeviceCapability>` node of the `<Capabilities>` section node. Although Visual Studio 2012 automatically adds the Internet (Client) capability, you can remove it if the app will not use the Internet, as is the case with our sample app.

If you run the application now, the Windows Runtime presents a dialog requesting the user's permission to use the webcam, as shown in Figure 2-3. The user can select only Allow or Block and cannot use other buttons. This is one of the standard user interfaces and message dialogs of the Windows Runtime.

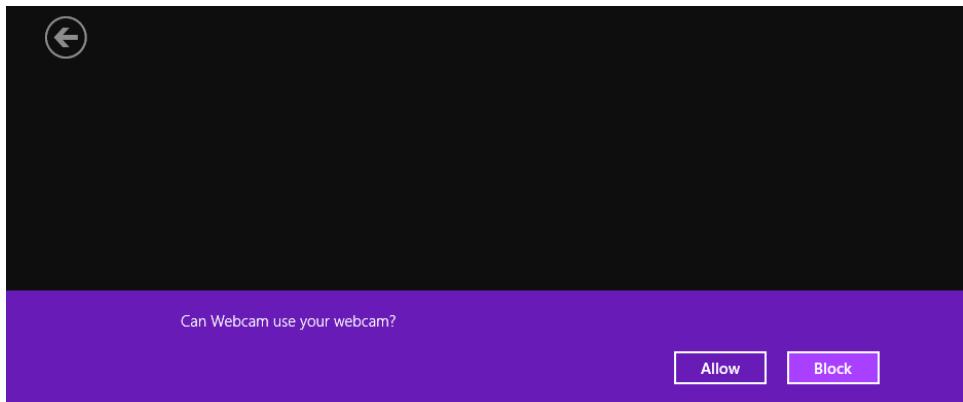


FIGURE 2-3 The standard UI for requesting user permission

To recap, the application needs to declare its capabilities in the application manifest, but the user has to provide permission to the application explicitly for each capability.

If the user blocks the webcam, the corresponding feature cannot be used. In the sample application, the webcam shows a black screen in which the user cannot do anything but click the Back button to return to the application. The system retains the user's choice forever.

However, users can remove a specific permission at any time for any application or restore a permission at any time.

You can deny the permission using the *Block* button. The webcam UI will close and the value returned to the code will be *null*. The code presented in Listing 2-1 can handle this situation, displaying a message dialog box that informs the user that no photo has been taken.

As stated, the system remembers the user's choice. If the user tries to open the camera again, the message in Figure 2-1 displays again, because the application has no permission to use the device.

To restore a permission, open the application's settings in the Windows 8 charms bar. (Move the mouse to the lower-right corner to open the charms bar and select **Settings**.) A panel appears on the right of the screen with some settings in the lower section, such as the network joined by the system, the volume level, the language, and a button to turn off/sleep/restart the system.

In the upper section of the panel, you can see the application's name, current user, version, and the permissions for the webcam (shown in Figure 2-4).

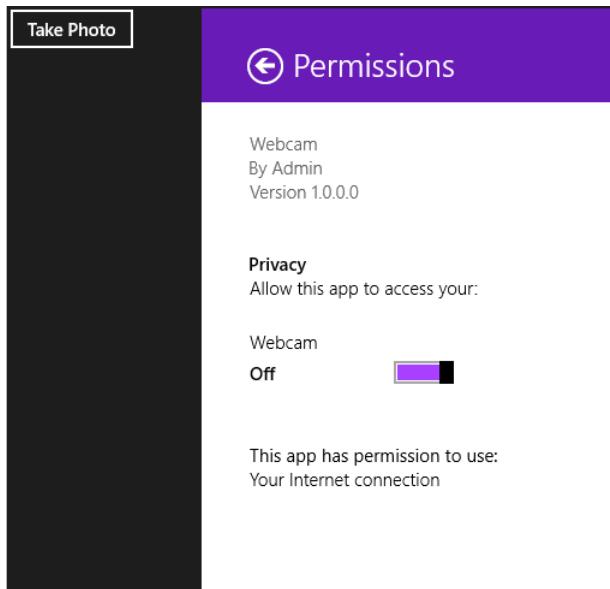


FIGURE 2-4 Settings for the application

The lower section of the pane presents the capabilities requested in the application manifest. Switching the slider to On immediately gives the application permission to use the webcam. The application presents the lens image, as shown in Figure 2-5.

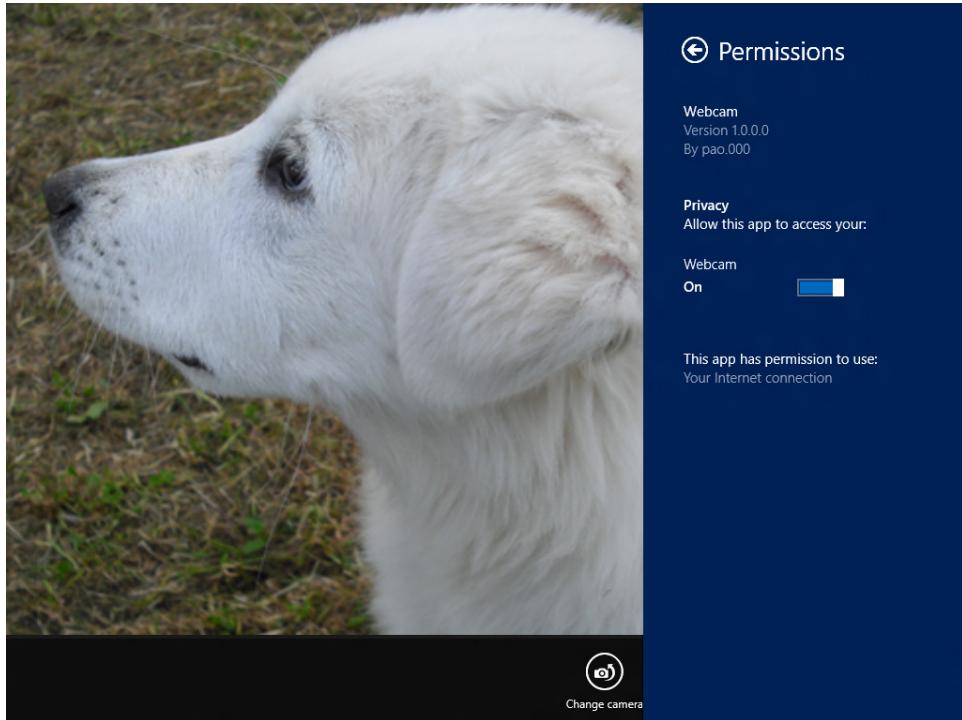


FIGURE 2-5 Standard UI for an enabled webcam

You can remove the permission at any time. If you move the slider to the Off position, the image will be blanked.

The standard UI enables you to choose some settings using the provided buttons. To take a photo, just tap the screen on a tablet or touch screen–enabled PC, or use the left mouse button directly on the image. The camera lets you crop the photo on the fly and, if you accept the shot, the image is returned to the code as an *IRandomAccessStream* interface.

The code in Listing 2-1 tests the result, and, if not *null*, the image is provided as the source for the *Image* tag.

The *CameraCaptureUI* enables you to define some initial settings to be used to take pictures. For example, the code in Listing 2-4 sets the *AllowCropping* to *true* to enable the user to crop the image on the fly, sets the image format to *Png*, and sets the maximum resolution.

LISTING 2-4 Leveraging the camera settings of the *CameraCaptureUI* class

```
function takePicture_click(args) {
    var captureUI = new Windows.Media.Capture.CameraCaptureUI();
    captureUI.photoSettings.format = Windows.Media.Capture
        .CameraCaptureUIPhotoFormat.png;
    captureUI.photoSettings.croppedAspectRatio = { width: 4, height: 3 };
    captureUI.photoSettings.allowCropping = true;
    captureUI.photoSettings.maxResolution = Windows.Media.Capture
        .CameraCaptureUIMaxPhotoResolution.highestAvailable;
```

```

captureUI.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
    .done(function (capturedItem) {
        if (capturedItem) {
            var objectUrl = URL.createObjectURL(capturedItem,
                { oneTimeOnly: true });
            document.getElementById("photo").src = objectUrl;
            photoMessage.innerHTML = "User shot a photo."
        }
        else {
            photoMessage.innerHTML = "User canceled the operation."
        }
    }, function (err) {
        photoMessage.innerHTML = "Something went wrong";
    });
}

```

The enumeration for the *CameraCaptureUIPhotoFormat* can be *Jpeg*, *JpegXR*, or *Png*. The values of the *CameraCaptureUIMaxPhotoResolution* enumeration and their descriptions are shown in Listing 2-5.

LISTING 2-5 *CameraCaptureUIMaxPhotoResolution* enumeration

```

#region Assembly Windows.winmd, v255.255.255.255
// C:\Program Files (x86)\Windows
// Kits\8.0\References\CommonConfiguration\Neutral\Windows.winmd
#endregion

using System;
using Windows.Foundation.Metadata;

namespace Windows.Media.Capture
{
    // Summary:
    //     Determines the highest resolution the user can select for capturing photos.
    [Version(100794368)]
    public enum CameraCaptureUIMaxPhotoResolution
    {
        // Summary:
        //     The user can select any resolution.
        HighestAvailable = 0,
        //

        // Summary:
        //     The user can select resolutions up to 320 X 240, or a similar 16:9
        //     resolution.
        VerySmallQvga = 1,
        //

        // Summary:
        //     The user can select resolutions up to 320 X 240, or a similar 16:9
        //     resolution.
        SmallVga = 2,
        //

        // Summary:
        //     The user can select resolutions up to 1024 X 768, or a similar 16:9
        //     resolution.
        LargeVga = 3
    }
}

```

```

        MediumXga = 3,
    //
    // Summary:
    //     The user can select resolutions up to 1920 X 1080, or a similar 4:3
    //     resolution.
    Large3M = 4,
    //
    // Summary:
    //     The user can select resolutions up to 5MP.
    VeryLarge5M = 5,
}
}

```

After a photo has been taken, you can examine the *CroppedAspectRatio* and the *CroppedSizeInPixels* properties of the *PhotoSettings* properties of the *CameraCaptureUI* instance. They tell you the aspect ratio of the cropping area used by the user and the relative size.

To record a video using the standard UI, set the parameter of the *CaptureFileAsync* method of the *CameraCaptureUI* to *CameraCaptureUIMode.Video*, as shown in the following code:

```
var captureUI = new CameraCaptureUI();
captureUI.CaptureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.video);
```

You can also set the value of the parameter to *CameraCaptureUIMode* to *PhotoOrVideo* to let the user choose between the two.

The video mode can be configured using the *VideoSettings* property of the *CameraCaptureUI* class, as follows:

```
var captureUI = new CameraCaptureUI();
captureUI.VideoSettings.allowTrimming = true;
captureUI.VideoSettings.format =
    Windows.Media.Capture.CameraCaptureUIVideoFormat.mp4;
captureUI.VideoSettings.maxDurationInSeconds = 30;
captureUI.VideoSettings.maxResolution =
    Windows.Media.Capture.CameraCaptureUIMaxVideoResolution.highDefinition;
```

The *CameraCaptureUIVideoFormat* property can be set to *Mp4* or *Wmv*, and the *MaxResolution* property can be set to a different video quality. Video resolutions are explained by the corresponding enum definition shown in Listing 2-6.

LISTING 2-6 The *CameraCaptureUIMaxVideoResolution* enum definition

```
using System;
using Windows.Foundation.Metadata;

namespace Windows.Media.Capture
{
    // Summary:
    //     Determines the highest resolution the user can select for capturing video.
    [Version(100794368)]
    public enum CameraCaptureUIMaxVideoResolution
    {
        // Summary:
        //     The user can select any resolution.
```

```

        HighestAvailable = 0,
    //
    // Summary:
    //     The user can select resolutions up to low definition resolutions.
    LowDefinition = 1,
    //
    // Summary:
    //     The user can select resolutions up to standard definition resolutions.
    StandardDefinition = 2,
    //
    // Summary:
    //     The user can select resolutions up to high definition resolutions.
    HighDefinition = 3,
}
}

```

If you also want to record the audio, you need to select the *Microphone* capability in the application manifest.

Even if the *CameraCaptureUI* class hides all the internal detail for the webcam and the relative drivers, do not forget to intercept and manage any exception you encounter working on such a device. For instance, if the webcam does not function or the driver is not correctly installed, you will receive the exception shown in Figure 2-6. You have to manage the exception using a *try/catch/finally* block for synchronous code or the exception handling code within the promises *then* and *done*. You will learn many details about handling errors in Chapter 6.

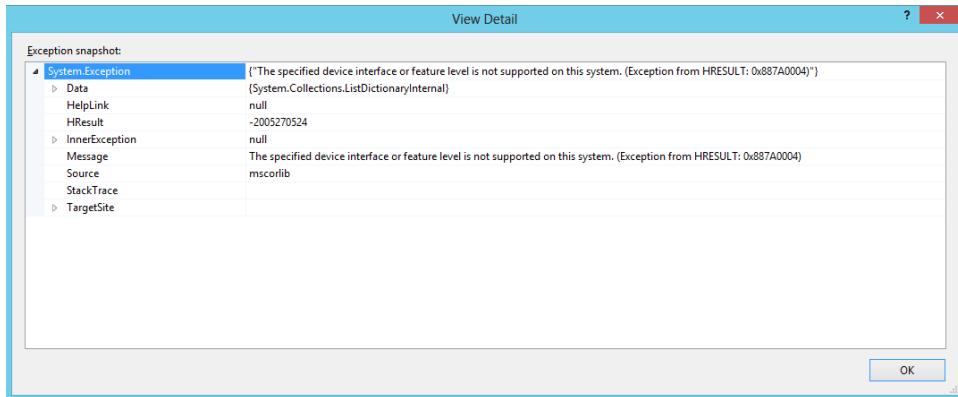


FIGURE 2-6 Exception properties for a malfunctioning device

Using *MediaCapture* to capture pictures, video, or audio

As mentioned at the beginning of this objective section, sometimes you might want more control over the flow of the capture operation. Perhaps you want to perform some sort of operation directly on the data stream or during the recording phase, or you simply want to create a custom UI rather than using the standard interface.

In this case, you can leverage the *MediaCapture* APIs (exposed by the same *Windows.Media.Capture* namespace of its companion class, *CameraCaptureUI*) to achieve more fine-grained control over the entire process of audio and video recording.

Listing 2-7 shows sample code that takes advantage of the *MediaCapture* class to display the webcam feed on the screen. Before using the sample code presented in this section, you need to add a few declarations to your application manifest. Because the sample code needs to access both the camera and the microphone, if present, the two capabilities you must declare are the Webcam and the Microphone capabilities in the *Package.appxmanifest* file of your app. Because the sample code saves the recorded video in the user's Videos Library and the picture captured in the Pictures Library, you also need to declare the corresponding capabilities as well. In the listing, note the *mediaCapture* variable definition at the page level and its use in the various methods presented.

LISTING 2-7 Using the *MediaCapture* class to display a video stream from the webcam

```
app.onload = function ()  
{  
    btnStartDevice.addEventListener("click", startDevice_click);  
}  
  
var mediaCapture;  
  
function startDevice_click(args)  
{  
    mediaCapture = new Windows.Media.Capture.MediaCapture();  
    mediaCapture.onrecordlimitationexceeded = recordLimitationExceeded;  
    mediaCapture.onfailed = mediaFailed;  
  
    mediaCapture.initializeAsync()  
        .done(null, function (err) {  
            errorMessage.innerHTML = "Something went wrong when initializing the camera.  
                Please check the permissions.";  
        });  
}  
  
function recordLimitationExceeded(args)  
{  
    errorMessage.innerHTML = "Record limitation exceeded!";  
}  
  
function mediaFailed(args)  
{  
    errorMessage.innerHTML = "Media capture failed!";  
}
```

After the *MediaCapture* class has been instantiated in the *startDevice_click* method, the code subscribes to the two events exposed by the *MediaCapture* class: the *Failed* event and the *RecordLimitationExceeded* event. The *Failed* event is raised when an error occurs during media capture (for example, because the user has revoked her permission to use the camera during the streaming). The *RecordLimitationExceeded* event is raised when the record limit is exceeded. (In Windows 8, the current record limit is three hours.)



EXAM TIP

When the *RecordLimitationExceeded* event is raised, the app is expected to finalize the file being recorded by calling the *StopRecordingAsync* method. If the file is not finalized, the capture engine will stop sending samples to the file.

The code then calls the *StartAsync* method of the *MediaCapture* class to initialize the capture device. The first time the app is executed, the *InitializeAsync* displays a consent prompt to get the user's permission to access the camera or the microphone. (For this reason, the *InitializeAsync* method should be called from the main UI thread of your app.)

After the device has been initialized, you can use the code in Listing 2-8 to start the camera stream.

LISTING 2-8 Displaying the camera stream

```
app.onload = function ()  
{  
    btnStartDevice.addEventListener("click", startDevice_click);  
    btnStartPreview.addEventListener("click", startPreview_click);  
}  
  
(code omitted)  
  
function startPreview_click(args)  
{  
    try {  
        previewVideo.src = URL.createObjectURL(mediaCapture, { oneTimeOnly: true });  
        previewVideo.play();  
    } catch (e) {  
        errorMessage.innerHTML =  
            "Something went wrong. You must start the device first";  
    }  
}
```

In the *startPreview_click* button click's event handler, the code sets the *src* property of the *video* element to point to the current *MediaCapture* object and then starts the video preview by calling the *Play* method. At this point, your app is able to display the stream from the camera on the screen, as shown in Figure 2-7.

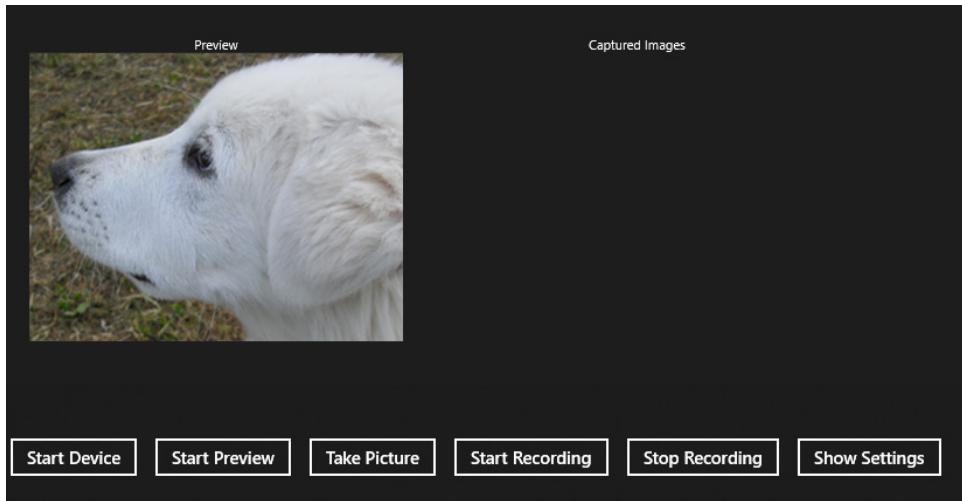


FIGURE 2-7 The camera preview

Another thing that is worth noticing is the ability to add video and audio effects to your stream by calling the *AddEffectAsync* method, as illustrated in the following revised version of Listing 2-8 (changes are in bold):

```
function startPreview_click(args){  
    try {  
        mediaCapture.addEffectAsync(Windows.Media.Capture.MediaStreamType  
            .videoRecord, Windows.Media.VideoEffects.videoStabilization, null)  
            .done(null, function (err){  
                // handle error  
            });  
  
        previewVideo.src = URL.createObjectURL(mediaCapture, { oneTimeOnly: true });  
        previewVideo.play();  
    } catch (e) {  
        errorMessage.innerHTML =  
            "Something went wrong. You must start the device first";  
    }  
}
```

The *AddEffectAsync* method is used in this sample to apply a video stabilization effect to a video preview stream. This method accepts three parameters. The first parameter is an instance of the *MediaStreamType* enum that specifies on what kind of stream you want the filter to be applied: photo, audio, video preview, or video recording. The second parameter is a string representing the effect Id: the Id of the runtime class that contains the internal calculations for applying the audio or video effect. Finally, the third parameter is represented by a dictionary implementing the *IPropertySet* interface (an empty interface at the time of this writing) and containing configuration parameters for the effect.

In the previous example, to retrieve the Id of the video stabilization effect, the code leverages the *VideoStabilization* property of the *VideoEffects* class, which returns a string with the corresponding Id. When you are done with the effect, you can invoke the *ClearEffectsAsync* method, which removes all audio and video effects from a certain stream.

MORE INFO CUSTOM EFFECTS

To browse an example of a custom video effect applied to a video stream, you can download the Windows 8 Software Development Kit (SDK) official sample at <http://code.msdn.microsoft.com/windowsapps/Media-capture-using-b65e221b>.

The next step is to add the option to take a picture from the webcam stream. Listing 2-9 shows one of several options available for your app.

LISTING 2-9 Using the *MediaCapture* class to take a picture from the webcam stream

```
app.onloaded = function () {
    btnStartDevice.addEventListener("click", startDevice_click);
    btnStartPreview.addEventListener("click", startPreview_click);
    btnTakePhoto.addEventListener("click", takePicture_click);
}

(code omitted)

function takePicture_click(args) {
    try {
        var photoStorage;
        Windows.Storage.KnownFolders.picturesLibrary.createFileAsync("snapshot",
            Windows.Storage.CreationCollisionOption.generateUniqueName)
            .then(function (newFile) {
                var photoStorage = newFile;
                var photoProperties = Windows.Media.MediaProperties
                    .ImageEncodingProperties.createJpeg();
                return mediaCapture.capturePhotoToStorageFileAsync(photoProperties,
                    photoStorage);
            })
            .done(function (result) {
                var url = URL.createObjectURL(photoStorage, { oneTimeOnly: true });
                photoImage.src = url;
            }, function (error) {
                errorMessage.innerText =
                    "Something went wrong when saving the photo";
            });
    } catch (e) {
        errorMessage.innerText =
            "Something went wrong when taking a picture.";
    }
}
```

This code takes advantage of the *CapturePhotoToStorageFileAsync* method of the *MediaCapture* class to save the photo in the user's Pictures library. (See Chapter 5, "Manage data and security," for further details on storage files and user libraries).

Figure 2-8 shows the UI with the new captured photo.

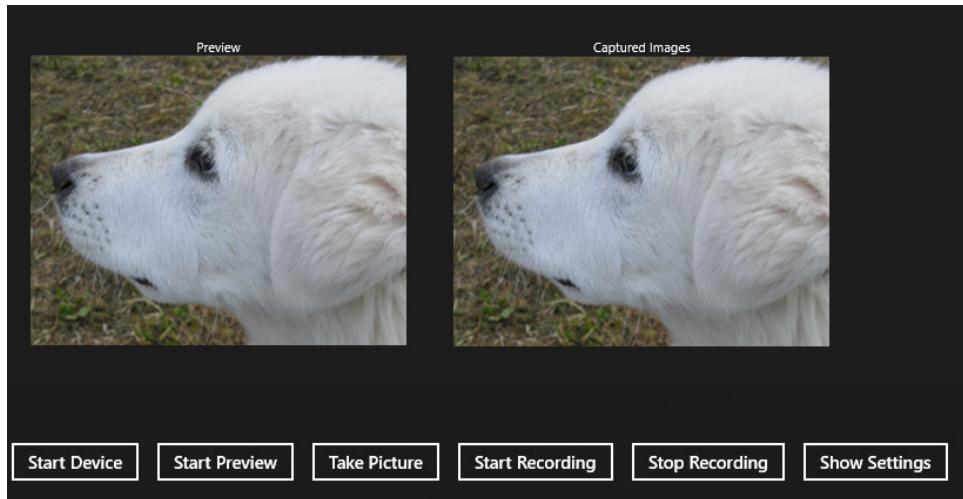


FIGURE 2-8 The captured photo

The *MediaCapture* class also exposes an overloaded version of the *InitializeAsync* method, which takes an object of type *MediaCaptureInitializationSettings*. As its name suggests, *MediaCaptureInitializationSettings* contains some initialization settings for the media capture device. You can choose, for example, among audio capture, video capture, a combined audio and video capture (through the *StreamingCaptureMode* property), or the stream source to use for the photo capture (through the *PhotoCaptureSource* property).

Listing 2-10 shows a revised version of the *startDevice_click* method presented in Listing 2-7 (changes are in bold) that takes advantage of the overloaded version of the *InitializeAsync* method.

LISTING 2-10 Initializing the media capture device

```
function startDevice_click(args) {  
  
    mediaCapture = new Windows.Media.Capture.MediaCapture();  
    mediaCapture.onrecordlimitationexceeded = recordLimitationExceeded;  
    mediaCapture.onfailed = mediaFailed;  
  
    var cameraSettings = new Windows.Media.Capture.MediaCaptureInitializationSettings();  
    cameraSettings.streamingCaptureMode =  
        Windows.Media.Capture.StreamingCaptureMode.audioAndVideo;  
    cameraSettings.photoCaptureSource =  
        Windows.Media.Capture.PhotoCaptureSource.videoPreview;  
    cameraSettings.realTimeModeEnabled = true;  
  
    mediaCapture.initializeAsync(cameraSettings)  
        .done(null, function (err) {  
            errorMessage.innerHTML = "Something went wrong initializing the camera";  
        });  
}
```

The *VideoDeviceId* and *AudioDeviceId* properties, exposed by the *MediaCaptureInitializationSettings* class, enable you to specify the device unique identification number of the video camera and the microphone, respectively, that you want to use. Listing 2-11 shows an example of their usage (changes to the code are shown in bold). You will learn details about finding and enumerating devices in Objective 2.3 of this chapter.

LISTING 2-11 Enumerating video devices available on the system

```
function startDevice_click(args) {  
  
    mediaCapture = new Windows.Media.Capture.MediaCapture();  
    mediaCapture.onrecordlimitationexceeded = recordLimitationExceeded;  
    mediaCapture.onerror = mediaFailed;  
  
    var deviceInfo = Windows.Devices.Enumeration.DeviceInformation  
        .findAllAsync(Windows.Devices.Enumeration.DeviceClass.videoCapture)  
        .done(function (devices) {  
            if (devices.length > 0) {  
                var cameraSettings = new Windows.Media.Capture  
                    .MediaCaptureInitializationSettings();  
                cameraSettings.videoDeviceId = devices[0].id;  
                cameraSettings.streamingCaptureMode =  
                    Windows.Media.Capture.StreamingCaptureMode.audioAndVideo;  
                cameraSettings.photoCaptureSource =  
                    Windows.Media.Capture.PhotoCaptureSource.videoPreview;  
                cameraSettings.realTimeModeEnabled = true;  
  
                mediaCapture.initializeAsync(cameraSettings)  
                    .done(null, function (err) {  
                        errorMessage.innerHTML =  
                            "Something went wrong when initializing the camera.  
                            Please check the permissions.";  
                    });  
            }  
            else {  
                errorMessage.innerHTML = "No camera device on the system";  
            }  
        }, function (err) {  
            // handle error  
        });  
    }  
}
```

Listing 2-12 shows the complete code to record a video from the webcam and save it in the user's Videos library (hence, the need to add the corresponding capability in the Package.appxmanifest of your app).

LISTING 2-12 Recording a video in the user's Videos library

```
app.onload = function () {  
    btnStartDevice.addEventListener("click", startDevice_click);  
    btnStartPreview.addEventListener("click", startPreview_click);  
    btnTakePhoto.addEventListener("click", takePicture_click);  
    btnStartRecording.addEventListener("click", startRecording_click);  
}  
}
```

(code omitted)

```
function startRecording_click(args) {  
  
    var profile = Windows.Media.MediaProperties.MediaEncodingProfile  
        .createMp4(Windows.Media.MediaProperties.VideoEncodingQuality.qvga);  
  
    try {  
        Windows.Storage.KnownFolders.videosLibrary.createFileAsync("sample.mp4",  
            Windows.Storage.CreationCollisionOption.generateUniqueName)  
            .then(function (file) {  
                mediaCapture.startRecordToStorageFileAsync(profile, file);  
            })  
            .done(null, function (err) {  
                errorMessage.innerHTML =  
                    "Something went wrong while recording the video.";  
            });  
    } catch (e) {  
        // handle exception  
    }  
}
```

The *startRecording_click* function calls the *CreateFileAsync* that, in turn, creates a sample file (named sample.mp4) in the user's Videos library. In the *then* promise, the code calls the *StartRecordToStorageFileAsync* method of the *MediaCapture* class to start recording the newly created file using the profile specified. The *MediaCapture* class also exposes a *StartRecordToStreamAsync* to start recording to a random-access stream as well as a *StartRecordToCustomSinkAsync* method, in two different versions, to start recording to a custom media sink.



EXAM TIP

The presented code samples use the *CreateMp4* method to create an encoding profile for an MP4 video, but the *MediaEncodingProfile* class also exposes methods to create encoding profiles for different audio and video files. These profiles include the Advanced Audio Codec (AAC) audio profile (corresponding to the M4a format), MP3 audio profile, as well as the Windows Media Audio (WMA) and Windows Media Video (WMV) profiles.

During the recording, you can adjust some basic camera settings by taking advantage of the *CameraOptionsUI* class, which exposes only a single method, *Show*, to display a flyout containing options for the capture of photos, audio recordings, and videos. The method accepts, as its only parameter, the *MediaCapture* object that the options refer to.

The following JavaScript code excerpt shows how to use it:

```
app.onloaded = function () {  
    btnStartDevice.addEventListener("click", startDevice_click);  
    btnStartPreview.addEventListener("click", startPreview_click);  
    btnTakePhoto.addEventListener("click", takePicture_click);  
    btnStartRecording.addEventListener("click", startRecording_click);  
    btnShowSettings.addEventListener("click", showSettings_click);  
}
```

```

}

(code omitted)

function showSettings_click(args) {
    if (mediaCapture != null) {
        Windows.Media.Capture.CameraOptionsUI.show(mediaCapture);
    }
}

```

The camera settings flyout is shown in Figure 2-9.

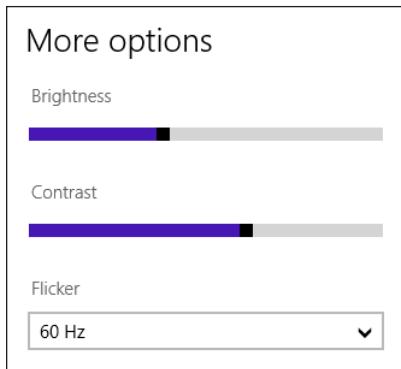


FIGURE 2-9 Camera settings flyout

Finally, to stop the video recording, call the *StopRecordAsync* method of the *MediaCapture* class, as shown in the following code snippet:

```

app.onload = function () {
    btnStartDevice.addEventListener("click", startDevice_click);
    btnStartPreview.addEventListener("click", startPreview_click);
    btnTakePhoto.addEventListener("click", takePicture_click);
    btnStartRecording.addEventListener("click", startRecording_click);
    btnShowSettings.addEventListener("click", showSettings_click);
    btnStopRecording.addEventListener("click", stopRecording_click);
}

(code omitted)

function stopRecording_click(args) {
    if (mediaCapture != null) {
        mediaCapture.stopRecordAsync().done(null, function (err) {
            // handle error
        });
    }
}

```

To test the preceding JavaScript code, you can use the HTML code shown in Listing 2-13 as a reference for your default.html page.

LISTING 2-13 The HTML definition of the default.html page

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Media Capture Sample JS</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- MediaCaptureSampleJS references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <div id="container">
        <div id="previewBox">
            <div class="caption" id="previewCaption">Preview</div>
            <video id="previewVideo" width="400" height="300" />
        </div>
        <div id="photoCapture">
            <div class="caption" id="photoCaption">Captured Images</div>
            
        </div>
        <div id="errorMessage"/>
    </div>
    <div id="buttonRow">
        <span>
            <button class="button" id="btnStartDevice">Start Device</button>
            <button class="button" id="btnStartPreview">Start Preview</button>
            <button class="button" id="btnTakePhoto">Take Photo</button>
            <button class="button" id="btnStartRecording">Start Recording</button>
            <button class="button" id="btnStopRecording">Stop Recording</button>
            <button class="button" id="btnShowSettings">Show Settings</button>
        </span>
    </div>
</body>
</html>
```

In the proposed sample, you have seen how to capture a video and take some pictures from a camera. The audio was recorded as part of the stream coming from the camera.

The usage of the *MediaCapture* class to capture audio coming from another source, typically a microphone, follows the same pattern of video and photo capturing. The JavaScript code in Listing 2-14 captures audio from the device's microphone.

LISTING 2-14 Capturing audio from the microphone

```
function startDevice_click(args) {  
  
    mediaCapture = new Windows.Media.Capture.MediaCapture();  
    mediaCapture.onrecordlimitationexceeded = recordLimitationExceeded;  
    mediaCapture.onerror = mediaFailed;  
  
    var deviceInfo = Windows.Devices.Enumeration.DeviceInformation  
        .findAllAsync(Windows.Devices.Enumeration.DeviceClass.audioCapture)  
        .done(function (devices) {  
            if (devices.length > 0) {  
                var audioSettings = new Windows.Media.Capture  
                    .MediaCaptureInitializationSettings();  
                audioSettings.audioDeviceId = devices[0].id;  
                audioSettings.streamingCaptureMode =  
                    Windows.Media.Capture.StreamingCaptureMode.audio;  
  
                mediaCapture.initializeAsync(audioSettings)  
                    .done(null, function (err) {  
                        errorMessage.innerHTML =  
                            "Something went wrong when initializing the mic.  
                            Please check the permissions.";  
                    });  
            }  
            else {  
                errorMessage.innerHTML = "No audio capture device on the system";  
            }  
        }, function (err) {  
            // handle error  
        });  
    }  
  
    function startRecording_click(args) {  
        var profile = Windows.Media.MediaProperties.MediaEncodingProfile  
            .createMp3(Windows.Media.MediaProperties.AudioEncodingQuality.high);  
  
        try {  
            Windows.Storage.KnownFolders.videosLibrary  
                .createFileAsync("sample.mp3",  
                    Windows.Storage.CreationCollisionOption.generateUniqueName)  
                .then(function (file) {  
                    mediaCapture.startRecordToStorageFileAsync(profile, file)  
                })  
                .done(null, function (err) {  
                    errorMessage.innerHTML = "Something went wrong while recording audio.";  
                });  
        } catch (e) {  
            // handle exception  
        }  
    }  
}
```



Thought experiment

Capturing pictures for a travel application

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are creating a travel application that lets the user capture pictures when visiting interesting places. The user can add audio comments to the photos, and tag the photos with text to find them more easily.

What strategy would you follow to manage the camera, shoot a photo, tag it with some text, and add audio comments?

Objective summary

- The *CameraCaptureUI* class provides a standard UI to capture pictures and/or video from a webcam.
- The *CameraCaptureUI* class enables you to define the format for the media and camera settings using the provided properties.
- Use the *MediaCapture* class, instead of the *CameraCaptureUI*, whenever you want complete control over the entire process of audio and video capturing, including customization of the UI.
- Start device initialization by calling the *StartAsync* method of *MediaCapture*, or use the overloaded version of this method if you want to supply the media capture device with one or more initialization settings.
- You can use the *CameraOptionsUI.Show* method to display some basic audio and video options in a flyout, or create your own custom controls.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. How can an application set the media format for capturing a video using the standard UI?
 - A. You cannot set the media format using the standard UI.
 - B. Use the *VideoSettings* properties of the *CameraCaptureUI* instance.
 - C. Use the *VideoSettings.MaxResolution* property of the *CameraCaptureUI* class.
 - D. Use *CameraCaptureUIMode* as a parameter of the *CaptureFileAsync* method of the *CameraCaptureUI* instance.

2. When is the *RecordLimitationExceeded* event raised?
 - A. When an error occurs during media capture
 - B. When the app does not have permission to access the capture device
 - C. When the user stops recording an audio or video stream
 - D. When the record limit is exceeded
3. Which class contains initialization settings for the *MediaCapture* object that can be passed as a parameter to the *MediaCapture.InitializeAsync* method?
 - A. *MediaCaptureInitializationSettings* class
 - B. *CameraCaptureUIPhotoFormat* class
 - C. *CameraCaptureUIVideoCaptureSettings* class
 - D. *CameraOptionsUI* class

Objective 2.2: Get data from sensors

Modern devices expose an incredible number of sensors, from web and GPS receiver to accelerometer and near field communication (NFC). In the past, interacting with these devices was challenging because you had to manage the various drivers by code or by using external libraries. The Windows Runtime aims to unify the programming model for all devices.

This objective covers how to:

- Determine the availability of a sensor (*Windows.devices.sensors*)
- Add sensor requests to the app manifest
- Handle sensor events
- Get sensor properties
- Determine location via GPS

Understanding sensors and location data in the Windows Runtime

All the APIs discussed throughout this objective are built on top of the Windows Sensor and Location platform, which greatly simplifies the integration and use of different sensors, such as accelerometers, inclinometers, light sensors, GPS sensors, and so on.

From a developer's point of view, the platform provides two namespaces for Windows Store apps: *Windows.Devices.Sensors*, which includes support for a variety of motion, device-orientation and light sensors, and the *Windows.Devices.Geolocation* namespace, which is capable of retrieving the computer's location using a location provider.

MORE INFO THE WINDOWS SENSOR LOCATION PLATFORM

For details on the platform, visit <http://msdn.microsoft.com/en-us/library/windows/hardware/gg463473.aspx>.

Accessing sensors from a Windows Store app

The *Windows.Devices.Sensors* namespace provides classes, methods, and types to access the various sensors that can be integrated in a device with Windows 8 onboard, such as accelerometer, gyrometer, compass, orientation, inclinometer, and light sensor. Each of these sensors can be accessed through simple APIs that follow similar patterns, with few variations.

First, you need to obtain a reference to the class that wraps the specific sensor you want to use by invoking the *GetDefault* static method of the sensor class. Then you can follow two strategies to retrieve the data from the sensor: polling the device at regular intervals or subscribing the *ReadingChanged* event to be notified of any update in the sensor's readings. You will learn details of these steps in the following subsections.

Responding to user movements with the accelerometer sensor

The accelerometer sensor measures the acceleration transmitted to a device along the three axes (X, Y, and Z), that is, the relationship between the speed variation and the time interval considered (expressed through the G-force acceleration). Figure 2-10 illustrates the axis orientation in a tablet and in a notebook.

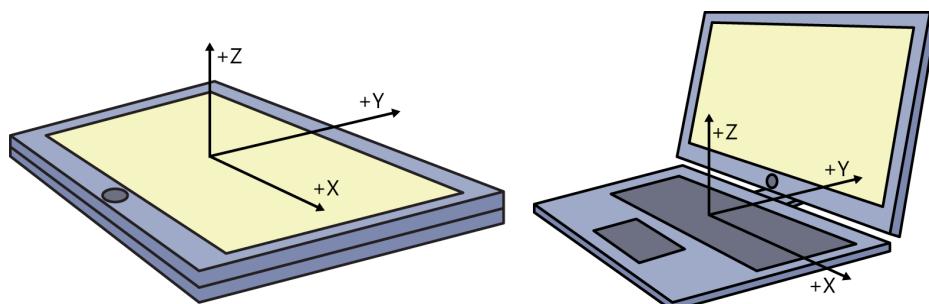


FIGURE 2-10 The position of the X, Y, and Z axes for a tablet (left) and a notebook (right)

Source: Adapted from the MSDN article, "Motion and device orientation for simple apps (Windows Store apps)," at <http://msdn.microsoft.com/en-us/library/windows/apps/jj155767.aspx>

To leverage the accelerometer from a Windows Store app, the first thing to do is to get a reference to an *Accelerometer* object by calling the *GetDefault* static method of the *Accelerometer* class, as follows:

```
var sensor = Windows.Devices.Sensors.Accelerometer.getDefault();
if (sensor == null) {
    // No accelerometer found
}
```



EXAM TIP

The `GetDefault` method returns a value only if an accelerometer is detected. A `null` value means the requested sensor is not available in the system, so it is important to check for the returning value before going any further.

The simple UI that presents accelerometer data can be similar to the one presented in the following HTML code excerpt:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Accelerometer Sensor Sample JS</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- AccelerometerSensorJS references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <div id="sensorResult" />
</body>
</html>
```

Listing 2-15 shows how to initialize the accelerometer sensor.

LISTING 2-15 Initializing the accelerometer sensor

```
app.onloaded = initializeSensor;

var sensor;

function initializeSensor() {
    sensor = Windows.Devices.Sensors.Accelerometer.getDefault();
    if (sensor == null) {
        sensorResult.innerText = "No sensor detected";
        return;
    }

    var minimumReportInterval = sensor.minimumReportInterval;
    var desiredInterval = minimumReportInterval > 16 ? minimumReportInterval : 16;
    sensor.reportInterval = desiredInterval;
}
```

The `ReportInterval` property indicates the time interval (expressed in milliseconds) between two readings. The report interval must be set to a non-zero value before registering an event handler or calling the `GetCurrentReading` method, to inform the sensors (through their drivers) to allocate the resources needed to satisfy the application's requirements.

To improve battery efficiency, this property should explicitly be set to its default value by setting the property to zero as soon as the app no longer needs the sensor. By setting this property to zero, you are telling the sensor to use its default report interval. The next code excerpt illustrates this point. (The same principle applies to all the sensors discussed in this section.)

```
app.onunload = function () {
    if (sensor != null) {
        sensor.reportInterval = 0;
    }
}
```

Before setting the report interval, you should check whether the sensor can keep up with the requested interval by calling the *MinimumReportInterval* property. If you set a value that is lower than the minimum supported interval, the code will either trigger an exception or get unexpected results. The code shown in Listing 2-15 checks whether the minimum supported interval is lower than 16 milliseconds and, in this case, sets 16 milliseconds as the desired report interval to avoid too many readings (hence saving power battery).

However, consider that even if you set a valid report interval, or you set it back to its default value (by setting the property to zero), the sensor might still use a different report interval, based on its internal logic.

NOTE SENSOR CHANGE SENSITIVITY

The Sensor platform automatically adjusts the sensitivity of the accelerometer sensor based on the current report interval. As the interval increases, the force that must be applied to the device to trigger a new reading must increase as well. The following table, repurposed from the official MSDN documentation, specifies the change sensitivity values for given intervals.

Current report interval (in milliseconds)	Change sensitivity (in G-force)
1 to 16	0.01
17 to 32	0.02
33 or greater	0.05

After initializing the sensor, there are three different ways to read the sensor's data: implement event-driven readings, define a polling strategy, or wait for a shake. The first two patterns are commonly used by all the sensors discussed in this section, whereas the last one is specifically supported only by the accelerometer sensor.

The first pattern leverages the *ReadingChanged* event, which is raised every time the accelerometer reports a new sensor reading. The code in Listing 2-16 shows an example of its usage (changes to Listing 2-15 are shown in bold).

LISTING 2-16 Leveraging the event-driven reading pattern through the *ReadingChanged* event

```
var sensor;

function initializeSensor() {
    sensor = Windows.Devices.Sensors.Accelerometer.getDefault();
    if (sensor == null) {
        sensorResult.innerHTML = "<p>No sensor detected</p>";
        return;
    }

    var minimumReportInterval = sensor.minimumReportInterval;
    var desiredInterval = minimumReportInterval > 16 ? minimumReportInterval : 16;
    sensor.reportInterval = desiredInterval;

    sensor.onreadingchanged = readingChangedEventHandler;
}

function readingChangedEventHandler(args) {
    sensorResult.innerHTML =
        "<p> Acceleration X = " + args.reading.accelerationX.toFixed(2) +
        "- Acceleration Y = " + args.reading.accelerationY.toFixed(2) +
        "- Acceleration Z = " + args.reading.accelerationZ.toFixed(2) + "</p>";
}
```

In the *ReadingChanged* event handler, you can inspect the *Reading* property (of type *AccelerationReading*) exposed by the *AccelerometerReadingChangedEventArgs* instance and received as a parameter to retrieve the values that represent the G-force acceleration along the three axes. The values are stored in the *AccelerationX*, *AccelerationY*, and *AccelerationZ* properties. The fourth property of the *AccelerationReading* class is *Timestamp*, which, as its name suggests, indicates the time at which the sensor reported the reading.

The second method to retrieve the data coming from the sensor is to poll the sensor device at regular intervals. Before polling the sensor, the application must set a desired value for the *ReportInterval* property. Listing 2-17 shows how to poll the sensor to retrieve the current reading (changes to Listing 2-15 are in bold).

LISTING 2-17 Polling the sensor at regular interval for the current reading

```
var sensor;

function initializeSensor() {
    sensor = Windows.Devices.Sensors.Accelerometer.getDefault();
    if (sensor == null) {
        sensorResult.innerHTML = "<p>No sensor detected</p>";
        return;
    }

    var minimumReportInterval = sensor.minimumReportInterval;
    var desiredInterval = minimumReportInterval > 16 ? minimumReportInterval : 16;
    sensor.reportInterval = desiredInterval;

    window.setInterval(pollAccelerometerReading, desiredInterval);
}
```

```

function pollAccelerometerReading() {
    var reading = sensor.getCurrentReading();
    if (reading != null) {
        sensorResult.innerHTML =
            "<p> Acceleration X = " + reading.accelerationX.toFixed(2) +
            " - Acceleration Y = " + reading.accelerationY.toFixed(2) +
            " - Acceleration Z = " + reading.accelerationZ.toFixed(2) + "</p>";
    }
}

```

The third method of detecting the user's movements through the accelerometer sensor relies on a different event. Besides the already explained *ReadingChanged* event, the *Accelerometer* class exposes a *Shaken* event that is raised by the system when the user shakes the device. When subscribing to the *Shaken* event, you do not get detailed readings of the acceleration along the three axes, nor any other indication about the G-force impressed by the user to the device. In this case, it is up to the system to determine whether a fast shake motion has occurred by analyzing the sequence of acceleration changes. Notice that, unlike the other two methods, setting a report interval prior to registering for the *Shaken* event is not required. Listing 2-18 shows a revised sample of the *InitializeSensor* method used in Listing 2-15 (changes in bold).

LISTING 2-18 Leveraging the *Shaken* event to detect movements

```

var sensor;

function initializeSensor() {
    sensor = Windows.Devices.Sensors.Accelerometer.getDefault();
    if (sensor == null) {
        sensorResult.innerHTML = "<p>No sensor detected</p>";
        return;
    }
    sensor.onshaken = accelerometer_shaken;
}

var numberOfShakens = 0;

function accelerometer_shaken(args) {
    numberOfShakens++;
    shakenCounter.innerHTML = "<p>Number of shakens: " + numberOfShakens.toString()
        + "</p>";
}

```

The HTML code in Listing 2-19 is a revised version of the default.html page that you can use as a reference for your default.html page to keep track of how many times the *Shaken* event has been raised.

LISTING 2-19 The HTML definition of the default.html page used in this sample

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Accelerometer Sensor Sample JS</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- AccelerometerSampleJS references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <div id="sensorResult" />
    <div id="shakenCounter">
        <p>Shake me!</p>
    </div>
</body>
</html>
```

Measuring angular velocity with the gyrometer

The gyrometer sensor measures the angular velocity along the three axes. Its usage is very similar to the accelerometer sensor. First, you get a reference to a *Gyrometer* object by calling the *GetDefault* static method of the *Gyrometer* class. Then you can subscribe to the *ReadingChanged* event to be notified of any change in the current angular velocity, or poll the sensor at regular intervals through the *GetCurrentReading* method.

You can use the following HTML code as a reference for your default.html page:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Gyrometer Sensor Sample JS</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- GyrometerSampleJS references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <div id="gyrometerResult"></div>
</body>
</html>
```

Listing 2-20 uses the *ReadingChanged* event handler to display the current reading values.

LISTING 2-20 Getting data from the gyrometer sensor by subscribing to the *ReadingChanged* event

```
var sensor;
app.onloaded = initializeSensor;

app.onunload = function () {
    if (sensor != null) {
        sensor.reportInterval = 0;
    }
}

function initializeSensor() {
    sensor = Windows.Devices.Sensors.Gyrometer.getDefault();
    if (sensor == null) {
        gyrometerResult.innerHTML = "<p>No sensor detected</p>";
        return;
    }
    var minimumReportInterval = sensor.minimumReportInterval;
    var desiredInterval = minimumReportInterval > 16 ? minimumReportInterval : 16;
    sensor.reportInterval = desiredInterval;

    sensor.onreadingchanged = gyrometerReadingChanged;
}

function gyrometerReadingChanged(args) {
    gyrometerResult.innerHTML =
        "<p> Angular velocity X = " + args.reading.angularVelocityX.toFixed(2) +
        " - Angular velocity Y = " + args.reading.angularVelocityY.toFixed(2) +
        " - Angular velocity Z = " + args.reading.angularVelocityZ.toFixed(2) + "</p>";
}
```

NOTE SENSOR CHANGE SENSITIVITY

The Sensor platform automatically sets the change sensitivity for the gyrometer sensor based on the current report interval. The following table, repurposed from the official MSDN documentation, specifies the change sensitivity values for given intervals.

Current report interval (in milliseconds)	Change sensitivity (degrees per second)
1 to 16	0.1
17 to 32	0.5
33 or greater	1.0

Alternatively, you can interrogate the sensor at regular intervals, as shown in Listing 2-21, which is similar to Listing 2-15. However, the code in Listing 2-21 uses a polling strategy (changes in bold):

LISTING 2-21 Polling the gyrometer sensor at regular interval

```
function initializeSensor() {
    sensor = Windows.Devices.Sensors.Gyrometer.getDefault();
    if (sensor == null) {
        gyrometerResult.innerHTML = "<p>No sensor detected</p>";
        return;
    }

    var minimumReportInterval = sensor.minimumReportInterval;
    var desiredInterval = minimumReportInterval > 16 ? minimumReportInterval : 16;
    sensor.reportInterval = desiredInterval;

    window.setInterval(pollGyrometerReading, desiredInterval);
}

function pollGyrometerReading() {
    var reading = sensor.getCurrentReading();
    if (reading != null) {
        gyrometerResult.innerHTML =
            "<p> Angular velocity X = " + args.reading.angularVelocityX.toFixed(2) +
            " - Angular velocity Y = " + args.reading.angularVelocityY.toFixed(2) +
            " - Angular velocity Z = " + args.reading.angularVelocityZ.toFixed(2) +
            "</p>";
    }
}
```

Retrieving compass data

The compass sensor indicates the heading in degrees relative to magnetic north and, depending on the implementation of the sensors, geographic (or true) north.

Use the following HTML code as a reference for your default.html page to test the compass sensor on your device:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Compass Sensor Sample JS</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- Compass Sensor Sample JS references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <div id="magneticNorthResult"></div>
    <div id="trueNorthResult"></div>
</body>
</html>
```

As for the other sensors, the first thing to do is to obtain a reference to the default compass through the *GetDefault* static method of the *Compass* class. Then you need to set the *ReportInterval* property to let the system know how many resources need to be allocated.

After the sensor has been initialized, you have the usual two ways to collect data from the sensor: adopting an event-based strategy or defining a polling technique to get data from the sensor at some interval.

Listing 2-22 shows an example of the first strategy for retrieving the values (in degrees) of the *HeadingMagneticNorth* and *HeadingTrueNorth* properties of the *CompassReading* class.

LISTING 2-22 Retrieving direction from the compass sensor by subscribing the *ReadingChanged* event

```
var sensor;
app.onloaded = initializeSensor;

app.onunload = function () {
    if (sensor != null) {
        sensor.reportInterval = 0;
    }
}

function initializeSensor() {
    sensor = Windows.Devices.Sensors.Compass.getDefault();
    if (sensor == null) {
        magneticNorthResult.innerHTML = "<p>No sensor detected</p>";
        return;
    }

    var minimumReportInterval = sensor.minimumReportInterval;
    var desiredInterval = minimumReportInterval > 16 ? minimumReportInterval : 16;
    sensor.reportInterval = desiredInterval;

    sensor.onreadingchanged = compassReadingChanged;
}

function compassReadingChanged(args) {
    magneticNorthResult.innerHTML = "<p>Magnetic North: " +
        args.reading.headingMagneticNorth.toFixed(2) + " degrees</p>";

    if (args.reading.headingTrueNorth != null)
        trueNorthResult.innerHTML = "<p>True North: " +
            args.reading.headingTrueNorth.toFixed(2) + " degrees</p>";
    else
        trueNorthResult.innerHTML = "True North: no data available";
}
```



EXAM TIP

Because not all of the sensors implement the ability to detect geographic (or true) north, it is important to check the corresponding *HeadingTrueNorth* property before using it.

NOTE SENSOR CHANGE SENSITIVITY

The Sensor platform automatically sets the change sensitivity for the compass sensor based on the current report interval. The following table, taken from the official MSDN documentation, specifies the change sensitivity values for given intervals.

Current report interval (in milliseconds)	Change sensitivity (in degrees)
1 to 16	0.01
17 to 32	0.5
33 or greater	2

Combining different data using the orientation sensor

The orientation sensor combines data coming from three different sensors to determine the orientation of the device: the accelerometer, the gyrometer, and the compass. The orientation sensor includes two different APIs, represented by the *SimpleOrientationSensor* class and the *OrientationSensor* class.

The *SimpleOrientationSensor* class uses the orientation sensor to detect the current quadrant orientation of the device. As its name suggests, the *SimpleOrientationSensor* provides an easy and intuitive way to determine the orientation of a device without the need to analyze complex data.

The detected orientation can assume one of the following values, expressed by the *SimpleOrientation* enum:

- **NotRotated** The device is not rotated. This corresponds to portrait-up orientation.
- **Rotated90DegreesCounterclockwise** The device is rotated 90 degrees counter-clockwise. This corresponds to landscape-left orientation.
- **Rotated180DegreesCounterclockwise** The device is rotated 180 degrees counter-clockwise. This corresponds to portrait-down orientation.
- **Rotated270DegreesCounterclockwise** The device is rotated 270 degrees counter-clockwise. This corresponds to landscape-right orientation.
- **Faceup** The device is positioned faceup (the display is visible to the user).
- **Facedown** The device is positioned facedown (the display is hidden from the user).

Using the *SimpleOrientationSensor* class follows the same patterns of the other sensors discussed in this section, with few differences.

The following code snippet shows the HTML definition of the default.html page that you can use as a reference to test the *SimpleOrientationSensor*.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>SimpleOrientation Sensor Sample JS</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- SimpleOrientationSensorSampleJS references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <div id="simpleOrientationResult">Rotate the display to start</div>
</body>
</html>
```

The code in Listing 2-23 displays the current orientation by leveraging the *SimpleOrientationSensor* class.

LISTING 2-23 Determining the device orientation through the *SimpleOrientationSensor* class

```
var sensor;

app.onloaded = initializeSensor;

app.onunload = function () {
    if (sensor != null) {
        sensor.reportInterval = 0;
    }
}

function initializeSensor() {

    sensor = Windows.Devices.Sensors.SimpleOrientationSensor.getDefault();
    if (sensor == null) {
        simpleOrientationResult.innerHTML = "<p>No sensor detected</p>";
        return;
    }

    sensor.onorientationchanged = simpleOrientationReadingChanged;
}

function simpleOrientationReadingChanged(args) {
    switch (args.orientation) {
        case Windows.Devices.Sensors.SimpleOrientation.notRotated:
            simpleOrientationResult.innerHTML =
                "<p>Current orientation: Portrait-Up</p>";
            break;
    }
}
```

```

        case Windows.Devices.Sensors.SimpleOrientation
            .rotated180DegreesCounterclockwise:
                simpleOrientationResult.innerHTML = "<p>Current orientation: Portrait-Down";
                break;
        case Windows.Devices.Sensors.SimpleOrientation.faceup:
            simpleOrientationResult.innerHTML = "<p>Current orientation: Face-Up";
            break;
        case Windows.Devices.Sensors.SimpleOrientation.facedown:
            simpleOrientationResult.innerHTML = "<p>Current orientation: Face-Down</p>";
            break;
        case Windows.Devices.Sensors.SimpleOrientation.rotated90DegreesCounterclockwise:
            simpleOrientationResult.innerHTML =
                "<p>Current orientation: Landscape-Left";
            break;
        case Windows.Devices.Sensors.SimpleOrientation
            .rotated270DegreesCounterclockwise:
            simpleOrientationResult.innerHTML =
                "<p>Current orientation: Landscape-right";
            break;
    }
}

```

The API hides most of the internal mechanisms used to determine the current orientation by combining complex data from three different sensors. Compared to the other sensors discussed in this section, the main difference is that, in the case of the *SimpleOrientationSensor* class, you do not have to set the report interval before subscribing to the *OrientationChanged* event. The *OrientationChanged* event replaces the *ReadingChanged* event commonly used by the other sensors, and it is raised whenever the user rotates the devices in a different position. Alternatively, you can leverage the polling strategy to interrogate the sensor at regular intervals.

The second API at your disposal is represented by the *OrientationSensor* class, which enables more fine-grained control over the data collected by the orientation sensor. The *OrientationSensor* is generally used for games and other apps that need to calculate the camera view based on screen orientation, such as those dealing with augmented reality. Listing 2-24 shows how to use an event-based strategy with the *OrientationSensor* class.

LISTING 2-24 Retrieving data from the *OrientationSensor* by leveraging the *ReadingChanged* event

```

var sensor;
app.onloaded = initializeSensor;

app.onunload = function () {
    if (sensor != null) {
        sensor.reportInterval = 0;
    }
}

function initializeSensor() {

    sensor = Windows.Devices.Sensors.OrientationSensor.getDefault();
    if (sensor == null) {
        // No sensor detected
    }
}

```

```

        return;
    }

    var minimumReportInterval = sensor.minimumReportInterval;
    var desiredInterval = minimumReportInterval > 16 ? minimumReportInterval : 16;
    sensor.reportInterval = desiredInterval;

    sensor.onreadingchanged = orientationReadingChanged;
}

function orientationReadingChanged(args) {

    var quaternion = args.reading.quaternion;
    var rotationMatrix = args.reading.rotationMatrix;

    // adjust the view displayed on screen based on rotation and orientation
}

```

In Listing 2-24, the pattern of the *OrientationSensor* is identical to the one followed by the other sensors discussed in this section, including the name of the event raised when a new reading is available and the need to set the *ReportInterval* property before subscribing to the event. The *Reading* property (of type *OrientationSensorReading*) exposed by the *OrientationSensorReadingChangedEventArgs* received as a parameter by the event handler encapsulates two properties: *RotationMatrix* and *Quaternion*. Rotation matrices and quaternions are used for describing mathematically the rotation of the device in the space. These mathematical objects can be used to adjust the scene rendered on the screen according to the device's movements.

NOTE ADVANCED MATHEMATICS

Illustrating the mathematical nature and usage of matrices and quaternions in game development and computer graphics are beyond the scope of this book.

Getting data from the inclinometer sensor

An inclinometer sensor can determine the rotation angles of a device around the three axes. The rotation around the vertical axis (Y) is also known as "roll" (or "gamma"), whereas the term "pitch" (or "beta") indicates the rotation around the lateral axis (X). The term "yaw" (or "alpha") indicates the rotation around the longitudinal axis (Z). Figure 2-11 shows the orientation of the three axes in a device running on Windows 8.

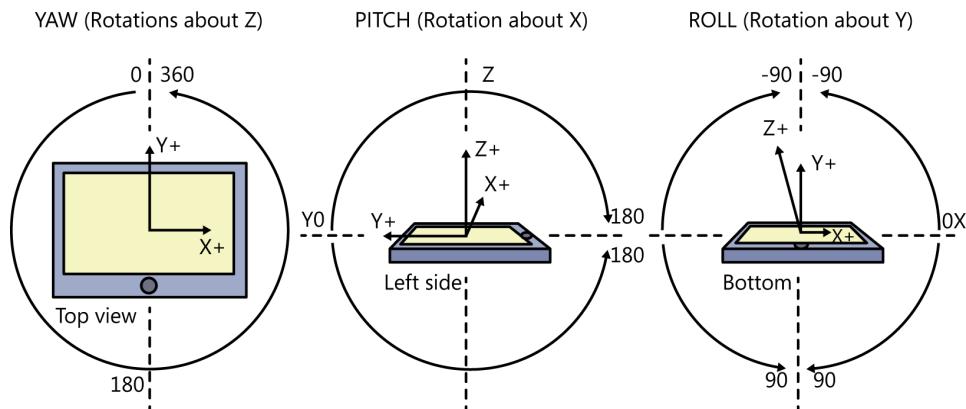


FIGURE 2-11 Yaw, pitch, and roll rotations on a tablet device

Source: Adapted from the MSDN article "Motion and device orientation for simple apps (Windows Store apps)," at <http://msdn.microsoft.com/en-us/library/windows/apps/jj155767.aspx>

To test the inclinometer on your device, you can use the following HTML code as a reference for your default page:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Inclinometer Sensor Sample JS</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- InclinometerSensorSampleJS references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <div id="inclinometerResult" />
</body>
</html>
```

Listing 2-25 shows an example of usage of the *Inclinometer* class that leverages the polling strategy to retrieve data from the sensor at regular intervals and displays the corresponding values of the *PitchDegrees*, *YawDegrees*, and *RollDegrees* properties.

LISTING 2-25 Polling the inclinometer to retrieve data about the inclination around the three axes

```
var sensor;

app.onloaded = initializeSensor;

app.onunload = function () {
    if (sensor != null) {
        sensor.reportInterval = 0;
    }
}
function initializeSensor() {

    sensor = Windows.Devices.Sensors.Inclinometer.getDefault();

    if (sensor == null) {
        inclinometerResult.innerHTML = "<p>No sensor detected</p>";
        return;
    }

    var minimumReportInterval = sensor.minimumReportInterval;
    var desiredInterval = minimumReportInterval > 16 ? minimumReportInterval : 16;
    sensor.reportInterval = desiredInterval;

    window.setInterval(pollInclinometerReading, desiredInterval);
}

function pollInclinometerReading() {
    var reading = sensor.getCurrentReading();
    if (reading != null) {
        inclinometerResult.innerHTML =
            "<p>" +
            "Pitch (X) rotation: " + reading.pitchDegrees.toFixed(2) + " | " +
            "Yaw (Z) rotation: " + reading.yawDegrees.toFixed(2) + " | " +
            "Roll (Y) rotation: " + reading.rollDegrees.toFixed(2) +
            "</p>";
    }
}
```

NOTE SENSOR CHANGE SENSITIVITY

The Sensor platform automatically sets the change sensitivity for the inclinometer sensor based on the current report interval. The following table, repurposed from the official MSDN documentation, specifies the change sensitivity values for given intervals.

Current report interval (in milliseconds)	Change sensitivity (in degrees)
1 to 16	0.01
17 to 32	0.5
33 or greater	2

Using the light sensor

A Windows Store app can use the light sensor to detect and respond to changes in ambient lighting. Changing the contrast between the background and the font used to render the content, for example, improves the readability of the content. If you want to measure ambient light through the light sensor of your device, you can use the following HTML code as a reference for your default.html page:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Light Sensor Sample JS</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- Light Sensor Sample JS references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <div id="lightResult">Illuminance in lux...</div>
</body>
</html>
```

The measurement of ambient lighting is expressed in lux and can be accessed through the *IlluminanceLux* property of the *LightSensorReading* class, as shown in Listing 2-26.

NOTE LUX AND LUMEN

A lux is a unit of measure of the intensity (as perceived by the human eye) of light that hits or passes through a surface. Lux is equal to one lumen per square meter. A lumen represents the intensity of visible light emitted by a source.

LISTING 2-26 Determining the intensity of the ambient light through the light sensor

```
var sensor;

app.onloaded = initializeSensor;

app.onunload = function () {
    if (sensor != null) {
        sensor.reportInterval = 0;
    }
}

function initializeSensor() {
    sensor = Windows.Devices.Sensors.LightSensor.getDefault();
    if (sensor == null) {
```

```

        lightResult.innerHTML = "<p>No sensor detected</p>";
        return;
    }

    var minimumReportInterval = sensor.minimumReportInterval;
    var desiredInterval = minimumReportInterval > 16 ? minimumReportInterval : 16;
    sensor.reportInterval = desiredInterval;

    sensor.onreadingchanged = lightReadingChanged;
}

function lightReadingChanged(args) {
    lightResult.innerHTML =
        "<p>Illuminance in lux: " + args.reading.illuminanceInLux + "</p>";
}

```

NOTE SENSOR CHANGE SENSITIVITY

The Sensor platform automatically sets the change sensitivity for ambient light sensors based on the current report interval. The following table, repurposed from the official MSDN documentation, specifies the change sensitivity values for given intervals.

Current report interval (in milliseconds)	Change sensitivity (percentage)
1 to 16	1%
17 to 32	1%
33 or greater	5%

Determining the user's location

The Windows Runtime can determine the user's location in two ways. The first way is to get the information from the Windows Location Provider, which uses Wi-Fi triangulation and Internet Protocol (IP) address data to determine the user's location. The second and more precise way to determine the user's location is to leverage a GPS sensor, if present.

NOTE LOCATION PROVIDER

A location provider is a hardware device or software that generates geographic data to determine the location of a computer or device.

It is up to the Location API to determine the most accurate location sensor for a given scenario. When the Windows Location Provider and GPS both exist on the same system and are providing data, the Location API will use the sensor with the most accurate data. In most cases, the GPS will be more accurate, and its data will be passed to the application. From a developer's viewpoint, however, the internal mechanisms used by the system to determine the user's location are completely transparent. Just use the same Location API and let the system determine the right strategy to retrieve the data from all available sources.

If you are developing Windows Store apps on a traditional desktop PC, you should not expect to get accurate data about your location. Determining location through Wi-Fi triangulation can give you only coarse-grained results. According to the official MSDN documentation, locations calculated from Wi-Fi data are accurate to within 350 meters in urban areas. If no Wi-Fi networks are available, the Windows Location Provider uses IP address resolution to get approximate location with an accuracy of 50 kilometers.

To test your app with more accurate data, you can use the Windows 8 Simulator. The simulator enables you to set geographic data such as longitude, latitude, altitude, and accuracy, which will be used as a simulated location to test your app's behavior. Figure 2-12 shows the Set Location dialog box in the Windows 8 Simulator.

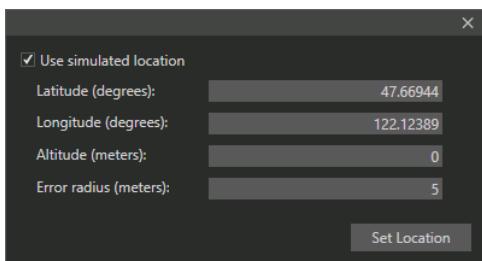


FIGURE 2-12 The Windows 8 Simulator showing the dialog box to use a simulated location

In Windows 8, location providers and GPS sensors are considered “sensitive devices” because they provide information that’s potentially harmful to the user’s privacy. For this reason, before accessing the Location API programmatically, you have to add the corresponding Location capability in the Package.appxmanifest file of your app, as shown in the Figure 2-13.

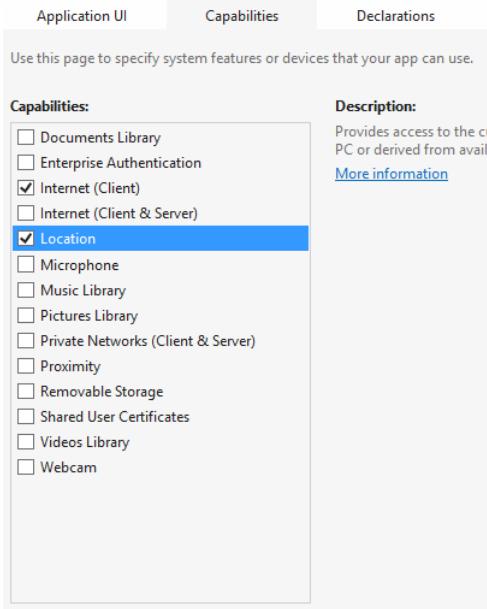


FIGURE 2-13 The App Manifest Designer with the Location capability enabled

The first time your app needs to access the user's location, a dialog appears asking for the user's permission to use the Location API. If a user denies permission to access the GPS sensor, make sure that your app is able to deal with this scenario gracefully.

MORE INFO PERMISSIONS

Chapter 6, Objective 6.2, "Design for error handling," provides details about user permissions for device access.

Retrieving geographic data

The `Windows.Devices.Geolocation` namespace contains all the classes, types, and methods that your app needs to access the computer's geographic location. The `Geolocator` class is responsible for retrieving data about the user's current location.

Listing 2-27 shows the HTML code for the default.html page that you can use as a reference to test the code presented in this section.

LISTING 2-27 The HTML definition of the default page

```
<!DOCTYPE html>

<html>
<head>
    <meta charset="utf-8" />
    <title>Geolocator Sample JS</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- Geolocator Sample JS references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <div class="buttonRow">
        <span>
            <button class="button" id="btnTrackPosition">Track Position</button>
            <button class="button" id="btnGetLocation">Get Location</button>
        </span>
    </div>
    <div id="coordinates">
        <div id="errorMessage"></div>
        <div id="sensorStatus"></div>
        <div id="latitude"></div>
        <div id="longitude"></div>
        <div id="accuracy"></div>
        <div id="timestamp"></div>
    </div>
</body>
</html>
```

Listing 2-28 shows how to use the *Geolocator* class to determine the user's current position.

LISTING 2-28 Retrieving the user's current location

```
var geolocator;

app.onloaded = function () {
    geolocator = new Windows.Devices.Geolocation.Geolocator();
    btnTrackPosition.addEventListener("click", trackPosition_click);
    btnGetLocation.addEventListener("click", getLocation_click);
}

function getLocation_click(args) {
    try {
        geolocator.desiredAccuracy = Windows.Devices.Geolocation.PositionAccuracy.high;
        geolocator.getGeopositionAsync().then(displayPosition, function (err) {
            errorMessage.innerHTML = "<p>Something went wrong when retrieving the
                location. Check the permission for the app</p>";
        });
    } catch (ex) {
```

```

        errorMessage.innerHTML =
            "<p>Something went wrong when retrieving the location</p>";
    }
}

function displayPosition(position) {
    latitude.innerHTML = "<p>Latitude: " + position.coordinate.latitude + "</p>";
    longitude.innerHTML = "<p>Longitude: " + position.coordinate.longitude + "</p>";
    accuracy.innerHTML = "<p>Accuracy: " + position.coordinate.accuracy + "</p>";
    timestamp.innerHTML = "<p>Timestamp: " + position.coordinate.timestamp + "</p>";
}

```

After instantiating a *Geolocator* object, you can optionally leverage the *DesiredAccuracy* property to indicate the accuracy level at which the *Geolocator* provides location updates. You should set the *DesiredAccuracy* property to *High* only if your app requires the most accurate data available. Otherwise, set it to *Default* to save battery life. Consider, however, that setting the *DesiredAccuracy* property does not guarantee improvement of the accuracy of data, which depends on several other factors.

To retrieve the user's current location, the only thing you have to do is to call the *GetGeopositionAsync* method using the *async/await* pattern. The method returns a *Geoposition* instance containing the required data. More specifically, the *Geoposition* class contains two kinds of data concerning the user's location.

The first type of data is exposed through the *Coordinate* property (of type *Geocoordinate*). This data represents geographic coordinates, such as longitude and latitude, and other data associated with the geographic location, such as altitude (expressed in meters above sea level), current heading (in degrees relative to true north), speed (meters per second), accuracy of the measurements, and a timestamp indicating the time at which the location was determined. The availability of some of this data, such as altitude and speed, depends on the implementation of the GPS device, so you should check for *null* values before using them.

NOTE WINDOWS LOCATION PROVIDER LIMITATIONS

The Windows Location Provider does not provide information about heading, speed, or altitude; only information about latitude and longitude coordinates and accuracy is provided.

The *Geoposition* class also exposes a *CivicAddress* property, which represents the civic address data associated with a geographic location. However, this information is not available unless a Civic Address provider has been installed. If no Civic Address provider is installed, the API returns the regional information accessible through Control Panel. Windows 8 does not include a Civic Address provider (as of this writing), nor are third-party providers available at this time. If your app needs to translate geographic coordinates into civic addresses, consider taking advantage of external services, such as the Bing Maps Geocode service.

NOTE BING MAPS GEOCODE SERVICE

For further information about the Bing Maps Geocode service, visit <http://msdn.microsoft.com/en-us/library/cc966793.aspx>.

The *Geolocator* class also exposes an overloaded version of the *GetGeopositionAsync* that accepts two *TimeSpan* objects as parameters. The first parameter indicates the maximum acceptable age of cached location data, whereas the second parameter represents the time-out to complete the operation.



EXAM TIP

According to the official MSDN documentation, if the *GetGeopositionAsync* method is called and the *Geolocator* instance cannot find any sensor within the next seven seconds, the call will time out. As a result, the *StatusChanged* event handler will be called, and the *PositionStatus* property of the *StatusChangedEventArgs* passed to the handler will be *NoData*.

Tracking the user's position

Besides determining the user's current location, you can use the *Geolocator* class to track the user's movements through the *PositionChanged* event. This event is raised every time the location provider detects a change in the user's geographical location. Listing 2-29 shows how to leverage the *PositionChanged* event to track the user's movements and display updated information on the screen.

LISTING 2-29 Tracking user position

```
function trackPosition_click(args) {
    try {
        geolocator.desiredAccuracy =
            Windows.Devices.Geolocation.PositionAccuracy.default;
        geolocator.movementThreshold = 5.0;

        geolocator.addEventListener("positionchanged", geolocatorPositionChanged);
        geolocator.addEventListener("statuschanged", geolocatorStatusChanged);

        geolocator.getGeopositionAsync()
            .then(displayPosition, function (err) {
                errorMessage.innerHTML = "<p>Something went wrong when
                    retrieving the location. Check the permission for the app</p>";
            });
    } catch (e) {
        errorMessage.innerHTML =
            "<p>Something went wrong when retrieving the location</p>";
    }
}
```

```

function geolocatorPositionChanged(args) {
    displayPosition(args.position);
}

function geolocatorStatusChanged(args) {
    if (args.status != null) {
        switch (args.status) {
            case Windows.Devices.Geolocation.PositionStatus.ready:
                sensorStatus.innerHTML = "<p>Geolocation: Ready</p>";
                break;
            case Windows.Devices.Geolocation.PositionStatus.initializing:
                sensorStatus.innerHTML = "<p>Geolocation: Initializing</p>";
                break;
            case Windows.Devices.Geolocation.PositionStatus.noData:
                sensorStatus.innerHTML = "<p>Geolocation: No data</p>";
                break;
            case Windows.Devices.Geolocation.PositionStatus.disabled:
                sensorStatus.innerHTML = "<p>Geolocation: Disabled</p>";
                break;
            case Windows.Devices.Geolocation.PositionStatus.notInitialized:
                sensorStatus.innerHTML = "<p>Geolocation: Not initialized</p>";
                break;
            case Windows.Devices.Geolocation.PositionStatus.notAvailable:
                sensorStatus.innerHTML = "<p>Geolocation: Not available</p>";
                break;
            default:
                sensorStatus.innerHTML = "<p>Geolocation: Unknown</p>";
                break;
        }
    }
}

```

If your app does not track even the smallest change in the user's position, you can leverage the *MovementThreshold* property to set the distance of required movement, in meters, for the location provider to raise the *PositionChanged* event (the default value for this property is zero).

The code in Listing 2-29 also shows how to leverage the *StatusChanged* event, which is raised when the ability of the *Geolocator* to provide location data changes. The status of the *Geolocator* is expressed by the *LocationStatus* property of the *PositionChangedEventArgs* supplied to the event handler as parameter. The *LocationStatus* property can assume one of the following values (expressed by the *PositionStatus* enum):

- **Ready** Location data is available.
- **Initializing** The location provider is initializing, looking for the required number of satellites in view to obtain an accurate position.
- **NoData** No location data is available from any location provider. After data becomes available, the *LocationStatus* property will change from the *NoData* state to the *Ready* state.

- **Disabled** The user has not granted the application permission to access location. Make sure your app can handle this kind of scenario gracefully. For further details about handling device errors, refer to Chapter 6, Objective 6.2, “Handling device capability errors” subsection.
- **NotInitialized** Neither the `GetGeopositionAsync` method has been called nor has an event handler for the `PositionChanged` event been registered yet.
- **NotAvailable** The Windows Sensor and Location platform is not available on this system.

The current status of the location provider can be checked at any time by inspecting the `LocationStatus` property of the `Geolocator` class.

Figure 2-14 shows the sample app running in the Windows 8 Simulator.

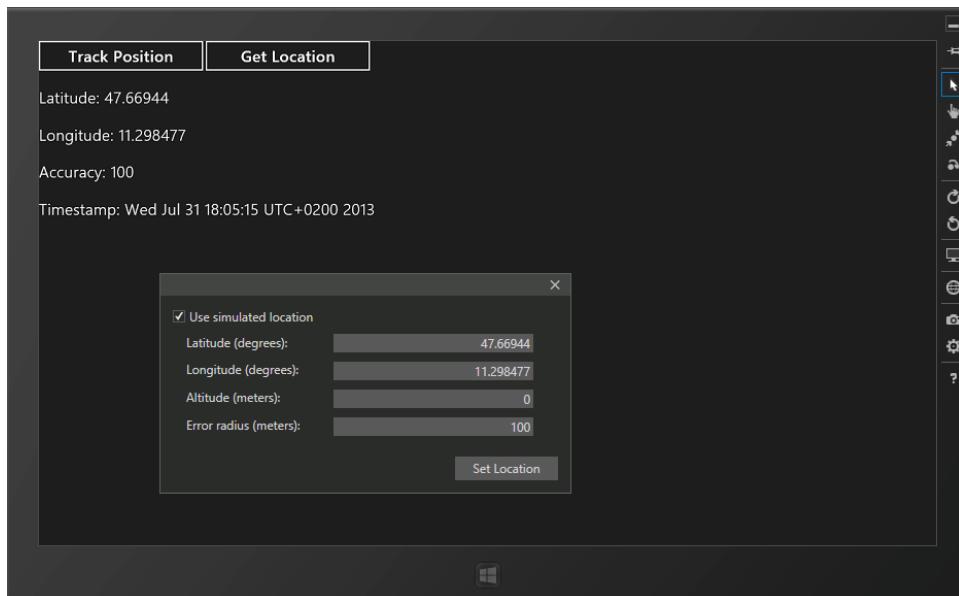


FIGURE 2-14 The sample app in the Windows 8 Simulator, showing the coordinates of a simulated location



Thought experiment

Determining the current heading

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are developing an app for hikers and trekkers. The app keeps track of the user's movements by recording a track that can be traced back to the original location at a later time. The app leverages a device's GPS sensor and uses the GPS data to determine the current heading. However, if the user is not moving, the data coming from the GPS sensor is not precise enough to display the direction the user is currently facing. For example, the user might stop walking and turn her head—and her device—from right to left and vice versa, trying to figure out which direction to take.

1. What kind of sensor could you leverage to gain more precise information about the direction in which the user is currently pointing the device, even if she is not moving?
2. How would you use the data to improve the user experience?

Objective summary

- The `Windows.Devices.Sensors` namespace includes support for a variety of sensors: accelerometer, gyrometer, compass, orientation, inclinometer, and light sensors.
- All corresponding APIs use similar patterns. First, you get a reference to the sensor through the `GetDefault` static method of the corresponding sensor class (such as `Accelerometer`, `Gyrometer`, `Compass`, and so on). Then you set the `ReportInterval` property of the sensor to let the system know how many resources need to be allocated for the sensor. (This step is not required for the `SimpleOrientationSensor` class.)
- To retrieve data from the sensor, you can follow an event-driven approach and subscribe to one of the events provided by the sensor class to notify when a new reading is available.
- Most sensor classes expose an almost identical `ReadingChanged` event, whereas the `Acceleration` class also exposes a `Shaken` event. The `SimpleOrientationSensor` exposes an `OrientationChanged` event.
- Alternatively, you can opt for a polling strategy, interrogating the sensor at regular intervals through the `GetCurrentReading` method exposed by all sensor classes (with the exception of the `SimpleOrientationSensor` class, which exposes a `GetCurrentOrientation` method).

- The *Windows.Devices.Geolocation* namespace exposes a Location API that enables retrieving the user's location through a location provider, which uses Wi-Fi triangulation and IP address data to determine the position, or a more precise GPS sensor, when available. The *Geolocator* class enables you to take advantage of these functionalities while hiding most of the inner work.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

- Which sensor class does not expose any *ReportInterval*?
 - Accelerometer* class
 - Gyrometer* class
 - Inclinometer* class
 - SimpleOrientationSensor* class
- What kind of data is provided by the *Gyrometer* class?
 - The acceleration of the device along the three axes
 - The rotation angles of a device around the three axes
 - The quadrant orientation of the device
 - The heading in degrees relative to magnetic north
- What information is provided by the Windows Location Provider based on Wi-Fi triangulation and IP data analysis?
 - Latitude
 - Altitude
 - Heading
 - Speed

Objective 2.3: Enumerate and discover device capabilities

Sometimes you need to query a system to retrieve the set of available devices. You might need to check whether certain devices are connected and enabled, or whether more devices of the same kind, such as microphones and cameras, are available at the same moment (and therefore let the user decide which device to use). The *DeviceInformation* class, exposed by the *Windows.Devices.Enumeration* namespace, provides two ways of enumerating devices,

represented by two static methods that can be used to retrieve information about the devices available on the system.

The first way leverages the *FindAllAsync* static method to perform a one-time search for available devices. This option is best suited for Windows Store apps that do not need to be notified when one of the existing devices changes or is removed from the system, or when new devices are added. The second way relies on the *DeviceWatcher* class, which not only enumerates all available devices but also raises specific events any time the device collection changes. In this section, you learn how to implement both strategies.

This objective covers how to:

- Discover the capabilities and properties of available devices, for example, GPS, removable storage, accelerometer, and near field communication

Enumerating devices

The code in Listing 2-30 uses the *FindAllAsync* method to retrieve all available devices. More precisely, the method returns a *Windows.Devices.Enumeration.DeviceInformationCollection* object, which represents a collection of *DeviceInformation* instances, each of which allows accessing device properties. Then, for each device in the collection, the code displays some of the information gathered, such as the device's name and unique ID, the thumbnail image that represents the device, and the device's glyph (the graphic symbol associated with that particular type of device). The last two pieces of information, thumbnail and glyph, can be retrieved asynchronously through two specific methods named *GetThumbnailAsync* and *GetGlyphThumbnailAsync*, respectively.

LISTING 2-30 Retrieving the collection of available devices through the *FindAllAsync* method

```
app.onloaded = function (args) {
    btnEnumerate.addEventListener("click", deviceEnumeration_click);
}

function deviceEnumeration_click() {

    Windows.Devices.Enumeration.DeviceInformation.findAllAsync()
        .done(function (devCollection) {

            var numDevices = devCollection.length;

            for (var i = 0; i < numDevices; i++) {
                displayDeviceInterface(devCollection[i], i);
            }
        },
        function (err) {
            // handle error
        });
}
```

```

function displayDeviceInterface(deviceInterface, index) {
    results.innerHTML +=
        "<h3>" + deviceInterface.name + "</h3>" +
        "<p>Device ID: " + deviceInterface.id + "</p>" +
        "<p>Thumbnail: <img class=\"thumbnail\" id=\"thumbnail" + index + "\" /></p>" +
        "<p>Glyph Thumbnail: <img class=\"glyph\" id=\"glyph" + index + "\" /></p>";

    deviceInterface.getThumbnailAsync().done(function (thumbnail) {
        document.getElementById("thumbnail" + index).src =
            window.URL.createObjectURL(thumbnail, { oneTimeOnly: true });
    });

    deviceInterface.getGlyphThumbnailAsync().done(function (glyph) {
        document.getElementById("glyph" + index).src =
            window.URL.createObjectURL(glyph, { oneTimeOnly: true });
    });
}

```

Listing 2-31 includes the HTML definition of the default.html page that you can use as a reference to test the presented code.

LISTING 2-31 The complete HTML definition of the default page for this sample

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Enumerating Devices Sample JS</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- EnumeratingDevicesJS references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <div id="container">
        <button class="button" id="btnEnumerate">Enumerate devices</button>
        <div id="results"></div>
    </div>
</body>
</html>

```

If you now run the app, you should see a result similar to Figure 2-15. (Your device enumeration will most likely be different.)



FIGURE 2-15 Enumerating the available devices

The *DeviceInformation* class exposes various overloaded versions of the *FindAllAsync* method that enables you to specify what type of device you are looking for. The first version accepts as a parameter a *DeviceClass* object that indicates the type of devices to enumerate. The *DeviceClass* enum can assume one of the following values:

- **All** Indicates that the user wants to enumerate all the devices
- **AudioCapture** Indicates that the user wants to enumerate all audio capture devices
- **AudioRender** Indicates that the user wants to enumerate all audio rendering devices
- **PortableStorageDevice** Indicates that the user wants to enumerate all portable storage devices
- **VideoCapture** Indicates that the user wants to enumerate all video capture devices

The following JavaScript code excerpt shows a revised version of the code presented in Listing 2-30 that enumerates only the portable storage devices present on the system (changes are in bold):

```
function deviceEnumeration_click() {
    Windows.Devices.Enumeration.DeviceInformation
        .findAllAsync(Windows.Devices.Enumeration.DeviceClass.portableStorageDevice)
        .done(function (devCollection) {
            var numDevices = devCollection.length;
            for (var i = 0; i < numDevices; i++) {
                displayDeviceInterface(devCollection[i], i);
            }
        })
}
```

```
        },
        function (err) {
            // handle error
        });
}
```

If you run the app, you should see a result similar to Figure 2-16.

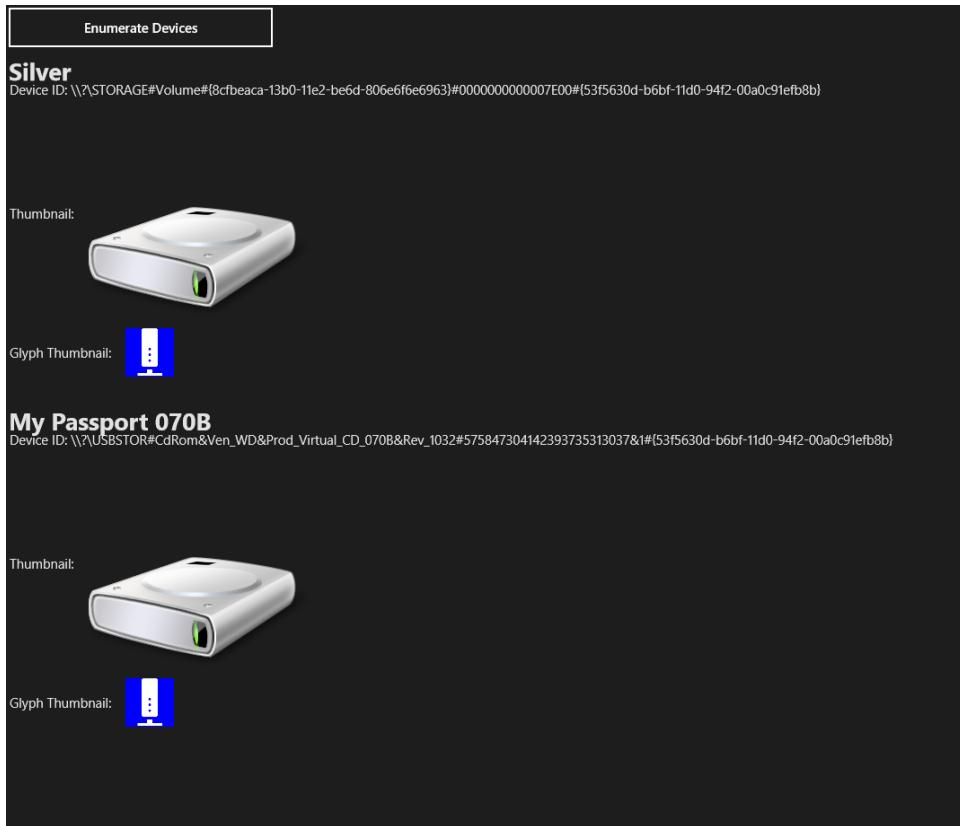


FIGURE 2-16 Enumerating only the storage devices

The second overloaded version of the *FindAllAsync* method accepts an Advanced Query Syntax (AQS) string as a parameter, a particular query syntax that Windows uses internally to refine and narrow search parameters, and, as a second parameter, an array of strings to specify additional properties for the desired device.

MORE INFO THE ADVANCED QUERY SYNTAX (AQS)

You can find more information on this topic at [http://msdn.microsoft.com/en-us/library/windows/desktop/bb266512\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/bb266512(v=vs.85).aspx).

You can use the AQS string to specify the device interface class implemented by the device you are looking for. In fact, any driver of a physical, logical, or virtual device must supply a name that uniquely identifies the device. Starting with Windows 2000, device drivers make use of a device interface class, which represents a way of exposing internal functionalities to other system components. Each device interface class is associated with a globally unique identifier (GUID).

NOTE RETRIEVING SPECIFIC SELECTORS

Some WinRT APIs expose a helper method that enables you to retrieve the ID of specific types of devices. For example, the *Windows.Networking.Proximity.ProximityDevice* class enables near field communication (NFC), in which two computers are connected within a certain range without the need for a network using Wi-Fi. The class provides a *GetDeviceSelector* method that returns the selector for that specific sensor, as shown in the following snippet:

```
var selector =
    Windows.Networking.Proximity.ProximityDevice.getDeviceSelector();
```

The following code snippet shows a revised version of Listing 2-30 that uses the AQS string to query for all devices implementing the device interface class for printers (represented by the GUID enclosed in curly brackets). For now, the code just passes *null* as second parameter.

```
function deviceEnumeration_click() {

    var selector =
        "System.Devices.InterfaceClassGuid:={0ECEF634-6EF0-472A-8085-5AD023ECBCCD}""";
    Windows.Devices.Enumeration.DeviceInformation.findAllAsync(selector, null)
        .done(function (devCollection) {

            var numDevices = devCollection.length;

            for (var i = 0; i < numDevices; i++) {
                displayDeviceInterface(devCollection[i], i);
            }
        },
        function (err) {
            // handle error
        });
}
```

If you run the app, you should see a result similar to Figure 2-17.



FIGURE 2-17 Enumerating only the devices implementing the printer interface class

The second parameter of the *FindAllAsync* method accepts, besides an ASQ string, an array of strings that enables you to specify additional properties you want to retrieve from the devices you are looking for. By default, when retrieved by the *FindAllAsync* method (or by the *CreateWatcher* method discussed in the next section), a *DeviceInfo* object contains only a little information encapsulated in five properties.



EXAM TIP

The properties exposed by the *DeviceInfo* class correspond to the ID and the name of the device (accessible through, respectively, the *Id* and *Name* properties of the *DeviceInfo* class), whether the device is enabled (represented by the *IsEnabled* property), whether the device is also the default device for certain operations (corresponding to the *IsDefault* property), and a path to the physical device location (*EnclosureLocation* property).

To retrieve more information about the device, provide the *FindAllAsync* method with an array of strings, as shown in the following code snippet. The code specifically asks to retrieve two more properties: the container ID of the device and a Boolean value indicating whether the device is enabled.

```
function deviceEnumeration_click() {  
  
    var selector =  
        "System.Devices.InterfaceClassGuid:=\\"{0ECEF634-6EF0-472A-8085-5AD023ECBCCD}\\"";  
    var properties = new Array("System.Devices.ContainerId",  
        "System.Devices.InterfaceEnabled");  
  
    Windows.Devices.Enumeration.DeviceInformation.findAllAsync(selector, properties)  
        .done(function (devCollection) {  
  
            var numDevices = devCollection.length;  
  
            for (var i = 0; i < numDevices; i++) {  
                displayDeviceInterface(devCollection[i], i);  
            }  
        },  
        function (err) {  
            // handle error  
        });  
}
```

MORE INFO FINDALLASYNC AND CREATEWATCHER METHODS

The complete list of properties that can be retrieved through the *FindAllAsync* method (or the *CreateWatcher* method discussed in the next section), visit <http://msdn.microsoft.com/en-us/library/windows/apps/hh464997.aspx>. Some of these properties refer to devices, some to device interfaces, and others to device containers.

Using the *DeviceWatcher* class to be notified of changes to the device collection

The *DeviceWatcher* class is responsible for enumerating devices dynamically, raising specific events every time devices are added, removed, or changed after the initial enumeration is completed. The events exposed by the *DeviceWatcher* class are:

- **Added** Raised when a device is added to the collection enumerated by the *DeviceWatcher* class
- **EnumerationCompleted** Raised when the enumeration of devices is completed
- **Removed** Raised when a device is removed from the collection of enumerated devices
- **Stopped** Raised when the enumeration operation has been stopped
- **Updated** Raised when a device is updated in the collection of enumerated devices

To test the code presented in this section, you can use the HTML code in Listing 2-32 as a reference for your app's default page.

LISTING 2-32 The HTML definition of the default.html page used in this sample

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Device Watcher Sample JS</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- DeviceWatcherSample references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <div id="content">
        <div id="buttonRow">
            <button id="btnStartWatcher">Start Watcher </button>
            <button id="btnStopWatcher">Stop Watcher</button>
        </div>
        <div id="deviceCounter"></div>
        <div id="watcherStatus"></div>
    </div>
</body>
</html>
```

To enumerate devices dynamically, you must first obtain a reference to a *DeviceWatcher* object by calling the *CreateWatcher* static method of the *DeviceInformation* class. Then, after subscribing to one or more events exposed by the *DeviceWatcher* object, you can begin the search for devices by calling the *Start* method.

NOTE ADDING FILTERS TO THE QUERY FOR THE DEVICE COLLECTION

The *CreateWatcher* method presents three overloaded versions that allow filtering and narrowing of devices being enumerated and under observation by the watcher. These overloaded versions use the same parameters as the *FindAllAsync* discussed in the previous section. For example, the following code adds removable storage devices to the device enumeration:

```
var watcher = new Windows.Devices.Enumeration.DeviceInformation
    .createWatcher(Windows.Devices.Enumeration.DeviceClass.portableStorageDevice);
```

During the initial enumeration, the *DeviceWatcher* raises an *Added* event for each device found until the enumeration is complete and the *EnumerationCompleted* event is raised. From this point on, the watcher continues to raise *Added*, *Removed*, and *Updated* events every time a device is added, removed, or updated, respectively.

When you no longer need to be notified of any change in the device collection, you can stop the watcher by calling the *Stop* method.

Listing 2-33 shows an example of its usage.

LISTING 2-33 Dynamically enumerating devices by using a *DeviceWatcher* object

```
var watcher;
var counter = 0;

app.onloaded = function (args) {
    btnStartWatcher.addEventListener("click", startWatcher_click);
    btnStopWatcher.addEventListener("click", stopWatcher_click);

    watcher = new Windows.Devices.Enumeration.DeviceInformation
        .createWatcher(Windows.Devices.Enumeration.DeviceClass.portableStorageDevice);

    watcher.onadded = watcher_added;
    watcher.onupdated = watcher_updated;
    watcher.onremoved = watcher_removed;

    watcher.onenumerationcompleted = watcher_enumerationCompleted;

    watcherStatus.innerHTML =
        "<p>The Device Watcher has been created. Click to start.</p>";
}

function startWatcher_click(args) {
    if (watcher.status != Windows.Devices.Enumeration.DeviceWatcherStatus.started &&
        watcher.status != Windows.Devices.Enumeration.DeviceWatcherStatus.stopping) {
        try {
            watcher.start();
            watcherStatus.innerHTML += "<p>The Device Watcher has been started.</p>";
        } catch (e) {
            //handle exception
        }
    }
}

function watcher_enumerationCompleted(args) {
    watcherStatus.innerHTML += "<p>Enumeration completed.</p>";
    deviceCounter.innerHTML = "<p>" + counter +
        " removable storage devices found on your system.</p>";
}

function watcher_added(args) {
    if(args != null){
        var name = args.properties["System.ItemNameDisplay"];
        counter++;
        watcherStatus.innerHTML += "<p>The following device has been added: " + name + "</p>";
    }
}
```

```

        + args.name + "</p>";
    deviceCounter.innerHTML = "<p>" + counter +
        " removable storage devices found on your system.</p>";
}
}

function watcher_updated(args) {
    watcherStatus.innerHTML += "<p>A removable storage device has been updated.</p>";
}

function watcher_removed(args) {
    counter--;
    watcherStatus.innerHTML += "<p>A removable storage device has been removed.</p>";
    deviceCounter.innerHTML = "<p>" + counter +
        " removable storage devices found on your system.</p>";
}

function stopWatcher_click(args) {
    if (watcher.status != Windows.Devices.Enumeration.DeviceWatcherStatus.stopped &&
        watcher.status != Windows.Devices.Enumeration.DeviceWatcherStatus.stopping) {
        watcher.stop();
        watcherStatus.innerHTML += "<p>The Device Watcher has been stopped</p>";
    }
}

```

Figure 2-18 shows the sample app reacting to changes in the collection of removable storage devices connected to the system.

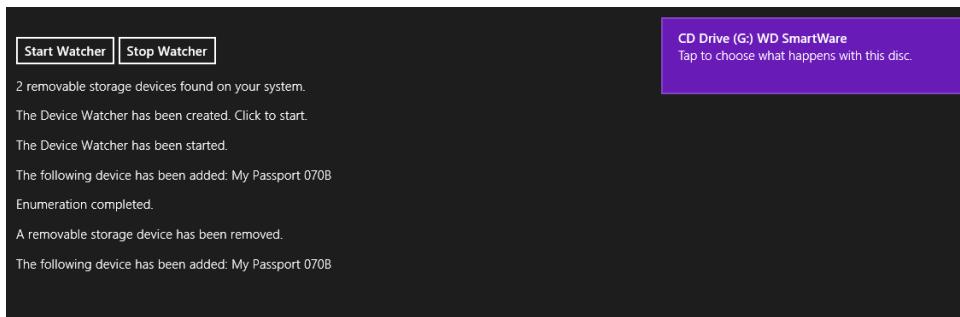


FIGURE 2-18 The sample app being notified of any change in the collection of removable storage devices connected to the system

The code illustrated in Listing 2-33 uses the *DeviceWatcherStatus* enum to check the state of the watcher before starting or stopping the watcher. The *DeviceWatcherStatus* enum can assume one of the following values:

- **Created** The initial state of a *Watcher* instance. During this state, clients can register event handlers.
- **Started** After the *Start* method has been called, the watcher starts enumerating the initial collection.

- **EnumerationCompleted** The watcher has finished enumerating the initial collection. Items can still be added, updated, or removed from the collection.
- **Stopping** The *Stop* method has been invoked by the client, and the watcher is still in the process of stopping. Events can still be raised during this state.
- **Stopped** The watcher has stopped. No further events will be raised.
- **Aborted** The watcher has aborted its operations. No subsequent events will be raised.

The *Start* method cannot be called when *DeviceWatcher* is in the Started or Stopping state. The *Stop* method, on the other hand, raises the *Stopped* event and the watcher state transitions to the Stopping state. The watcher will pass to the Stopped state after all events that are already in the process of being raised have completed. Figure 2-19 illustrates the complete flow of transitions between the states.

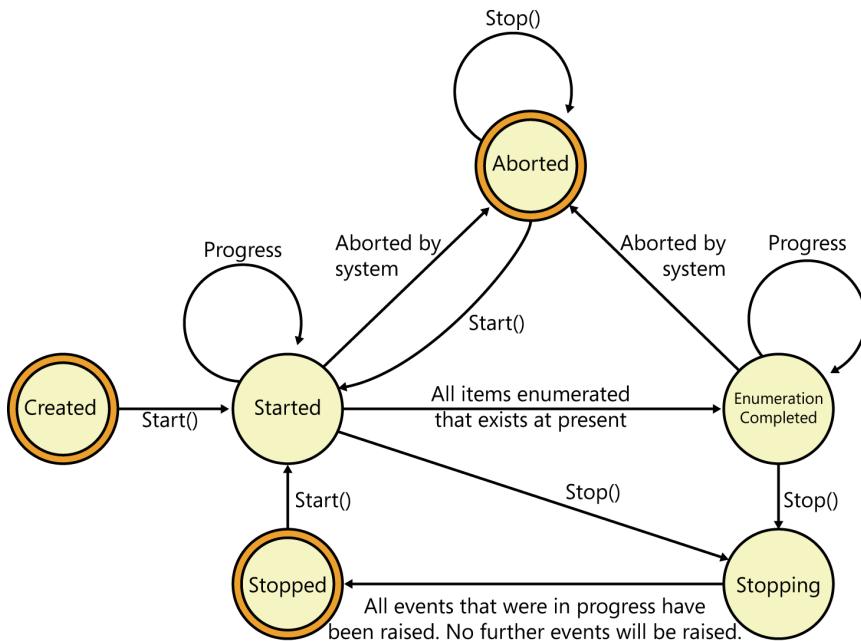


FIGURE 2-19 Diagram illustrating state transitions
Source: Adapted from MSDN documentation, <http://i.msdn.microsoft.com/dynimg/IC557460.png>

Enumerating Plug and Play (PnP) devices

The *Windows.Devices.Enumeration.PnP* namespace enables you to enumerate devices, device interfaces, device interface classes, and device containers. A device interface is a symbolic link to a PnP device that an application can use to access the device. A device interface class is the interface that represents functionalities exposed by a class of devices. A device container represents the physical device as seen by the user. The device container enables you to access information that pertains to the entire device hardware product, rather than only one of its

functional interfaces. Examples of device container properties are manufacturer or model name.

The *Windows.Devices.Enumeration.PnP* namespace provides a *PnpObject* class that presents the same properties and methods of the *DeviceInformation* class discussed in the preceding sections, with a few differences. For example, for both the *FindAllAsync* and *CreateWatcher* methods exposed by the *PnpObject* class, you have to provide an instance of the *PnpObjectType* enum to indicate what kind of object you are looking for. The *PnpObjectType* enum can assume one of the following values:

- **Unknown** Indicates an object of unknown type. This value is not normally used.
- **DeviceInterface** Indicates a device interface.
- **DeviceContainer** Indicates a device container.
- **DeviceInterfaceClass** Indicates a device interface class.

For example, the following JavaScript code snippet searches only the device containers available on the system.

```
function pnpEnumeration() {  
    var properties = new Array(  
        "System.ItemNameDisplay",  
        "System.Devices.ModelName",  
        "System.Devices.Connected",  
        "System.Devices.FriendlyName",  
        "System.Devices.Manufacturer",  
        "System.Devices.ModelNumber" );  
  
    Windows.Devices.Enumeration.Pnp.PnpObject.findAllAsync(  
        Windows.Devices.Enumeration.Pnp.PnpObjectType.deviceContainer,  
        properties)  
        .done(doCompleted, doError);  
}
```

For each *PnpObject* retrieved and added to the collection, the code also retrieves the device display name, the model name, the connection status, the friendly name, the manufacturer, and the model number.

MORE INFO DEVICE CONTAINER

For further information about device containers and container IDs, visit [http://msdn.microsoft.com/en-us/library/windows/hardware/ff549447\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/hardware/ff549447(v=vs.85).aspx).

The *PnpObject* class also provides the possibility of leveraging a *PnpDeviceWatcher* object (which is practically identical to the *DeviceWatcher* class) to monitor changes in the device collection. Like the *FindAllAsync* method, this option follows the same pattern already illustrated in the preceding section. The following code snippet shows an example of its usage:

```
var properties = new Array(  
    "System.ItemNameDisplay",  
    "System.Devices.ModelName",
```

```

    "System.Devices.Connected",
    "System.Devices.FriendlyName",
    "System.Devices.Manufacturer",
    "System.Devices.ModelNumber");

var watcher = Windows.Devices.Enumeration.Pnp.PnpObject.createWatcher(
    Windows.Devices.Enumeration.Pnp.PnpObjectType.deviceInterfaceClass, properties);

watcher.onadded += watcher_added;
watcher.onupdated += watcher_updated;
watcher.onremoved += watcher_removed;

watcher.onenumerationcompleted += watcher_enumerationCompleted;

watcher.start();

```



Thought experiment

Knowing the device you are working with

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are developing a Windows Store app for video capturing. The app enables the user to apply several video effects to the recorded stream. To apply them, you need to know the manufacturer and the model of the camera.

Which steps do you have to perform to retrieve this kind of information from the system?

Objective summary

- The *DeviceInformation* class provides two ways of enumerating devices, represented by two static methods that can be used to retrieve information about the devices available on the system: the *FindAllAsync* method and the *DeviceWatcher* class.
- The *FindAllAsync* static method performs a one-time search for available devices. This method returns a collection of devices that can be inspected and enumerated. Overloaded versions of this method enable you to provide different search parameters to narrow down the resulting devices.
- The *DeviceWatcher* class, instantiated through the *CreateWatcher* method of the *DeviceInformation* class, not only enumerates available devices but also raises specific events any time the device collection changes.
- Use the *Start* method of the *DeviceWatcher* instance to start retrieving the collection of devices. Use the *Added*, *Updated*, and *Removed* events to be notified of changes in the device collection. Call the *Stop* method when you no longer need to be notified of changes in the devices.

- You can leverage the *PnpObject* class of the *Windows.Devices.Enumeration.PnP* namespace to enumerate Plug And Play (PnP) devices, device interfaces, and device containers. Use the *FindAllAsync* method for a one-time search for available PnP devices, or create a *PnpDeviceWatcher* object to be notified of any change in the collection.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. Which parameter can be passed to the *FindAllAsync* method of the *DeviceInformation* class to retrieve all video capture devices available on a system?
 - A string with the content "VideoCapture"
 - An instance of the *DeviceClass* enum
 - An instance of the *PnpObjectType*
 - An instance of the *DeviceWatcher* class
2. Which class raises the *Added* event when a new device is added to the collection of available devices?
 - PnpObject* class
 - DeviceInformation* class
 - DeviceWatcher* class
 - DeviceThumbnail* class
3. Which of the following capabilities must be declared in the application manifest before enumerating the devices?
 - Location capability
 - Webcam capability
 - Removable Storage capability
 - None; no declaration needs to be added

Chapter summary

- To capture pictures or video, you can use the *CameraCaptureUI* class that encapsulates all the low-level details and provides the standard UI for simple operations.
- Use the *MediaCapture* class instead of the *CameraCaptureUI* whenever you want complete control over the entire process of audio and video capturing, including customization of the UI.

- The *Windows.Devices.Sensors* namespace includes support for a variety of sensors: accelerometer, gyrometer, compass, orientation, inclinometer, and light sensors. All corresponding APIs use similar patterns. To retrieve data from a sensor, you can follow an event-driven approach and subscribe to the event specifically provided by the sensor's class (usually the *ReadingChangedEvent*) and be notified when a new reading is available, or you can opt for a polling strategy (interrogating the sensor at regular intervals).
- The *Windows.Devices.Geolocation* namespace exposes a Location API that enables retrieving the user's location through a location provider, which uses Wi-Fi triangulation and IP address data to determine the position, or a more precise GPS sensor, when available. Use the *Geolocator* class to take advantage of these functionalities.
- The *DeviceInformation* class, under the *Windows.Devices.Enumeration* namespace, provides two ways of enumerating devices. The *FindAllAsync* static method performs a one-time search for available devices, whereas the *DeviceWatcher* class not only enumerates all available devices, but also raises specific events any time the device collection changes.
- You can leverage the *PnpObject* class of the *Windows.Devices.Enumeration.PnP* namespace to enumerate PnP devices, device interfaces, and device containers.

Answers

This section contains the solutions to the thought experiments and answers to the lesson review questions in this chapter.

Objective 2.1: Thought experiment

A possible approach is to use the *CameraCaptureUI* API to display the webcam preview using the standard Windows 8 UI. After the picture has been taken, you have to retrieve the picture stream and then show the picture to the user to enable her to insert the additional information. This way, the app uses the well-known standard interface, improving the overall user experience.

Unfortunately, this approach suffers from some limitations. First, the user could do something during the capture of the photo that is not allowed by the standard UI. Second, you still have to solve the problem of recording audio comments to associate to a particular photo. The *CameraCaptureUI* API enables a user to record video and capture photos. Audio is recorded only as part of the video stream coming from the webcam, which means you cannot handle audio and video separately when using the *CameraCaptureUI* API.

The second, more sophisticated, approach is to use the *MediaCapture* API to display the webcam preview using a custom UI. This way, you can achieve finer-grained control over the entire flow. You can use the *MediaCapture* class to take pictures from the video stream, display the picture in the same form, and add text information. Then you can use the *MediaCapture* class again to record audio comments to associate to a photo, all without leaving the current page. Besides, by using the *MediaCapture* class, you can enable the user to add some video or audio effects, such as video stabilization or other custom effects.

Objective 2.1: Review

1. Correct answer: B

- A. Incorrect:** You can set the media format.
- B. Correct:** *VideoSettings* is the property to use to set the media format.
- C. Incorrect:** *MaxResolution* is not designed to set the media format.
- D. Incorrect:** The *CameraCaptureUIMode* parameter does not set the media format. It sets the type of capture.

2. Correct answer: D

- A. Incorrect:** When an error occurs during media capture, the *Failed* event is raised.
- B. Incorrect:** When the app does not have permission to use the capture device, an *UnauthorizedAccessException* is raised.
- C. Incorrect:** No event is raised when the user stops recording the stream.
- D. Correct:** The *RecordLimitationExceeded* event is raised when the record limit is exceeded.

3. Correct answer: A

- A. Correct:** The *MediaCaptureInitializationSettings* class contains initialization settings for the *MediaCapture* object that can be passed as parameters to the *InitializeAsync* method.
- B. Incorrect:** The *CameraCaptureUIPhotoFormat* enum determines the format for storing photos captured through the *CameraCaptureUI* API.
- C. Incorrect:** The *CameraCaptureUIVideoCaptureSettings* class provides settings for capturing videos through the *CameraCaptureUI* API.
- D. Incorrect:** The *CameraOptionsUI* class provides access to the UI setting for the webcam.

Objective 2.2: Thought experiment

1. You can leverage the compass sensor to determine which direction the device is pointing. This sensor notifies you of the direction your device is facing, expressed in degrees relative to magnetic north and, when available, to geographic (or true) north.
2. You should use the data gathered by the compass sensor to adjust the orientation of the map displayed on the screen according to the user's current heading. Doing so aligns what the user sees on the displayed map with what he sees in front of him in the real world.

Objective 2.2: Review

1. Correct answer: D

- A. Incorrect:** The *Accelerometer* class exposes a *ReportInterval* property.
- B. Incorrect:** The *Gyrometer* class exposes a *ReportInterval* property.
- C. Incorrect:** The *Inclinometer* class exposes a *ReportInterval* property.
- D. Correct:** The *SimpleOrientationSensor* class does not expose a *ReportInterval* property.

2. Correct answer: B

- A. Incorrect:** The acceleration of the device along the three axes is measured by the *Accelerometer* class.
- B. Correct:** The rotation angles of a device around the three axes (also known as yaw, pitch, and roll) are measured by the *Gyrometer* class.
- C. Incorrect:** The quadrant orientation of the device is measured by the *SimpleOrientationSensor* class.
- D. Incorrect:** The heading in degrees relative to magnetic north is measured by the *Compass* class.

3. Correct answer: A

- A. Correct:** The location provider uses Wi-Fi triangulation and IP address data to determine latitude and longitude coordinates.
- B. Incorrect:** The altitude is not provided by the location provider, only by the GPS sensor.
- C. Incorrect:** The current heading is not provided by the location provider, only by the GPS sensor or the compass sensor.
- D. Incorrect:** Current speed is not provided by the location provider, only by the GPS sensor.

Objective 2.3: Thought experiment

Manufacturer or model information, together with all other information related to the hardware of the device, is generally exposed through device containers, which represent the physical devices as seen by the user.

The best option to retrieve information about manufacturer and model of a media capture device is to use the *FindAllAsync* method of the *PnpObject* class to retrieve all pertinent information exposed by the various device containers available on the system. You can also explicitly look only for devices manufactured by certain vendors. To do that, you have to supply the appropriate AQS string to the *FindAllAsync* method. When doing so, be sure to specify the right properties to retrieve.

For example, the following code uses one of the overloaded versions of the *FindAllAsync* method to retrieve all device containers that satisfy the provided condition, expressed through an AQS string that asks only for devices manufactured by Microsoft Corporation. The array of strings specifies which properties should be returned by the query, including device interface class ID, manufacturer, and model number.

```
var properties = new Array()  
    "System.ItemNameDisplay",  
    "System.Devices.InterfaceClassGuid",  
    "System.Devices.ModelName",  
    "System.Devices.Connected",  
    "System.Devices.FriendlyName",  
    "System.Devices.Manufacturer",  
    "System.Devices.ModelNumber");  
  
var selector = "System.Devices.Manufacturer:=\"Microsoft Corporation\"";  
Windows.Devices.Enumeration.Pnp.PnpObject.findAllAsync(  
    Windows.Devices.Enumeration.Pnp.PnpObjectType.deviceContainer,  
    properties,  
    selector)  
.done(doCompleted, doError);
```

Objective 2.3: Review

1. Correct answer: B

- A. Incorrect:** To retrieve all video capture devices available on the system, you have to provide the *FindAllAsync* method of the *DeviceInformation* class with a *DeviceClass* instance.
- B. Correct:** To retrieve all video capture devices available on the system, you have to provide the *FindAllAsync* method of the *DeviceInformation* class with an instance of the *DeviceClass* enum (more precisely, the *DeviceClass.VideoCapture* value).
- C. Incorrect:** The *PnpObjectType* enum is part of the *Windows.Devices.Enumeration.PnP* namespace.
- D. Incorrect:** The *DeviceWatcher* class is used to enumerate the devices available on the system and to be notified of any change in the collection.

2. Correct answer: C

- A. Incorrect:** The *PnpObject* class is used for one-time search of available devices.
- B. Incorrect:** The *DeviceInformation* class is used for one-time search of available devices.
- C. Correct:** The *DeviceWatcher* class exposes the *Added* event, which is raised when a new device is added to the collection of available devices.
- D. Incorrect:** The *DeviceThumbnail* class represents the image of a device.

3. Correct answer: D

- A. Incorrect:** The Location capability is required to access data about a user's location.
- B. Incorrect:** The Webcam capability is required to access the device's webcam.
- C. Incorrect:** The Removable Storage capability is required to access a device's removable storage.
- D. Correct:** You do not need to add declarations to enumerate available devices.

CHAPTER 3

Program user interaction

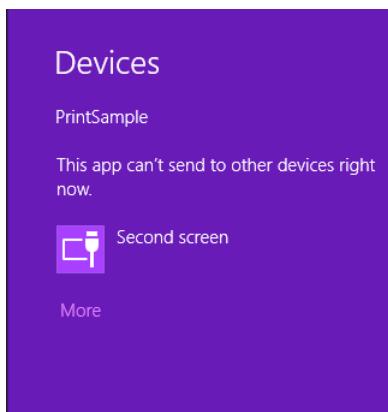
In this chapter, you learn how to implement the Print contract, which handles the entire printing flow, in your Windows Store apps. The chapter also focuses on the Play To contract, which enables users to stream a media element from any Windows Store app to a Play To-certified receiver connected to the network. You learn how to implement a Play To receiver application to receive a media element streaming from other Windows Store apps. Finally, you deal with the Windows Push Notification Service (WNS) and learn how to authenticate to the service; how to code against the library to create, request, and save the notification channel; and, how to interact with the service and send notifications to users.

Objectives in this chapter:

- Objective 3.1: Implement printing by using contracts and charms
- Objective 3.2: Implement Play To by using contracts and charms
- Objective 3.3: Notify users by using Windows Push Notification Service (WNS)

Objective 3.1: Implement printing by using contracts and charms

By default, a Windows Store app does not allow the user to access the Windows print system to print content. If you activate the Devices charm in the charms bar, a message appears: This app can't send to other devices right now.



NOTE *WINDOW.PRINT*

As explained in the official Microsoft documentation, in your Hypertext Markup Language (HTML)/JavaScript Windows Store app, you are still able to use the classic JavaScript function `window.print` from code to print your app's content. However, this function allows only the content that is displayed on the screen to be printed by using the default print experience. Because the content displayed on the screen of a Windows Store app doesn't always produce high-quality printed output, using the `window.print` function might not be the best approach. For an optimal customer experience, Microsoft recommends that you use the Windows Runtime functions and register for the Print contract as illustrated in this objective.

A Windows Store app needs to implement the Print contract to be able to print. Implementing the Print contract in Windows Store apps written in languages such as C#, Visual Basic (VB), and C++ can require a considerable amount of effort. You need to paginate the content that you want to print and handle numerous events raised at different stages of the printing process. However, the entire print process is considerably simpler in a Windows Store app written in HTML/JavaScript. The drawback is that you encounter some limitations in an HTML/JavaScript Windows Store app. For example, you cannot create customized printing options, nor can you modify the document after it has been sent to the print subsystem for previewing.

In the following sections, you learn how to manage all the required steps, from formatting the content you want to print to creating custom print options.

This objective covers how to:

- Implement the Print contract
- Construct a print preview
- Handle print pagination
- Create a custom print template
- Expose printer settings within your app
- Implement in-app printing

Registering a Windows Store app for the Print contract

Registering a Windows Store app for the Print contract requires three steps. First, obtain a *PrintManager* object for each view that you want your users to be able to print. Second, implement a *PrintTask* object representing the actual printing operation. Third, define which options will be displayed in the print window and in which order.

NOTE REQUIRED CLASSES FOR THE PRINTING PROCESS

Most of the classes required by the printing process are included in the *Windows.Graphics.Printing* namespace.

The *PrintManager* class is responsible for managing and orchestrating the complete printing process for a Windows Store app. A *PrintManager* object cannot be shared across different pages, but must be specific for each view that you want your users to be able to print. You have to get a reference to the *PrintManager* object for the current view by calling its *GetForCurrentView* method. Then you have to define a handler for the *PrintTaskRequested* event. Listing 3-1 shows the complete code of the default.js file of a sample app that registers itself for the Print contract.

LISTING 3-1 Registering for the Print contract

```
// For an introduction to the Blank template, see the following documentation:  
// http://go.microsoft.com/fwlink/?LinkId=232509  
(function () {  
    "use strict";  
  
    WinJS.Binding.optimizeBindingReferences = true;  
    var app = WinJS.Application;  
    var activation = Windows.ApplicationModel.Activation;  
  
    app.onactivated = function (args) {  
        if (args.detail.kind === activation.ActivationKind.launch) {  
            if (args.detail.previousExecutionState !==  
                activation.ApplicationExecutionState.terminated) {  
                // TODO: This application has been newly launched. Initialize  
                // your application here.  
            } else {  
                // TODO: This application has been reactivated from suspension.  
                // Restore application state here.  
            }  
            args.setPromise(WinJS.UI.processAll());  
        }  
    };  
  
    app.oncheckpoint = function (args) {  
        // TODO: This application is about to be suspended. Save any state  
        // that needs to persist across suspensions here. You might use the  
        // WinJS.Application.sessionState object, which is automatically  
        // saved and restored across suspension. If you need to complete an  
        // asynchronous operation before your application is suspended, call  
        // args.setPromise().  
    };  
  
    app.onloaded = function () {  
        registerForPrintContract();  
    }  
  
    function registerForPrintContract() {
```

```

        var printManager = Windows.Graphics.Printing.PrintManager.getForCurrentView();
        printManager.onprinttaskrequested = onPrintTaskRequested;
    }

    app.start();
}();

```



EXAM TIP

The *PrintTaskRequested* event is raised by the system when the user requests a print operation by activating the Devices charm in the charms bar.

The event handler for the *PrintTaskRequested* event is where your code needs to create the *PrintTask* object that represents a specific printing operation for a Windows Store app. It includes the content to be printed, as well as information describing how that content is to be printed.

Listing 3-2 shows the basic implementation of the *PrintTaskRequested* event handler.

LISTING 3-2 Implementing the *PrintTaskRequested* event handler

```

function onPrintTaskRequested(printEvent) {

    var printTask = printEvent.request.createPrintTask("Windows 8 Print Sample",
        function (args) {

            (code omitted)

        });
}

```

The *PrintTask* object is created by invoking the *CreatePrintTask* method exposed by the *Request* property. The *Request* property is an instance of the *PrintTaskRequest* class and can be accessed from the *PrintTaskRequestedEventArgs* object passed to the *PrintTaskRequested* event.

This method requires two parameters: the name for the task and a *PrintTaskSourceRequestedHandler* delegate. The first parameter represents the name that appears in the print queue, as shown in Figure 3-1.

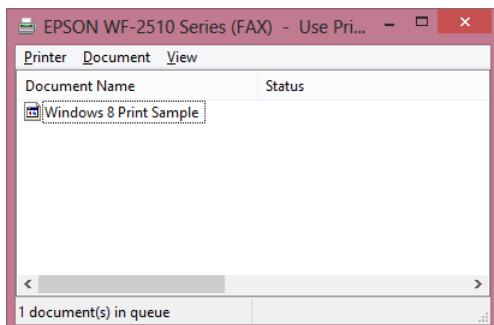


FIGURE 3-1 The print queue showing the print operation's name provided through the *CreatePrintTask* method

The function passed as the second parameter holds a reference to the method that is called when the user is ready to begin the printing process. (In Listing 3-2, the delegate is represented by an anonymous method.)

If you launch the app in Microsoft Visual Studio 2012 and activate the Devices charm, the flyout shows that your app is now registered to the Windows print system (see Figure 3-2).

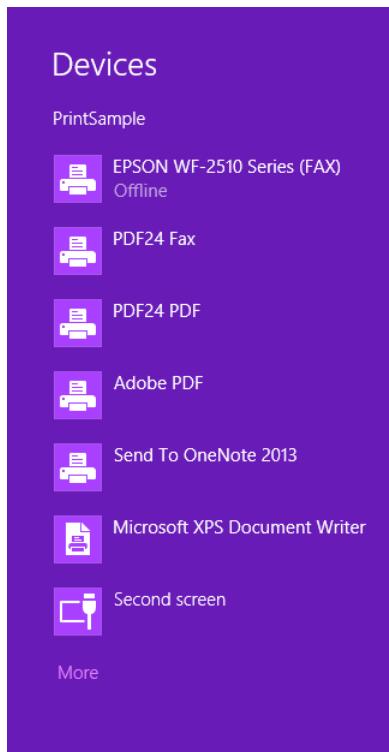


FIGURE 3-2 The Devices flyout showing available print targets

If you select a print target, the system displays a message: The app didn't provide anything to print. (See Figure 3-3.)

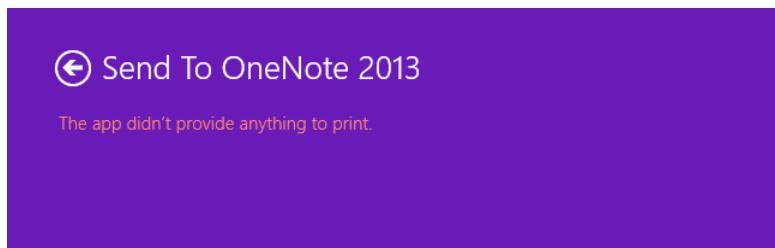


FIGURE 3-3 The default message displayed by the Windows Runtime when the user selects a print target and your app has not provided any content to print yet

NOTE UNREGISTERING AN EVENT

As suggested in the official Microsoft documentation, you should unregister the event handler when your app is showing content that does not need (or is not supposed) to be printed. Unregistering the event handler lets Windows know that your app cannot print at the present time. This ensures that the print flow will not fail due to a lack of content to print.

After creating the *PrintTask*, you need to provide some content to be printed. To implement the *PrintTask*, provide it with a reference to the HTML document to be printed by invoking the *SetSource* method exposed by the *PrintTaskRequestedEventArgs*. The document is retrieved by using the *MSApp.GetHtmlPrintDocumentSource* method, which returns the source content of the document that is going to be printed.



EXAM TIP

The content supplied to the *SetSource* method can consist of the root document, a document within an *IFrame*, a document fragment, or a Scalable Vector Graphics (SVG) document. Be aware that it must be a document, not just an HTML element.

The object passed to this method represents the actual content that will be printed as soon as the delegate is invoked. (See the next section for more details.) Listing 3-3 shows the revised *PrintTaskRequested* event handler originally shown in Listing 3-1, with changes in bold.

LISTING 3-3 The *PrintTaskRequested* event handler using the *SetSource* method

```
function onPrintTaskRequested(printEvent) {  
  
    var printTask = printEvent.request.createPrintTask("Windows 8 Print Sample",  
        function (args) {  
            args.setSource(MSApp.getHtmlPrintDocumentSource(document));  
        });  
}
```

The *PrintTaskRequestedEventArgs* also exposes, through its *Request* property, a *GetDeferral* method that can be used whenever you need to perform a lengthy operation asynchronously. By using a deferral, the Windows Runtime (WinRT) will wait for the content to be retrieved until the deferral is marked as completed by invoking the *Complete* method of the deferral object. The following code shows the basic pattern for this scenario:

```
function onPrintTaskRequested(printEvent) {  
  
    var deferral = printEvent.request.getDeferral();  
  
    var printTask = printEvent.request.createPrintTask("Windows 8 Print Sample",  
        function (args) {  
            // perform some lengthy operations here  
        });  
  
    deferral.complete();  
}
```

Handling *PrintTask* events

After the *PrintTask* object has been created, your application can optionally subscribe to the events exposed by this class. These events are raised by the system at different stages of the printing process and can be used to provide feedback to the user:

- **Previewing** Raised when the system initializes the preview mode of the printing process
- **Submitting** Raised when a print task begins submitting content to the print subsystem
- **Progressing** Raised to provide progress information about how much of the printed content has been submitted to the print system
- **Completed** Fired when the printing operation is finished

Regarding the *Completed* event, it is important to understand that a task is marked as completed not only when the content has been actually submitted to the system for printing, but also when the task has been canceled by the user, has failed, or has been otherwise abandoned.

You should consider using the *Completed* event to provide the user with feedback about the result of the printing task by accessing the *Completion* property of the *PrintTaskCompletedEventArgs* object passed as a parameter to the event handler. In particular, the *Completion* property (of type *PrintTaskCompletion*) can assume one of the following values:

- **Abandoned** The task has been abandoned.
- **Canceled** The task has been canceled.
- **Failed** The task has failed.
- **Submitted** The task has been submitted.

The following code excerpt shows a possible usage of the *Completed* event handler informing the user that the print task has failed:

```
function onPrintTaskCompleted(printTaskCompletionEvent) {  
  
    if (printTaskCompletionEvent.completion ===  
        Windows.Graphics.Printing.PrintTaskCompletion.Failed) {  
        Windows.UI.Popups.MessageDialog("Print Task Failed!");  
    }  
}
```

In the *PrintTaskRequested* event handler, you can subscribe to the *Completed* event as shown in Listing 3-4.

LISTING 3-4 Subscribing to the *Completed* event

```
function onPrintTaskRequested(printEvent) {  
  
    var printTask = printEvent.request.createPrintTask("Windows 8 Print Sample",  
        function (args) {
```

```

        args.setSource(MSApp.getHtmlPrintDocumentSource(document));

    printTask.oncompleted = onPrintTaskCompleted;
}
}

```

Creating the user interface

It is time to create a basic user interface (UI) that displays some sample text that the user can print. Listing 3-5 shows the complete definition of the default.html file, which you can use as a reference.

LISTING 3-5 The complete HTML definition of the default page

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>PrintToSample</title>
    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- PrintToSample references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <div id="myDocument">
        <div id="myTitle">
            <h1>Windows 8 Print Sample in JS</h1>
        </div>
        <div id="myDocumentBody">
            <p>
                
                Lorem ipsum dolor sit amet, consectetur adipiscing elit.
                Nunc nec urna ut turpis pretium cursus imperdiet non metus.
                (code omitted)
            </p>
            <p>
                Sed rhoncus consectetur mi ut molestie.
                Nunc faucibus mi suscipit metus facilisis facilisis.
                (code omitted)
            </p>
            <p>
                Donec faucibus ullamcorper metus, eu aliquam lectus dictum sed.
                Cras egestas hendrerit orci, vitae rutrum est ullamcorper pharetra.
                (code omitted)
            </p>
        </div>
    </div>
</body>
</html>

```

Figure 3-4 shows the default page displayed on the screen.

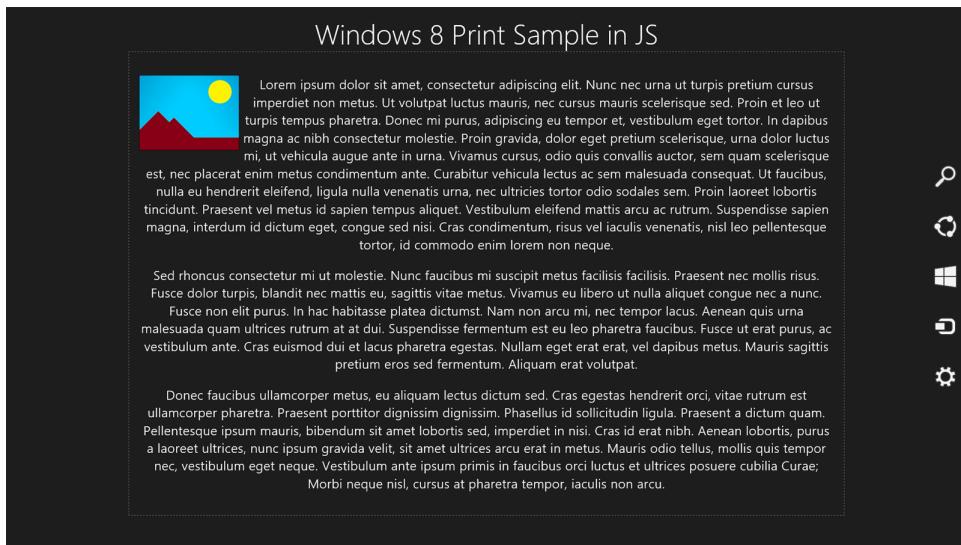


FIGURE 3-4 Sample content displayed in the UI

Creating a custom print template

So far, we have sent the HTML document to the print subsystem exactly as it appeared on the screen. This is not very different from calling the `window.print` method in a web application. To improve the overall user experience, you should consider providing the print system with a document specifically formatted for print (reducing, for example, the font size in the document to be printed, modifying the margins, and so on).

There are at least two ways to implement this scenario by simply using the appropriate HTML/Cascading Style Sheets (CSS) attributes.

The first and simplest way is to leverage the `media` attribute of the HTML document. The `media` attribute indicates that a style sheet, or a section of a style sheet, applies only to certain media types, such as print media. The following code excerpt shows the header of the HTML document that uses the `media` attribute to point to a CSS file specifically designed for printing.

Sample of HMTL code

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>PrintToSample</title>
    <link rel="stylesheet" type="text/css"
        href="/css/default.css" />
    <link rel="stylesheet" type="text/css"
        href="/css/print.css" media="print" />
    <script src="/js/default.js"></script>
</head>
```

(code omitted)

The same result can be achieved by leveraging the analogous CSS *media* attribute, instead of the previous one, as exemplified in the following snippet:

Sample of CSS code

```
@media print {  
    body {  
        width: 100%;  
        height: 100%;  
        font-size: 18px;  
    }  
}
```

(code omitted)

The result is the same in both cases: The document that will be sent to the print system will be formatted according to the style specifically customized for print.

The other way to provide specific content to the print system is by leveraging the *rel* attribute of the *link* element, in combination with the *media* attribute, to provide the print subsystem with a completely different HTML document, rather than just the same document with different formatting. The following snippet shows the revised code of Listing 3-1, with changes in bold:

Sample of HTML code

```
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8" />  
    <title>PrintToSample</title>  
    <link rel="alternate" href="printsample.html" media="print" />  
    <!-- WinJS references -->  
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />  
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>  
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>  
  
</head>  
(code omitted)
```

Listing 3-6 contains the complete definition of the secondary page (named *printsample.html*, in this case) that is provided to the Windows print system. Changes to the original HTML page displayed on the screen are highlighted in bold.

LISTING 3-6 The complete HTML definition of the secondary page passed to the print subsystem

```
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="utf-8" />  
    <title>PrintToSample</title>  
    <!-- WinJS references -->  
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
```

```

<script src="//Microsoft.WinJS.1.0/js/base.js"></script>
<script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

<!-- PrintToSample references -->
<link href="/css/print.css" rel="stylesheet" />
<script src="/js/default.js"></script>
</head>
<body>
    <div id="header">
        
    </div>
    <div id="myDocument">
        <div id="myTitle">
            <h1>Windows 8 Print Sample in JS</h1>
        </div>
        <div id="myDocumentBody">
            (code omitted)           </div>
        </div>
    </div>
</body>
</html>

```

The new page contains a reference to a new CSS file named print.css (instead of the default.css file automatically added to the project by Visual Studio 2012) and a header section with the company's logo that will appear in the print output, as shown in Figure 3-5. The figure also illustrates the differences between the page displayed on the screen and the one shown in the preview window.



FIGURE 3-5 The preview window showing the new HTML document specifically formatted for print

So far, we have leveraged HTML/CSS rules and attributes to provide the print subsystem with content specifically formatted for print. You can achieve the same result

programmatically by manipulating the Document Object Model (DOM) structure of the document before sending it to the print subsystem, as shown in the following code excerpt:

Sample of JavaScript code

```
var alternateLink = document.createElement("link");
alternateLink.setAttribute("id", "alternateContent");
alternateLink.setAttribute("rel", "alternate");
alternateLink.setAttribute("href", "printsample.html");
alternateLink.setAttribute("media", "print");

document.getElementsByTagName("head")[0].appendChild(alternateLink);
args.setSource(MSApp.getHtmlPrintDocumentSource(document));
```

Remember that the *MSApp.GetHtmlPrintDocumentSource* method accepts any HTML document. You can download the document from the web or create it programmatically. For example, the following JavaScript snippet shows how to send to the print subsystem only a fragment of the current document (in the form of an *IFrame* object) by isolating the content to be printed before sending it to the print subsystem.

```
var newdoc = document.getElementById("frameToPrint");
args.setSource(MSApp.getHtmlPrintDocumentSource(frame.contentDocument));
```

Another way to create an HTML document fragment programmatically is by leveraging the *CreateDocumentFragment* method, as follows:

```
var p = document.createElement("p");
var txt = 'Sample paragraph';
var t = document.createTextNode(txt);
p.appendChild(t);
var doc = document.createDocumentFragment();
doc.appendChild(p);

args.setSource(MSApp.getHtmlPrintDocumentSource(doc));
```

Understanding the print task options

Most of the options exposed by the print target selected by the user in the preview window can be accessed via the *Options* property of the *PrintTask* class. This property (of type *PrintTaskOptions*) can be used to customize the print user experience through a rich set of print options, such as color, number of pages, orientation, print quality, and so on.

The options available through the *PrintTaskOptions* class are:

- **Binding** Represents the binding option for the print task. The range of possible values is defined in the *Windows.Graphics.Printing.PrintBinding* enum.
- **Collation** Represents the collation option of the print tasks. The range of possible values is defined in the *Windows.Graphics.Printing.PrintCollation* enum.
- **ColorMode** Represents the color mode option of the print task. The range of possible values is defined in the *Windows.Graphics.Printing.PrintColorMode* enum.

- **Duplex** Represents the duplex option of the print task. The range of possible values is defined in the *Windows.Graphics.Printing.PrintDuplex* enum.
- **HolePunch** Represents the hole punch option of the print task. The range of possible values is defined in the *Windows.Graphics.Printing.PrintHolePunch* enum.
- **MaxCopies** Represents a read-only property that indicates the maximum number of copies supported for the print task.
- **MediaSize** Represents the media size option of the print task. The range of possible values is defined in the *Windows.Graphics.Printing.PrintMediaSize* enum.
- **MediaType** Represents the media type option for the print task. The range of possible values is defined in the *Windows.Graphics.Printing.PrintMediaType* enum.
- **MinCopies** Represents a read-only property that indicates the minimum number of copies allowed for the print task.
- **NumberOfCopies** Represents the value for the number of copies for the print task.
- **Orientation** Represents the orientation option for the print task. The range of possible values is defined in the *Windows.Graphics.Printing.PrintOrientation* enum.
- **PrintQuality** Represents the print quality option for the print task. The range of possible values is defined in the *Windows.Graphics.Printing.PrintQuality* enum.
- **Staple** Represents the staple option for the print task. The range of possible values is defined in the *Windows.Graphics.Printing.PrintStaple* enum.

Although each option presents its specific set of values, they all share three common values (which represent three common scenarios):

- *Default*
- *NotAvailable*
- *PrinterCustom*

The semantics of these values might be slightly different depending on whether they are acquired from the print target or set from code.

When the *Default* value is retrieved by accessing the corresponding print target property, it means the actual value for that property has not yet been determined. On the contrary, setting an option to the *Default* value instructs the target device to use its default value (whatever it might be) for that property. If the selected print target has no default value for that option, no changes are made (that is, the selected target keeps its original value).



EXAM TIP

When a *NotAvailable* value is retrieved, it means the specific option is not available for the current print target. Setting an option to *NotAvailable* is not allowed and results in a runtime exception.

Most of these options can be used to set a specific value to display in the print window, or to retrieve the information provided by the print device currently selected by the user. For

example, you can leverage the *PrintTaskOptions* class to set the initial value of one or more options for the preview window. If you try to set an option that is not supported by the print target, that option is simply ignored and not displayed to the user. Listing 3-7 shows how to provide some default values to the print system. (Changes from Listing 3-4 are highlighted in bold.)

LISTING 3-7 Setting the print option initial values

```
function onPrintTaskRequested(printEvent) {  
    var printTask = printEvent.request.createPrintTask("Windows 8 Print Sample",  
        function (args) {  
  
            args.setSource(MSApp.getHtmlPrintDocumentSource(document));  
            printTask.oncompleted = onPrintTaskCompleted;  
  
            printTask.options.orientation =  
                Windows.Graphics.Printing.PrintOrientation.landscape;  
            printTask.options.colorMode =  
                Windows.Graphics.Printing.PrintColorMode.monochrome;  
            printTask.options.mediaSize =  
                Windows.Graphics.Printing.PrintMediaSize.isoA4;  
        });  
}
```

Figure 3-6 shows the preview window with the values provided via the *PrintTask* object. The default orientation is now set to Landscape and the color mode to Monochrome, while the media size option is not displayed to the user. In the next section, you learn which options should be displayed in the preview window and in what order.

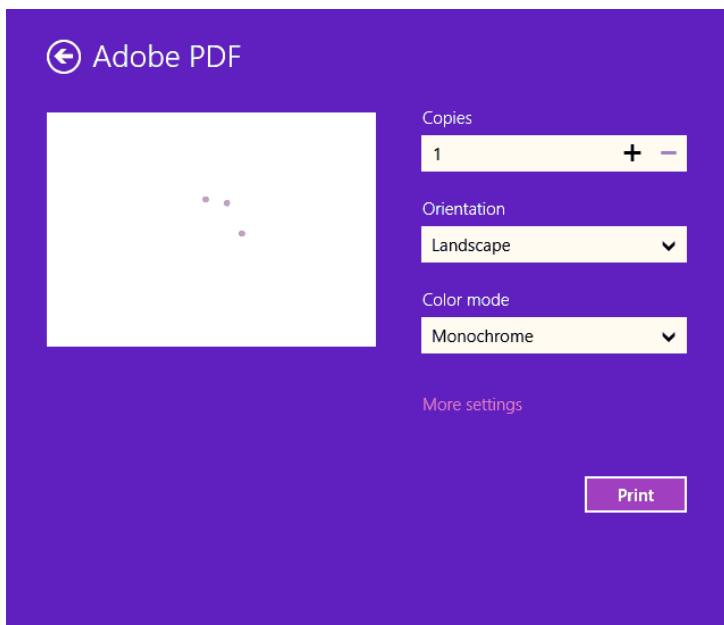


FIGURE 3-6 The preview window with two of the new default options

Choosing options to display in the preview window

You can define the options that display in the preview window, and in which order. To do so, clear the default collection of print options, represented by the *DisplayedOptions* property, and add the options that you want to display in the preview window one by one, in the order you want them to appear. Add the options by passing to the *Append* method one of the various properties exposed by the *StandardPrintTaskOptions* class, which provides access to the canonical names for the options represented by the *PrintTaskOptions* class. Listing 3-8 shows the complete process. (Changes from Listing 3-7 are shown in bold.)

LISTING 3-8 Choosing which print options to display in the preview window

```
function onPrintTaskRequested(printEvent) {  
  
    var printTask = printEvent.request.createPrintTask("Windows 8 Print Sample",  
        function (args) {  
  
            args.setSource(MSApp.getHtmlPrintDocumentSource(document));  
            printTask.oncompleted = onPrintTaskCompleted;  
  
            printTask.options.displayedOptions.clear();  
            printTask.options.displayedOptions.append(  
                Windows.Graphics.Printing.StandardPrintTaskOptions.orientation);  
            printTask.options.displayedOptions.append(  
                Windows.Graphics.Printing.StandardPrintTaskOptions.colorMode);  
            printTask.options.displayedOptions.append(  
                Windows.Graphics.Printing.StandardPrintTaskOptions.mediaSize);  
  
            printTask.options.orientation =  
                Windows.Graphics.PrintOrientation.landscape;  
            printTask.options.colorMode =  
                Windows.Graphics.PrintColorMode.monochrome;  
            printTask.options.mediaSize =  
                Windows.Graphics.PrintMediaSize.isoA4;  
        });  
}  
}
```

After you run the application, activate the Devices charm, and select a print target. The three options added to the *DisplayedOptions* collection display in the preview window (see Figure 3-7). If any option added to the collection is not supported by the print target selected by the user, that option is ignored.

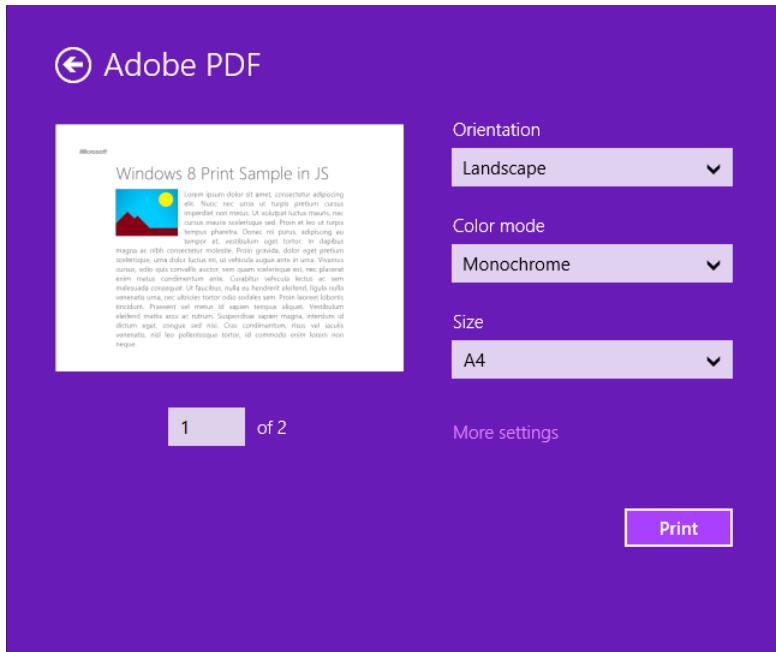


FIGURE 3-7 The preview window with all options added by using the *DisplayedOptions* property, which now includes the Size option

When dealing with print options, you should consider two things. First, as suggested in the Microsoft official documentation, you should not change the order of the settings shown in the print window unless it is necessary. This maintains consistency in the user experience. The same caveat also applies whenever you need to add more print settings to the print window. If the printer manufacturer made printer-specific settings available for your printer, users of your app can click the More settings link in the print window (shown in Figure 3-7) to display the additional settings.

Reacting to print option changes

Because the user can change one or more print options in the preview window, it would be ideal for your application to react to those changes and modify the formatted document accordingly. Unfortunately, in a Windows Store app using JavaScript, it is not yet possible to modify the previewed document or its pagination programmatically based on the new options.



EXAM TIP

The official Microsoft documentation states: “If you develop in JavaScript, however, you must use Direct2D printing to change the preview or print content based on option changes. This is because print template is not supported.” In other words, it appears that after the HTML document has been provided to the print subsystem through the *SetSource* method, it cannot be modified via JavaScript. However, if the user changes the orientation or the size of the page in the preview window, the system will rearrange the document pagination based on the new values. Any other changes to the print options will not affect the previewed document.

NOTE DIRECT2D

You can find more information about Direct2D printing at [http://msdn.microsoft.com/en-us/library/windows/desktop/hh448418\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/hh448418(v=vs.85).aspx).

Nevertheless, the *PrintTaskOptionDetails* class exposes an *OptionChanged* event, which is raised every time the user changes one of the properties of the print target. In Listing 3-9, changes from Listing 3-8 are highlighted in bold.

LISTING 3-9 Subscribing to the *OptionChanged* event

```
function onPrintTaskRequested(printEvent) {  
  
    var printTask = printEvent.request.createPrintTask("Windows 8 Print Sample",  
    function (args) {  
  
        args.setSource(MSApp.getHtmlPrintDocumentSource(document));  
        printTask.oncompleted = onPrintTaskCompleted;  
  
        printTask.options.displayedOptions.clear();  
        printTask.options.displayedOptions.append(  
            Windows.Graphics.Printing.StandardPrintTaskOptions.orientation);  
        printTask.options.displayedOptions.append(  
            Windows.Graphics.Printing.StandardPrintTaskOptions.colorMode);  
        printTask.options.displayedOptions.append(  
            Windows.Graphics.Printing.StandardPrintTaskOptions.mediaSize);  
  
        printTask.options.orientation =  
            Windows.Graphics.Printing.PrintOrientation.landscape;  
        printTask.options.colorMode =  
            Windows.Graphics.Printing.PrintColorMode.grayscale;  
        printTask.options.mediaSize =  
            Windows.Graphics.Printing.PrintMediaSize.isoA4;  
  
        var detailedOptions = Windows.Graphics.Printing.OptionDetails  
            .PrintTaskOptionDetails.getFromPrintTaskOptions(printTask.options);  
        detailedOptions.onoptionchanged = onPrintOptionChanged;  
    });  
}
```

```
function onPrintOptionChanged(optionEvents) {
    if (optionEvents !== null && optionEvents.optionId === "PageOutputColor") {
        // do something
    }
}
```

The code first retrieves a reference to the *PrintTaskOptionDetails* object for the current task by calling the *GetFromPrintTaskOptions* static method, which accepts a *PrintTaskOptions* object as parameter. It then subscribes to the *OptionChanged* event. In the event handler, you can verify which option has been modified by checking the *OptionId* property of the *PrintTaskOptionChangedEventArgs* instance received as a parameter.

Implementing in-app printing

Normally, the user starts the printing process by manually activating the Devices charm in the charms bar (which, in turn, fires the *PrintTaskRequested* event). The Devices charm can also be “triggered” via code, by leveraging the *ShowPrintUIAsync* method of the *PrintManager* class:

```
function printButton_Click(args) {
    Windows.Graphics.Printing.PrintManager.showPrintUIAsync();
}
```

Before calling this method, make sure the application has already retrieved the *PrintManager* object and registered a listener for *PrintTaskRequested*. The method returns a *Boolean* value that indicates the success or failure of the procedure.

Keep in mind that Microsoft official documentation explicitly discourages the use of this method in generic applications, in which you should let users start the printing process via the Devices charm. However, the *ShowPrintUIAsync* method can be used in scenarios in which you want to provide guidance to users to help them print some content, such as a ticket purchase, a flight check-in, or a boarding pass.



Thought experiment

Adding a custom print option for your app

In this thought experiment, apply what you’ve learned about this objective. You can find answers to these questions in the “Answers” section at the end of this chapter.

You are developing a Windows Store app using JavaScript that handles very large HTML documents. You want your customers to be able to select which pages to print. For example, customers can decide whether to print the entire document, the current page only, or all pages within a certain range.

Given that you cannot create custom print options in an HTML/JavaScript Windows Store app as you would for an app written in C#, VB, or C++, what work-around can you use to allow users to print only selected pages?

Objective summary

- To be able to print, a Windows Store app must implement the Print contract. To register with the Windows print system, you have to get a reference to the *PrintManager* object associated with the current view, subscribe to the *PrintTaskRequested* event, and create a print task representing a specific printing operation in the corresponding event handler.
- Retrieve and/or set printer settings by leveraging the *PrintTaskOptions* class and define the options to be displayed in the preview window by adding them to the *DisplayedOptions* property.
- Provide your users with in-app printing guidance by invoking the *ShowPrintUIAsync* static method.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the “Answers” section at the end of this chapter.

1. What class is responsible for orchestrating the complete printing process for a Windows Store app?
 - A. *PrintManager* class
 - B. *PrintTaskOptionDetails* class
 - C. *PrintTaskOptions* class
 - D. *PrintTask* class
2. What event is raised when the user requests a print operation by activating the Devices charm in the charms bar, before selecting a print target?
 - A. The *Submitting* event exposed by the *PrintTask* class.
 - B. The *Completed* event exposed by the *PrintTask* class.
 - C. The *PrintTaskRequested* event exposed by the *PrintManager* class.
 - D. No events are raised until the user selects a print target from the list of available targets.
3. What does the function passed to the *CreatePrintTask* method as a second parameter hold a reference to?
 - A. The function called when the user is ready to begin the printing process
 - B. The function called when the *Submitting* event of the *PrintTask* instance is raised
 - C. The function that notifies the user of the outcome of the printing operation
 - D. The function that keeps the user up to date about how much content has been processed by the print subsystem during the print operation

Objective 3.2: Implement Play To by using contracts and charms

The Play To feature enables you to stream multimedia elements, such as movies, music, and photos, from a Windows Store app to other networked devices, such as TVs, Xbox 360s, speakers, and other receivers. For example, if you are watching a video on your tablet, with a simple gesture you can decide to stream it to your TV in the living room and share the experience with your family and friends. Besides streaming toward other devices, a Windows Store app can also receive that stream by implementing the Play To receiver contract.

This objective covers how to:

- Register an app for Play To
- Use *PlayToManager* to stream media assets
- Register an app as a *PlayToReceiver*

Introducing the Play To contract

All native Windows 8 apps that are focused on media, such as Music, Video, Photos, and even Internet Explorer 10, implement the Play To contract.

The Play To feature is designed to work with Windows-certified Play To devices. Certified devices compatible with this feature expose the Windows 8/Windows RT logo. As of this writing, the manufacturers that offer compatible devices include Microsoft (Xbox 360), Samsung, Sony, and Western Digital.



For the Play To feature to work, the user must enable the Sharing option when connecting to a network. After the sharing feature is enabled, Windows automatically finds and installs all supported devices connected to the network. You can check the list of available devices in the PC Settings | Devices panel, as shown in Figure 3-8.

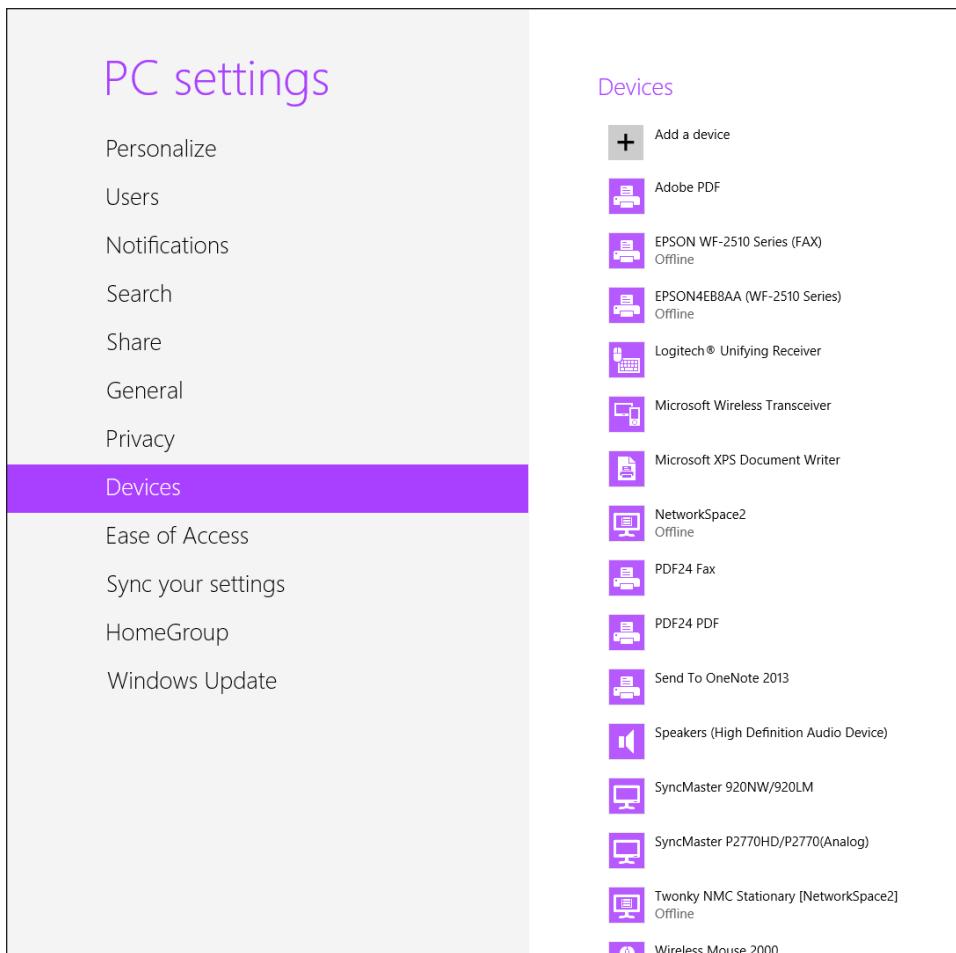


FIGURE 3-8 The PC Settings panel listing all available devices

If there are no Play To devices connected to your network, or if you have turned off the network sharing option, the message shown in Figure 3-9 appears when you click the Devices charm from a Play To application.

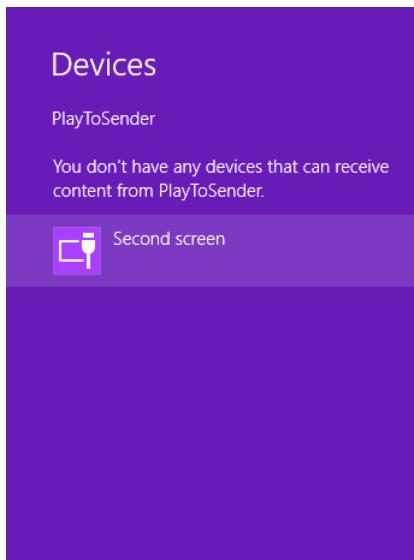


FIGURE 3-9 The Devices charm indicating that no Play To-certified devices are available

It is important to understand that any Windows 8 app containing media elements has the Play To feature enabled by default. If a user selects the Devices charm while running the app and clicks on a target device, Play To streams the media from the first audio, video, or image element on the current page.

You might wonder why you should implement the Play To contract if your app can leverage this functionality out of the box. The answer is that the built-in capability to stream media content is best suited for individual media elements. If you want to provide your application with more sophisticated interactions with users, such as playlists or slide shows, you need to implement the Play To contract.

NOTE DISABLING THE DEFAULT BEHAVIOR OF THE PLAY TO FEATURE

You can disable the default behavior of the Play To feature by setting the *DefaultSourceSelection* of the *PlayToManager* class to *false*.

The Play To contract introduces a high level of abstraction from the underlying streaming technologies, formats, and protocols, making the implementation of this feature in your Windows 8 app plain and simple, as shown in the following code:

```
app.onloaded.=.function().{  
  
    var pm.=.Windows.Media.PlayTo.PlayToManager.getForCurrentView();  
    pm.onsourcerRequested.=.sourceRequestedHandler;  
}  
  
function.sourceRequestedHandler(args).{
```

```

var videoPlayer= document.getElementById("videoPlayer");
if (videoPlayer.!==.null).{
    args.sourceRequest.setSource(videoPlayer.msPlayToSource);
}
}

```

The basic code to implement the contract in your source application is quite straightforward. First, get a reference to the *Windows.Media.PlayTo.PlayToManager* object for the current view by calling the *PlayToManager.GetForCurrentView* method. After you have a reference to the *PlayToManager* instance, register your application for the *SourceRequested* event exposed by the *PlayToManager* class.

In the corresponding event handler, you have to pass the media element that you want to stream to the *SetSource* method of the *PlayToSourceRequestedEventArgs* object as a parameter to the event handler.

When the user activates the Devices charm in the charms bar, the system raises the *SourceRequested* event. Then, when the user selects one of the Play To devices in the Devices panel, the *PlayToManager* starts streaming the HTML5 media element passed to the *SetSource* method to the device selected by the user.

The Play To feature continues to stream the media element to the target device, even if the application has been moved to the background because the user has launched another app. An app that is currently streaming to other devices using the Play To feature is not suspended by the system as long as the Play To receiver is still playing the video or music, or if new images are still being sent to the receiver.

In other words, the system automatically keeps the app running while a Play To session is active. According to the official MSDN documentation, an application has about 10 seconds to send another audio or video file after the current one has ended, or to send a new image after the current one is displayed, before being suspended by the system.

Testing sample code using Windows Media Player on a different machine

Before implementing a basic source application, you need to set up your development environment to be able to test an application.

The Play To feature works between two or more devices on the local network. If you do not have any Play To-certified target device at hand to test your application, you can always use Windows Media Player on another computer connected to the network (PC, notebook, tablet, or virtual machine) as a target device to test the behavior. To do so, launch Windows Media Player on the target device. In the Stream menu, enable the Allow remote control of my Player option, as shown in Figure 3-10. Do not close the Windows Media Player instance.

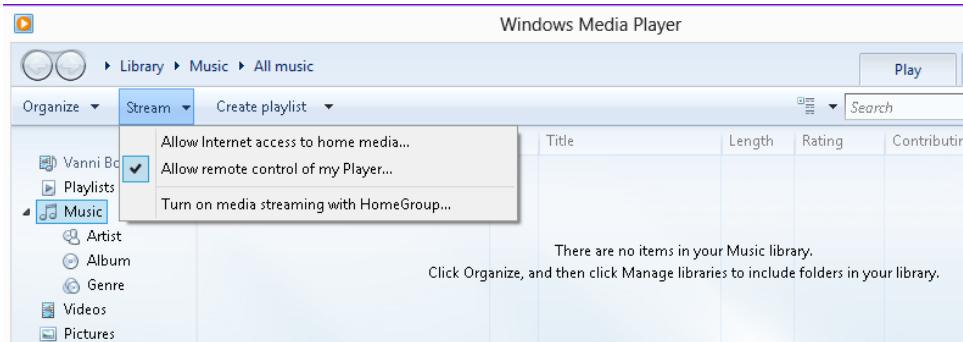


FIGURE 3-10 The Windows Media Player stream options

Go back to your local machine; in the PC Settings panel, click the **Devices** menu item. Windows should find and install the new device as soon as it becomes available. If you do not see the remote instance of Windows Media Player, click the **Add Device** button in the top-right portion of the panel. Figure 3-11 shows a Windows Media Player instance running on a virtual machine.

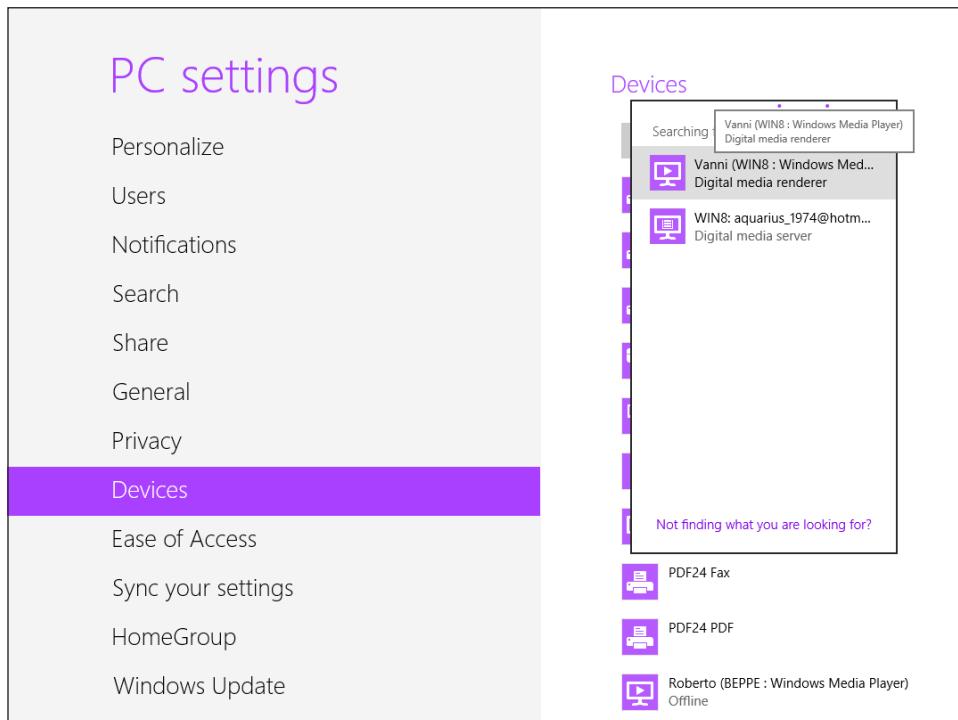


FIGURE 3-11 The Windows Media Player instance added to the available device list

Now you can use the Play To feature to stream an HTML5 media element to the remote machine. In the Windows 8 Start screen, launch one of the native Windows 8 apps that support the Play To feature, such as Photos or Video, select a media element from your libraries, activate the Devices charm, and select the Windows Media Player instance running on the remote machine. Figure 3-12 shows an example of an image streaming from the local machine to the remote instance of Windows Media Player running on a virtual machine equipped with Windows 8.

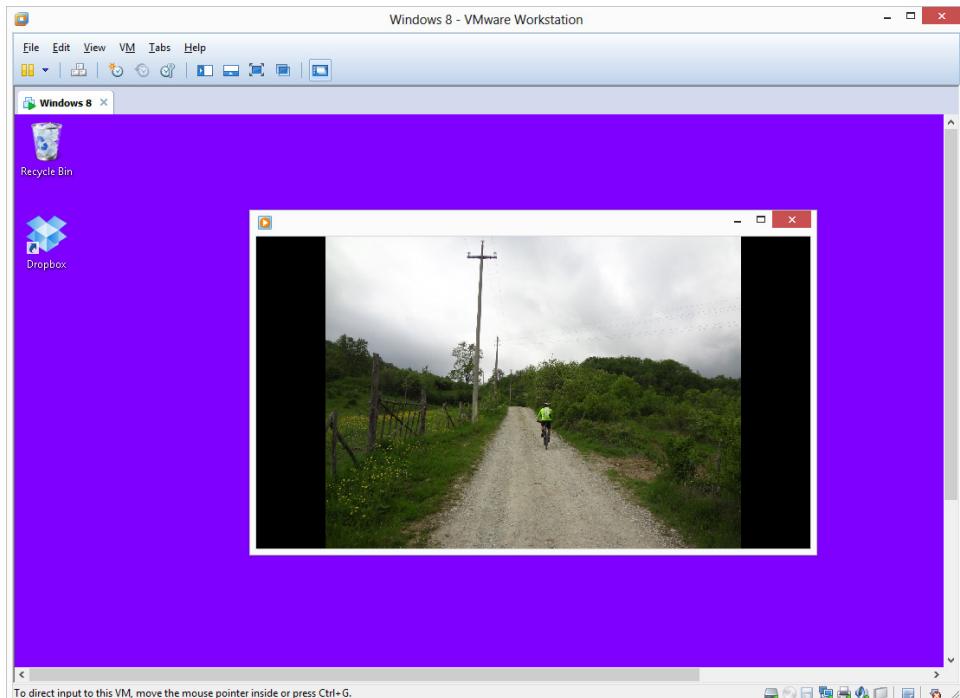


FIGURE 3-12 Streaming an image to a remote instance of Windows Media Player

Implementing a Play To source application

In this section, you learn how to implement a basic Windows Store application that leverages the Play To feature to stream a video to a Play To–certified device.

Listing 3-10 shows the HTML document used in this sample.

LISTING 3-10 Displaying a video using HTML and JavaScript

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Play To Sample</title>
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
```

```

<script src="//Microsoft.WinJS.1.0/js/base.js"></script>
<script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

<link href="/css/default.css" rel="stylesheet" />
<script src="/js/default.js"></script>
</head>
<body>
    <div id="videoFrame">
        <video id="videoPlayer" width="800" height="640" controls="controls" />
    </div>
    <div id="connectionStatus"></div>
    <div id="errorMessage"></div>
</body>
</html>

```

To make the code work, you need something to stream. In this case, the source application plays the first video file available in the user's Videos library. (There must be at least one video in your library if you want to try this sample.) Because the application accesses the user's libraries, you must declare the corresponding capability in your Package.appxmanifest file, or an exception will be raised.

Figure 3-13 illustrates the application manifest with the declared capability.

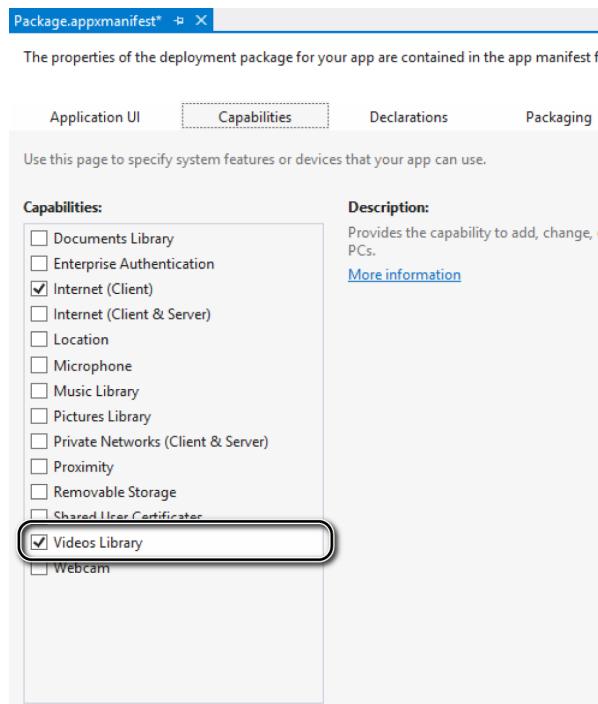


FIGURE 3-13 The Package.appxmanifest file with the required declaration

NOTE MANIFEST DECLARATIONS

In this case, the declaration in the manifest is required because the application will access the user library without asking the user's permission. If you decide to let the user choose which file to stream by using a file picker (as you probably would do in a real application), a specific declaration is not necessary.

In Listing 3-11, the button click handler loads the first video found in the user's Videos library by leveraging the *GetFilesAsync* of the *VideosLibrary* class, and then plays it in a loop by setting the *Loop* attribute of the video media element to *true*.

LISTING 3-11 Playing the first video available in the user's Videos library

```
app.onloaded = function () {  
  
    var videoPlayer = document.getElementById("videoPlayer");  
  
    Windows.Storage.KnownFolders.videosLibrary.  
        getFilesAsync().then(function (resultLibrary) {  
            if (resultLibrary.length > 0) {  
                videoPlayer.src = URL.createObjectURL(resultLibrary[0]);  
                videoPlayer.loop = true;  
                videoPlayer.play();  
            }  
        });  
  
    var pm = Windows.Media.PlayTo.PlayToManager.getForCurrentView();  
    pm.onsourcerRequested = sourceRequestedHandler;  
    pm.onsourceSelected = sourceSelectedHandler;  
}
```

NOTE VIDEO FORMATS

The list of video formats supported by Windows Store apps written in different languages can be found at <http://msdn.microsoft.com/en-us/library/windows/apps/hh986969.aspx>.

After retrieving a video media element from the user's library, the code gets a reference to the *PlayToManager* class for the current page by invoking the *GetForCurrentView* static method. It then wires up both the events exposed by the *PlayToManager* class: *SourceRequested* and *SourceSelected*.

The *SourceRequested* event is raised when the user activates the Devices charm but before the selection of the target device. In the corresponding event handler, the code uses the *SourceRequest* property of the *PlayToSourceRequestedEventArgs* object to set the media element source. The *SourceRequest* property (of type *PlayToSourceRequest*) represents a request to connect a media element with a Play To target device. Listing 3-12 shows the procedure.

LISTING 3-12 The *SourceRequested* event handler

```
function sourceRequestedHandler(args) {  
    try {  
        var deferral = args.sourceRequest.getDeferral();  
        args.sourceRequest.setSource(videoPlayer.msPlayToSource);  
  
        var controller = document.getElementById("videoPlayer").msPlayToSource;  
        controller.connection.addEventListener("error",  
            playToConnectionErrorHandler);  
        controller.connection.addEventListener("statechanged",  
            playToConnectionStateChangedHandler);  
        controller.connection.addEventListener("transferred",  
            playToConnectionTransferredHandler);  
  
        deferral.complete();  
    }  
    catch (ex) {  
        document.getElementById("errorMessage").innerHTML =  
            "Exception encountered: " + ex.message;  
    }  
}
```

**EXAM TIP**

The Windows Runtime gives your application a slice of 200 milliseconds (ms) to provide the media element to stream; otherwise, the *SourceRequested* event times out and the Devices charm does not display any Play To targets for your app. To avoid this problem, you can leverage the *GetDeferral* method of the *PlayToSourceRequest* class to create a deferral before making an asynchronous call to retrieve the media element. The Windows Runtime waits for the media element until the deferral is marked as complete.

NOTE DEADLINE PROPERTY

You can verify the time left to supply the Play To source element by checking the *Deadline* property of the *PlayToSourceRequest* class.

The *PlayToSource* object passed to the *SetSource* method represents the media element to stream to the Play To target. Its *Connection* property (of type *PlayToConnection*) provides some useful information about the state of the connection with the device. In this sample, you use the three events exposed by the *PlayToConnection* class to be notified of any variation in the communication with the target device. More specifically, the *Error* event is raised when an error is encountered for the Play To connection (for example, the target device is not responding, is locked, or has encountered an error). The *StateChanged* event occurs when the state of the Play To connection has changed. The possible values of the connection are

Disconnected, Connected, or Rendering. Finally, the *Transferred* event is raised every time the Play To connection is transferred to the next Play To source.

Listing 3-13 shows the three handlers for the events exposed by the *PlayToConnection* class. For illustrative purposes, the sample uses a simple function, *getName*, to cast the values expressed by the *CurrentState* and *PreviousState* properties into more meaningful strings.

LISTING 3-13 Handling the *PlayToConnection* events

```
function playToConnectionErrorHandler(args) {

    if (args.code == Windows.Media.PlayTo.PlayToConnectionError.deviceError ||
        args.code == Windows.Media.PlayTo.PlayToConnectionError.deviceNotResponding) {
        document.getElementById("connectionStatus").innerHTML +=
            "Error occurred. Disconnecting...<br />";
    }
}

function playToConnectionStateChangedHandler(args) {

    var status = document.getElementById("connectionStatus");
    var states = Windows.Media.PlayTo.PlayToConnectionState;

    status.innerHTML +=
        "StateChanged: PreviousState = " + getName(states, args.previousState) + "<br/>" +
        "StateChanged: CurrentState = " + getName(states, args.currentState) + "<br/>";
}

function getName(enumtype, enumvalue) {
    for (var enumName in enumtype) {
        if (enumtype[enumName] == enumvalue)
            return enumName;
    }
}

function playToConnectionTransferredHandler(args) {

    document.getElementById("connectionStatus").innerHTML +=
        "Transferred: PreviousSource = " +
        args.previousSource + "<br /> CurrentSource = " + args.currentSource;
}

function sourceSelectedHandler(args) {
    if (!args.supportsVideo) {
        var errorMessage = document.getElementById("errorMessage").innerHTML = "<p>" +
            args.friendlyName + " does not support video.</p>";
    }
}
```

Figure 3-14 shows a portion of the app's window, which displays different states of the connection with the target device during the streaming.



```
State changed: PreviousState = Disconnected  
State changed: CurrentState = Connected  
State changed: PreviousState = Connected  
State changed: CurrentState = Rendering  
State changed: PreviousState = Rendering  
State changed: CurrentState = Disconnected  
State changed: PreviousState = Rendering  
State changed: CurrentState = Disconnected
```

FIGURE 3-14 Changes in the connection state of the target device

The *SourceSelected* event is fired when the user selects the target device. If the user leaves the Devices charm without selecting a device, the event is not raised. The corresponding event handler can be the right place to make sure that the device is selected because a Play To target actually supports the specific media element provided for the stream, as illustrated in the following code:

```
function sourceSelectedHandler(args) {  
    if (!args.supportsVideo) {  
        document.getElementById("errorMessage").innerHTML = "<p>" +  
            args.friendlyName + " does not support video.";  
    }  
}
```

The *FriendlyName* property of the *PlayToSourceSelectedEventArgs* object passed as a parameter to the *sourceSelectedHandler* function identifies the Play To target device on the local network.

To start the streaming of a media element, as well as to stop it and disconnect the target device, you have to rely on the Play To flyout that can be activated by clicking the Devices charm. Although you cannot start streaming or disconnect the target device programmatically, you can take advantage of the *ShowPlayToUI* static method exposed by the *PlayToManager* class. Calling this method, as shown in the following code, results in the visualization of the above-mentioned Play To flyout:

```
function ShowPlayToUserInterface(args) {  
    Windows.Media.PlayTo.PlayToManager.showPlayToUI();  
}
```

Registering your app as a Play To receiver

Now that you have seen how to implement the Play To contract that enables a Windows Store app to stream multimedia files to a certified device, let's move on to build a Windows 8 application that can receive and display streams coming from a client source. Whereas the implementation of the Play To contract in a source application relies on the *PlayToManager* class, a Windows 8 Play To receiver application has to leverage the *PlayToReceiver* class.

First, you must update the Package.appxmanifest file to enable the target application to act as a server on the private network, as shown in Figure 3-15.

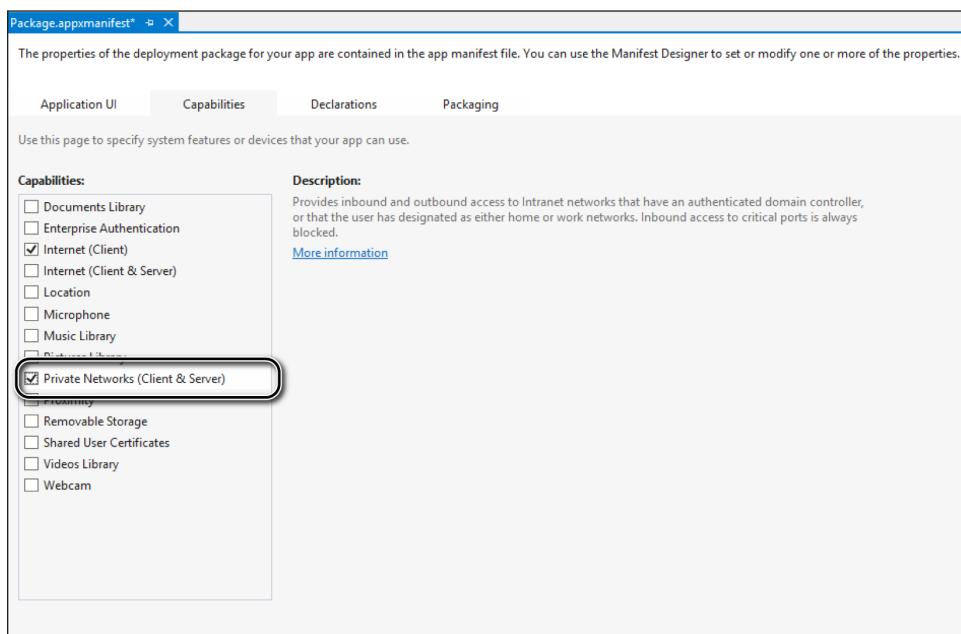


FIGURE 3-15 The Package.appxmanifest editor with the Private Networks (Client & Server) capability enabled



EXAM TIP

The Private Networks (Client & Server) capability provides inbound and outbound access to private networks through the firewall. This capability is typically used for games and apps that share data across the network. If you forget to add the declaration mentioned previously, when you try to start the receiver by calling the *StartAsync* method of the *PlayToReceiver* class, you end up with an exception.

After the required capability is declared in the application manifest, you can start coding the target application. First, you need to add a few HTML tags to show the video streaming from the source application. Listing 3-14 shows the complete definition of the default.html file.

LISTING 3-14 The complete default.html file

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>PlayToReceiverJS</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <table>
        <tr>
            <td><button id="btnStartReceiving">Start Play To Receiver</button></td>
            <td style="text-align:right"><button id="btnStopReceiving">
                Stop Play To Receiver</button></td>
        </tr>
        <tr>
            <td colspan="2"><video id="videoPlayer" width="800" height="600" /></td>
        </tr>
        <tr>
            <td colspan="2"><div id="message" /></td>
        </tr>
    </table>
</body>
</html>
```

To act as a Play To receiver, the application must instantiate a *PlayToReceiver* object to handle the communication with Play To client devices. The application can then display the content streamed from those devices using the user's own media elements or controls.

The *StartReceivingButton_Click* handler method shown in Listing 3-15 performs this operation.

LISTING 3-15 Initializing the Play To receiver app

(code omitted)

```
app.onloaded = function (args) {
    btnStartReceiving.addEventListener("click", startReceiverButton_click);
    btnStopReceiving.addEventListener("click", stopReceiverButton_click);
}

var receiver;
var display;
var videoPlayer;

function startReceiverButton_click() {
    try {
        if (receiver == null) {
            receiver = new Windows.Media.PlayTo.PlayToReceiver();
            receiver.friendlyName = "MyPlayToReceiver";
            receiver.addEventListener("currenttimechangerequested",
                receiver_CurrentTimeChangeRequested);
            receiver.addEventListener("mutechangerequested",
                receiver_MuteChangeRequested);
            receiver.addEventListener("pauserequested", receiver_PauseRequested);
            receiver.addEventListener("playbackratechangerequested",
                receiver_PlaybackRateChangeRequested);
            receiver.addEventListener("playrequested", receiver_PlayRequested);
            receiver.addEventListener("sourcechangerequested",
                receiver_SourceChangeRequested);
            receiver.addEventListener("stoprequested", receiver_StopRequested);
            receiver.addEventListener("timeupdaterequested",
                receiver_TimeUpdateRequested);
            receiver.addEventListener("volumechangerequested",
                receiver_VolumeChangeRequested);

            receiver.supportsVideo = true;
            receiver.supportsAudio = false;
            receiver.supportsImage = false;

            videoPlayer = document.getElementById("videoPlayer");
            videoPlayer.addEventListener("error", videoPlayer_Error);
            // you can subscribe to the other video player events here

            receiver.startAsync().done(function () {
                if (display == null) {
                    display = new Windows.System.Display.DisplayRequest();
                }
                display.requestActive();
                message.innerHTML = ''' + receiver.friendlyName + '' started.''';
            });
        }
    } catch (e) {
        receiver = null;
        message.innerHTML = "Failed to start receiver.";
    }
}
```

Let's review the code in more detail. After creating an instance of the *PlayToReceiver* class, the code sets its *FriendlyName* property. Setting this property is fundamental because it identifies the name of the Play To receiver as it appears on the network (see Figure 3-16).

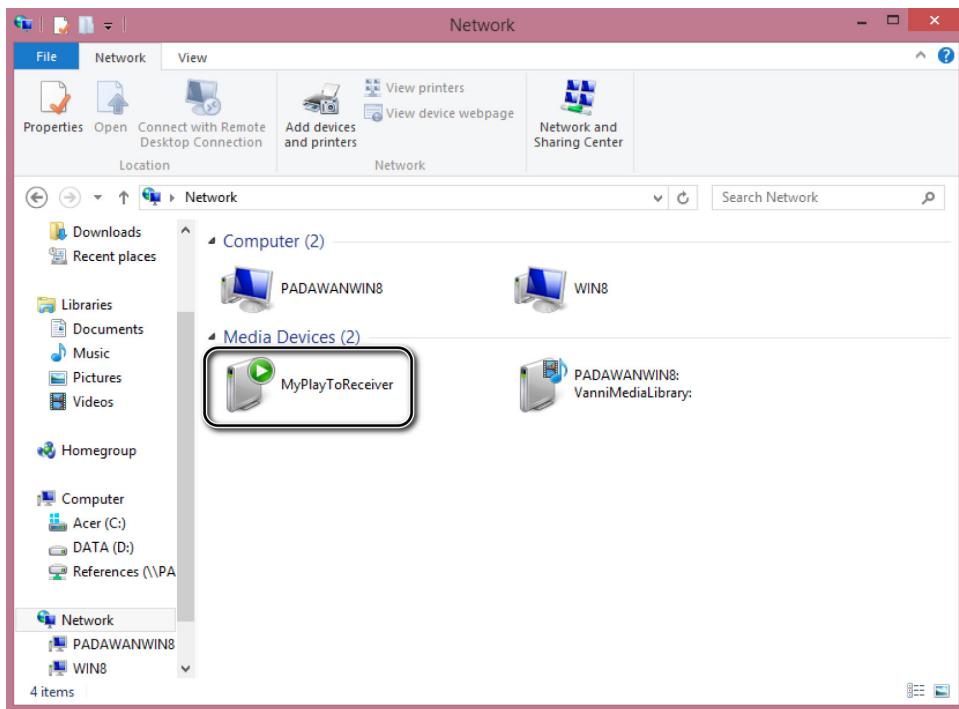


FIGURE 3-16 The Play To target app as it appears on the network

After the *PlayToReceiver* class is instantiated and a friendly name assigned, you have to subscribe to the various events exposed by that class. These events are fired by the system in response to the actions performed by the user on the client application. Here is the complete list of events exposed by the *PlayToReceiver* class:

- **CurrentTimeChangeRequested** The playback time location has changed.
- **MuteChangeRequested** The audio has been (un)muted.
- **PauseRequested** The playback has been paused.
- **PlaybackRateChangeRequested** The rate of the playback has changed.
- **PlayRequested** The playback has started.
- **SourceChangeRequested** The media source for the Play To receiver has changed.
- **StopRequested** A request has been made for the Play To receiver to stop playing the streamed media.
- **TimeUpdateRequested** The current playback position has changed.
- **VolumeChangeRequested** The volume has changed.



EXAM TIP

If you forget to subscribe even to one of the preceding events, when you invoke the `StartAsync` method to start receiving the stream, you get an exception informing you that “A method was called at an unexpected time.” The same exception is raised in case you have forgotten to assign a friendly name to the receiver.

For this reason, it is crucial to remember to subscribe to *all* the events exposed by the `PlayToReceiver` class, even if some of them make sense only for certain types of media elements, such as videos and music. This behavior might seem odd, but if the streaming target application did not respond to all the events raised by the source application, a poor user experience would result.

NOTE HANDLERS

You can decide not to implement the handlers of the events that do not relate to the media elements that your receiver application is targeting. For example, if your receiver targets only images, implementing the `MuteChangeRequested` event does not make much sense.

To filter the type of media that your receiver application can stream, you can leverage the three `Boolean` properties at the end of the previous code snippet (that is, `SupportsAudio`, `SupportsVideo`, and `SupportsImage`). They indicate to the system what kind of media element the receiver actually supports (in this example, just audio elements).

Finally, the `StartAsync` method of the `PlayToReceiver` class starts the Play To receiver and advertises it on the network as a digital media renderer, ready to receive streams from a source application:

```
receiver.startAsync().done(function () {
    message.innerHTML = "'" + receiver.friendlyName + "' started.";
});
```

Analogously, when you want your target application to stop receiving the streaming from the source application, you can invoke the `StopAsync` method of the `PlayToReceiver` class and remove all the event handlers:

```
function stopReceiverButton_click(e) {
    try {
        if (receiver != null) {

            receiver.stopAsync().done(function () {

                // Unsubscribe all the events
                // (code omitted)

                message.innerHTML = "The receiver has been stopped.";
            });
        }
    }
    catch (e) {
```

```

        message.innerHTML = "Failed to stop '" + receiver.FriendlyName + "'." ;
    }
}

```

The next step is to implement the handlers for the events exposed by the *PlayToReceiver* class so that the target application can react to the actions performed on the client application. Because the communication between the client and the target application is two-way, you can notify the client about a change in the Play To receiver by using one of the *Notify** methods exposed by the *PlayToReceiver* class, as shown in Listing 3-16.

LISTING 3-16 *Notify** methods exposed by the *PlayToReceiver* class

```

function receiver_CurrentTimeChangeRequested(args) {
    if (videoPlayer.currentTime !== 0 || args.time !== 0) {
        videoPlayer.currentTime = args.time / 1000;
        receiver.notifySeeking();
    }
}

function receiver_MuteChangeRequested(args) {
    videoPlayer.muted = args.mute;
}

function receiver_PauseRequested() {
    videoPlayer.pause();
    receiver.notifyPaused();
}

function receiver_PlaybackRateChangeRequested(args) {
    videoPlayer.playbackRate = args.rate;
}

function receiver_PlayRequested() {
    videoPlayer.play();
}

function receiver_SourceChangeRequested(args) {
    if (args.stream != null) {
        var mediaStream = MSApp.createBlobFromRandomAccessStream(
            args.stream.contentType, args.stream);
        videoPlayer.src = URL.createObjectURL(mediaStream, false);
    }
}

function receiver_StopRequested() {
    if (videoPlayer.readyState != 0) {
        videoPlayer.pause();
        videoPlayer.currentTime = 0;
        receiver.notifyStopped();
    }
}

function receiver_TimeUpdateRequested() {
    receiver.notifyTimeUpdate(videoPlayer.currentTime * 1000);
}

```

```
}

function receiver_VolumeChangeRequested(args) {
    videoPlayer.volume = args.volume;
    receiver.notifyVolumeChange(videoPlayer.volume, videoPlayer.muted);
}

The same pattern can be used to handle the events exposed by the video player. The following code excerpt shows the handler for the Error event, as an example:
```

```
function videoPlayer_Error() {
    receiver.notifyError();
    receiver.notifyStopped();
}
```



Thought experiment

Interacting with a Play To target app

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

Suppose you are developing a Windows Store app that is able to present the user's pictures in a slide show. It also provides the ability to stream those images to a Play To receiver app by implementing the Play To contract. When the slide show is playing only locally, it has a timer that triggers a new photo to be displayed at regular intervals. However, when streaming to the target device, your app needs to know if the target device has already finished rendering the preceding image before sending a new image to the target.

How would you implement this behavior? What event could you leverage to verify that the target app is ready for a new stream?

Objective summary

- By implementing the Play To contract, a Windows Store app can stream media elements to any Play To-certified device connected to the same private network (source app), as well as receive a stream from another Windows Store app (receiver app).
- To stream media elements to a Play To target, you have to get a reference to the *PlayManager* object for the current view, subscribe to the *SourceRequested* event, and provide a media element to stream.
- To implement a Play To receiver app, you have to instantiate a *PlayToReceiver* object to handle the communication with Play To client devices and display the content streamed from those devices using your own media elements or controls.

- Before starting to receive the stream by calling the *StartAsync* method of the *PlayToReceiver* instance, remember to set the *FriendlyName* property (which will broadcast over the network) and wire up all the events exposed by the *PlayToReceiver* class.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. What is the first step of implementing the Play To contract in a source application?
 - Create a new *PlayToManager* object through the *new* keyword.
 - Register your application for the *SourceRequested* event exposed by the *PlayToManager* class.
 - You do not need any reference to a *PlayToManager* object, just use its static methods as needed.
 - Get a reference to the *PlayToManager* instance for the current view by calling the static method *GetForCurrentView*.
2. In a Play To source app, what event is raised when the user activates the Devices charm but before selecting the target device?
 - SourceRequested* event of the *PlayToManager* class.
 - SourceSelected* event of the *PlayToManager* class.
 - StateChanged* event of the *PlayToSource* class.
 - No event is raised.
3. In a Play To receiver app, what happens if you do not set a value for the *FriendlyName* property of the *PlayToReceiver* instance?
 - In the Devices charm, the receiver appears as Unknown Device.
 - The receiver app is simply ignored and cannot receive the stream.
 - The receiver app raises an exception as soon as the *StartAsync* method of the *PlayToReceiver* instance is invoked.
 - The receiver raises an exception as soon as you launch the application.
4. In a Play To receiver app, which of the following capabilities must be declared in the Package.appxmanifest file to be able to receive the stream?
 - Internet (Client)
 - Internet (Client & Server)
 - Music library and/or Videos library, depending on the type of the media element
 - Private Networks (Client & Server)

Objective 3.3: Notify users by using Windows Push Notification Service (WNS)

The WNS lets an application send toasts or tile and badge updates to a Windows 8 user. The service, hosted in the cloud, manages the communication mechanism to the user's device in a transparent and efficient way. To bring the notification to the correct address, the client application needs to request a notification channel to the client application programming interfaces (APIs) and send the returned uniform resource identifier (URI) to the application back end. This URI represents the client univocally and lets the back end send notifications to the desired device.

This objective covers how to:

- Request, create, and save a notification channel
- Authenticate with WNS
- Call and poll the WNS

Requesting and creating a notification channel

A notification channel is represented by a URI and is used to send notifications to the user via the WNS.

A client application can request the channel URI to the Windows Runtime and send it to a back-end service that stores it in persistent storage (independent from the WNS), and can use it to send toasts or tile and badge updates. For example, a weather app can request the channel to the Windows Runtime and send it to the application back end together with some user preferences, such as favorite cities and temperature thresholds. This data can be used by the back-end logic to alert the user if the temperature is going higher than a user-entered value or simply update the application tile with the current temperature.

A *channel* represents a single user on a single device for a specific application, which means that two applications on the same device never share the notification channel. If the user uses two devices with the same application installed, the user receives two different channels. The back-end service needs to manage these situations. For example, the back-end service can simply store two channels for a user that works with the weather application on two devices.

To begin working with the push notification APIs, you do not need to add a reference to the main project. If you want, add the *Windows.Networking.PushNotification* namespace.

To create a channel, you can use the *CreatePushNotificationChannelForApplicationAsync* method, which creates and returns the notification channel in the form of a *PushNotificationChannel* instance. Always check for errors in the call to this method using a promise, as in the following code:

Sample of JavaScript

```
var channel;
var pushNotifications = Windows.Networking.PushNotifications;
var channelOperation = pushNotifications.PushNotificationChannelManager
    .createPushNotificationChannelForApplicationAsync();

return channelOperation.then(function (reqChannel) {
    channel = reqChannel;

    // Everything fine. The channel URI is in by reqChannel.uri.

    var dialog = new Windows.UI.Popups.MessageDialog(req.Channel.uri);
    dialog.showAsync();
},
function (error) {
    // Could not create a channel. The error number is in error.number.
}
);
});
```

The *PushNotificationChannel* exposes the *Uri* property that represents the channel's unique URI, to which notifications are sent. The result of the presented code is shown in Figure 3-17.

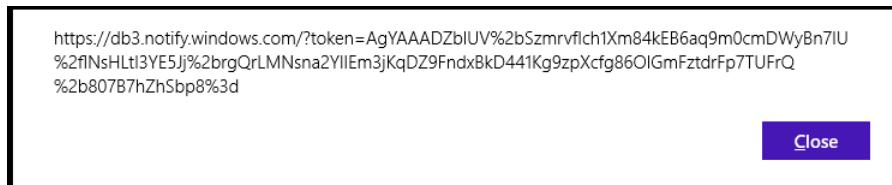


FIGURE 3-17 Notification channel URI in a message dialog box

The application needs to send the URI to the application back end in a secure way. If you send this information to a web service, consider using a form of encryption of the message or at least use HTTPS as the transport.



EXAM TIP

The application need to request a channel each time the application is launched and update the back end when the URI changes.

Because the application needs to request a channel URI on launch, the app can compare the returned information with the previous channel. If the channel URI is different from the previous one, the application must send the new channel to the back end. If the channel is the same, the application does not need to send the channel URI to the server. The application can send the channel to the back end to perform this check, but it is more convenient to store the channel locally to avoid useless round trips.

Best practices suggest performing the following steps during the first application launch:

1. Request a channel to the Windows Runtime.
2. Store the channel in the local storage for later check.
3. Send the channel URI to the back end.

When the application is launched, the code can check local storage for a previous channel URI and compare the value with the new one. If the channel is the same, no other actions are required. If the channel is different, the app sends the new channel URI to the back end. Remember to include logic to send a new channel if:

- The application has never requested a channel.
- The application has never sent the channel URI to the server.
- The last attempt to send the channel URI was not successful.

An application can have multiple valid channels at the same time without any problems. Moreover, each channel remains valid until it expires. There is no need to delete channels or perform any sort of operations.

It is highly important to request a channel every time the application starts because a channel expires in 30 days. There are more chances to work with a fresh channel if the user does not work with the app for some time.

If you are concerned that the user might not launch the application for 30 days, you can implement a background task to request the channel and perform the described operations on a regular basis.

MORE INFO BACKGROUND TASKS

For more information on background tasks, see Chapter 1, “Develop Windows Store apps.”

If the Internet is not available when the application is launched (and a channel is requested using *CreatePushNotificationChannelForApplicationAsync*), an exception is thrown. The best way to handle this exception is to retry the operation three times in 30 seconds, with a delay of 10 seconds each time. If all three attempts fail, there are few chances to obtain a channel; the application should wait for the next application launch to try the described process again.

Once the application no longer needs a notification channel, the related code can call the *Close* method of the *PushNotification* instance to close the channel. If the notification channel is used to update the tile, after closing the channel, the application can clear the corresponding tile calling the *Clear* method of the *TileUpdater* class.

Sending a notification to the client

The back end that receives the notification channel URI is responsible for storing this information and making it available to the code that sends the notification on the channel.

For example, the web service can store the channel URI in a SQL Azure database or in a Windows Azure Storage Account table together with the user preferences. The business logic of the back end can inspect the user preferences and use the URI to send to the WNS the notification message that will be delivered to the client.

NOTE **CLIENT AND SERVICE ARE DECOUPLED**

There is no direct connection between the back end and the client; the WNS manages the communication transparently.

The back end can send tile updates, badge updates, and/or toasts to the client using the corresponding XML syntax. This syntax is identical to the one the client application uses to perform the same operation locally. For example, to send a toast with some text in it, the XML payload might look like this:

```
<toast launch="">
  <visual lang="en-US">
    <binding template="ToastText01">
      <text id="1">The weather is changing to cloudy</text>
    </binding>
  </visual>
</toast>
```

If you need to include an image in the toast, you can use the corresponding template:

```
<toast launch="">
  <visual lang="en-US">
    <binding template="ToastImageAndText01">
      <image id="1" src="Cloudy.jpg" />
      <text id="1"> The weather is changing to cloudy </text>
    </binding>
  </visual>
</toast>
```

The same applies to tiles and badges. Use the same template you normally use in the client code.

NOTE **XML**

In your client code you probably used the *NotificationExtensions* library that hides all the XML details. This library creates the XML under the hood.

The XML fragment must be sent using HTTPS to the WNS using an authentication token provided by Windows Live services. Therefore, the first thing the server code needs to do is to ask for the token to the Windows Live service.

To request the authentication token, you need to use the *HttpClient* class and make a round trip to the *login.live.com* authentication service, as shown in the following code. We have chosen C# for the back-end service, but you can write the service in any language and host it on any server-side platform:

Sample of C# code

```
var encSecret = HttpUtility.UrlEncode(secret);
var encSid = HttpUtility.UrlEncode(sid);

var body = String.Format("grant_type=client_credentials&client_id={0}
    &client_secret={1}&scope=notify.windows.com", encSid, encSecret);

string response;
using (var client = new WebClient())
{
    client.Headers.Add("Content-Type", "application/x-www-form-urlencoded");
    response = client.UploadString("https://login.live.com/accesstoken.srf", body);
}
```

The code encodes the “password” (called the secret) and the security identifier (SID), and builds up the string to provide this information to the Windows Live service for the *notify.windows.com* scope. In practice, the code asks for a token to be used on the scope for the secret and SID provided.

To authenticate the request to the Windows Live service, you need to provide the SID and the password you receive during the WNS subscription. Store them in a secure place.

The Windows Live service’s response is in JavaScript Object Notation (JSON) format and directly contains the authentication token. You can deserialize the response by using this code:

```
using (var ms = new MemoryStream(Encoding.Unicode.GetBytes(response.ToString())))
{
    var ser = new DataContractJsonSerializer(typeof(OAuthToken));
    var oAuthToken = (OAuthToken)ser.ReadObject(ms);
    return oAuthToken;
}
```

The code uses a *MemoryStream* to encapsulate the JSON string contained in the response. Then it instantiates the *DataContractJsonSerializer* to read the stream and transform it in the OAuth authentication token.

You need to define the *OAuthToken* class as follows:

```
[DataContract]
public class OAuthToken
{
    [DataMember(Name = "access_token")]
    public string AccessToken { get; set; }
    [DataMember(Name = "token_type")]
    public string TokenType { get; set; }
}
```

The *OAuthToken* exposes a property called *AccessToken* that represents the authentication header to be passed to the service to authenticate the back end during the notification request.

The complete HTTP message is similar to the one presented in the following excerpt:

```
POST https://db3.notify.windows.com/?token=AfUAABCQmGg70M1Cg%2fK0K8rBPcBqHuy%2b1rTSNPM
uIzF6BtvPdT7DM4j%2fs%2bNNm8z511QKZMtyjByKW5uXqb9V7hIAeA3i8FoKR%2f49ZnGgyUkAhzix%2fuSua
sL3jalK7562F4Bpw%3d HTTP/1.1
Authorization: Bearer agFaAQDAAAEGAAACoAAPzCGedIbQb9vRfPF2Lxy3K//QZB78mLTgK
X-WNS-RequestForStatus: true
X-WNS-Type: wns/toast
ContentType: text/xml
Host: db3.notify.windows.com
Content-Length: 189
<toast launch="">
  <visual lang="en-US">
    <binding template="ToastImageAndText01">
      <image id="1" src="Cloudy.jpg" />
      <text id="1"> The weather is changing to cloudy </text>
    </binding>
  </visual>
</toast>
```

Assuming the code to request the token is refactored in a *GetOAuthToken* method, the complete code to create a notification and send it to the service can be similar to the following:

```
var token = GetOAuthToken(secret, sid);
byte[] contentInBytes = Encoding.UTF8.GetBytes(toastXML);

HttpWebRequest request = HttpWebRequest.Create(uri) as HttpWebRequest;
request.Method = "POST";
request.Headers.Add("X-WNS-Type", notificationType);
request.ContentType = contentType;
request.Headers.Add("Authorization", String.Format("Bearer {0}",
    token.AccessToken));

using (Stream requestStream = request.GetRequestStream())
    requestStream.Write(contentInBytes, 0, contentInBytes.Length);

using (HttpWebResponse webResponse = (HttpWebResponse)request.GetResponse())
    return webResponse.StatusCode.ToString();
```

The code asks for the token, as explained previously, and uses it to create an authentication header for the request to the WNS. The notification type is expressed by another HTTP header called X-WNS-Type. The code performs an HTTP POST to the service using the content of the XML representation of the notification.

To request the SID and the secret, the developer must register the application with the WNS using the Windows Store Dashboard (see Figure 3-18).

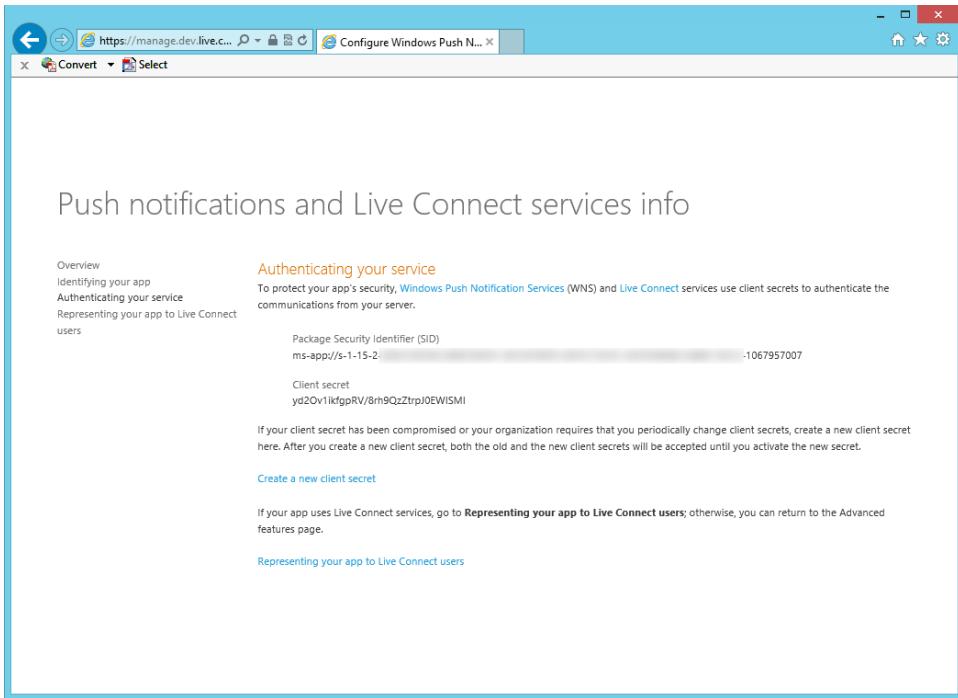


FIGURE 3-18 Notification channel URI in a message dialog

After creating the application in the Windows Store App Development portal, you must go to Advanced Features and then select **Push notification and live connect services info**.

NOTE PACKAGE UPLOAD

You do not need to upload the application package. It is sufficient to reserve the application name.

On this page, select the Identifying your app link and then the Authenticating your service link to display the package SID and the client secret. (Refer to the dashboard in Figure 3-18.)

The request parameters for authenticating the cloud service to the Windows Live service are described in Table 3-1.

TABLE 3-1 Parameters for authenticating a cloud service to a Windows Live service

Parameter	Description
<i>grant_type</i>	Must be set to <i>client_credentials</i>
<i>client_id</i>	Package SID for your cloud service
<i>client_secret</i>	Secret key for your cloud service
<i>scope</i>	Must be set to <i>notify.windows.com</i>

The request receives an HTTP status of 200 OK if everything is okay, and the response contains the access token. If something goes wrong, the error codes are described in the OAuth 2.0 protocol (in draft at the time of this writing). If the authentication fails, you receive a 400 Bad Request HTTP status code.

The response parameters are described in Table 3-2.

TABLE 3-2 Response parameters for authentication to a Windows Live service

Parameter	Description
<i>access_token</i>	The access token that the cloud service uses when it sends a notification
<i>token_type</i>	Always returned as <i>bearer</i>

The following is an example of a sample successful response:

```
HTTP/1.1 200 OK
Cache-Control: no-store
Content-Length: 422
Content-Type: application/json

{
    "access_token": "FgBaAQMBAAAALYAAY/c+Huwi3Fv4Ck10UrKNmtxR06Njk2MgA=",
    "token_type": "bearer"
}
```

You already saw the class to deserialize this response in this section.

To avoid round-trips, you can cache the received token, but be sure to catch the exception for an expired token, as shown in the following code:

```
catch (WebException webException)
{
    string exceptionDetails = webException.Response.Headers["WWW-Authenticate"];
    if (exceptionDetails.Contains("Token expired"))
    {
        GetAccessToken(secret, sid);

        // Retry. Set a maximum retry policy
    }
}
```

NOTE WNS REQUEST AND RESPONSE PARAMETERS

For a complete list of request and response parameters to the WNS, refer to the official MSDN documentation at <http://msdn.microsoft.com/library/windows/apps/hh868245.aspx>.

If the application is up and running when the notification is sent to the device, the application can intercept the notification and handle it in many ways. The app can intercept the notification before a toast is displayed, or the tile or badge is updated, and then update the notification using internal logic. The app can also suppress the notification. For example, if a weather application receives the current temperature from the WNS as a toast, but the user is already watching the current temperature on the app page, the app can simply intercept and suppress the toast.

The JavaScript code to intercept the notification is shown in Listing 3-17.

LISTING 3-17 Code to intercept notification arrival

```
var pushNotifications = Windows.Networking.PushNotifications;
var channel;
var channelOperation = pushNotifications.PushNotificationChannelManager
    .createPushNotificationChannelForApplicationAsync();
return channelOperation.then(function (newChannel) {
    channel = newChannel;
    channel.addEventListener("pushnotificationreceived",
        onNotificationReceived, false);
},
function (error) {
    // ...
});
});
```

The corresponding event handler is shown in Listing 3-18.

LISTING 3-18 Event handler for receiving a push notification

```
var content;
function onNotificationReceived(e) {
    var notification;

    switch (e.notificationType) {
        case pushNotifications.PushNotificationType.toast:
            notification = e.toastNotification.content.getXml();
            break;
        case pushNotifications.PushNotificationType.tile:
            notification = e.tileNotification.content.getXml();
            break;
        case pushNotifications.PushNotificationType.badge:
            notification = e.badgeNotification.content.getXml();
            break;
        case pushNotifications.PushNotificationType.raw:
            notification = e.rawNotification.content;
            break;
    }

    e.cancel = true;
}
```

The code tests the *NotificationType* property of the received *PushNotificationEventArgs* and the type using the WinRT *PushNotificationType* enum. For each kind of notification, it retrieves the corresponding XML that can eventually be evaluated to perform some action. For a toast notification, the application performs some logic (probably on the XML content) and decides to suppress the notification.

You can use a raw notification, which is a type of push notification, to send short messages from your app's cloud service to an app through the WNS. Unlike the XML payload of other push notification types (tile, toast, and badge), the payload contained in a raw notification is strictly app-defined and is not used to convey UI data.

The background task that will be activated when the system receives the notification from the WNS needs to be registered in the application manifest. Use the *pushNotification* value for the type of the task, as follows:

```
<Extension Category="windows.backgroundTasks" StartPage="js\rawbackgroundtask.js" >
  <BackgroundTasks>
    <Task Type="pushNotification"/>
  </BackgroundTasks>
</Extension>
```

The application registers the task by using a *PushNotificationTrigger* as the trigger to fire the task:

```
var builder = new Windows.ApplicationModel.Background.BackgroundTaskBuilder();
var trigger = new Windows.ApplicationModel.Background.PushNotificationTrigger();

builder.name = "rawBGTTask";
builder.taskEntryPoint = "js\\ rawbackgroundtask.js";
builder.setTrigger(trigger);
```

The task is activated when a raw notification arrives from the WNS. You can use *TriggerDetails* to retrieve the instance of the raw notification and use the related *Content* property to inspect the content of the notification. Use the following code as a reference:

```
(function () {
  "use strict";
  var bgTaskInstance = Windows.UI.WebUI.WebUIBackgroundTaskInstance.current;
  function doWork() {
    var notificationContent = bgTaskInstance.triggerDetails.content;

    // ...

    close();
  }
  doWork();
})();
```



Thought experiment

Receiving notifications

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are developing a simple gaming application that needs to receive notifications from the cloud when a player beats your best score.

Which should be the application data flow from the application to the WNS?

Objective summary

- Push notifications are a powerful and lightweight way to send toasts to the user or to update tiles and badges.
- The application asks the Windows Runtime library for a channel and sends the related URI to the application back end.
- The back end uses the channel URI to send notification to the device using an HTTP request to the WNS.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. Which class creates a channel from the back end of a Windows Store app?
 - A. *PushNotificationChannelManager* class.
 - B. Every class derived from *PushNotificationChannelManager*.
 - C. There is no need to create a channel.
 - D. *PushNotificationChannel* class.
2. Where can you find the SID to authenticate a request to send a notification to the client?
 - A. The information is in the WNS Authentication Service login page available at <https://db3.notify.windows.com>.
 - B. You receive this info from the Windows Store Dashboard.
 - C. You do not need the SID to send a notification because the client already received the channel.
 - D. The SID is not for authentication; it is the security identifier in the received token.

3. Is it possible to activate a background task when a notification arrives to the application?
- A. Yes, if the application is up and running, it can activate a background task.
 - B. No, the application can receive a notification only when it is not up and running, so it cannot activate a background task.
 - C. Yes, but only for lock screen–enabled applications.
 - D. Yes, by registering the task with a *PushNotificationTrigger*.

Chapter summary

- The Print contract enables a Windows Store app to access the Windows print system. First, you need to obtain a reference to a *PrintManager* instance for each view that you want users to be able to print. Second, you need to implement a *PrintTask* instance representing the actual printing operation. Third, you have to create a *PrintDocument* instance to hold a reference to the content that you want to print and handle the events raised during the printing process.
- The Play To feature enables you to stream multimedia from a Windows Store app to other Play To–certified devices, such as TVs, Xbox 360s, speakers, and other receivers. Besides streaming toward other devices, a Windows Store app can also receive that stream by implementing the Play To receiver contract.
- The Windows Push Notification Service (WNS) sends notifications to the user by using toasts. The client application requests a channel to the API and sends the returned channel to some back end, such as a cloud-based service. The back end can then store the channel to use later to send notification to the user.

Answers

This section contains the solutions to the thought experiments and answers to the lesson review questions in this chapter.

Objective 3.1: Thought experiment

A possible work-around that would allow your users to print only a subset of the entire document is to handle pagination of the HTML document before sending it to the print subsystem. You might let the user decide which pages to print through some UI control, such as a text box, before activating the Devices charm. (As you learned, after the document has been sent to the print subsystem, it cannot be modified.)

Let's say, for example, that the user has chosen to print only the first few pages of a large document. When the user activates the Devices charm (or clicks the Print button, in case you want to leverage in-app printing), the app creates an HTML document on the fly, putting together only the selected pages. Then it sends the (reduced) document to the print subsystem for preview. Keep in mind that, after the document has been sent for preview, the user cannot modify the initial selection. If she wants to print a different range of pages, her only option is to close the flyout and start over. However, this kind of operation would alter the standard print user experience and should therefore be used only when no other options are available.

Objective 3.1: Review

1. **Correct answer:** A
 - A. **Correct:** This is the class responsible for orchestrating the entire printing process.
 - B. **Incorrect:** This class defines a reusable object that sends output to a printer.
 - C. **Incorrect:** This class represents a collection of methods and properties for managing the options that define how the content is to be printed.
 - D. **Incorrect:** This class represents a specific printing operation.
2. **Correct answer:** C
 - A. **Incorrect:** This event is raised when a print task begins submitting content to the print subsystem.
 - B. **Incorrect:** This event is fired when the printing operation completes.
 - C. **Correct:** The *PrintTaskRequested* event exposed by the *PrintManager* class is raised when the user activates the Devices charm.
 - D. **Incorrect:** When the user activates the Devices charm, the system raises the *PrintTaskRequested* event.

3. Correct answer: A

- A. Correct:** The delegate passed as a second parameter to the *CreatePrintTask* method holds a reference to the method that will be called when the user activates the Devices charm.
- B. Incorrect:** The *Submitting* event of the *PrintTask* instance is raised when a print task begins submitting content to the print subsystem to be printed.
- C. Incorrect:** To notify the user about the outcome of the printing operation, you can use the *Completed* event exposed by the *PrintTask* class.
- D. Incorrect:** To inform the user about the status of the printing operation, you can use the *Progressing* event exposed by the *PrintTask* class.

4. Correct answer: C

- A. Incorrect:** In this event handler, you need to format the page that you want to print based on the settings retrieved from the print target selected by the user. The resulting pages can then be added to the collection of preview pages to be displayed in the preview window.
- B. Incorrect:** In this event handler, you have to supply the print system with the requested page to be displayed in the preview window.
- C. Correct:** In this event handler, you have to add all the pages resulting from the preceding operations to the final collection to be sent to the print target. To accomplish this task, call the *AddPage* method of the *PrintDocument* object. After all the pages are added, signal the system that the collection is ready to be printed by invoking the *AddPagesComplete* method.
- D. Incorrect:** In this event handler, you can leverage the *InvalidatePreview* method exposed by the *PrintDocument* class to invalidate the current print preview every time the user changes one of the properties of the print target.

Objective 3.2: Thought experiment

To understand whether the target device has finished rendering the streamed image before sending a new one, you can leverage the *StateChanged* event exposed by the *Connection* property of the *PlayToSource* media file being streamed. More in detail, when this event occurs, you can check the *PreviousState* and *CurrentState* properties of the *PlayToConnectionStateChangedEventArgs*. For example, if the previous state was *Rendering* and the new state is *Disconnected*, you can assume that the target device has finished rendering the current image, and it is now ready to receive a new stream.

On the other hand, if the current state is *Connected* and the previous state was *Disconnected*, the image that raised the event has just been connected to the Play To receiver. You should wait before sending a new stream. You can also detect whether the slideshow on the target device has been paused by the user. In this case, the previous state would be *Rendering*, whereas the current state would now be *Connected*. When the slide show on the target

device is unpause, the current state would be Rendering, while the previous state would be Connected.

NOTE PLAY TO CONTRACT

You can find a more sophisticated case study that goes through all the possible combinations at <http://code.msdn.microsoft.com/Media-Play-To-sample-ab941e3d>.

Objective 3.2: Review

1. Correct answer: D

- A. Incorrect:** You cannot instantiate a *PlayToManager* object by using the *new* keyword.
- B. Incorrect:** You can register your application for the *SourceRequested* event only after you have obtained a reference to a *PlayToManager* object by calling the *GetForCurrentView* method.
- C. Incorrect:** You need to get a reference to the *PlayToManager* instance before using it.
- D. Correct:** Each view needs a reference to its own specific *PlayToManager* instance that can be retrieved by calling the *GetForCurrentView* static method of the *PlayToManager* class.

2. Correct answer: A

- A. Correct:** The *SourceRequested* event of the *PlayToManager* class is fired as soon as the user activates the Devices charm in the charms bar.
- B. Incorrect:** The *SourceSelected* event is fired when the user selects the target device.
- C. Incorrect:** This event occurs when the state of the Play To connection changes.
- D. Incorrect:** When the user activates the Devices charm in the charms bar, the system raises the *SourceRequested* event.

3. Correct answer: C

- A. Incorrect:** As soon as the *StartAsync* method is invoked, the code throws an exception. Without a valid call to the *StartAsync* method, your app will not be registered on the network.
- B. Incorrect:** As soon as the receiver app tries to start receiving, if the *FriendlyName* property has not been set, the system raises an exception.
- C. Correct:** The receiver app raises an exception as soon as the *StartAsync* method of the *PlayToReceiver* instance is invoked.
- D. Incorrect:** Until the *StartAsync* method is invoked, no exceptions are raised.

4. Correct answer: D

- A. Incorrect:** This capability provides outbound access to the Internet and public networks through the firewall, whereas the Play To feature requires a Play To device connected to the same private network.
- B. Incorrect:** This capability provides inbound and outbound access to the Internet and public networks through the firewall, whereas the Play To feature requires a Play To device connected to the same private network.
- C. Incorrect:** These two capabilities provide programmatic access to the user's Music and Videos libraries and have nothing to do with the Play To feature.
- D. Correct:** This capability provides inbound and outbound access to home and work networks through the firewall. This capability is typically used for games that communicate across the local area network (LAN) and for apps that share data across a variety of local devices, including Play To devices.

Objective 3.3: Thought experiment

During application launch, the application requests a channel to the Windows Runtime, stores the channel in local storage for later check, and sends the channel URI to the back end.

The back end stores the channel URI in remote storage, assigning it to the player name or id (the unique key that represents a player).

For the first game run, the application stores the reached point in the local store and sends it to the remote back end that, in turn, records this information with the channel URI in the player record.

For each subsequent game run, the application compares the score with the one stored locally. If the score is less than the previous one, no action is required. If the score is greater, it is considered to be the "record" (that is, the value is greater than any other user record). The app then sends the score to the remote back end to store the record server-side.

The remote back end compares the new score with the record of every other player. If the new score is the game record, it sends to the WNS an XML message representing a toast with a message stating that a new record has been reached by another user.

Objective 3.3: Review

1. Correct answer: C

- A. Incorrect:** This class is used in the client application.
- B. Incorrect:** There is no need to derive this class.
- C. Correct:** The server code uses a simple HTTP request to send notification to the client.
- D. Incorrect:** This class is used in the client application.

2. Correct answer: B

- A. **Incorrect:** This URL represents the WNS service to send notification. It is not a login page.
- B. **Correct:** The Windows Store Dashboard provides the SID and the client secret to authenticate a request to the WNS.
- C. **Incorrect:** You need the SID and the client secret to send notification requests to the WNS.
- D. **Incorrect:** The SID is one of two pieces of information needed to authenticate the request to the WNS.

3. Correct answer: D

- A. **Incorrect:** A background task can be activated regardless of application states.
- B. **Incorrect:** A background task can be activated regardless of application states.
- C. **Incorrect:** A background task can be activated for every kind of application.
- D. **Correct:** It is the only thing to do to activate a background task for push notifications.

CHAPTER 4

Enhance the user interface

The most visible part of your app is your user interface (UI). If you follow the design guidelines, you can create beautiful apps that will appeal to your users. This chapter helps you further enhance your app's UI.

The chapter first focuses on creating a responsive UI by using asynchronous code with promises and web workers. That way you can make sure that your UI will always respond to user input.

After that, you look at animations. You can use animations to create a great UI but they can also have a more functional benefit. Correctly used animations can help your users more easily find their way around your app. You can create animations by using the built-in animation library or by creating your own animations using Cascading Style Sheets 3 (CSS3).

One other aspect you focus on is creating new controls. Although Windows Library for JavaScript (WinJS) has a lot of standard controls, you sometimes want to create your own controls or extend the existing controls.

Finally, you learn about an often overlooked area: globalization and localization. Planning for globalization and localization can help you create apps with a wide user base and improved usability.

Objectives in this chapter:

- Objective 4.1: Design for and implement UI responsiveness
- Objective 4.2: Implement animations and transitions
- Objective 4.3: Create custom controls
- Objective 4.4: Design apps for globalization and localization

Objective 4.1: Design for and implement UI responsiveness

One of the important design principles of a Windows 8 app is being fast and fluid. To accomplish this with your app, you need to create a responsive UI. One way to achieve this is to never block the UI by performing a long-running operation, which can make users leave your app because they think it's stopped working. Microsoft added the concept of asynchronous operations to the WinJS environment. You can use *promises* and *web workers* to create a fluid app that will appeal to your users.

This objective covers how to:

- Choose an asynchronous strategy between web workers and promises
- Implement web workers
- Nest and chain promises
- Make custom functions promise-aware

Choosing an asynchronous strategy

JavaScript is a single-threaded language. This means that your app is executed on a single thread, the *UI thread*. If you block this thread with a long-running operation, your app appears unresponsive and is certainly not fast and fluid.

You can see an example of this in the following code. Depending on the speed of your CPU, this code takes a while to execute. If you bind this code to a button in your user interface, clicking that button freezes the UI.

```
function mySlowCalculation() {  
    var total = 0;  
    for (var i = 0; i <= 1000000000; i++) {  
        total += i;  
    };  
    return total;  
};
```

To avoid this, you have a few options that all come down to making sure you don't block your UI thread.

One option is to use *asynchronous programming*. This means you execute a long-running operation, like a call to a web service, and schedule some code to execute after the operation is finished. Your asynchronous call finishes immediately and doesn't block the UI. When your asynchronous operation finishes, your scheduled code runs.

One way to achieve this is by using callbacks. When you call an asynchronous method, you pass it a function as one of its parameters that needs to be called when the operation finishes. The problem with callbacks, however, is that they break the logical flow of your code and make your code much harder to read and maintain. The following code sample uses callbacks for asynchronous programming:

```
doSomething(function(foo){  
    foo = processFoo(foo);  
    doMoreStuff(foo, function(bar){  
        var baz = getBaz(foo);  
        stillMore(baz, function(quux){  
            gettingPrettyBad(function(widget){  
                return widget + baz;  
            });  
        });  
    });  
});
```

This is a contrived example, but you can see how callbacks make your code much harder to read. This is why Microsoft chose to use the promise pattern in WinJS. When using promises, you can create asynchronous code in an easy way. The first part of this objective discusses the promise pattern in further depth.

Another option is to use *web workers*, a new addition to Hypertext Markup Language 5 (HTML5). Web workers offload work to a separate thread so that your UI thread is free to respond to user input while another thread is doing all the work. The second part of this objective will provide details on how to use web workers.

Both strategies have their uses. Web workers are ideal for situations in which you want to execute a complex, CPU-bound operation. Examples are cryptography, sorting a large amount of data, and processing an operation in parallel. Promises are an essential part of WinJS. You need to use them when interacting with the existing framework functions. You can also create your own promises to wrap an operation that could be potentially long running, whether it is input/output (I/O) or CPU bound.

Implementing promises and handling errors

A promise is just a JavaScript object. If you look at the source code for the *WinJS.Promise* object in the *base.js* file in your Windows Store app, you'll see that a promise has all kinds of methods, such as *then*, *done*, and *join*.

MORE INFO PROMISES

Microsoft implements the Common JS Promises/A proposal for promise support in Windows Store apps. You can find this proposal at <http://wiki.commonjs.org/wiki/Promises/A>.

A promise, as the name suggests, promises to give you a value in the future. You don't know exactly when that value will be available, but you can schedule code to be executed when your promise object is finished.

An advantage of promises compared to callbacks is that promises provide a standard interface for dealing with asynchronous code. With a callback, the method signature defines which callback methods you can attach (for example, a method for completion, error handling, and progress). However, the number and order of arguments can be different for each callback.

A promise has standard methods you can call on every promise object. This way, you have a consistent way of working with promises.

A promise can have three different states:

- Unfulfilled
- Fulfilled
- Failed

A promise starts out with a state of *unfulfilled*, meaning that it has no value available yet. From that state it can move to either *fulfilled* or to *failed*. After the fulfilled or failed state has been reached, the value of the promise object doesn't change.

Using promises

For an example of how to use promises, look at the *WinJS.UI.processAll* method that is used at the start of your application to initialize all your controls. This method returns a promise object. That promise object will be finished when binding to all your controls has completed. This can potentially take quite some time, which is why the method is executed asynchronously.

Let's say that you want to add a rating control to your app, and you want to initialize this control to have an average rating of 3. You can initialize a control without waiting for the promise to finish by using the following code:

```
WinJS.UI.processAll();
// Don't do this:
var control = document.getElementById("ratingControlHost").winControl;
control.averageRating = 3;
```

The *processAll* method runs asynchronously and returns immediately. Therefore, it's possible the runtime is still initializing your controls when you try to access your rating control. If so, the *winControl* property will be *null* and your code will throw an error.



EXAM TIP

Always use a promise on the *WinJS.UI.processAll* method when you want to access a WinJS control.

Instead of executing your initialization code after the *processAll* method, you need to use the *then* method of the promise object to execute your code, as follows:

```
WinJS.UI.processAll().then(function () {
    var control = document.getElementById("ratingControlHost").winControl;
    control.averageRating = 3;
});
```

By using a promise, you create asynchronous code that's still easy to read. You can also chain multiple *then* methods together, as follows:

```
longTaskAsyncPromiseA()
    .then(function () {
        // Execute some code
        return longTaskAsyncPromiseB();
    })
    .then(function () {
        // Execute some code
        return longTaskAsyncPromiseC();
    });
}
```

This code begins by executing a long-running method. When the method is finished, the code starts another asynchronous operation that in turn starts another operation. It's important to notice that chaining is done as a flat list. You could nest the chaining, but that makes it harder to read your code and deal with errors, as shown in the following example:

```
WinJS.UI.processAll().then(
    function (value) {
        console.log("controls set up");
        WinJS.Binding.processAll().then(
            function (value) {
                console.log("data binding done");
            });
    });
});
```

Next to the *then* method, you also have a *done* method on the promise object. The *done* method does not return a new promise object. Therefore, it should be the last method in your promise chain. It can be used to help you with progress reporting and error handling over all your chained promises.

Handling errors

It's possible your asynchronous function could return an error. When using promises, you can register a method that executes when an error occurs.

The following code snippet shows an example of using the *WinJS.xhr* method. You can use this method to wrap calls to the *XMLHttpRequest* class in a promise object. The code attaches a method for success and a method for error handing.

```
WinJS.xhr({ url: 'invalidurl' }).then(
    function (result) { },
    function (error) {
        return new Windows.UI.Popups.MessageDialog(error.message).showAsync();
    });
});
```

Because of the invalid *url* parameter, the code ends up in the error handler, which shows a message dialog using the *Windows.UI.Popups.MessageDialog* class to the user with the message "The system cannot locate the resource specified."

The *MessageDialog* class also implements the promise pattern. You won't find a regular *Show* method on the *MessageDialog*, only a *ShowAsync* method.

When chaining multiple promises, you can use the *done* method to attach an error handler for all promises in one pass, as follows:

```
longTaskAsyncPromise()
    .then(function () { return longTaskAsyncPromiseB(); })
    .then(function () { return longTaskAsyncPromiseC(); })
    .done(null, function (error) { });
```

This example passes only an error handler to the *done* method. If one of the three async methods throws an exception, the error handler will catch it.

There is one important difference between using a *done* and a *then* method with respect to unhandled exceptions. Because a *then* method returns a promise, the exception is silently captured as a part of the state of the promise. Other chained *then* or *done* methods can then access this promise state and handle the exception.

The *done* method doesn't return a value that can capture the exception state. Therefore, an exception in a *done* method is always thrown.

To make sure you catch all errors that can occur in a promise (to log them, for example), you can add a general error handler. The following code shows how to add a global error handler for promises:

```
// Add this line to your initialization code
WinJS.Promise.onerror = errorHandler;

// Add this method as a global error handler
function errorHandler(event) {
    var ex = event.detail.exception;
}
```

Handling progress notifications

An important aspect of creating a great user experience when dealing with asynchronous operations is to provide progress notifications to the user. That way, the user can be certain the app is still functioning and get a sense of how long the operation is going to take.

In addition to a completion and error handler, you can also pass a progress handler to your method. For example, you can add a progress handler to a *WinJS.xhr* call to check the current state of your request and retrieve updates when it changes. If you add a *div* element to your page with an *id* of *xhrReport*, you can use the code in Listing 4-1 to implement progress notifications.

LISTING 4-1 Using progress notifications

```
var xhrDiv = document.getElementById("xhrReport");

WinJS.xhr({ url: "http://www.microsoft.com" })
    .done(function complete(result) {
        xhrDiv.innerText = "Downloaded the page";
        xhrDiv.style.backgroundColor = "#00FF00";
    },
    function error(error) {
        xhrDiv.innerHTML = "Got error: " + error.statusText;
        xhrDiv.style.backgroundColor = "#FF0000";
    },
    function progress(progress) {
        xhrDiv.innerText = "Ready state is " + progress.readyState;
        xhrDiv.style.backgroundColor = "#0000FF";
    });
});
```

Cancelling promises

Because promises represent asynchronous operations that could be potentially long running, it's important to be able to cancel a promise. To cancel a promise, you need to store a reference to the promise object and then call the *cancel* method on it.

Consider the following example for using progress notifications. The *longTaskAsyncPromise* returns a progress value every second. The promise is stored in a variable called *p*. This reference is then used in the click handler of a button to stop the asynchronous operation.

```
var p = longTaskAsyncPromise().then(
    null,
    null,
    function (progress) {
        report.textContent = progress;
    }
);

operationButton.addEventListener('click', function () {
    p.cancel();
});
```

When an asynchronous operation is cancelled, the error handler of your promise is called with all properties (*name*, *message*, and *description*) set to *Canceled*.

Using the *join* method

Running an operation asynchronously is different from running multiple operations in parallel. If you use a promise with a *then* method, the *then* method executes after the promise has finished.

For example, the code in Listing 4-2 shows chained promises running sequentially. It takes four seconds before it displays the Done message.

LISTING 4-2 Sequentially chained promises run after each other

```
WinJS.Promise.timeout(2000)
    .then(function () { return WinJS.Promise.timeout(1000); })
    .then(function () { return WinJS.Promise.timeout(1000); })
    .then(function () { return new Windows.UI.Popups.MessageDialog("Done") .
showAsync(); });
```

But sometimes you want certain operations to run in parallel. For example, if you are firing off multiple web requests and you want to combine the resulting data on the client, you don't have to run them sequentially.

The promise pattern supports this by using the *join* method. You call *join* on a set of promises. The *join* method finishes when all promises are done. Listing 4-3 shows how you can change the code in Listing 4-2 to use the *join* method.

LISTING 4-3 Running multiple promises concurrently by using *join*

```
var promiseArray = [];
promiseArray[0] = WinJS.Promise.timeout(2000);
promiseArray[1] = WinJS.Promise.timeout(1000);
promiseArray[2] = WinJS.Promise.timeout(1000);

WinJS.Promise.join(promiseArray)
    .done(function () { return new Windows.UI.Popups.MessageDialog("Done")
        .showAsync(); });
```

In this case, the Done message will be shown after two seconds. Because the three promises run in parallel, the other two are already finished.

Using time-outs

You can also use a time-out to cancel an asynchronous operation. This way, you can specify a number of milliseconds that the operation can take. If it takes longer, your operation is cancelled and the error handler is executed.

You create such a time-out by using the *WinJS.Promise.timeout* method. This method has two overloads:

- ***WinJS.Promise.timeout(timeout)*** Returns a promise that completes after the specified number of milliseconds. You can use this to execute some code at a specified time in the future.
- ***WinJS.Promise.timeout(timeout, promise)*** Is linked to a promise object. If the time-out expires, the other promise is cancelled and the handlers are executed.

The following code shows an example of creating a time-out of three seconds on a promise:

```
var p;
var timeout = WinJS.Promise.timeout(3000, p = longTaskAsyncPromise());

p.done(null,
    function (error) {
        var e = error.message;
    });
});
```

Creating your own promises

Because the promise class in WinJS is just a regular class, you can also create your own promises. The following code shows how you can create a method that returns a promise. The promise sleeps for one second at a time until it reaches the delay time you can pass to the method. It also sends update notifications to the caller of the method.

```
function longTaskAsyncPromise(secondsDelay) {
    return new WinJS.Promise(function (c, e, p) {
        var seconds = 0;
        var intervalId = window.setInterval(function () {
            seconds++;
            p(seconds);
        }, 1000);
    });
}
```

```

        if (seconds >= secondsDelay) {
            window.clearInterval(intervalId);
            c();
        }
    }, 1000);
});
}

```

The promise constructor takes three parameters: *complete*, *error*, and *progress*. In this example, the error callback isn't used. The progress callback is used to report the number of seconds that have passed, and the completion callback is called when the promise is completed.

In addition to constructing a new promise, you can wrap an existing value into a promise. For example, you already have a concrete value, but you want to pass it to a method that expects a promise. You can then use the *WinJS.xhr* method. This method takes any object and wraps it in a promise. If the value you pass to the *as* method is already a promise, it is returned without any modification.

The next code snippet shows an example in which you have a regular value, wrap it in a promise, and schedule a continuation method.

```
WinJS.Promise.as(5).then(function (arg) {
    return new Windows.UI.Popups.MessageDialog(arg).showAsync();
});
```

In this case, the first promise finishes immediately, and the value of the promise is passed to the *then* handler.

You can also change an existing method to wrap it in a promise, making it promise-aware. Take the following method:

```
function mySlowFunction() {
    var done = false;
    setTimeout(function () {
        console.log("poll");
        done=true;
    }, 10000);
    return done;
};
```

This method returns *false* immediately. The time-out executes the anonymous method only after 10 seconds have passed. To fix this, you can wrap the method in a promise, as follows:

```
function mySlowFunction() {
    var mypromise = new WinJS.Promise(
        function(complete,error,progress){
            var done = false;
            setTimeout(function () {
                console.log("poll");
                done = true;
                complete(done);
            }, 10000);
    });
}
```

```
});  
return mypromise;  
};
```

When you call this method, it returns a promise object. You can then add a continuation method, like this:

```
mySlowFunction().then(function(value){console.log(value)});
```

Using web workers

Using web workers is another way to run code asynchronously. You use a web worker to create a new thread and execute a script on that thread.

The following code creates a new web worker and passes it the script it needs to run:

```
var myWorker = new Worker('worker.js');
```

You communicate with the worker by sending and receiving messages. You send a message by using the *postMessage* method, and you listen for messages in the *onmessage* event.

If you add the following code to a new JavaScript file, you can execute that code as a worker that will listen to your messages and echo them back to the calling thread.

```
self.onmessage = function (e) {  
    self.postMessage("Echo:" + e.data);  
};
```

Exploring available features

Web workers have access to a subset of JavaScript features:

- The *navigator* object
- The *location* object (read-only)
- *XMLHttpRequest* class
- Time-out application programming interfaces (APIs)
- The Application Cache (or AppCache)
- Importing external scripts using the *importScripts* method
- Spawning other web workers

Workers don't have access to the following:

- The Document Object Model (DOM)
- The *window* object
- The *document* object
- The *parent* object

This may seem like a problematic limitation, but it actually simplifies your code and creates a nice model for multithreading in JavaScript. Other languages that do allow you to share

state between two threads, such as C#, have a lot of complicated concepts for managing synchronization between threads. Because web workers don't allow you to share state, you avoid all of those mechanisms.

You can communicate only by using the *postMessage* method. This method enables you to send JavaScript primitives, such as *number* and *string* values. Listings 4-17 and 4-18 show you how this works. Listing 4-4 contains the code for a simple HTML form, and the JavaScript code that starts your worker and sends the messages. Listing 4-5 contains the worker code. Your worker can access both the *message* and *timeout* properties. It uses these to set a *timeout* that returns the data to the caller. (If you want to run this sample, you can put the code in Listing 4-4 in the *body* part of a blank Windows Store app and the web worker code in a file called *echo.js* in the *js* folder.)

LISTING 4-4 Sending an object to a web worker

```
<script>
    var echo = new Worker('js/echo.js');
    echo.onmessage = function (e) {
        new Windows.UI.Popups.MessageDialog(e.data).showAsync();
    }

    window.onload = function () {
        var echoForm = document.getElementById('echoForm');
        echoForm.addEventListener('submit', function (e) {
            echo.postMessage({
                message: e.target.message.value,
                timeout: e.target.timeout.value
            });
            e.preventDefault();
        }, false);
    }
</script>
<form id="echoForm">
    <p>Echo the following message after a delay.</p>
    <input type="text" name="message" value="Input message here." /><br />
    <input type="number" name="timeout" max="10" value="2" />
    seconds.<br />
    <button type="submit">Send Message</button>
</form>
```

LISTING 4-5 Receiving an object in a web worker

```
onmessage = function (e) {
    setTimeout(function () {
        postMessage(e.data.message);
    },
    e.data.timeout * 1000);
}
```

You access the data by using the *data* property on your event. The data you send from your main page to the worker or vice versa is cloned, not shared. This means that changes in your main page or in the worker will not be visible at the other end. If you want to share the changes, you will have to use the *postMessage* method.



EXAM TIP

It's important to understand that you can't directly share data between your main page and a web worker. Instead, you need to use the `postMessage` method and `onmessage` event for communication.

Stopping a web worker

A web worker stays alive until you explicitly kill it. This is important to know because a web worker consumes some memory and will keep using this memory until you stop it.

You can stop a web worker in two different ways:

- From the main page by calling the `terminate` method
- From the web worker itself by calling the `close` method

Listings 4-6 and 4-7 show how you can change the previous example to allow the user to stop the web worker. If the user clicks the button or sends an empty message to the web worker, the web worker is stopped.

LISTING 4-6 Stopping a web worker from the main page

```
<script>
    var echo = new Worker('js/echo.js');
    echo.onmessage = function (e) {
        new Windows.UI.Popups.MessageDialog(e.data).showAsync();
    }

    window.onload = function () {
        var echoForm = document.getElementById('echoForm');
        echoForm.addEventListener('submit', function (e) {
            echo.postMessage({
                message: e.target.message.value,
                timeout: e.target.timeout.value
            });
            e.preventDefault();
        }, false);

        var stopButton = document.getElementById('stopButton');
        stopButton.addEventListener('click', function () {
            echo.terminate();
        });
    }
</script>
```

LISTING 4-7 Stopping a worker from inside the worker

```
onmessage = function (e) {
    if (e.data.message) {

        setTimeout(function () {
            postMessage(e.data.message);
        },

```

```
        e.data.timeout * 1000);
    }
    else {
        self.close();
    }
}
```

Handling errors

When an error occurs in a web worker, the *onerror* event handler is called. This handler receives an event named *error*. The event has the following three fields that are of interest:

- **message** A human-readable error message
- **filename** The name of the script in which the error occurred
- **lineno** The line number in the script file where the error occurred

You can register an event listener for the *onerror* event in your main page. The following code shows how to log the error to the console:

```
echo.onerror = function (e) {
    console.log(['ERROR: Line ', e.lineno, ' in ', e.filename, ': ', e.message].join(''));
}
```

Loading external scripts

Another important feature of web workers is the ability to load external scripts from within your web worker. The worker script itself is loaded asynchronously from your main page. By using the *importScripts* method, you can asynchronously load scripts into your worker scope. You can see some examples in this code:

```
importScripts();
importScripts('foo.js');
importScripts('foo.js', 'bar.js');
```

All these examples are valid. However, the first example won't load any scripts, the second one will load one script, and the third will load two scripts.

After the scripts have been loaded and executed, you can access all global objects from those scripts in your web worker. If a script can't be loaded, a *NETWORK_ERROR* is thrown. Your scripts can be downloaded in any order but they will be executed in the order you passed them.

MORE INFO WEB WORKERS

For an in-depth look at how web workers were designed, see the official W3C information at <http://www.w3.org/TR/workers/>.



Thought experiment

Choosing the correct asynchronous strategy

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are working on a Windows 8 business application for time tracking. You need to issue multiple requests to fetch data from a server with project and client information. In the app, you are processing the data to show it to the user with his personal preferences. This process is quite resource intensive and you are experiencing performance issues.

Answer the following questions:

1. How can you optimize data fetching?
2. What can you do to improve the performance of data processing?
3. With these answers in mind, what are the main scenarios for using a promise or a web worker?

Objective summary

- Promises are an essential element of the WinRT JavaScript libraries for making your code asynchronous.
- You can use the *then* and *done* methods to attach code to a promise. The *join* method can be used to execute multiple promises in parallel.
- You can cancel a promise by using the *cancel* method or by setting a time-out through the *WinJS.Promise.timeout* method.
- You can convert a value to a promise by using the *as* method.
- Web workers can be used to execute code on another thread. You create a web worker by pointing it to a script file and then start it by calling *postMessage*.
- You communicate with a web worker by using *postMessage* and the *onmessage* event. The data you send is cloned.
- You stop a web worker from your main page by calling *terminate* or from within your web worker by calling *close*.
- To load external scripts from a web worker, you use the *importScripts* method.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. You want to create a *ToggleSwitch* in your application. You want to initialize the control and attach a handler to the *onchange* event. Where do you put this code?
 - A. At the top of your HTML file
 - B. After the *processAll* call
 - C. In a *then* method on the *processAll* call
 - D. At the end of your HTML file
2. You are using a web worker to check for spelling of the content of a text field in your app. How does your web worker access the content of the text field?
 - A. By using *document.getElementById* from within your web worker and then attaching to the *onchange* event
 - B. By using *document.getElementById* from within your main page, attaching to the *onchange* event, and sending the text to your web worker with *postMessage*
 - C. By using *document.getElementById* from within your main page and sending the element to the web worker with *postMessage* so the web worker can subscribe to the *onchange* event
 - D. By using *document.getElementById* from within your main page, attaching an event handler, and sending the handler to the web worker with *postMessage*
3. You are chaining a few promises that each execute an asynchronous operation. You want to make sure you handle all errors that occur. What do you do?
 - A. Add an error handler to each promise.
 - B. Add a *then* method with only an error handler.
 - C. Wrap your code in a *try/catch* block.
 - D. Add a *done* method with only an error handler.

Objective 4.2: Implement animations and transitions

When creating your app, animations can help to create a compelling user experience. Using a shared set of animations enables users to feel familiar with how your app works. To achieve this, you can use CSS3 animations to animate your screens and the objects on them. Microsoft created a predefined set of animations that build on CSS3 and bundled them in the *WinJS.UI.Animation* API. Understanding these animations and leveraging them in your own app will enhance your UI.

One of the advantages of building a Windows 8 app is that you can focus on Internet Explorer 10. Whereas older browsers don't support CSS3 features and HTML5, Microsoft Internet Explorer 10 is a new browser with good CSS3 support.

By using animations and transitions, you can create complex animations without having to use any custom JavaScript. For example, you can make things move, fade in and out, and change color.

This objective covers how to:

- Create and customize animations and transitions by using CSS
- Apply transformations
- Create animations by using keypoints
- Apply timing functions
- Apply animations from the animation library (*WinJS.UI.animation*)
- Animate with the HTML5 *canvas* element

Using CSS3 transitions

When working with CSS, you often change properties or classes on an element. For example, you could apply a *hover* selector to apply some effects when the user hovers over your element. You can also use JavaScript to change classes or individual properties.

An example of this can be found in Listing 4-8. If you insert this code in the default.html file of a new blank Windows Store app, you get a centered rectangle with a green background color. When you move your mouse over the rectangle, it instantly switches colors to red.

LISTING 4-8 Changing the background color with CSS

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Transitions demo</title>
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <style>
        body {
            display: -ms-flexbox;
            -ms-flex-align: center;
            -ms-flex-pack: center;
        }
    </style>
```

```

#demoDiv {
    width: 350px;
    height: 100px;
    border: 1px solid white;
    background-color: green;
}

#demoDiv:hover {
    background-color: red;
}

```

</style>

```

<div id="demoDiv">
</div>

```

</body>

</html>

However, the change is quite abrupt. This is where *transitions* can be used. A transition helps you make a smooth change from one state to another.

The advantage of CSS3 transitions (and animations) is that the UI thread doesn't have to perform extra work, such as a layout pass when you move an object. This ensures smooth transitions and animations, and helps to avoid performance problems.

Adding and configuring a transition

To add a transition to the example in Listing 4-8, you can insert the following line in the `#demoDiv:hover` selector:

```
transition: background-color .5s ease-in-out;
```

Now when you move your mouse over the rectangle, the color changes smoothly from green to red.

The transition uses both a start and an end state. In this case, this is the start state:

```

#demoDiv {
    width: 350px;
    height: 100px;
    border: 1px solid white;
    background-color: green;
}

```

The end state is in the *hover* declaration:

```

#demoDiv:hover {
    transition: background-color .5s ease-in-out;
    background-color: red;
}

```

The transition property interpolates between the start state and the end state. Your app calculates the values between those two states and gradually changes those values.

A typical transition defines the following three properties:

- ***transition-property*** Specifies the CSS property for which you want your transition to listen for changes
- ***transition-duration*** Specifies how long the transition should take
- ***transition-timing-function*** Specifies the rate at which you want the property values to change

In this example, any changes to the *background-color* property will be noticed by your transition and will be gradually changed.

You can also specify multiple properties for your transition, like this:

```
#demoDiv:hover {  
    background-color: red;  
    border: 10px solid yellow;  
    transition-property: background-color, border;  
    transition-duration: .5s;  
    transition-timing-function: ease-in-out;  
}
```

This code transitions your *background-color* and *border* properties, both with a 0.5-second duration and an *ease-in-out* timing function. You can specify different settings for the duration and timing functions for each property by separating them with a comma, like this:

```
#demoDiv:hover {  
    background-color: red;  
    border: 10px solid yellow;  
    transition-property: background-color, border;  
    transition-duration: .5s, 1s;  
    transition-timing-function: ease-in-out, ease-in;  
}
```

If you have more values than properties, the excess values at the end of your list are not used.

If you have a lot of properties you want to transition on, you can also use the keyword *all* to transition on all CSS changes.

MORE INFO TRANSITION PROPERTIES

The CSS3 specification lists which properties can be animated. You can find these at <http://www.w3.org/TR/2009/WD-css3-transitions-20091201/#animatable-properties-> and <http://www.w3.org/TR/SVG/propidx.html>. In practice, these are all the properties you will ever need so you don't have to worry about this too much.

The second important property for your transition is *transition-duration*. It specifies how long you want the transition to take. The larger the value, the longer the transition will take to complete.

The third property is *transition-timing-function*. This setting controls the rate at which property values change from their initial state to the final state. By using different settings, you can create animations that change linearly (they don't slow down or speed up) but also accelerate, decelerate, or both.

You have the following timing functions at your disposal:

- **ease** Slow start, then fast, then end slowly.
- **linear** Same speed from start to end.
- **ease-in** Slow start.
- **ease-out** Slow end.
- **ease-in-out** Slow start and end.
- **cubic-bezier()** Specifies a cubic Bézier curve.
- **steps()** Defines a step function or staircase function. All steps are equal in length. Each step is one step closer to the goal state.
- **step-start** Equivalent to steps(1, start).
- **step-end** Equivalent to steps(1, end).

Figure 4-1 shows some examples of what these timing functions look like.

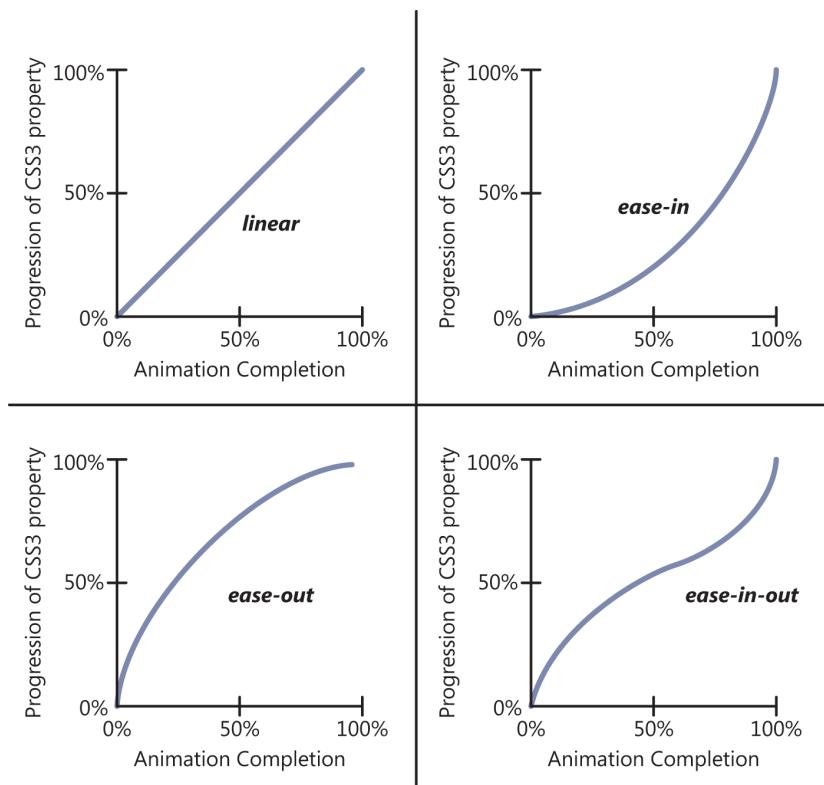


FIGURE 4-1 Timing functions

All those easing functions are implemented by means of a *stepping function* or a *cubic Bézier curve*. A cubic Bézier curve makes sure you have a smooth transition that can slow down or speed up.

A stepping function creates an animation that increases in discrete steps from start to finish. The easiest way to visualize the different timing functions is to try them out. For creating cubic Bézier curves, you have several different websites that can help you. One can be found at <http://cubic-bezier.com/>. Here you can visually experiment with your curve settings to get the type you want.

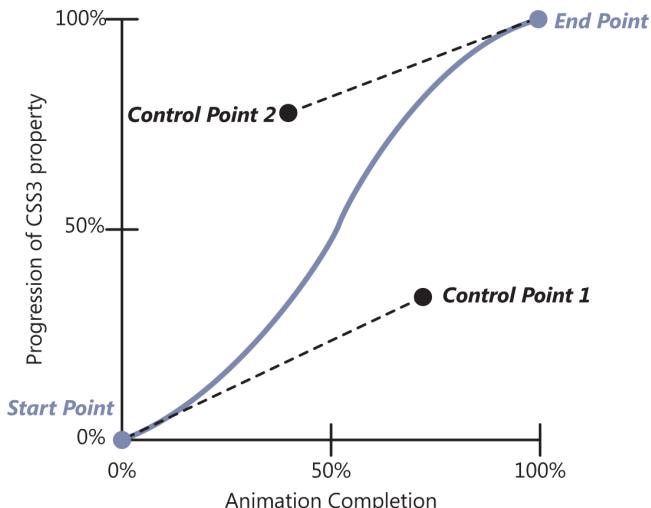


FIGURE 4-2 An example of a cubic Bézier curve

In addition to the commonly used properties—*transition-property*, *transition-timing-function*, and *transition-duration*—you can use one other setting: *transition-delay*. This property defines when the transition will start. If the value is a positive number, it will result in a delay before your transition starts playing. A negative value will skip the first part of your transition and start playing from the offset you specified.

Using JavaScript to activate transitions

One way to connect some JavaScript to the transition is by using the *transitionEnd* event. When your transition finishes, this event is fired. It's fired for each property that undergoes a transition. Each event will tell you the name of the property that changed and the duration of the transition.

The following code shows how you can subscribe to the *transitionEnd* event when using the code from Listing 4-8.

```
var demoDiv = document.getElementById('demoDiv');
demoDiv.addEventListener("transitionend", fireAtTheEnd, false);

function fireAtTheEnd(e) {
}
```

If you change multiple properties, the event fires multiple times. This is also true for short-hand properties. If you change the *border* property, it fires for all four sides of your border.

One way to use the *transitionEnd* event is to fire off a new transition or other action. Maybe you have a few animations that should follow each other, or you want to do an Asynchronous JavaScript and XML (AJAX) call when a transition ends. Listing 4-9 shows how to extend the previous examples to animate the *div*, first to red, and then immediately following that animation, to blue.

LISTING 4-9 Chaining transitions by using the *transitionEnd* event

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Transitions demo</title>
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <style>
        body {
            display: -ms-flexbox;
            -ms-flex-align: center;
            -ms-flex-pack: center;
        }

        #demoDiv {
            width: 350px;
            height: 100px;
            border: 1px solid white;
            background-color: green;
        }

        #innerDemoDiv {
            transition: all .5s ease-in-out;
            width: 100%;
            height: 100%;
        }

        #demoDiv:hover {
            background-color: red;
            border: 10px solid yellow;
            transition-property: background-color, border;
            transition-duration: .5s;
            transition-timing-function: cubic-bezier(.70, .35, .41, .78);
        }
    </style>
    <div id="demoDiv">
        <div id="innerDemoDiv">
        </div>
    </div>
```

```

<script>
    var demoDiv = document.getElementById('demoDiv');
    demoDiv.addEventListener("transitionend", fireAtTheEnd, false);
    function fireAtTheEnd(e) {
        if (e.propertyName == 'background-color') {
            var innerDemoDiv = document.getElementById('innerDemoDiv');
            innerDemoDiv.style.background = 'blue';
        }
    }
</script>
</body>
</html>

```

Integrating transitions with JavaScript can help you create some really nice effects. You will explore animations in the next section. However, know that when you want to use JavaScript, it's easier to use transitions than animations.

An example of using JavaScript in combination with transitions can be found in Listing 4-10. The code in this listing shows you how to create a simple ball that moves to wherever you click on the screen.

LISTING 4-10 Creating an animation by combining JavaScript and transitions

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Transitions demo</title>
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <style>
        #thing {
            position: relative;
            left: 50px;
            top: 50px;
            transition: left .5s ease-in, top .5s ease-in;
            width: 100px;
            height: 100px;
            background: #fc2e5a;
            border-radius: 50px;
        }
    </style>

    <div id="thing"></div>

    <script>
        document.querySelector("body")
            .addEventListener("MSPointerDown", getClickPosition, false);

        var theThing = document.querySelector("#thing");
        function getClickPosition(e) {

```

```

        theThing.style.left = e.clientX - (theThing.clientWidth / 2) + "px";
        theThing.style.top = e.clientY - (theThing.clientHeight / 2) + "px";
    }

```

</script>

</body>

</html>

Creating and customizing animations

Besides transitions, CSS3 also supports custom *animations*. Where transitions are entirely controlled by the start and end values in your CSS, an animation gives you more flexibility.

Another difference between animations and transitions is the way in which they are executed. A transition executes whenever a CSS property changes. An animation, however, is explicitly executed whenever the animation properties are applied.

As an example, review the code in Listing 4-11. If you add a simple *div* with an *id* of *cloud*, this CSS creates a cloud with a blue color.

LISTING 4-11 Creating a cloud shape with CSS

```

/*Base of the cloud*/
#cloud {
    width: 350px;
    height: 120px;
    background: #64b7f2;
    border-radius: 100px;
    position: relative;
}

/*Top part of the cloud*/
#cloud:after, #cloud:before {
    content: '';
    position: absolute;
    background: #64b7f2;
    z-index: -1;
}

/*Left part of the cloud*/
#cloud:after {
    width: 100px;
    height: 100px;
    top: -50px;
    left: 50px;
    border-radius: 100px;
}

/*Right part of the cloud*/
#cloud:before {
    width: 180px;
    height: 180px;
    top: -90px;
    right: 50px;
    border-radius: 200px;
}

```



EXAM TIP

You must understand the difference between an animation and a transition. A transition executes as a result of a CSS change and moves from one style to another. An animation is more complex and uses key frames to control the animation. It is implicitly started the moment the animation is applied.

Understanding key frames

Key frames (referred to as keypoints in the exam material) are used to specify the values for your animation at varying points in time. With key frames, you can create a detailed animation in which you control what occurs at each moment.

A key frame starts with the name of the animation. This animation is then referenced from the *animation-name* property in your other styles. You can reuse an animation by referencing it from multiple styles.

You build key frames by specifying multiple percentages and defining properties for each step. For example, you could use *0%*, *50%*, and *100%*, but you can use any percentages that are suitable for your animation. You can also use the *from* and *to* keywords that stand for *0%* and *100%*.

You specify the properties you want to animate inside a key frame. You can use the same list of animatable properties as for transitions.

One other useful property is *animation-timing-function*. This property is used to animate from the current key frame to the next key frame. Logically, you don't need to add this setting to the *100%* key frame. Now let's see how to use key frames in an animation.

Adding and configuring an animation

A completely motionless cloud is not that interesting, but by using animations, it's quite easy to give your cloud some movement. An animation consists of two different parts: the animation itself and a link between your element and the animation.

If you add the following line to your *#cloud* selector, you say that you want to animate this element:

```
animation: bobble 2s infinite;
```

Then you need to create the actual *bobble* animation key frames. You can find these key frames in Listing 4-12.

LISTING 4-12 Key frames for the *bobble* animation

```
@keyframes bobble {  
    0% {  
        transform: translate3d(50px, 40px, 0px);  
        animation-timing-function: ease-in;  
    }  
  
    50% {  
        transform: translate3d(50px, 50px, 0px);  
    }  
}
```

```
    animation-timing-function: ease-out;
}

100% {
    transform: translate3d(50px, 40px, 0px);
}
}
```

If you add the key frames to your CSS, you suddenly have a cloud that moves up and down. Now let's look at how this actually works. The first step is the `animation` property:

```
animation: bobble 2s infinite;
```

By applying this line directly in your style sheet, the animation will fire at document load. If you apply this style through JavaScript at some later point in code, it will start at that moment. By applying the `display:none` style to an element, you disable the animation. This is important to do when the animation is no longer visible, especially for animations that take up a lot of resources (such as animations that run infinitely). You can restart the animation by removing `display:none` from the element.

The `animation` property consists of three parts:

- The name of your animation (*bobble*, in this case)
- The duration of the animation (*2s*)
- The number of times your animation should play (*infinite*)

The `animation-name` property defines a list of animations you want to apply to the selected elements. If you have a cascading list of animations (animations inherited from parent elements), you can set the animation name to *none* to remove all of those animations.

The `animation-duration` property specifies how long your animation should run. You use a number of seconds for this property. A negative value is not allowed.

The third property is `animation-iteration-count`. This property specifies the number of times your animation should play. The default value is *1*, which means your animation executes once. In the *bobble* example, you used a value of *infinite*, which means your animation should never end.

In addition to `animation-name`, `animation-duration`, and `animation-iteration-count`, there are some other settings you can configure directly on your animation.

One such setting is the `animation-direction` property. This property can reverse an animation on some or all cycles. You can use the following options for this property:

- **normal** All iterations of the animation are played as specified.
- **reverse** All iterations of the animation are played in the reverse direction from the way they were specified.
- **alternate** The animation cycle plays the odd counts normally and the even iterations in reverse.
- **alternate-reverse** The animation cycle plays the even counts normally and the odd iterations in reverse.

For example, by using *alternate*, you can easily create an animation that moves an object from left to right and then from right to left. By repeating this animation, you get a nice, smooth experience of an object going backward and forward.

Let's say your animation should stop at some point in time (or should start delayed). The problem is that when your animation stops, the calculated values are discarded and your element changes back to the original state.

You can change this by using the *animation-fill-mode* property. A value of *backwards* applies the property values of your first key frame before the animation is started. A value of *forwards* ensures the calculated values on your last key frame remain in use even after the animation ends. You can also use a value of *both* to combine *forwards* and *backwards*.

Although an animation is normally started at the moment it's applied, you can also use the *animation-play-state* property to pause an animation. The property starts with a default value of *running*, but you can change it to *paused* and back again to *running*. Your animation will continue exactly from the moment where you paused it. This results in smooth animations when you pause and resume them.

Finally, the *animation-delay* property helps you define when the animation should start. A positive value delays execution, whereas a negative value skips a part of the animation. That is, it begins part-way through its animation cycle.

Using the animation library

Animations and transitions are an essential part of a Windows 8 app. A fluid interface is defined as one in which everything comes from somewhere and goes somewhere. In other words, elements shouldn't suddenly appear or disappear. Instead, your animations should tell the story and ensure that your user understands what's happening.

You should also be careful not to overuse animations. Using an animation for each and every element could irritate users and hinder them in using your app. Consider your animations as a form of communication. You don't want to shout at the user, distract him with a lot of noise, or speak so softly that he can't hear you. Used wisely, animations are an integral part of Windows 8 and can add great value to your app.

Animations are even required for app certification. Without animations, it can be hard for a touch user to use your app and get feedback on the actions he takes.

You could start building animations yourself, but that would result in a lot of work and could lead to inconsistency between different apps. Microsoft created the *WinJS.UI.Animation* API to handle desirable animations and transitions. Deciding which animations to use is something you should discuss with a designer. Knowing which animations are available will help you choose the ones that are right for your app.

Table 4-1 lists the animations that are a part of the WinJS animation library.

TABLE 4-1 Animations in the WinJS animation library

Name	Description	Methods
Add/Delete from list	Adds or deletes an item from a list	<i>createAddToListAnimation</i> <i>createDeleteFromListAnimation</i>
Add/Delete from search list	Adds or deletes an item from a list when filtering search results	<i>createAddToSearchListAnimation</i> <i>createDeleteFromSearchListAnimation</i>
Badge update	Updates a numerical badge on a tile	<i>updateBadge</i>
Content transition	Animates one piece or set of content into or out of view	<i>enterContent</i> <i>exitContent</i>
Crossfade	Refreshes a content area	<i>crossfade</i>
Expand/Collapse	Shows additional inline information	<i>createExpandAnimation</i> <i>createCollapseAnimation</i>
Fade in/out	Shows transient elements or controls	<i>fadeln</i> <i>fadeout</i>
Page transition	Animates the contents of a page into or out of view	<i>enterPage</i> <i>exitPage</i>
Peek	Updates the contents of a tile	<i>createPeekAnimation</i>
Pointer up/down	Gives visual feedback of a tap or click on a tile	<i>pointerUp</i> <i>pointerDown</i>
Reposition	Moves an element into a new position	<i>createRepositionAnimation</i>
Show/Hide edge UI	Slides edge-based UI into or out of view	<i>showEdgeUI</i> <i>hideEdgeUI</i>
Show/Hide panel	Slides large edge-based panels into or out of view	<i>showPanel</i> <i>hidePanel</i>
Show/Hide popup	Displays contextual UI on top of the view	<i>showPopup</i> <i>hidePopup</i>
Start/End a drag or drag-between	Gives visual feedback during a drag-and-drop operation	<i>dragSourceStart</i> <i>dragSourceEnd</i> <i>dragBetweenEnter</i> <i>dragBetweenLeave</i>
Swipe hint	Hints that a tile supports the swipe interaction	<i>swipeReveal</i>
Swipe select/deselect	Transitions a tile to a swipe-selected state	<i>swipeSelect</i> <i>swipeDeselect</i>

MORE INFO ANIMATION EXAMPLES

You can see videos of different animations at <http://msdn.microsoft.com/en-us/library/windows/apps/hh465165.aspx>. To find sample code for all animations, visit <http://code.msdn.microsoft.com/windowsapps/using-the-animation-787f3720/>.



EXAM TIP

Review the samples for all animations listed in the previous More Info sidebar. Knowing how animations work and when to use them is an important part of app development and the 70-482 exam.

The animations in Table 4-1 are implemented in the ui.js file that's referenced from your Windows App projects. By searching for the animation name inside this file, you can see how Microsoft implemented each animation.

Examples of how to use the add/delete from list animations are shown in Listings 4-13 for the HTML and 4-14 for the JavaScript. A toggle is included to turn the animation on or off. You can see how the principle of “everything comes from somewhere and goes somewhere” is followed.

LISTING 4-13 HTML for the add/delete from list animations

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Animation library demo</title>
    <link href="//Microsoft.WinJS.1.0/css/ui-light.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <style>
        .listItem {
            background: #336699;
            height: 50px;
            width: 120px;
            margin: 0px 20px 20px 20px;
        }

        #errorMessage {
            color: #f00;
        }
    </style>
    <div data-win-control="WinJS.UI.ToggleSwitch" id="animationToggleSwitch"
        data-win-options="{labelOff:'Without animation',
                           labelOn:'With animation', checked:true}">
    </div>
    <button id="addItem">Add an item</button>
    <button id="deleteItem">Delete an item</button>
    <div id="output">
        <div id="errorMessage"></div>
        <p>Click on a button to add or delete an item from the list.</p>
        <div id="list">
            <div class="listItem"></div>
        </div>
    </div>
</body>
</html>
```

LISTING 4-14 JavaScript for the add/delete from list animations

```
(function () {
    "use strict";

    WinJS.Binding.optimizeBindingReferences = true;

    var app = WinJS.Application;
    var activation = Windows.ApplicationModel.Activation;
    var useAnimation = true;
    app.onactivated = function (args) {
        if (args.detail.kind === activation.ActivationKind.launch) {
            WinJS.UI.processAll().then(function () { });

            list = document.querySelector("#list");

            addItem.addEventListener("click", runAddToListAnimation, false);
            deleteItem.addEventListener("click", runDeleteFromListAnimation, false);

            var toggleSwitch = document.getElementById("animationToggleSwitch");
            toggleSwitch.addEventListener("change", toggleAnimationHandler);
        };
    }

    function toggleAnimationHandler() {
        var obj = document.getElementById("animationToggleSwitch").winControl;
        useAnimation = obj.checked;
    }

    var list;

    function runAddToListAnimation() {
        errorMessage.innerHTML = "";

        // Create new item.
        var newItem = document.createElement("div");
        newItem.className = "listItem";
        newItem.style.background = randomColor();

        if (useAnimation) {
            // Create addToList animation.
            var addToList = WinJS.UI.Animation
                .createAddToListAnimation(newItem, affectedItems);
            // Set affected items to all items that may
            // move in response to the list addition.
            var affectedItems = document.querySelectorAll(".listItem");

            list.insertBefore(newItem, list.firstChild);
            // Execute the animation.
            addToList.execute();
        }
        else {
            // Just add the element to the DOM without any animation
            list.insertBefore(newItem, list.firstChild);
        }
    }
})()
```

```

function runDeleteFromListAnimation() {
    var listItems = document.querySelectorAll(".listItem:not([deleting])");

    if (listItems.length === 0) {
        errorMessage.innerHTML = "No items to delete. Please add an item.";
    } else {
        var deletedItem = listItems[0];
        deletedItem.setAttribute("deleting", true);

        if (useAnimation) {
            var affectedItems = document
                .querySelectorAll(".listItem:not([deleting])");

            // Create deleteFromList animation.
            var deleteFromList = WinJS.UI.Animation
                .createDeleteFromListAnimation(deletedItem, affectedItems);

            // Take deletedItem out of the regular document layout
            // flow so remaining list items will change position in response.
            deletedItem.style.position = "fixed";
            // Set deletedItem to its final visual state (hidden).
            deletedItem.style.opacity = "0";

            // Execute deleteFromList animation, then clean up.
            deleteFromList.execute().done(
                // After animation is complete, remove item from the DOM tree.
                function () { list.removeChild(deletedItem); });
        }
        else {
            // Just remove the element from the DOM without any animation
            list.removeChild(deletedItem);
        }
    }
}

function randomColor() {
    // Return a random color in #rgb format
    return '#' + Math.floor((1 + Math.random()) * 4096).toString(16).substr(1);
}

app.start();
})();

```

As you can see in this example, you create a new animation and then use the *execute* method to start it. Because each animation returns a promise object, you can use the *then* and *done* methods to schedule work that should be executed when the animation finishes.

When you use the standard WinJS controls, you get these animations for free. The standard controls are also a great way to learn about all the different animations and understand when they should be used.

Animating with the HTML5 *canvas* element

The *canvas* element was introduced in HTML5 and provides an area in which you can draw graphics on the fly in a web page or in your app. The *canvas* element doesn't have actual state. You can draw lines, curves, images, and other elements on the canvas, but it views each item as a set of pixels.

You create a canvas by adding the *canvas* element to your app:

```
<canvas id="canvas" width="640" height="480"></canvas>
```

Using JavaScript, you can get the *canvas* by its *id* and then access the 2D context of the *canvas* where you can do the actual drawing:

```
var c = document.getElementById("#canvas").getContext("2d");
```

The *canvas* element offers a few methods you can use to draw elements on it. The following code draws a solid circle:

```
var r = 20, x = r, y = r;
c.beginPath();
c.fillStyle = "darkorange";
c.arc(x, y, r, 0, Math.PI * 2, true);
c.closePath();
c.fill();
```

When you run this code, a solid circle with a radius of 20 appears in the top-right corner of your screen. To animate the circle to bounce around on your canvas, call a *draw* method on a regular interval, such as by using the *setInterval* method. Listing 4-15 shows an example of using *draw* and *setInterval*.

LISTING 4-15 Animating a *canvas* element

```
var dx = 5, dy = 5, r = 20, x = r, y = r;

var c = q("#canvas").getContext("2d");

setInterval(function () { draw(); step(); }, 10);

function draw() {
    c.clearRect(0, 0, c.canvas.width, c.canvas.height);
    c.beginPath();
    c.fillStyle = "darkorange";
    c.arc(x, y, r, 0, Math.PI * 2, true);
    c.closePath();
    c.fill();
}

function step() {
    if (x > c.canvas.width - r || x < r) dx *= -1;
    if (y > c.canvas.height - r || y < r) dy *= -1;
    x += dx;
    y += dy;
}
```

On each interval, the *canvas* is first cleared and then the image is redrawn. The *step* function changes the x and y position of the circle so you get a smooth animation.

You can use the *canvas* element to create complex games with a lot of moving objects. However, because the canvas is simply a place to draw pixels, you would have to build all the logic yourself to draw complex shapes and animate them.



Thought experiment

Creating a complex animation

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are developing a game that will include a bouncing ball. The user can "hit" the ball by touching it and then throwing it in some direction. You want to create an animation for your ball so it slowly stops bouncing.

Answer the following questions:

1. Should you use a transition or an animation for the bouncing ball? Why?
2. What will the force of the ball strike affect?
3. Which properties should you animate?
4. Which timing functions should you use?

Objective summary

- Animations and transitions are an important part of a Windows 8 app. They help you create a fluid interface that not only appeals to users but also helps them use your app.
- CSS transitions are used to move from one CSS style to another in a smooth way. The most used properties are *transition-property*, *transition-duration*, and *transition-timing-function*.
- You can use the *transitionEnd* event to execute code when a transition finishes.
- CSS animations consist of two parts: a set of key frames that describe the animation and an *animation* property that links the animation to a specific element.
- WinJS provides a built-in library of animations you can integrate with a Windows 8 app. The built-in controls all use this library to create a compelling user experience.
- The HTML5 *canvas* element draws elements on screen and animates them with a method you call on a regular interval.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the “Answers” section at the end of this chapter.

1. You are building a game in which a user can move tiles around by grabbing a tile and swiping it in the desired direction. You want to animate this movement. What do you use?
 - A. From JavaScript, call the *jQuery.animate* method to customize all settings for the animation.
 - B. Define the basics for the key frames in CSS, and adjust the settings through JavaScript when the animation should play.
 - C. Use transitions to move the tile by changing the *top* and *left* properties.
 - D. Use pure JavaScript for performance reasons, and include a timer to ensure the UI remains fluid.
2. You want to shrink an object in an application. You use a transition to do this. However, your designer says your transition is too fast. What should you do?
 - A. Change the *transition-duration* property.
 - B. Change the *transition-timing-function* property.
 - C. Switch to use animations with key frames so you can control the exact steps of the animation.
 - D. Use the *transitionEnd* event to slow down the transition.
3. You have created an animation that changes the size and color of an object. However, when testing the animation, you notice the object keeps changing back to the original size and color. What should you do?
 - A. Change the *animation-direction* to *reverse*.
 - B. Change the *animation-fill-mode* to *forwards*.
 - C. Change *animation-delay* to a negative value.
 - D. Change the *animation-timing-function* to *ease-out*.

Objective 4.3: Create custom controls

When building Windows Store apps, you can use the standard HTML controls and WinJS controls created by Microsoft. However, you will occasionally want additional or alternative functionality that the system does not provide. Fortunately, WinJS helps you create your own controls, which you can reuse in your app or even across different apps. In this objective, you learn how to create your own controls and extend existing controls.

This objective covers how to:

- Create custom controls using *WinJS.Namespace*, *WinJS.Class.define*, and HTML
- Bind to custom controls with *data-win-bind*
- Inherit from and extend an existing WinJS control

Understanding how existing controls work

A WinJS control is not some kind of magic that's created in another language like C# or C++. Instead, a WinJS control consists of HTML, CSS, and JavaScript. You can examine the complete source code of all WinJS controls by browsing the ui.js file.

Consider the rating control, for example. Listing 4-16 shows the HTML generated by the rating control.

LISTING 4-16 Generated HTML for the rating control

```
<div tabindex="0" class="win-rating" id="Div1" role="slider"
    aria-readonly="false" aria-valuenow="Unrated" aria-valuemin="0"
    aria-valuemax="5" aria-label="User Rating" aria-valuetext="Unrated">
    <div class="win-star win-empty win-user"></div>
    <div class="win-star win-average win-full win-user"
        style="-ms-flex: 0 0 auto; padding-right: 0px; padding-left: 0px;
        border-right-color: currentColor; border-left-color: currentColor;
        border-right-width: 0px; border-left-width: 0px;
        border-right-style: none; border-left-style: none; display: none;">
    </div>
</div>
```

There is nothing particularly special about the generated HTML. It uses a *div* for the containing element and *div* elements for all the stars. It also uses the *class* element to ensure the control is styled. The *role* property and *aria* properties are used to improve accessibility. Some inline styles are applied to the last inner *div*.

But this isn't all there is to the control. When creating a custom control, you also want to supply a default style sheet. Listing 4-17 shows you the CSS for the rating control.

LISTING 4-17 CSS for the rating control

```
.win-rating {
    display: -ms-inline-flexbox;
    height: auto;
    width: auto;
    white-space: normal;
    -ms-flex-align: stretch;
    -ms-flex-pack: center;
}
```

```

.win-rating .win-star {
    -ms-flex: 1 1 auto;
    height: 28px;
    width: 28px;
    padding: 0 6px;
    font-family: "Segoe UI Symbol";
    font-size: 28px;
    overflow: hidden;
    text-indent: 0;
    line-height: 1;
    cursor: default;
    position: relative;
    letter-spacing: 0;
    -ms-touch-action: none;
}
.win-rating.win-small .win-star {
    width: 14px;
    height: 14px;
    font-size: 14px;
    padding: 0 3px;
}
.win-rating .win-star:before {
    content: "\E082";
}
.win-rating .win-star.win-disabled {
    cursor: default;
    -ms-touch-action: auto;
}

```

The code sample uses CSS to define the rating control itself and the stars. However, the JavaScript contains the functionality for your control. The rating control has 1,086 lines of JavaScript. If you want to view the JavaScript for the rating control, we encourage you to open the code in Microsoft Visual Studio. You can learn a lot by looking at existing controls. To understand how you can create your own controls, let's walk through a few important aspects of the code.

The rating control starts with a few constant definitions, such as the average rating and CSS class names. After those definitions, the actual control is implemented with the following line of code:

```
WinJS.Namespace.define("WinJS.UI", { ... });
```

All WinJS controls are declared in the *WinJS.UI* namespace. JavaScript enables you to add new members to an existing namespace. This way, you can create your own namespace and extend it by each control that you add. It's important to create a namespace for your controls so you don't pollute the global namespace.

After that, the first element that's defined is the constructor for the rating control. You can find the code for the constructor in Listing 4-18.

LISTING 4-18 The constructor for the rating control

```
Rating: WinJS.Class.define(function Rating_ctor(element, options) {
    element = element || document.createElement("div");
    options = options || {};
    this._element = element;

    //initialize properties with default value
    this._userRating = 0;
    this._averageRating = 0;
    this._disabled = DEFAULT_DISABLED;
    this._enableClear = true;
    this._tooltipStrings = [];

    this._controlUpdateNeeded = false;
    this._setControlSize(options.maxRating);
    if (!options.tooltipStrings) {
        this._updateTooltips(null);
    }
    WinJS.UI.setOptions(this, options);
    this._controlUpdateNeeded = true;
    this._forceLayout();

    // Remember ourselves
    element.winControl = this;
    this._events();
}
```

Within your namespace, you use the *WinJS.Class.define* method to create a new custom control. You assign the return value of this method to a property with the name you want to use in your *data-win-control* attributes. So, if your namespace is *MyCustomControls* and you want to name your control *MyControl*, you use *MyControl* as the property name to store the return value of your constructor method. You then use your control as *MyCustomControls.MyControl*.

Two values are passed to your constructor: *element* and *options*. The first value, *element*, points to the root element that will contain your control. If this value is *null*, you create a new empty *div* and use that. That's done in the following code:

```
element = element || document.createElement("div");
```

You have total freedom in determining the options you want to pass in. The helper method *WinJS.UI.setOptions(this, options)*; loops through all the properties on your *options* object and copies them. It can also be used to pass in event handlers for events you expose.

The last required part of your constructor is storing the element. You store the original DOM element in *this.element* and your custom control in *this.element.winControl*.

The *WinJS.Class.define* method takes as a second parameter an object that contains your instance public methods and properties. Within this object you put all methods and properties your object should expose.

One property the rating control exposes is the *userRating* property. The following code shows how this property is defined.

```

userRating: {
    get: function () {
        return this._userRating;
    },
    set: function (value) {
        // Coerce value to a positive integer between 0 and maxRating
        this._userRating = Math.max(0, Math.min(Number(value) >> 0, this._maxRating));
        this._updateControl();
    }
}

```

This property exposes both a *get* and a *set* accessor (commonly called a *getter* and a *setter*). Within your *setter*, you can validate the new value. In this case, it checks that the value falls within the range of the rating control.

In the same way, you can add methods to your control that can be called by users. As a convention, you prefix methods with an underscore to mark them as private. This doesn't mean a user of your control can't call the method, but Visual Studio won't show these methods in IntelliSense, which makes them much harder to find.

The *rating* class also exposes three events: *cancel*, *change*, and *preview_change*. The events are encapsulated in a function called *_events*. You can raise an event from your custom control by calling *this.dispatchEvent* method and passing an event name and the data for the event.

The rating control also subscribes to events in the DOM, such as key presses and mouse or touch actions.

After the *WinJS.Class.define* method ends, a special helper class is called to add the *on<eventName>* properties to the rating control. This is done by calling *WinJS.Utilities.createEventProperties* method and passing the result to the *WinJS.Class.mix* method.

The *createEventProperties* method takes a variable list of strings and creates event properties for all strings that you pass by prefixing *on* to them. The *mix* method then adds those new properties to your existing control.

Aside from adding custom events of your own, you also need to add a few standard events. You can handwrite those events if you want custom behavior, or you can use a special class called *WinJS.UI.DOMEventMixin* and pass that to the *WinJS.Class.mix* method to add them to your class. The events are:

- ***addEventListener*** Adds an event listener to the control
- ***dispatchEvent*** Raises an event of the specified type, adding the specified additional properties
- ***removeEventListener*** Removes an event listener from the control
- ***setOptions*** Adds the set of declaratively specified options (properties and events) to the specified control. If the name of the options property begins with *on*, the property value is a function and the control supports *addEventListener*. This method calls the *addEventListener* method on the control.

Now that you understand the elements of the rating control, you can use the same concepts to create your own custom control.



EXAM TIP

All *WinJS.UI* controls are plain JavaScript, HTML, and CSS. By inspecting the source code for these controls, you can learn a lot about how they work.

Creating a custom control

When creating a custom control, you should follow the same steps for built-in controls:

1. Encapsulate your control in a self-calling function to make sure you don't pollute the global namespace.
2. Define a namespace for your control.
3. Define a class constructor by using *WinJS.Class.define*.
 - A. Set your options by using *WinJS.UI.setOptions(this, options)*.
 - B. Make sure your *element* is valid or create a new *div*. Store a reference to your *element* and set *element.winControl* to *this* (the *winControl* you are creating).
4. Pass an object containing your instance methods and properties as a second parameter to your *define* method. As the third parameter, you can pass an object containing your static properties and methods. A static property or method is accessible without calling or creating a new instance of your control.
5. Use the *WinJS.Class.mix* method to add custom event handlers and the *WinJS.UI.DomEventMixin* class to add the default methods.

Now let's look at an example of creating such a custom control. You can find the JavaScript for a simple *HelloWorld* control in Listing 4-19.

LISTING 4-19 A custom *HelloWorld* control

```
(function helloWorldInit(global) {
    "use strict";

    // Constants definition
    var DEFAULT_MESSAGE = "Hello World",
        CHANGE = "change";

    var utilities = WinJS.Utilities;

    // CSS class names
    var helloWorld = "hello-world";

    WinJS.Namespace.define("MyCustomControls.UI", {
        HelloWorld: WinJS.Class.define(function HelloWorld_ctor(element, options) {
            element = element || document.createElement("div");
            element.className = helloWorld;
            element.textContent = options.message || DEFAULT_MESSAGE;
            element.winControl = this;
            this.element = element;
            this._options = options;
        },
        _change: function () {
            this.element.textContent = this._options.message;
        }
    });
});
```

```

        element.winControl = this;
        this._element = element;

        options = options || {};
        WinJS.UI.setOptions(this, options);
    }, {
        _message: DEFAULT_MESSAGE,

        message: {
            get: function () {
                return this._message;
            },
            set: function (value) {
                this._message = value;
                this.element.innerText = value;
                utilities.addClass(this.element, helloWorld);
                this._onMessageChanged(value);
            }
        },
        element: {
            get: function () {
                return this._element;
            }
        },
        _onMessageChanged: function (newMessage) {
            this.dispatchEvent("messagechanged", { message: newMessage });
        }
    })
});
}

WinJS.Class.mix(MyCustomControls.UI.HelloWorld,
    WinJS.Utilities.createEventProperties("messagechanged"));
WinJS.Class.mix(MyCustomControls.UI.HelloWorld, WinJS.UI.DOMEventMixin);
})(this, WinJS);

```

You can then use the control like this:

```
<div id="helloWorld" data-win-control="MyCustomControls.UI.HelloWorld"
    data-win-options="{ message: 'Hello, World'}"></div>
```

The control displays the message "Hello, World" inside the *div*.

The code starts by creating an anonymous function that wraps your control definition. Inside this function, you can declare constants and other helper variables that you want to use inside your control.

Then it defines a namespace for your control and the class constructor. Inside the constructor, the *element* is initialized and the options are set.

The second parameter contains a property to get the element and one for the message to display. When the message is set, a CSS class is added to the element and an event is raised.

After the class definition, the first call to *mix* adds the custom event *messagechanged*. The second adds the default *DOMEvents*.

A user of your control can subscribe to the event, as shown in the following code. In this case, you display a message dialog with the new message each time it's changed.

```
WinJS.UI.processAll().then(function () {
    var helloWorld = document.getElementById("helloWorld").winControl;

    helloWorld.onmessagechanged = function (e) {
        new Windows.UI.Popups.MessageDialog(e.message).showAsync();
    }
});
```

You can create custom controls that are much more complicated than this one. However, they all follow the same steps. By adding more HTML, methods, and properties you can extend your controls. You can also let your control subscribe to events like touch or mouse events to respond to user input.

It's a best practice to group your controls in per-control folders that contain all the HTML, JavaScript, and CSS files for your control.

Binding to custom controls with *data-win-bind*

When using controls, you can also use a template to configure how a control renders and the data to which it binds. For example, you can extend the *HelloWorld* control from Listing 4-19 to bind to a list of strings that are then rendered inside the control. First, you need to define the template that you want to render:

```
<div id="templateDiv" data-win-control="WinJS.Binding.Template">
    <span data-win-bind="innerText: content" />
</div>
```

You use the *WinJS.Binding.Template* value for the *data-win-control* option. Inside your template, you can use *data-win-bind* to bind values of your data source to specific attributes of your HTML. In this case, you bind the *innerText* property of *span* to a *content* property on your data source items.

You can change the JavaScript for your control to use this template to render the items that you pass it. Listing 4-20 shows the property and the method that you need to add.

LISTING 4-20 Using a template for the *HelloWorld* control

```
itemsSource: {
    set: function (source) {
        this._source = source;
        this._doBinding();
    }
},
_doBinding: function () {
    if (!this._source)
        return;

    var template = document.getElementById('templateDiv').winControl;
    this._items = [];
```

```

var createItem = function (data) {
    var item = document.createElement("div");
    template.render(data, item);

    return item;
};

for (var index = 0; index < this._source.length; index++) {
    var data = this._source.getAt(index);

    this._element.appendChild(createItem(data));
}
}

```

The *itemsSource* property sets the new value and then calls the *_doBinding* method. This method retrieves the template and then renders the template for each item in the item source. The *render* method of your template processes the *data-win-bind* property and sets the *innerText* of *span* to the *content* property on your items.

You can now use the control like this:

```

var c = document.getElementById('helloWorld').winControl;
var list = new WinJS.Binding.List();

list.push({ content: "Hello World 1" });
list.push({ content: "Hello World 2" });
list.push({ content: "Hello World 3" });
list.push({ content: "Hello World 4" });
list.push({ content: "Hello World 5" });
list.push({ content: "Hello World 6" });

c.itemsSource = list;

```

Adding documentation

When you use built-in controls, you might have noticed a lot of instructions appear in IntelliSense. They increase productivity and will help users of your controls if you add the instructions.

You can add instructions by inserting some special comments in the code that tell Visual Studio to treat it as IntelliSense documentation.

Listing 4-21 shows an example of how to add documentation to a method.

LISTING 4-21 Adding XML comments to a JavaScript method

```

function getArea(radius)
{
    /// <summary>Determines the area of a circle that has the specified radius
    /// parameter.</summary>
    /// <param name="radius" type="Number">The radius of the circle.</param>
    /// <returns type="Number">The area.</returns>
    var areaVal;
    areaVal = Math.PI * radius * radius;
    return areaVal;
}

```

The important elements you should add are *summary*, *param*, and *returns*. With these elements, you can describe what the function does, which values you expect from the caller, and what you are going to return.

MORE INFO EXTENSIBLE MARKUP LANGUAGE (XML) COMMENTS

For more information on how to add XML comments to your code, visit <http://msdn.microsoft.com/en-us/library/vstudio/bb514138.aspx>.

Extending controls

In addition to building your own controls, you can extend existing controls. You can do this in two different ways:

- By replacing functionality on existing control through the *prototype* object
- By creating a new control that derives from an existing control

Changing the prototype on a control

The *prototype* functionality in JavaScript enables you to simulate the idea of classes. If you access a property or method on an object, the runtime first checks if it's present on the object. If not, it checks the entire chain of prototype objects to find the method.

The *WinJS.Class.define* method supports this by adding the instance members that you pass to it to the prototype. You can add methods to all the objects that you create of the specific class type.

To illustrate how the *WinJS.Class.define* method works, Listing 4-22 shows how a custom class is defined with a *doSomething* method.

LISTING 4-22 Creating a custom class

```
(function (global) {
    "use strict";

    WinJS.Namespace.define("MyLibrary", {
        MyClass: WinJS.Class.define(function MyClass_ctor() { },
        {
            doSomething: function (message) {
                // Function body
            }
        }
    });
})(this, WinJS);
```

The *doSomething* method is now present on all instances of the *MyClass* class. When using this class, you can replace the *doSomething* method with your own implementation, as follows:

```
MyLibrary.MyClass.prototype.doSomething = function (message) {
    new Windows.UI.Popups.MessageDialog(message).showAsync();
};
```

After this code executes, a call to `MyClass.doSomething` displays a message dialog. If you don't want to replace the original implementation entirely but simply extend it, you can use the following approach:

```
var doSomething_old = MyClass.prototype.doSomething;
MyClass.prototype.doSomething = function (message) {
    doSomething_old(message);
    new Windows.UI.Popups.MessageDialog(message).showAsync();
};
```

By first saving a reference to the old implementation, you can call this from inside your new function, extending the original functionality instead of replacing it.

You can also extend the functionality of a control. Let's say you want to extend the rating control by displaying a number behind the stars that displays the number of stars you selected, both in preview and in update mode. You can do this by using the code in Listing 4-23.

LISTING 4-23 Extending the rating control

```
(function (global) {
    "use strict";

    WinJS.UI.Rating.prototype._createControlOld =
        WinJS.UI.Rating.prototype._createControl;
    WinJS.UI.Rating.prototype._createControl = function () {
        this._createControlOld();

        var html = "<div>0</div>";
        this._element.insertAdjacentHTML('beforeend', html);

        this._ratingNumberElement = this._element.lastElementChild;
    };

    WinJS.UI.Rating.prototype._updateControlOld =
        WinJS.UI.Rating.prototype._updateControl;
    WinJS.UI.Rating.prototype._updateControl = function () {
        this._updateControlOld();
        this._ratingNumberElement.innerText = this._userRating;
    };

    WinJS.UI.Rating.prototype._showTentativeRatingOld =
        WinJS.UI.Rating.prototype._showTentativeRating;
    WinJS.UI.Rating.prototype._showTentativeRating = function () {
        this._showTentativeRatingOld();
        this._ratingNumberElement.innerText = this._tentativeRating;
    };
})(this, WinJS);
```

This code changes three methods on the `rating` class. The `_createControl` method is used to add some extra HTML to the control and store the extra `div` inside a property on your control for later access. The `_updateControl` and `_showTentativeRating` methods are used to update the number inside the `div` with the selected or tentative rating.

All three methods call the original method to make sure the rating control does all the work and then add a little code for the extra work.

One problem with this approach is that you affect all existing rating controls. If somewhere in your app you don't want this extra number to show up, you have to make sure this code isn't included. You don't have an explicit opt-in model for your changes. If you want that, you can choose another strategy: deriving from an existing control.

Deriving from an existing control

You can create a new control that extends the functionality of an existing control. That way, your users can explicitly choose to use your control by changing their HTML to point to your control.

Extending an existing control can be done by *deriving* from an existing control by calling the `WinJS.Class.derive` method. The code for the `WinJS.Class.derive` method is shown in Listing 4-24.

LISTING 4-24 The `WinJS.Class.derive` method

```
function derive(baseClass, constructor, instanceMembers, staticMembers) {
    if (baseClass) {
        constructor = constructor || function () { };
        var basePrototype = baseClass.prototype;
        constructor.prototype = Object.create(basePrototype);
        WinJS.Utilities.markSupportedForProcessing(constructor);
        Object.defineProperty(constructor.prototype, "constructor", { value: constructor,
            writable: true, configurable: true, enumerable: true });
        if (instanceMembers) {
            initializeProperties(constructor.prototype, instanceMembers);
        }
        if (staticMembers) {
            initializeProperties(constructor, staticMembers);
        }
        return constructor;
    } else {
        return define(constructor, instanceMembers, staticMembers);
    }
}
```

This method implements something called *prototypical inheritance*. This means that it takes the class you want to derive from and stores that in the `prototype` property of your new class. That way, your new class supports all functionality of your old class. If you now change anything on the derived class, those changes won't be visible on your base class.

Listing 4-25 shows how you can use the `WinJS.Class.derive` method to implement the previous example of extending the rating control in Listing 4-23.

LISTING 4-25 Deriving from the rating control

```
(function (global) {
    "use strict";

    WinJS.Namespace.define("MyLibrary", {
        MyRating: WinJS.Class.derive(WinJS.UI.Rating,
            function MyRating_ctor(element, options) {
                WinJS.UI.Rating.apply(this, [element, options]);
            },
            {
                _createControl: function () {
                    WinJS.UI.Rating.prototype._createControl.call(this);
                    var html = "<div>0</div>";
                    this._element.insertAdjacentHTML('beforeend', html);

                    this._ratingNumberElement = this._element.lastElementChild;
                },
                _updateControl: function () {
                    WinJS.UI.Rating.prototype._updateControl.call(this);
                    this._ratingNumberElement.innerText = this._userRating;
                },
                _showTentativeRating: function () {
                    WinJS.UI.Rating.prototype._showTentativeRating.call(this);
                    this._ratingNumberElement.innerText = this._tentativeRating;
                }
            }
        );
    })(this, WinJS);
```

You pass the class you want to derive from as the first argument to the *WinJS.Class.derive* method. The other parameters are the same as those used for the *WinJS.Class.define* method: a constructor function, instance members, and static members.

You can use the *apply* method inside the constructor function to call the base constructor of the rating control. Unlike in other languages like C#, the base constructor isn't called automatically.

When you want to call another method or property on your base class, you use the *call* method, as follows:

```
WinJS.UI.Rating.prototype._createControl.call(this);
```

The methods you define on your derived class don't affect your base class.



Thought experiment

Creating your own custom control

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You need to create a few chart controls for a Windows 8 line of business app. You need both line and bar charts that will display the data you supply them with. The user should be able to easily switch between different chart types.

Answer the following questions:

1. How can you share as much code as possible between your different chart controls?
2. Why is it a good idea to add XML comments to your control?

Objective summary

- A control is composed of HTML, JavaScript, and CSS. You can inspect the code for all *WinJS.UI* controls inside Visual Studio.
- You can create a custom control in your own namespace by using the *WinJS.Namespace.define* method.
- To create a new control, use the *WinJS.Class.define* method and pass it a constructor function, the instance members, and the static members.
- Inside your constructor you need to store references to the element and to your control. You also initialize the options that were passed to you.
- You can define properties, methods, and events on your class. By convention, when prefixing an item with an underscore, you mark it as private.
- You use the *WinJS.Class.mix* method to add members from one object to another. You can use this in combination with the *WinJS.Utilities.createEventProperties* method to add default event implementations for your custom events. You can also use it with the *WinJS.UI.DOMEventMixin* method to add the default DOM members to your control.
- By using XML comments, you can add documentation to your controls that appear in IntelliSense.
- You can change an existing control by changing methods on the *prototype*.
- You can create a new control that derives from an existing control by using the *WinJS.Class.derive* method.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the “Answers” section at the end of this chapter.

1. Your organization has created a line of business app for your consultants to track customer data. The app contains a toggle switch that's used to switch from billable to non-billable hours. Your users report that the app stops responding randomly when they use the toggle. You have studied the performance reports but can't find the exact cause of the problem. You want to instrument the toggle with some log methods that will show you where the error comes from. What do you do?
 - A. Create an entirely new toggle control with the logging capabilities and replace the billable toggle.
 - B. Change the `WinJS.UI.ToggleSwitch.prototype` methods to add the logging functionality.
 - C. Bind to the `onchange` event and add logging in your event handler.
 - D. Use the `WinJS.Class.derive` method to create a `LoggableToggleSwitch` and replace the billable toggle with the new one.
2. You are creating a new control to show images in a carousel. You use a default value for the number of images that are shown. You want to add a method to change the default value. How should you do this? (Choose all that apply.)
 - A. Directly add a new method to the prototype of your class: `Carousel.prototype.changeDefaultNumberOfImages = ...`
 - B. Add the method to the third parameter of your `WinJS.Class.define` call.
 - C. Add the method to the second parameter of your `WinJS.Class.define` call.
 - D. Directly add a new method to the prototype of your class: `Carousel.changeDefaultNumberOfImages = ...`
3. You have derived a custom control from an existing built-in control by using `WinJS.Class.derive`. However, the constructor of your base class isn't called. What do you do?
 - A. Add a call to `<BaseClass>.apply(this)` to your constructor.
 - B. Use the `WinJS.Class.mix` method to add the existing constructor to your class.
 - C. Add a call to `<BaseClass>.call(this)` to your constructor.
 - D. Add an XML comment to your new control that states the user should call the constructor manually.

Objective 4.4: Design apps for globalization and localization

The Windows Store helps you bring your app to a large audience. It provides support for selling your app in many countries and takes care of all the financial ramifications. You can create an English-only version of your app, or perhaps an app in your own language, but that would reduce your potential audience.

WinJS contains functionality you can use to both globalize and localize your app. Globalization is the process of making sure your app functions in multiple cultures. Localization is the process of customizing your app for a specific culture and locale.

In this objective, you examine the various options you have for globalizing and localizing your app.

This objective covers how to:

- Implement .resjson files to translate text
- Implement collation and grouping to support different reading directions
- Implement culture-specific formatting for dates and times
- Bind JSON properties to resources by using the *data-win-res* property

Planning for globalization

When you think about a global audience for your app, one of the first things that comes to mind is the text you display in your app. This is indeed one of the largest visible parts of your application that needs to be translated to other languages.

However, string data is not the only thing you must consider. Some of the things that vary around the world are the representation of dates and times, calendars, numbers, measures (units), phone numbers and addresses, currencies, paper sizes, the way text is sorted, and the direction of text and fonts for specific character sets.

Creating a globalized application means you don't hardcode any of these items. For example, you shouldn't display a date directly on the screen. Instead, you should format it with the current settings of the user. WinJS contains APIs that can help you display these types of data in the correct format.

Although globalization is mainly about using the correct APIs, there are other factors you need to consider.

Think of images, for example. One of your images might contain some text that needs to be translated in order for your image to be understood correctly in other parts of the world. Or maybe your image can't be mirrored to support a right-to-left language.

The same is true of text. Maybe your app uses expressions or phrases that won't be understood by people throughout the world and are difficult to translate. To save money on translation costs, you should try to avoid these expressions.

When sorting text, don't assume that sorting is always alphabetic. Japanese Kanji characters have the unique property of having more than one pronunciation, depending on the word and context they are used in. This means that one character can be sorted in multiple different ways, depending on its use. A special feature called Furigana is used to specify the phonetics for a character and uses that for sorting. That's one example, but there are many variations. Some languages sort on the number of pen strokes, for example.

What about text that a user enters in a specific language? In right-to-left environments, the user can switch the text direction on the on-screen keyboard. If you need to display the entered text again, you need to save the text direction with it so you can display it in the correct way.

You should also exercise caution when working with maps. Perhaps you think the borders drawn for countries in a particular map are correct, but someone in another country/region disagrees with you. The official MSDN documentation advises you to refer to "country/region" rather than only "country." Displaying a name in a list of countries that isn't recognized as a country by other countries/regions could cause problems.

Another consideration is the web services you are using. You can use different URLs to get localized content back from a service, or you can specify the language you want through a header or query string parameter.

In Windows, you can configure the languages you want to use. You can add multiple languages and specify your most preferred language. To access language settings, open the Clock, Language, and Region applet in Control Panel and select Add a language. Figure 4-3 shows the Language window.

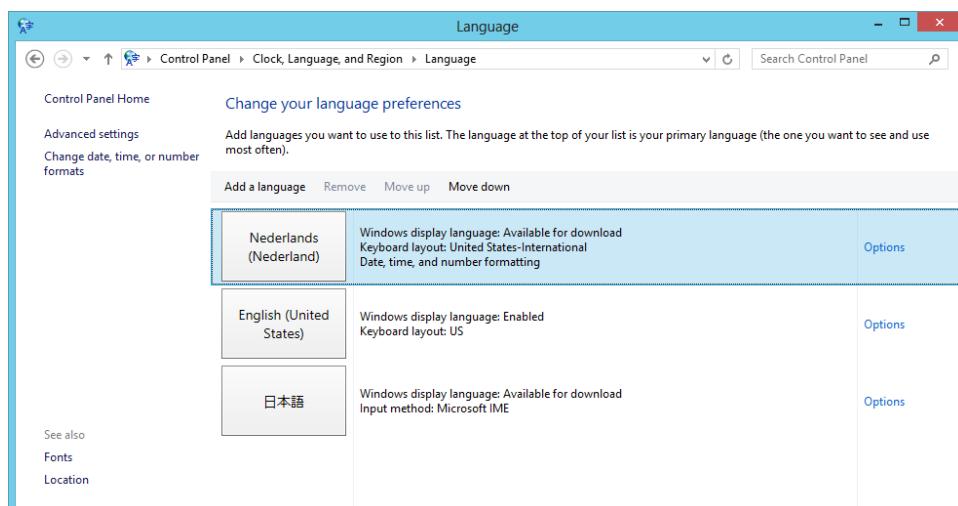


FIGURE 4-3 The language preferences screen

In Figure 4-3, three languages are installed: Dutch, English, and Japanese. These settings are used by Windows for loading resources in your app. Windows will search for the best matching language, or it will fall back to the default language of the app.

For each language, you can also configure the date, time, and number formats, as shown in Figure 4-4.

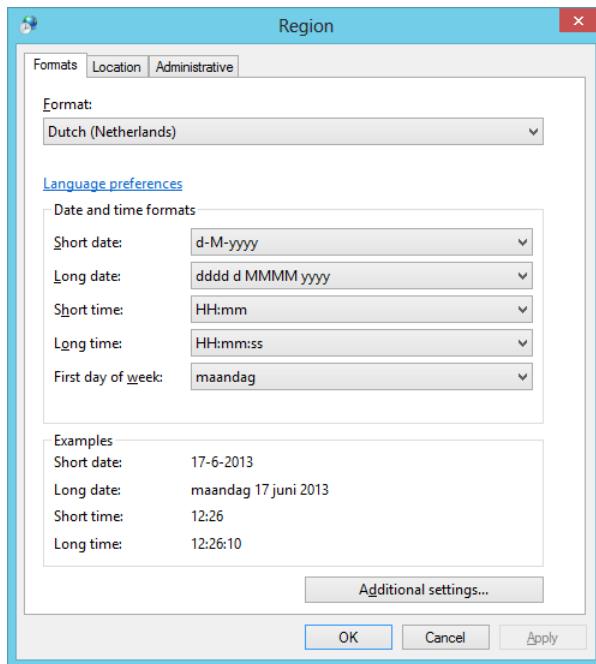


FIGURE 4-4 Regional settings in the Region dialog box

Globalization APIs help you deal with all of these settings. You can get these preferences manually by using `Windows.System.UserProfile.GlobalizationPreferences`. Listing 4-26 shows an example of getting the user's globalization preferences.

LISTING 4-26 Using `Windows.System.UserProfile.GlobalizationPreferences`

```
var userLanguages = Windows.System.UserProfile.GlobalizationPreferences.languages;
var userCalendar = Windows.System.UserProfile.GlobalizationPreferences.calendars;
var userClock = Windows.System.UserProfile.GlobalizationPreferences.clocks;
var userHomeRegion = Windows.System.UserProfile.GlobalizationPreferences.
    homeGeographicRegion;
var userFirstDayOfWeek = Windows.System.UserProfile.GlobalizationPreferences.
    weekStartsOn;
```

You can also get the specific language tag through the `.GlobalizationPreferences class`. You could use this string to pass it to a web service to get localized content. Listing 4-27 shows how to do this.

LISTING 4-27 Getting the *languageTag*

```
var topUserLanguage = Windows.System.UserProfile.GlobalizationPreferences.languages[0];
var userLanguage = new Windows.Globalization.Language(topUserLanguage);
var languageTag = userLanguage.languageTag;
```

MORE INFO GLOBALIZATION PREFERENCES

You can find a globalization preferences sample as a part of the Windows Software Development Kit (SDK) at <http://code.msdn.microsoft.com/windowsapps/Globalization-preferences-6654eb36>.

Localizing your app

Whereas globalization is the process of making sure your app doesn't depend on any hard-coded cultural settings, localization is the process of adapting your app for a specific culture and language. To localize an app, you have to translate text, change images, and format data to customize the app for user preferences.

String data

In your app, you probably use quite a lot of text data. When building your app, you put text directly into your HTML and JavaScript code. But when you want to translate your app to another language, you have to move all the strings to separate files called *resource files*.

A resource file has the extension *.resjson* and is placed in a special directory called *strings* inside your project. Listing 4-28 shows a typical resource file for the English spoken in the United States.

LISTING 4-28 A resource file with English (United States) text

```
{  
    "greeting" : "Hello",  
    "_greeting.comment" : "A welcome greeting.",  
  
    "farewell" : "Goodbye",  
    "_farewell.comment" : "A goodbye."  
}
```

Listing 4-29 contains the Dutch translation of these strings. The resource identifiers are exactly the same; you simply translate the value.

LISTING 4-29 A resource file with a Dutch translation and English comments

```
{  
    "greeting" : "Hallo",  
    "_greeting.comment" : "A welcome greeting.",  
  
    "farewell" : "Tot ziens",  
    "_farewell.comment" : "A goodbye."  
}
```

Identifiers that begin with an underscore are ignored by the runtime. You can add a `_<identifier>.comment` key to add a hint for the translator. You can also add a `_<identifier>.source` to keep the original phrase linked to the translation.

To bind these resource files to your HTML, use the `WinJS.Resources.processAll` method. This method goes through your code and retrieves all resources for you.

By subscribing to the `contextchanged` event, you can reload your string resources when a user changes the language settings on his device. You can do this with the following code:

```
WinJS.Resources.addEventListener("contextchanged", function () {
    WinJS.Resources.processAll();
});
```

It's important to translate complete sentences instead of words. For example, the sentence "The task could not be created" translates into Dutch as "De taak kan niet worden aangemaakt." The sentence "The document could not be created" translates into "Het document kan niet worden aangemaakt." In English, you can just replace "task" with "document," but in Dutch you also have to change "de" to "het."

To solve this, localize the entire sentence instead of separate words. This may seem like a lot of work, but it will increase the quality of your app. The same is true for reusing words in different contexts. Maybe you can use the same word in your language, but in other languages you might need to use another word.

You can use string resources from HTML by using the `data-win-res` property. The following code shows how to use the resource file from Listing 4-28 and 4-29 in your HTML.

```
<h2><span data-win-res="{textContent: 'greeting'}"></span></h2>
<h2><span data-win-res="{textContent: 'farewell'}"></span></h2>
```

If you want to translate a specific attribute, like `alt` on an `img` element, you can use the syntax `data-win-res="{<attribute> : '<identifier>'}"`. You can add multiple attributes by separating them with a comma. If you have a hyphen in your attribute name (like with the `aria-label` attribute), enclose your attribute name in single quotes, like this:

```
<div id="newItemsSection"
    data-win-res="{attributes: {'aria-label':'aria_newItems'}}"
/>
```

You can also use resource strings for control properties. You then use this syntax: `{winControl: {<property> : '<identifier>'}}`.

When working with strings from JavaScript, use the `WinJS.Resources.getString` method. This returns an object with a `value` and `lang` property, and an `empty` flag indicating the resource wasn't found. You can use the `lang` property to ensure that the correct font is used. The following code shows how to get a resource string into JavaScript.

```
var statusDiv = document.getElementById("statusMessage");
var str = WinJS.Resources.getString('scenario3Message');
statusDiv.textContent = str.value;
statusDiv.lang = str.lang;
```

This method loads resources in the user's default language. If you want to load resources in another language, use the *ResourceLoader* class.

When sorting and grouping data, you shouldn't depend on the rules you are accustomed to from your own alphabet. Instead, you should use the *localeCompare* method that's defined on the string class. This method returns -1, 0, or +1, depending on the sort order of the system's default locale. It returns -1 if your string sorts before the other string, 1 if your string comes after the other string, and 0 if both strings are equivalent.

This code snippet shows how to use the *localeCompare* method:

```
var str1 = "def";
var str2 = "abc"
document.write(str1.localeCompare(str2)); // Output: 1
var str3 = "ghi";
document.write(str1.localeCompare(str3)); // Output: -1
var str4 = "def";
document.write(str1.localeCompare(str4)); // Output: 0
```

When grouping items with the correct collation for a specific language, you can use the *CharacterGroupings* class that you can find in the *Windows.Globalization.Collation* namespace. This class gives you the functionality to get a label for any string that you pass in. The following code sample shows how to get character groupings for the current locale.

```
var characterGroupings = new Windows.Globalization.Collation.CharacterGroupings();
var size = characterGroupings.size;
if (size > 0) {
    // Get the first characterGrouping.
    var characterGrouping = characterGroupings.getAt(0);
    var first = characterGrouping.first;
    var label = characterGrouping.label;
}
```

When using the *en-US* locale, you will get groupings for special characters (such as the ampersand, or &) and numbers (0 through 9), and for all letters of the Latin alphabet. You can then use the *characterGroupings.lookup* method to get the label for a specific string.

Images

You do not have to localize images if they do not contain text or other culture-specific information. Localizing images adds to the download size of your app, which can affect the user experience, so avoid image localization if possible.

If you must localize images, put them in a resource-specific folder. For example, you can place an image in *images/en-US* and a copy of the same image in *images/NL-nl* to create both an English and a Dutch version of your image.

Within your HTML, you can reference the image without any qualifiers. It might look like you are pointing to an actual file on disk, but you are actually loading the image from the package. The resource manager responsible for loading images loads the most appropriate version of the image. Use the following code to reference the image within HTML:

```
Images/Logo.png
```

In addition to language, you can specify a scale factor and contrast mode. All of these items are considered when loading an image. The following code shows an example of how to reference an image's location, scale factor, and contrast mode:

Images/en-US/homeregion-USA/logo.scale-100_contrast-white.png

MORE INFO RESOURCE QUALIFIERS

For more information on the contrast, scale, and language qualifiers, visit <http://msdn.microsoft.com/en-us/library/windows/apps/hh965372.aspx>.

If you need to reverse an image for a right-to-left language, you can use CSS. Applying the *transform:scaleX(-1)* style flips the image. If you can't flip your image but need to replace it with another version in a right-to-left language, you can use the *layoutdir-RTL* qualifier in the image name.

Dates and times

As shown in Figure 4-4, there are a lot of possible settings when it comes to working with dates and time. You should never work with a date or time manually to display it on the screen.

Instead, use the *Windows.Globalization.DateTimeFormatting.DateTimeFormatter* class. This class helps you format a date and time easily while following the user's preferences.

When creating an instance of the *DateTimeFormatter* class, you pass it a string that describes the result you want, as follows:

```
var sdatefmt = new Windows.Globalization.DateTimeFormatting.  
    DateTimeFormatter("shortdate");
```

Now you can use this formatter to format a date:

```
var sdate = sdatefmt.format(new Date());
```

You can pass all kinds of format strings to your constructor. However, be careful to avoid representations that are not valid in all languages. Specifying that you want a *month day* pattern, for example, is valid in the United States but not in the Netherlands.

Some examples of format templates you can use are:

- *shortdate*
- *shorttime*
- *dayofweek*
- *day*
- *month*
- *year*
- *day month year*

- *hour*
- *minute*
- *second*

MORE INFO FORMAT TEMPLATES

For a complete list of all the format templates and how to combine them, visit <http://msdn.microsoft.com/library/windows/apps/BR206828>.

Numbers and currencies

When working with numbers and currencies, you can use the following built-in formatters:

- **CurrencyFormatter** Formats and parses currencies
- **DecimalFormatter** Formats and parses decimal numbers
- **PercentFormatter** Formats and parses percentages
- **PermilleFormatter** Formats and parses permillages

These classes can be used to convert a number to a string or to parse a string to a number. You use these classes by creating a new formatter with or without a language code, and set any necessary properties before calling *format* or *parse*. The following code shows how to use the *CurrencyFormatter* to format and parse a number:

```
var userCurrency = Windows.System.UserProfile.GlobalizationPreferences.currencies;
var currencyFormat =
    new Windows.Globalization.NumberFormatting.CurrencyFormatter(userCurrency);
var currencyNumber = 1234.56;
var currencyFormatted = currencyFormat.format(currencyNumber);
var currency1Parsed = currencyFormat.parseDouble(currencyFormatted);
```

When working with the formatters, you can set the *isGrouped* and *fractionDigits* properties. These properties control whether you want to display the group separator, and a minimum number of fractional digits you want to display.

Calendars

When working with dates and times, it's important to avoid manual arithmetic. Things can go wrong when another calendar has another number of days in a month (such as the Hebrew calendar), or when you have to deal with adjustments for daylight saving times and leap years. Instead, you should use the *Windows.Globalization.Calendar* class.

You can use the code in Listing 4-30 to display information about the current calendar settings.

LISTING 4-30 Getting calendar data

```
var CalendarIdentifiers = Windows.Globalization.CalendarIdentifiers;
var ClockIdentifiers = Windows.Globalization.ClockIdentifiers;

var calendarDate = new Date();

var currentCal = new Windows.Globalization.Calendar();
currentCal.setDateTime(calendarDate);
var calDesc = "User's default calendar: " + currentCal.getCalendarSystem() + "\n" +
    "Name of Month: " + currentCal.monthAsString() + "\n" +
    "Day of Month: " + currentCal.dayAsPaddedString(2) + "\n" +
    "Day of Week: " + currentCal.dayOfWeekAsString() + "\n" +
    "Year: " + currentCal.yearAsString();
```

When you have an instance of a *Calendar* object, you can use this to perform arithmetic on a date. A *Calendar* object has methods for adding nanoseconds, seconds, minutes, hours, days, weeks, and years to a date. It can also give you information such as how many days there are in a year or hours in a day. This code shows how to add and subtract a day from a date:

```
var currentCal = new Windows.Globalization.Calendar();
currentCal.setDateTime(new Date(2013, 1, 1));

var endDate = currentCal.clone();
currentCal.addDays(-1);
endDate.addDays(1);
```

**EXAM TIP**

You should never perform manual math on a date or time. Always use the built-in classes so you don't introduce mistakes.

Localizing your manifest

When distributing your app to multiple markets, you should localize your manifest. This enables you to show a translated description for your app in the Windows Store.

You can localize several items in your manifest:

- The display name, short name, and description
- Descriptions on the Declarations page
- URLs on the Content URLs page

To set the default language, open the Package.appxmanifest file in Visual Studio Editor. On the Application UI tab, specify the default language for your app. This language is used as a *fallback language* for your app. If the user requests a language that's not available in your app, it switches to the fallback language.

Inside your manifest use *ms-resource:<identifier>* to reference resource strings. For example, to set your application title, you can add an *app_title* to your resource file and reference it with *ms-resource:app_title*.

Figure 4-5 shows the Application UI tab with a default language of *en-US*.

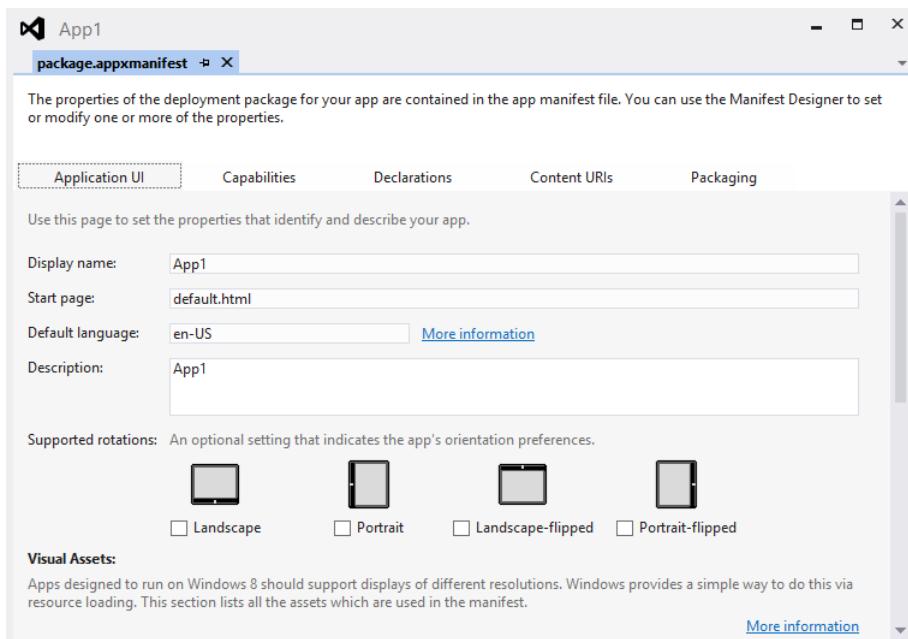


FIGURE 4-5 The Visual Studio App Manifest Designer

MORE INFO LOCALIZING THE MANIFEST

For more information on how to localize your manifest, visit <http://msdn.microsoft.com/en-us/library/windows/apps/hh454044.aspx>.

At runtime, the Windows Runtime determines the language preferences of the user, matches them with the supported language in the application manifest, and creates what is called an *application language list*. This list serves to determine the language or languages for application resources, dates and times, numbers, and other objects.

Microsoft recommends that you let Windows handle all of the intricacies of matching languages because there are many factors that influence the priority of language tags.

MORE INFO LANGUAGE TAGS AND QUALIFY RESOURCES

The MSDN documentation provides information on managing language and region in Windows Store apps using JavaScript and HTML at <http://msdn.microsoft.com/en-us/library/windows/apps/hh967758.aspx>.

Using the Multilingual App Toolkit

One tool you should definitely use is the Multilingual App Toolkit. This toolkit is an extension for Visual Studio that helps you localize your app and manage translations.

After you create a default resource file for your app, you can use the Multilingual App Toolkit to translate your app to other languages. The tool helps you create resource files for other languages and also for a pseudo language. The pseudo language is helpful for testing your app and making sure that all text is moved to a resource file. The pseudo language is also designed to show you any places in your app where translations can lead to layout errors (because some languages use more characters, which can take up more space).

To enable the toolkit, select Enable Multilingual App Toolkit from the Tools menu, and then select the Add Translation Languages option from the Project menu.

Some languages support Microsoft automatic translation. This is a machine translation that can get you started on creating a new translation. A human native speaker must still check the translation, but automatic translation can save you a lot of time.

After adding languages and building your project, the toolkit adds resource files for you. By using the supplied editor, you can then execute a machine translation. The tool uses *XLF files*, which are used by most translators. You can send\ to the translator, let them translate your resources, and then import their changes into your app.

MORE INFO MULTILINGUAL APP TOOLKIT

You can download the toolkit from <http://msdn.microsoft.com/en-us/windows/apps/hh848309.aspx>. A complete discussion of how to use the toolkit is at <http://msdn.microsoft.com/en-us/library/windows/apps/jj569303.aspx>.



Thought experiment

Globalizing and localizing your app

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You created an app that tracks holidays and shows you when your next day off will be. You have built the app for your local language and now you want to translate it. Your app uses HTML and JavaScript, and includes a few images.

Answer the following questions:

1. Which elements should you globalize and localize?
2. Why is it important to use the built-in APIs? Which APIs do you plan on using?
3. How can you organize your languages?

Objective summary

- Globalization is the process of making sure your app does not depend on a hard-coded language. Instead, you use built-in APIs for things like formatting numbers, currencies, and dates.
- Localization is the process of translating text and images to a specific locale so that your app supports that language.
- Use JavaScript Object Notation (JSON) resource files (.resjon) for storing resource strings. Call *WinJS.Resources.processAll* to bind all your resources. Use the *data-win-res* attribute to bind resources in HTML. In JavaScript, use *WinJS.Resources.getString*.
- Use the *localeCompare* method on a string and the *CharacterGrouping* class for sorting and grouping string data.
- You can localize images by moving them to a folder that contains the current locale. Reference the image by name as if it were located in the root folder.
- You can use the *DateTimeFormatter* class to format and parse dates and time. You can use the *CurrencyFormatter*, *DecimalFormatter*, *PercentFormatter*, and *PermilleFormatter* classes to work with numbers.
- You can translate resources in your manifest to localize content for the Windows Store. The Multilingual App Toolkit is a useful tool for translating your app.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. You want to localize an image for Hebrew. What do you do?
 - A. Put the image in images/IW/<yourimagename>.png and reference image/<yourimagename>.png from your HTML.
 - B. From JavaScript, load the correct image from your images/IW folder and bind that to the *img* tag in your HTML.
 - C. Use *transform: scaleX(-1)* in your CSS to mirror your image.
 - D. Put the image path in a resource string and use that to load the image.
2. You want to format a number so that it always has two-digit fractions. Which class do you use?
 - A. *CurrencyFormatter*
 - B. *DecimalFormatter*
 - C. *PercentFormatter*
 - D. *PermilleFormatter*

3. You are building an app that uses aria labels for accessibility. You want to bind the aria labels to your resource file. Which syntax do you use?
- A. `data-win-res="{aria-label: 'aria_greeting'}`
 - B. `data-win-res="{ attributes: {'aria-label': 'aria_greeting'}}"`
 - C. `data-win-res="{winControl: {aria-label : 'aria_greeting'}}"`
 - D. `data-win-res="{aria-label: WinJS.Resources.getString("aria_greeting"))}`

Chapter summary

- By using promises and web workers you can execute code asynchronously or on another thread. This way, you avoid blocking the UI thread, which makes sure that your user experience remains fast and fluid.
- CSS3 transitions can be used to smoothly change from one CSS style to another.
- CSS3 animations use key frames to create complex animations that can then be linked to an element.
- The built-in animation library is based on CSS3 and supports all the advised Windows Store animations.
- You can create custom controls by replacing functionality on the prototype object of an existing control (which affects all controls of that type) or by deriving a new control and making your changes in your new control.
- When globalizing your app, it's important to use the built-in Windows Globalization APIs when working with date, time, numbers, and currencies.
- You use JSON resource files (`.resjson`) to store translated resources, which can then be used from HTML and JavaScript.
- You can localize images by moving them to a locale-specific folder.

Answers

This section contains the solutions to the thought experiments and answers to the lesson review questions in this chapter.

Objective 4.1: Thought experiment

1. Fetching data is an I/O operation that's not CPU bound. You can use promises and the *join* method to execute multiple requests in parallel. That way, you will bring your download time down to the longest request instead of the sum of all requests.
2. You can use web workers to split the work over multiple threads. Especially on multi-core machines (as are most devices today), this will improve performance because you can parallelize the CPU-bound processing.
3. Web workers move work to a separate thread. This is useful for CPU-bound work that can be split to multiple cores. It is also useful for moving work out of the UI thread so you don't block that thread for too long, which would make the UI unresponsive. Promises are used for executing an asynchronous operation that will eventually return. You can then attach methods that will be executed asynchronously when the promise finishes. Using promises is ideal in a single-threaded language like JavaScript to ensure an operation returns immediately so that it doesn't block your UI thread.

Objective 4.1: Review

1. **Correct answer:** C
 - A. **Incorrect:** When your script executes immediately at the beginning, your HTML won't be ready yet. The *processAll* method, which creates the toggle control for you, hasn't executed yet and you won't be able to find the control.
 - B. **Incorrect:** Your *processAll* method returns a promise object and then executes asynchronously. Because of this, it is possible the *processAll* method isn't finished yet and your toggle control is not created yet. Instead, you should attach a handler to the *processAll* method that will be executed after the *processAll* method finishes.
 - C. **Correct:** The *processAll* method initializes all your controls. It returns a promise object and, when this promise finishes, all your controls are ready. The attached *then* method will then be executed and will be able to access your toggle control.
 - D. **Incorrect:** Although your HTML will be ready at the end of the document, your *processAll* method could still be running. Only when that method finishes can you access your toggle object and configure it.

2. Correct answer: B

- A. Incorrect:** Your web worker doesn't have access to the DOM. This makes it impossible to use `document.getElementById` to get the text field in your worker.
- B. Correct:** Your main page has access to the control and the event. You need to use `postMessage` to communicate with your web worker.
- C. Incorrect:** When you use `postMessage` to send an object to the web worker, the object is cloned. The `onchange` event inside the web worker will be different than the `onchange` event on the main page. Because of this, your event will never fire inside your worker.
- D. Incorrect:** The `postMessage` method clones objects and only supports basic JavaScript types. You won't be able to send a method through `postMessage`.

3. Correct answer: D

- A. Incorrect:** This is unnecessary. You can add a `done` handler with an error handler to your list of promises. That method will be called whenever an error occurs in one of the promises.
- B. Incorrect:** You shouldn't add a `then` method with an error handler. Instead, you should add a `done` method that is called whenever one of the promises in the chain throws an exception.
- C. Incorrect:** Because of the asynchronous nature of the promises, your `catch` block won't be called when an error occurs because your code already exited the `try/catch` block.
- D. Correct:** The `done` method will be called whenever one of the promises throws an exception. You don't need to specify a method for a correct result; instead, you can simply add an error handler.

Objective 4.2: Thought experiment

1. The easiest way to create this is by using an animation with *key frames*. A transition is not suitable because you have a few different states you want to use in your animation (each with a little less bounce).
2. The force of the ball strike will affect the length of the animation.
3. You need to animate the `top` property and a move to the right or the left. By using different sets of key frames for `top`, `left`, and `right`, you can easily compose the animation and apply it to your element.
4. You can use *ease-in* (slow to fast) for the animation from top to bottom, and you can use *ease-out* (fast to slow) for the animation from bottom to top (the bounce).

Objective 4.2: Review

1. Correct answer: C

- A. **Incorrect:** You shouldn't use JavaScript for these kinds of animations. Instead, you can use CSS3 transitions, which are better performing and easier to use.
- B. **Incorrect:** Changing key frames from JavaScript is too unwieldy. Using transitions is much easier for these types of animations.
- C. **Correct:** By applying a transition for the *top* and *left* properties, you can change those properties from JavaScript, after which the transition will do the smooth animation for you.
- D. **Incorrect:** This isn't necessary. CSS3 transitions don't block the UI thread and are much less complex.

2. Correct answer: A

- A. **Correct:** The *transition-duration* property controls how long your transition takes. By increasing the value, your transition will be slower.
- B. **Incorrect:** The *transition-timing-function* controls the rate at which the property values change. It doesn't control how long the animation takes.
- C. **Incorrect:** Although key frames give you a lot of freedom, you still control the length of the animation through the *duration* property.
- D. **Incorrect:** This event only fires after the transition is already finished. You can't use it to extend the animation.

3. Correct answer: B

- A. **Incorrect:** The *animation-direction* property controls whether your animation moves from beginning to end or from end to beginning. You can use this to change the direction on alternate cycles. For example, move from left to right and then from right to left.
- B. **Correct:** By setting *animation-fill-mode* to *forwards*, the last calculated properties will be used when your animation ends.
- C. **Incorrect:** Changing the *animation-delay* property controls when your animation starts.
- D. **Incorrect:** The *animation-timing-function* controls the rate at which your properties change. It doesn't retain those values when the animation ends.

Objective 4.3: Thought experiment

1. You can share code by using a base class that implements all the code that is equal for both chart controls. You define this control by using `WinJS.Class.define`. Then you create both of your chart controls and let them derive from the base control by using `WinJS.Class.derive`. That way, both classes will share all the code that's implemented in the base class.
2. XML comments will make it much easier for users to use your controls. You should include `summary`, `param`, and `return` comments, at a minimum. That way, users will see these comments show up in IntelliSense when they start using your controls.

Objective 4.3: Review

1. **Correct answer:** D
 - A. **Incorrect:** Creating a completely new toggle would be a waste of time. Instead, you can just extend the original toggle with a few lines of code.
 - B. **Incorrect:** This will change the implementation for all toggles throughout your entire app. By deriving from the toggle and adding logging to the new control, you can use that new control on the billable page.
 - C. **Incorrect:** The `onchange` event is fired after the toggle is switched, and the error has already occurred.
 - D. **Correct:** With this technique, you have a new control you can explicitly add to your billable page to only add logging in that scenario.
2. **Correct answers:** B, D
 - A. **Incorrect:** Adding to the prototype of your object will create an instance, not a static member.
 - B. **Correct:** With this technique, you will create a static method that can be called without creating a new instance of `Carousel`.
 - C. **Incorrect:** The second parameter is for instance members, not for static members.
 - D. **Correct:** This will add a static method to your class that can be called without creating a new instance of `Carousel`.
3. **Correct answer:** A
 - A. **Correct:** You use the `apply` method to call the base constructor and apply that constructor to your object.
 - B. **Incorrect:** The `WinJS.Class.mix` method is used to add members of a source class to your target class. It doesn't copy the constructor, and it doesn't call any of those members for you.

- C. **Incorrect:** The `call` method should be used to call instance members. It can't be used to call a constructor.
- D. **Incorrect:** This won't solve the problem. Users of your class aren't able to call the base constructor themselves. Instead, you can solve the problem by using the `apply` call.

Objective 4.4: Thought experiment

1. You need to globalize all code that deals with date and time functions, both for formatting and parsing. You should also check whether your images need to be localized. You have to create resource files for all text, both in HTML and JavaScript code, and in your manifest.
2. The globalization APIs are built to work with all the user preferences for language settings, and for settings that display date, time, number, and currency data. It would be difficult to write this code yourself. You can use the `DateTimeFormatter` class when working with date and time data.
3. You need to organize your languages by moving all resources (both images and text) into specific folders with the locale name in the folder name.

Objective 4.4: Review

1. **Correct answer:** A
 - A. **Correct:** In this way, the resource manager can find the correct file for the Hebrew language.
 - B. **Incorrect:** Binding to a localized image can be done from HTML. The resource manager will automatically load the correct version.
 - C. **Incorrect:** This will mirror your image in CSS. It won't load the localized version of your image.
 - D. **Incorrect:** Although this is possible, it's unwieldy. You can just bind to the image from your HTML, and the resource manager will load the correct version.
2. **Correct answer:** B
 - A. **Incorrect:** This class is specifically for displaying currency. It can display currency amounts with two-digit fractions, but it will also add the local currency symbol.
 - B. **Correct:** `DecimalFormatter` can be configured to display a two-digit fraction.
 - C. **Incorrect:** `PercentFormatter` adds a percent symbol to your formatted number.
 - D. **Incorrect:** `PermilleFormatter` adds a per mille symbol to your formatted number.

3. Correct answer: B

- A. Incorrect:** To bind to *aria-label*, you should use the attribute syntax.
- B. Correct:** This code will bind your *aria-label* attribute to a resource.
- C. Incorrect:** The *winControl* syntax is used to bind control options to resources.
- D. Incorrect:** The *WinJS.Resources.getString* method is used to load a resource from JavaScript code.

CHAPTER 5

Manage data and security

In this chapter, you learn how to use the local file system in an efficient and effective way, implement a data caching strategy, and secure application data using certificates and encryption algorithms.

Objectives in this chapter:

- Objective 5.1: Design and implement data caching
- Objective 5.2: Save and retrieve files from the file system
- Objective 5.3: Secure application data

Objective 5.1: Design and implement data caching

Windows Store applications have various options to store data, and each one has its advantages and disadvantages. The right place to store information or cache it depends on the application, the scenario in which the user works, and whether the application relies on HTML5/JavaScript or on Extensible Application Markup Language (XAML) with C#, Microsoft Visual Basic (VB), or C++.

This objective covers how to:

- Choose which types of items (user data, settings, application data) in your app should be persisted to the cache based on requirements
- Choose when items are cached
- Choose where items are cached (Windows Azure, remote storage)
- Select a caching mechanism
- Store data by using indexDB, LocalStorage, and SessionStorage

Understanding application and user data

A Windows Store app works with different kinds of data, mainly application data and user data. Application data, also referred to as *app data*, represents information the application creates, updates, and deletes, or, in general terms, uses. This kind of data is bound to the

application and has no meaning outside of it: The data cannot live without the application because it has no sense outside of the app. Types of application data include:

- **User preferences** Any setting the user can personalize in an application, such as choices for browsing, searching, and receiving notifications. For example, the preferred city for a weather app is a user preference, which has no meaning outside that weather application.
- **Runtime state** Data entered by the user and preserved in a page preserved in the navigation state. Runtime state data has meaning only inside the application.
- **Reference content** An item related to the application. For example, the list of cities for a weather application or the list of terms for a translation app is bound to the application.

User data is independent from the application and is usually stored outside the application in a database, nonrelational storage, or file. This kind of data can be used by other applications. Types of user data include:

- **Entities** The main classes a developer works on. Entities can include invoices, orders, and customer data, such as for an accounting app.
- **Media files** Audio files, videos, and photos.
- **Documents** Files that can be stored via a web service, Microsoft SkyDrive, or a Microsoft SharePoint service.

Application data must be stored in a specific per-app per-user store. This data cannot be shared across users and needs to be placed in isolated storage to prevent access from other apps and users. If the user installs an application update, this kind of data must be preserved, but the data must be cleaned if the user removes the application from the system. It is important to store this kind of data in a type of storage that provides these features. Fortunately, Microsoft Windows 8 exposes a storage mechanism that perfectly suits these needs.

You cannot use the same storage mechanism for user data, such as application entities or documents, however. You can use user libraries, SkyDrive, or a cloud service to store user data.

Caching application data

The Windows Runtime (WinRT) offers many solutions to store application data in local and remote storage. This section analyzes the most important ones:

- Application data application programming interfaces (APIs)
- IndexedDB
- Extensible Storage Engine (ESE)
- HTML5 Web Storage
- Windows Library for JavaScript (WinJS)—specific classes, such as *sessionState*, *local*, *roaming*

Application data APIs

WinRT application data APIs provide access to local and roaming data stores. Simply choose the class and the name of the object; these classes can save and retrieve content from both local and roaming data.

Every Windows 8 application, regardless of the language it is written, can save settings and unstructured data files to local and roaming storage. The storage also provides a place to store temporary data. This area is subject to clean-up by the system automatically.

LOCAL STORAGE

You can save and retrieve application settings with a single line of code, and compose them using a simple data type or complex data type. The settings provide a mechanism to store composite types, which are sets of application settings to be managed as a single unit in an atomic operation. The following code references the *LocalSettings* and *LocalFolder* properties of the *ApplicationData* class in a local variable to use them throughout the code.

Sample of JavaScript code

```
var localSettings = Windows.Storage.ApplicationData.current.localSettings;  
var localFolder = Windows.Storage.ApplicationData.current.localFolder;
```

Settings can be saved and retrieved in a synchronous way. File and folder access follows the *async/await* pattern implemented by JavaScript promises.

MORE INFO PROMISES

Promises are covered in detail in Chapter 4, “Enhance the user interface.”

The following code saves and then retrieves a setting called *PageSize*:

```
localSettings.Values["PageSize"] = "20";  
  
(code omitted)  
  
var pageSize = localSettings.Values["PageSize"];  
if(!pageSize)  
{  
    // There is no value in the PageSize setting  
}  
else  
{  
    // pageSize has the saved value  
}
```

Settings are limited to 8 kilobytes (KB) per single setting, whereas composite settings are limited to 64 KB. Listing 5-1 demonstrates the use of a composite setting. The *PageSize* and *Title* key-value pair are stored in the *PrintSettings* composite setting.

LISTING 5-1 Saving and retrieving a composite setting

```
var localSettings = Windows.Storage.ApplicationData.current.localSettings;
var composite =
    new Windows.Storage.ApplicationDataCompositeValue();

composite["PageSize"] = 20;
composite["Title"] = "DevLeap Members";

localSettings.values["PrintSettings"] = composite;

var composite = localSettings.values["PrintSettings"];

if (!composite)
{
    // No data saved in PrintSettings
}
else
{
    // Access data in composite["PageSize"] and composite["Title"]
}
```

If you have a complex settings structure, you can create a container to better understand and manage settings categories. Use the *ApplicationDataContainer.CreateContainer* method to create a settings container. The following code excerpt creates a settings container named *PrintSettingsContainer* and adds a setting value named *PageSize*:

```
var localSettings = Windows.Storage.ApplicationData.current.localSettings;
var container =
    localSettings.createContainer("PrintSettingsContainer",
        Windows.Storage.ApplicationDataCreateDisposition.always);

if (localSettings.containers.hasKey("PrintSettingsContainer"))
{
    localSettings.containers.lookup("PrintSettingsContainer").values["PageSize"] = "20";
}
```

The second line of code creates a container named *PrintSettingsContainer* using the *CreateContainer* method. The second parameter represents the *ApplicationDataCreateDisposition*. The value of *Always* means the container should be created if it does not exist. The *ApplicationDataCreateDisposition* enumeration exposes a second value, *Existing*, that activates the container only if the resource it represents already exists.

To reference a container and retrieve its settings values, you can use the *Containers* enum to test for container existence and then reference the object to ask for its settings and relative values. The following code uses the *HasKey* method to test for container presence. If the method returns *true*, it retrieves the value for the *PageSize* setting using the *LookUp* method of the *Containers* collection.

```
var localSettings = Windows.Storage.ApplicationData.current.localSettings;
bool hasContainer = localSettings.containers.HasKey("PrintSettingsContainer");
bool hasSetting = false;
```

```

if (hasContainer)
{
    hasSetting = localSettings
        .containers.lookup("PrintSettingsContainer").values.hasKey("PageSize");
}

```

To delete a setting, you simply call the *Remove* method, as follows:

```
localSettings.values.remove("PageSize");
```

To delete a container, you can use the *DeleteContainer* method of the *ApplicationDataContainer* class:

```
localSettings.deleteContainer("PrintSettingsContainer");
```

Remember that this area of storage is suitable for settings, not for user data. Do not store more than 1 megabyte (MB) of data in this location.

If you have a highly complex setting structure or you need to store information locally, you can choose to store these values in a local file instead of a local settings container. As you learn in the next section, local storage gives you a simple way to manage files and folder.

For example, if the application wants to track the time the user performs some operations, the settings container becomes difficult to manage. A plain text file can be a simple but effective solution to store this log information. The code in Listing 5-2 creates a file called log.txt to store the current date and time.

LISTING 5-2 Logging to files in local user storage

```

function CreateLog()
{
    var localFolder = Windows.Storage.ApplicationData.current.localFolder;
    localFolder.createFileAsync("log.txt",
        Windows.Storage.CreationCollisionOption.replaceExisting)
        .then(function (logFile) {
            var formatter = new
                Windows.Globalization.DateTimeFormatting.DateTimeFormatter("longtime");
            var logDate = formatter.format(newDate());
            return
                Windows.Storage.FileIO.writeTextAsync(logFile, logDate);
        }).done(null, function (err) {
            // Something wrong during save on log file
        });
}

```

The code creates a file in the local folder storage called log.txt using the *CreateFileAsync* method. Then it references the file in the *WriteTextAsync* method of the *FileIO* class (of the *Windows.Storage* library). The *CreationCollisionOption* can be *ReplaceExisting* to create a new file or *FailIfExists* to return an error if the file already exists; *OpenIfExists* to create a new file if it does not exist or to open the existing one; or *GenerateUniqueName* to create a new file with an autogenerated number if a file with the same name already exists.

To read a value from a file, use the code in Listing 5-3.

LISTING 5-3 Reading values from files

```
function ReadLog()
{
    var localFolder = Windows.Storage.ApplicationData.current.localFolder;

    localFolder.getFileAsync("Log.txt")
        .then(function (logFile) {
            return
                Windows.Storage.FileIO.readTextAsync(logFile);
        }).done(function (logDate) {
            // Everything fine. The content is available in the logDate variable
        }, function (err) {
            // Something wrong during read
        });
}
```

If you want to use WinJS, you can also use the following syntax to obtain a reference to the local folder:

```
var localFolder = WinJS.Application.local.folder;
```

ROAMING STORAGE

Settings and files in local storage can be roamed to “follow” the user on different devices and places. For example, a user can set application printing preferences on his Windows 8 desktop and retrieve the same value on his tablet WinRT device. You do not need to manage data transfers because the Windows Runtime gives you the *RoamingSettings*.

Roaming settings are synchronized by the system that manages the overall process in respect to battery life and bandwidth consumption.

First, you can retrieve the roaming settings or folder reference in a similar way as the local one:

```
var roamingSettings =
    Windows.Storage.ApplicationData.current.roamingSettings;
var roamingFolder =
    Windows.Storage.ApplicationData.current.roamingFolder;
```

You can then access the settings in a synchronous way. This code saves and then retrieves a setting called *PageSize*:

```
roamingSettings.values["PageSize"] = "20";
(code omitted)
var value = roamingSettings.values["PageSize"];
if(!value)
{
    // No data saved to PageSize roaming settings
}
else
{
    // value contains the saved data
}
```

If you need to store complex values, you can use the same strategy you saw for the local settings in Listing 5-1. Create an *ApplicationDataCompositeValue* instance, assign the values to it, and then use the *RoamingSettings* reference to store the instance. See Listing 5-4.

LISTING 5-4 Composite setting stored in the roaming user profile

```
var roamingSettings =
    Windows.Storage.ApplicationData.current.roamingSettings;

var composite =
    new Windows.Storage.ApplicationDataCompositeValue();

composite["PageSize"] = 20;
composite["Title"] = "DevLeap Members";

roamingSettings.values["PrintSettings"] = composite;

var composite = roamingSettings.values["PrintSettings"];

if (!composite)
{
    // No data stored in the composite setting yet
}
else
{
    // Access data in composite["PageSize"] and composite["Title"]
}
```

You can group roaming settings in containers to manage complex structures easily.

The following code excerpt creates a settings container named *PrintSettingsContainer* and adds a setting value named *PageSize*:

```
var roamingSettings =
    Windows.Storage.ApplicationData.current.roamingSettings;

var container =
    roamingSettings.createContainer(
        "PrintSettingsContainer",
        Windows.Storage.ApplicationDataCreateDisposition.always);

if (roamingSettings.containers.hasKey("PrintSettingsContainer"))
{
    roamingSettings.containers.lookup("PrintSettingsContainer").values["PageSize"] =
        "20";
}
```

The first line of code creates a container named *PrintSettingsContainer* using the *CreateContainer* method. The second parameter represents the *ApplicationDataCreateDisposition*. The value of *Always* means the container should be created if it does not exist yet. The enum also contains a second value, *Existing*, that activates the container only if the resource it represents already exists.

To reference a container and retrieve its settings values, you can use the *Containers* dictionary to test for container existence and then reference the object to ask for its settings and relative values. The following code uses the *HasKey* method to test for container presence. If the method returns *true*, it retrieves the value for the *PageSize* setting.

```
Windows.Storage.ApplicationDataContainer roamingSettings =
    Windows.Storage.ApplicationData.current.roamingSettings;

bool hasContainer = roamingSettings.Containers.HasKey("PrintSettingsContainer");
bool hasSetting = false;

if (hasContainer)
{
    hasSetting = roamingSettings
        .containers.lookup("PrintSettingsContainer").values.HasKey("PageSize");
}
```

To delete a setting, you simply call the *Remove* method:

```
roamingSettings.values.remove("PageSize");
```

To delete a container, you can use the *DeleteContainer* method of the *ApplicationDataContainer* class.

```
roamingSettings.deleteContainer("PrintSettingsContainer");
```

As for local storage, you can use a class to store a file in the roaming user profile. The code in Listing 5-5 creates a file called log.txt to store the current time in the roaming profile.

LISTING 5-5 Logging to a file in the roaming user storage

```
function CreateLog()
{
    var roamingFolder = Windows.Storage.ApplicationData.current.roamingFolder;
    localFolder.createFileAsync("log.txt",
        Windows.Storage.CreationCollisionOption.replaceExisting)
    .then(function (logFile) {
        var formatter = new
            Windows.Globalization.DateTimeFormatting.DateTimeFormatter("longtime");
        var logDate = formatter.format(new
Date());
        return
            Windows.Storage.FileIO.writeTextAsync(logFile, logDate);
    }).done(null, function (err) {
        // Something went wrong during log save
    });
}
```

This code is similar to the code for accessing local storage. It creates a file in the roaming folder storage called log.txt using the *CreateFileAsync* method. Then it references the file in the *WriteTextAsync* method of the *FileIO* class (of the *Windows.Storage* library). The *CreationCollisionOption* can be *ReplaceExisting* to create a new file or *FailIfExists* to return an error if the file already exists; *OpenIfExists* to create a new file if it does not exist or to open

the existing one; or *GenerateUniqueName* to create a new file with an auto-generated number if a file with the same name already exists.

To read a value from a file, use the code in Listing 5-6.

LISTING 5-6 Reading values from a file in the roaming profile

```
function ReadLog()
{
    var roamingFolder = Windows.Storage.ApplicationData.current.roamingFolder;

    localFolder.getFileAsync("Log.txt")
        .then(function (logFile) {
            return Windows.Storage.FileIO.readTextAsync(logFile);
        }).done(function (logDate) {
            // Everything fine. Date in logDate variable
        }, function (err) {
            // Something wrong during read
        });
}
```

Files in the local settings space have no limits, whereas roaming settings are limited to the quota specified by the *RoamingStorageQuota* property of the *ApplicationData* class. If you exceed that limit, synchronization does not occur.

You can be notified when data changes in the roaming profile of the user. Let's say an application on the desktop changes some date in the roaming profile. The application on a different user device can be notified and react to this change, refreshing the user interface.

The following code uses the *DataChanged* event to handle this situation:

```
Windows.Storage.ApplicationData.current.addEventListener("datachanged",
    roamingProfileChanged);

function roamingProfileChanged (appData)
{
    // Refresh data or inform the user.
}
```

If you want to use WinJS, you can use the following syntax to obtain a reference to the roaming folder:

```
var roamingFolder = WinJS.Application.roaming.folder;
```

TEMPORARY STORAGE

If your application needs temporary data, you can use the temporary space reserved by the Windows Runtime to store this information. The code is similar to code for local and roaming storage but uses different classes to obtain the reference to the store.

Temporary storage is not limited in any way apart from the physical disk space.

Let's say that the log that tracks the time of every operation presented in the previous samples (Listings 5-2 and 5-5) is not permanent; the application simply stores log information that is useful just for a user session on the application. The code in Listing 5-7 uses the

TemporaryFolder property of the *ApplicationData.Current* class to retrieve the temporary storage folder and then create a file in it.

LISTING 5-7 Writing log files in the temporary folder

```
function CreateLog()
{
    var tmpFolder = Windows.Storage.ApplicationData.current.temporaryFolder;
    tmpFolder.createFileAsync("log.txt",
        Windows.Storage.CreationCollisionOption.replaceExisting)
    .then(function (logFile) {
        var formatter = new
            Windows.Globalization.DateTimeFormatting.DateTimeFormatter("longtime");
        var logDate = formatter.format(new Date());
        return Windows.Storage.FileIO.writeTextAsync(logFile, logDate);
    }).done(null, function (err) {
        // Something went wrong during log save on temporary folder
    });
}
```

The code is similar to the code for accessing local storage. It creates a file in the temporary folder storage called *log.txt* using the *CreateFileAsync* method. Then it references the file in the *WriteTextAsync* method of the *FileIO* class (of the *Windows.Storage* library). The *CreationCollisionOption* can be *ReplaceExisting* to create a new file or *FailIfExist* to return an error if the file already exists; *OpenIfExists* to create a new file if it does not exist or to open the existing one; or *GenerateUniqueName* to create a new file with an autogenerated number if a file with the same name already exists.

To read a value from a file, use the code in Listing 5-8.

LISTING 5-8 Reading a value from a temporary file

```
function ReadLog()
{
    var tmpFolder = Windows.Storage.ApplicationData.current.temporaryFolder;

    tmpFolder.getFileAsync("Log.txt")
    .then(function (logFile) {
        return Windows.Storage.FileIO.readTextAsync(logFile);
    }).done(function (logDate) {
        // Everything fine. Date in logDate variable
    }, function (err) {
        // Something wrong during read of temporary data
    });
}
```

If you want to use WinJS, you can use the following syntax to obtain a reference to the local folder:

```
var tmpFolder = WinJS.Application.temp.folder;
```

Using temporary storage enables your app to behave fast and fluid when the use of a network is limited. For example, a weather application can immediately display weather

information that was cached to disk from a previous session. After the latest information is available, the app can update its content gracefully, ensuring that the user has content to view immediately upon launch while waiting for new content updates.

IndexedDB

IndexedDB technology enables a Windows Store app written in HTML and JavaScript to store Indexed Sequential Access Method (ISAM) files that can be read using sequential or indexed cursors. This storage has some limits:

- 250 MB per application.
- 375 MB for all applications if the hard drive capacity is less than 30 gigabytes (GB).
- 4 percent of the total drive capacity for disks greater than 30 GB, with a limit of 20 GB.
- Page's URL must be listed in the *ApplicationContentUriRules* of the application manifest, unless you set the *ms-enable-external-database-usage* meta tag in the application home (start) page.

The following JavaScript code demonstrates how to access IndexedDB:

```
try
{
    var db = null;
    var sName = "CustomerDB";
    var nVersion = 1.2;

    if (window.indexedDB)
    {
        var req = window.indexedDB.open( sName, nVersion );
        req.onsuccess = function(evt) {
            db = evt.target.result;
            doSomething( db );
        }
        req.onerror = handleError( evt );
        req.onblocked = handleBlock( evt );
        req.onupgradeneeded = handleUpgrade( evt );
    }
}
catch( ex ) {
    handleException( ex );
}
```

Extensible Storage Engine (ESE)

This engine provides ISAM storage technology to save and access files. Access can be done using indexed or sequential navigation. ESE can be used in transactional code and provides a robust data consistency with transparent mechanisms to the developer. It also provides for fast data retrieval based on caching technology; it is considered a scalable technology for large-size files (over 40 GB).

The drawback of this technology is the fact that the API is provided only in C/C++. The call to this API must be wrapped in a WinRT component to be accessed by a Windows Store application written in a language different from C++.

HTML5 Web Storage

Any HTML and JavaScript application or a Windows Store app using C++, C#, or Visual Basic using the *WebView* control can use HTML5 Web Storage.

These APIs are suitable for storing key-value pairs of strings using the web standard. The APIs are synchronous and enable the application to store 10 MB per single user. The storage is isolated on a per-user and per-app basis, so there is no conflict between apps; and it offers both local and session storage.

You can use the provided *localStorage* class to store data locally if you need lightweight storage in a web context and the Windows Runtime is not available. Be aware that if you use the *sessionStorage* class instead of the *localStorage* class, data exists only in memory and does not survive application termination. The *sessionStorage* can be used for transient data.

WinJS.Application.sessionState

As the name implies, this store is available for Windows Store apps using JavaScript in the local context, and is suitable to store transitory application state during suspension to resume them when the application resumes.

This store can contain JavaScript objects with application-defined properties and is automatically persisted during suspension to the local *ApplicationData* class. There is no size limit. The serialization is automatic as well as the repopulation of the object when the application is re-launched.

If you want to use a web standard to store settings that can be shared to non-Windows Store apps, using *WinJS.Application.sessionState* is one of the preferred methods.

WinJS.Application.local

The second storage option dedicated to Windows Store apps using JavaScript is a wrapper for the *ApplicationData* class that makes the use of the local context or web context transparent for the developer. In the local context, the wrapper uses local storage to persist data; in the web context, it falls back to in-memory.

WinJS.Application.roaming

The third storage option dedicated to Windows Store apps using JavaScript is a wrapper for the *ApplicationData* class that makes the use of roaming profiles or the web context transparent for the developer. In the local context, the wrapper uses the roaming storage to persist data; in the web context, it falls back to in-memory.

Understanding Microsoft rules for using roaming profiles with Windows Store apps

Microsoft recommends some general guidelines for using roaming profiles with Windows Store apps. Use roaming settings for settings the user can reuse on a different device. Many users commonly work on two or more different devices, such as a desktop at work, a tablet at home, and/or a notebook while traveling. Using roaming settings enables the user to reuse all preferences on every device. When the user installs the application on a secondary device, all user preferences are already available in the roaming profile. Any changes to user settings are propagated to the roaming profile and applied to the settings on all devices.

These considerations apply to session and state information as well. Windows 8 roams all this data, enabling the user to continue to use a session that was closed or abandoned on one device when he uses a secondary device. For example, Vanni is playing his favorite game at home on his desktop. If he leaves the house, he can take his tablet and continue playing the game.

Using user data in a roaming profile is as easy as using local data, with just a few differences in class names. Use roaming data for anything that can be reused on a different device without modifications, such as color preferences, gaming scores, and state data. Do not use roaming profiles for settings that can differ from device to device. For example, be careful about display preferences. It is not practical to store the number of items to be displayed in a list because a desktop screen can easily represent 50 elements in an ordered list when a small tablet screen can become unreadable with just 20 elements.

To give you some examples, use the roaming profile for the following:

- Favorite football team (sports news app)
- First category to display in a page for a news magazine
- Color customization
- View preferences
- Ordering preferences

Some examples of state data include:

- The last page read for an e-book app
- The score for a game and relative level
- Text inserted in a text box

Do not use roaming data for information bound to a specific device, such as the following:

- The number of items to be displayed in a list
- History of visited pages
- Global printing preferences

Do not use roaming data for large amounts of data. Remember that the roaming profile has quota limits. The roaming profile does not work as a synchronization mechanism, either. Windows 8 transfers data from a device to the roaming profile using many different

parameters to determine the best time to start a transfer. This is also true for downloading the profile or changing the profile from other devices. This method is not reliable for instantly passing information from one device to another.



EXAM TIP

To use roaming profiles, verify that all prerequisites are met:

- The user must have a Microsoft account.
- The user has used the Microsoft account to log on to the device.
- The network administrator has not switched off the roaming feature.
- The user has made the device "trusted."

Caching user data

The following points summarize the complete reference of where to store the different kinds of information and the pros and cons of each storage mechanism.

User data can be stored in:

- Libraries
- SkyDrive
- HTML5 API
- External services, server applications, third-party databases, and cloud storage

Libraries

Every Windows Store app, whether written in C++, JavaScript, C#, or Visual Basic, can use libraries for storage. Even DirectX apps written in C++ can use libraries.

The *StorageFile* class and file pickers are the primary mechanisms for accessing user libraries. There are no size limitations and all the APIs are asynchronous.

File and folder pickers are useful when you want the user to participate in the selection of library items to manage or create.

Programmatic access requires a capability in the application manifest for each library the application needs to access. In the Downloads library, the code can write any file but can read only the file the app wrote. In practice, you cannot access files downloaded from other applications.

This storage is intended to survive application updates and installation because it represents user libraries, not application libraries. In practice, libraries exist by means of Windows 8. A library can also be used independently from the user that runs the application.

SkyDrive

All Windows Store apps, whether written in C++, JavaScript, C#, or Visual Basic, can use SkyDrive for storage. Even DirectX apps written in C++ can use SkyDrive.

SkyDrive supports storage for user data in the cloud and can be shared across devices, applications, and even platforms. SkyDrive is not a relational database but a shared storage service for items such as documents, photos, music, and videos. A size quota can be applied based on the level of the user account.

NOTE SKYDRIVE-SUPPORTED FILE FORMATS

For a list of supported file formats, refer to the SkyDrive documentation available at <http://msdn.microsoft.com/en-us/library/live/hh826521.aspx>.

You can use Live Connect Representational State Transfer (REST) APIs with JavaScript Object Notation (JSON) payloads to work with SkyDrive.

HTML5 File API

Like SkyDrive and libraries, all Windows Store apps can use HTML5 File API storage, including DirectX apps. HTML5 File APIs are web standard APIs for uploading and downloading files from a running application. As you learned in Chapter 1, an application can be suspended if it's not in the foreground. For transfers that continue while the application is not in the foreground, use the *BackgroundTransfer* class.

HTML5 Application Cache API

A Windows Store app using JavaScript supports the AppCache (formerly Application Cache API), as defined in the HTML5 specification, which enables pages to use offline content. AppCache enables webpages to save resources locally, including images, script libraries, style sheets, and other HTML-defined resources. In addition, AppCache enables URLs to be served from cached content using standard uniform resource identifier (URI) notation. By using AppCache to save resources locally, you improve webpage performance by reducing the number of requests made to the Internet.

External service, server application, third-party database, and cloud storage

Windows 8 does not provide a native local database for Windows Store apps nor a way to interact with data access libraries or a database. You cannot access SQL Azure directly, nor can you access SQL Server or SQL Express locally. There are no relational database APIs in the Windows Runtime.

To overcome this limitation, you can store and access data locally using a third-party solution, or you can rely on web services to interact with a service-oriented architecture (SOA)/REST back end. For example, a Windows Communication Foundation (WCF) service can be

hosted in the Microsoft Windows Azure platform to bridge an SQL Azure database from Windows Store apps.

If the application uses some form of web service or website, the app can use other ways to store data. For example, if the application makes HTTP calls to a website or web application, it can use the standard web cookie mechanism to share data with each request to the server. Applications written in JavaScript for both web and the local context can use cookies directly; applications written in C++, C#, and Visual Basic can leverage cookies via the *WebView* control. Any other apps can use cookies via the *IXMLHTTPRequest2* interface.

Cookies are a web standard and well-known technology but have a 4 KB limit per cookie. Cookies in the local context to the web via *XMLHttpRequest* (XHR) are subject to web cross-domain rules.

If a JavaScript-based application reads data from the web, you can leverage the HTML5 AppCache to make data available offline. This is another web standard technique that caches server response content locally and enables prefetching of web-based markup, scripts, CSS, and resources in general that do not contain code.

Any Windows Store app can consume web services using SOAP or REST and become the new user interface for every kind of existing or new solutions.



Thought experiment

Storing user settings

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

Your application needs to store some user settings as the number of items to display in a page. Which kind of storage do you plan to use, and why?

Objective summary

- The most important thing to remember is the difference between application data and user data.
- Application data is related to the application and can be stored in local, roaming, or temporary storage using different techniques.
- Use the *LocalSettings* property of the *ApplicationDataContainer* class to store data locally.
- Use the *RoamingSettings* property of the *ApplicationDataContainer* class to store data in the user roaming profile. The APIs hides the complexity of storing data remotely.
- User data has a meaning outside the application. Orders, customers, and so on are examples of user data.
- User data is normally stored in outside stores, such as databases or via web services.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the “Answers” section at the end of this chapter.

1. Which data types are considered application data? (Choose all that apply.)
 - A. User preferences
 - B. Application entities such as invoices
 - C. All data related to the application itself, such as users and groups
 - D. Reference content
2. How you can store application data? (Choose all that apply.)
 - A. Via the *LocalSettings* property.
 - B. Via the *RoamingSettings* property.
 - C. Via the *ApplicationDataCreateDisposition* enumeration.
 - D. There are no classes related to the application data type.
3. Is it possible to store a file locally?
 - A. No, Windows Store apps cannot access the file system.
 - B. No, Windows Store apps can only use web services to store data.
 - C. Yes, you can use the *ApplicationData* class properties to cache data locally.
 - D. No, you can only save settings locally.

Objective 5.2: Save and retrieve files from the file system

As you learned in the previous section, Windows 8 applications have various options for storing data and each one has its pros and cons. The right place to store information or to cache it depends on the application, and the scenario in which the user works. You can save application data in local, roaming, or temporary storage in the form of an application settings dictionary or in files. Storing information in files let you organize the data as you want: You can represent complex structures or hierarchical data using, for example, XML, or you can store a comma-separated value (.csv) file with log information.

This section is dedicated to the different ways the application can leverage WinRT classes to store information in files and folders. You can access the file system to save and retrieve information by using a file picker, by accessing files programmatically, or by accessing the HomeGroup content.

This objective covers how to:

- Handle file streams
- Save and retrieve files by using the *StorageFile* and *StorageFolder* classes
- Set file extensions and associations
- Save and retrieve files by using file pickers and the folder picker
- Compress files to save space
- Access libraries, for example, pictures, documents, and videos

Using file pickers to save and retrieve files

Accessing files through pickers enables the user to choose files and folders explicitly during the process. Simply put, the code asks the user to choose a file or a folder to work on, giving control to the default Windows 8 user interface. When the user chooses a file or folder, the Windows Runtime gives control to the calling code to perform one or more operations on the item chosen by the user. In practice, the developer can use an easy and standard way to let the user choose the location to perform the input/output operation the code will work with.

Many apps work with individual files or folders (or small lists of files in a folder), so file pickers are often the best choice for providing a unified interface for users to work with file system resources. By using pickers, you can also minimize the application's capability declarations and simplify the process to certify your application for the Windows Store. For example, Figure 5-1 shows an application that asks the user to choose a photo from his Pictures library using the file picker. The user can select images, sort them, navigate back, and open one of them.

You can use the *FileOpenPicker* class to gain access to files. The methods to work with resources are asynchronous and follow the *async/await* pattern.

The file picker has different areas on the screen:

- Upper left, which is the current location
- Below upper left, which is the command zone with the *Up* command that goes to the previous folder, and the type of sorting to choose how to sort images
- Drop-down list of locations the user can use to browse for files
- An area at the center to select one file or multiple files

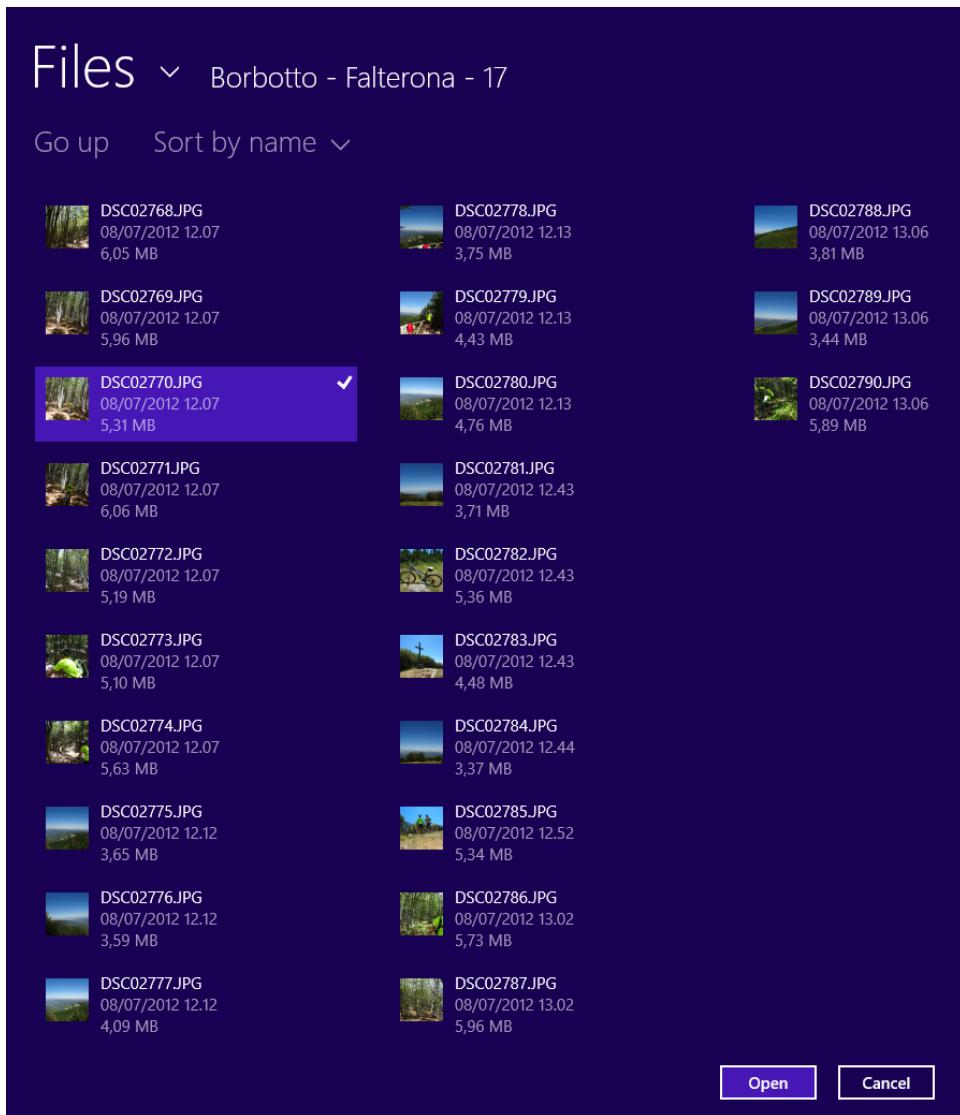


FIGURE 5-1 A file picker enables a user to choose files

The drop-down list also presents some system locations, like the Music library or the Downloads folder.

The number of available locations list depends on the user profile. For example, if the user has a SkyDrive account, the picker shows the SkyDrive of the user as a possible selection for the libraries. If the user installed applications that registered themselves as pickers (through the Picker contract), they are visible in the drop-down list. Figure 5-2 shows that the user can choose the Bing application in the provided drop-down list and then a photo from a standard picker.

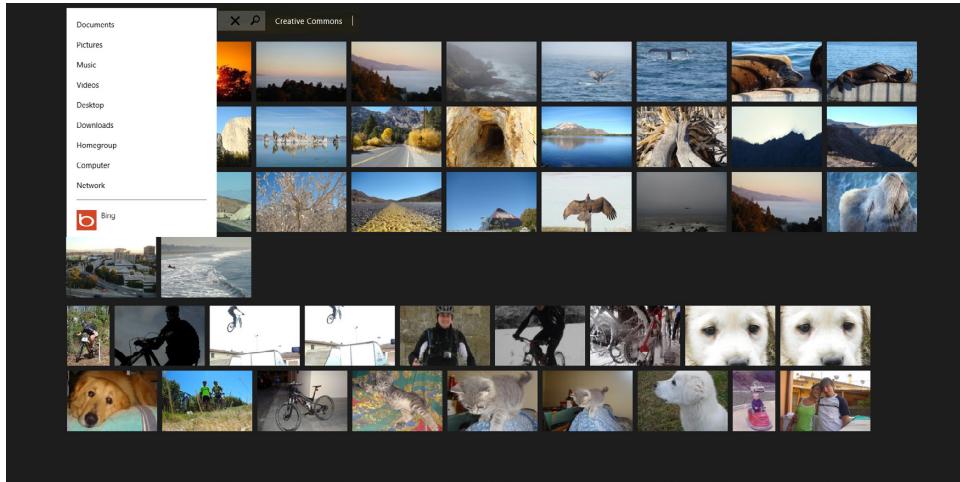


FIGURE 5-2 Example of a file picker

To activate the file picker, you can use a new instance of the *FileOpenPicker* class, asking it to select a single file or multiple files and letting the picker know which kind of files to select. For example, in the following code, the main page contains a simple button that opens the file picker for JPEG images. See the JavaScript code in Listing 5-9.

LISTING 5-9 Using the *FilePicker* class

```
var filePicker = new Windows.Storage.Pickers.FileOpenPicker();
filePicker.viewMode = Windows.Storage.Pickers.PickerViewMode.thumbnail;
filePicker.suggestedStartLocation =
    Windows.Storage.Pickers.PickerLocationId.picturesLibrary;
filePicker.fileTypeFilter.replaceAll([".jpg", ".jpeg"]);
filePicker.pickSingleFileAsync().then(function (file) {
    if (file) {
        // the picked file is available for read/write access
    } else {
        // No file selected by the user (Cancel)
    }
});
```

The *FileTypeFilter* collection cannot be empty, so you have to explicitly set file extensions (each one begins with a period).

To clear the filter or to add every file extension, use the following code:

```
picker.fileTypeFilter.Clear();
picker.fileTypeFilter.Add("*");
```

The `PickSingleFileAsync` follows the `async` pattern. It is `awaitable`, so you can manage user selection awaiting without blocking the user interface thread, as you can see from the use of the `then` promise.

The result can be `null` if the user does not select a file, or it can be the file in the form of a `StorageFile` class of the `Windows.Storage` library.

There are no differences between choosing a file from the file system or from other applications. They are always returned as `StorageFile` objects.

The complete flow is represented by Figure 5-3. The application calls the file picker to enable the user to pick a file or a list of files to open. The system invokes the picker that will display the user interface and present all the available system folders and applications able to provide files. If the user chooses a file, the picker returns the file immediately. If the user selects a file picker application, the system activates the application (Bing in Figure 5-2) that will provide its own user interface in the form of a page that the picker presents to select the file.

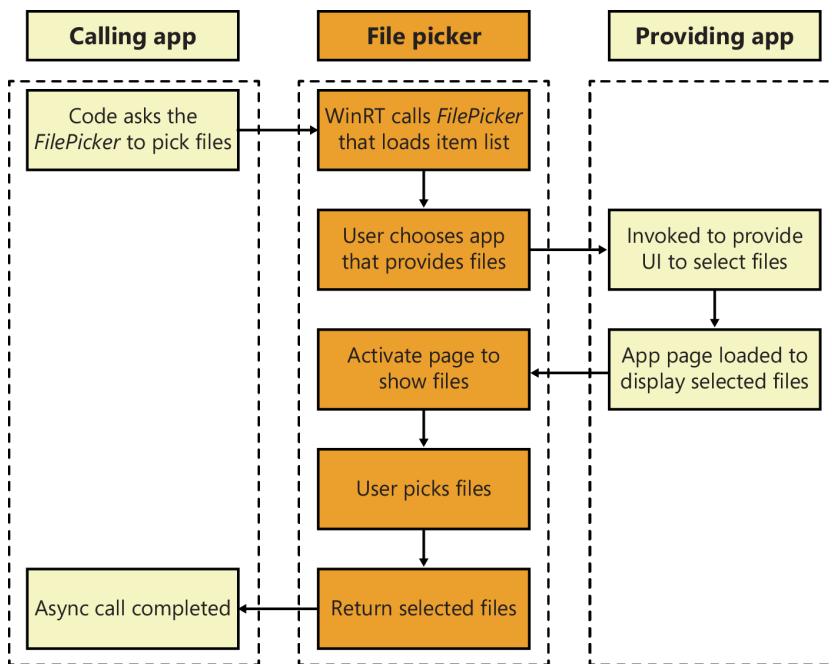


FIGURE 5-3 File pickers flow and contracts

The code can ask the *FileOpenPicker* to select different file types and provide a starting point to the user as well as the type of suggested view. The following code assumes a paragraph on the page called *chosenFile*:

```
var filePicker = new FileOpenPicker();
filePicker.viewMode = PickerViewMode.thumbnail;
filePicker.suggestedStartLocation = PickerLocationId.picturesLibrary;
filePicker.fileTypeFilter.insert(".jpg");
filePicker.fileTypeFilter.insert(".jpeg");
filePicker.fileTypeFilter.insert(".png");

filePicker.pickSingleFileAsync().then(function (file) {
    if (file) {
        // the picked file is available for read/write access
    } else {
        // No file selected by the user (Cancel)
    }
});
```

The code can also request multiple files obtaining a collection of a *Windows.Storage.StorageFile* instances:

```
var filePicker = new FileOpenPicker();
filePicker.viewMode = PickerViewMode.thumbnail;
filePicker.suggestedStartLocation = PickerLocationId.picturesLibrary;
filePicker.fileTypeFilter.insert(".jpg");
filePicker.fileTypeFilter.insert(".jpeg");
filePicker.fileTypeFilter.insert(".png");

filePicker.pickMultipleFilesAsync().then(function (fileCollection) {
    if (fileCollection.size > 0) {
        // The user select at least one file
        var output = "Files selected:<p>";
        for (var i = 0; i < fileCollection.size; i++) {
            output = output + "<p>" + files[i].name + "</p>";
        }
        chosenFile.innerHTML = outputString;
    } else {
        // no file selected
    }
});
```

Use the *FolderPicker* using the same pattern to gain access to folders.

If you try to show a file or folder picker when the application is snapped, the picker will not be shown, and the system will throw an exception that you can catch to inform the user.

The result is shown in Figure 5-4, where the application named “Learn with Wild Animals” occupies the bigger portion of the screen, and the sample picker application appears in the snapped view.

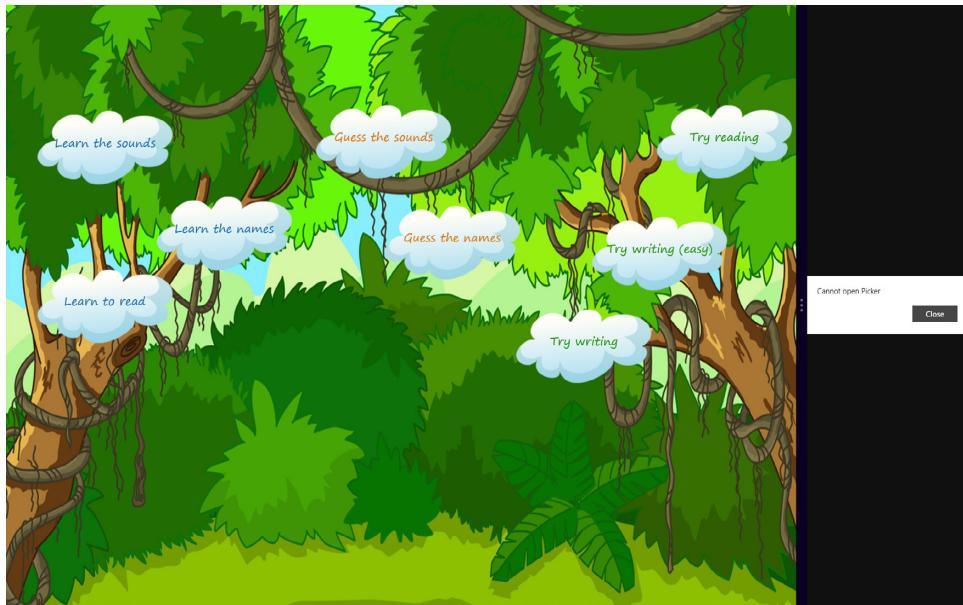


FIGURE 5-4 Snapped application displaying the error during the exception for trying to open the file picker in snapped view

Although this method works well from a developer perspective, it would be better to inform the user before trying to open the picker, or unsnap the application from code before opening the picker.

For example, you can use the following code to try to unsnap the application automatically:

```
var currentState = Windows.UI.ViewManagement.ApplicationView.value;
if (currentState === Windows.UI.ViewManagement.ApplicationViewState.snapped &&
    !Windows.UI.ViewManagement.ApplicationView.tryUnsnap()) {
    // Inform the user we cannot unsnap automatically
    return;
}
```

You can also save files using pickers. The first thing to do is define and set the *FileSavePicker* instance:

```
var savePicker = new Windows.Storage.Pickers.FileSavePicker();
savePicker.SuggestedStartLocation =
    Windows.Storage.Pickers.PickerLocationId.documentsLibrary;
savePicker.fileTypeChoices.insert("Text", [".txt"]);
savePicker.suggestedFileName = "DevLeap Description";
```

Then you can call the *PickSaveFileAsync* method to retrieve the file in which the user wants to save content:

```
savePicker.pickSaveFileAsync().then(function(file) {...});
```

If the user chooses a file, the resulting variable *file* will not be *null*. Before calling the *WriteTextAsync* method of the *FileIO* class, remember to prevent updates to the remote version of the file until the change is completed. The *CachedFileManager* class is very useful to perform this kind of lock:

```
savePicker.pickSaveFileAsync().then(function (file) {
if (file) {
    Windows.Storage.CachedFileManager.deferUpdates(file);
    Windows.Storage.FileIO.writeTextAsync(file, file.name).done(
        function () {
            Windows.Storage.CachedFileManager.completeUpdatesAsync(file).done(
                function (updateStatus) {
                    if (updateStatus ==
                        Windows.Storage.Provider.FileUpdateStatus.complete) {
                        // Everything fine
                    } else {
                        // Something wrong
                    }
                });
        });
} else {
    // Operation cancelled by the user
}
});
```

The code can also access HomeGroup content using the *SuggestedStartLocation* property:

```
var filePicker = new Windows.Storage.Pickers.FileOpenPicker();
filePicker.viewMode = Windows.Storage.Pickers.PickerViewMode.thumbnail;
filePicker.suggestedStartLocation = Windows.Storage.Pickers.PickerLocationId.homeGroup;
filePicker.fileTypeFilter.clear();
filePicker.fileTypeFilter.insert("*");
```

Accessing files and data programmatically

Application code can access files and folders in locations such as libraries, devices, and network paths, and can query the file system for files and folders matching the search expression. Before using the code, a Windows Store app must explicitly set the permission to use user libraries in the application manifest. Figure 5-5 shows the application manifest with all the user libraries selected in the capability set.

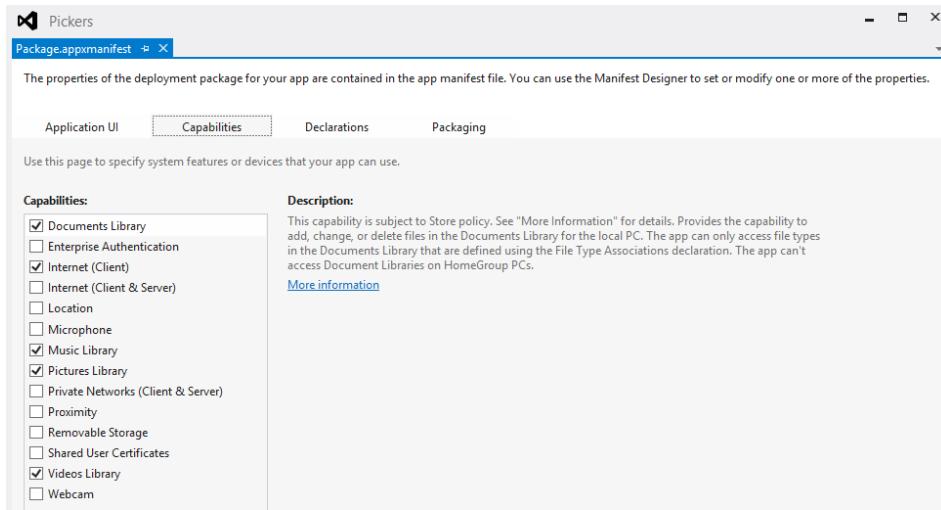


FIGURE 5-5 Application manifest file and folder-related capabilities

After setting the capabilities, in the Pictures library for example, the code can call the *StorageFolder* class method to work with files. For example, the following code gets a collection of all files in the user's Pictures library:

```
function getAllUserPictures() {
    var picturesFolder = Windows.Storage.KnownFolders.picturesLibrary;
    picturesLibrary.getItemsAsync().then(function (items) {
        items.forEach(function (item) {
            // You can test item type for folder or file
            if (item.isOfType(Windows.Storage.StorageItemTypes.folder)) {
                // This is a subfolder: use item.name for its name;
            }
            else {
                // This is a file: use item.fileName for its name;
            }
        });
    });
}
```

You can use the *GetItemsAsync* to retrieve the list of both files and folders.

To query the system for particular files and group them by one of their properties (for example, the year when pictures were captured), use the *CreateFolderQuery* method of the *StorageFolder* class:

```
var picturesFolder = Windows.Storage.KnownFolders.picturesLibrary;

var queryResult = picturesFolder.createFolderQuery(
    Windows.Storage.Search.CommonFolderQuery.groupByYear);
```

Then you can get the query result using the *GetFoldersAsync* method and iterate through each group to obtain the reference to the internal items using the *GetFileAsync* method:

```
query.getFoldersAsync().then(function (yearList) {
    yearList.forEach(function (year) {
        year.getFileAsync().then(function item) {
```

Working with files, folders, and streams

After obtaining a reference to a file or folder, you can perform read and write operations on it. Remember that you need to define the explained capabilities to perform operations on system folders.

For example, you can create a file in a folder using the following code:

```
var folder = Windows.Storage.KnownFolders.documentsLibrary;
folder.createFileAsync("log.txt",
    Windows.Storage.CreationCollisionOption.replaceExisting).then(function (logFile) {
    // logFile represents the StorageFile class
});
```

To write some text to the file, use the *WriteTextAsync* method as in the following code sample:

```
Windows.Storage.FileIO.writeTextAsync(logFile, "Operation Logged");
```

To write bytes to a file, you can use the following code:

```
var buffer = Windows.Security.Cryptography.CryptographicBuffer
.convertStringToBinary(
    "DevLeap is a group of professionals . . .",
    Windows.Security.Cryptography.BinaryStringEncoding.utf8);

var folder = Windows.Storage.KnownFolders.documentsLibrary;
folder.createFileAsync("Document.txt",
    Windows.Storage.CreationCollisionOption.replaceExisting).then(function (doc) {
    // doc represents the StorageFile class
    Windows.Storage.FileIO.writeBufferAsync(doc, buffer);
});
```

The code converts the string "DevLeap is a group of professionals . . ." to a binary object and then writes the obtained buffer to the file named Document.txt in the *DocumentsLibrary* user folder.

You can also use streams to write text to a file, as follows:

```
var folder = Windows.Storage.KnownFolders.documentsLibrary;
folder.createFileAsync("Document.txt",
    Windows.Storage.CreationCollisionOption.replaceExisting).then(function (doc) {
    // doc represents the StorageFile class
    doc.openTransactedWriteAsync().then(writeTheStreamToDoc);
});

function writeTheStreamToDoc(tx) {
    var dataWriter = new Windows.Storage.Streams.DataWriter(tx.stream);
    dataWriter.writeString("DevLeap is a group of professionals . . .");
    dataWriter.storeAsync().then(function () {
        tx.commitAsync().done(function () {
            // Text in the stream saved
            tx.close();
        });
    });
}
```

The code creates the file named Document.txt, opens the stream to the file by calling the *OpenTransactedWriteAsync* method, and returns the stream of the file's content when the transacted operation (class *StorageStreamTransaction*) completes. The promise function uses the *DataWriter* class to write a string on the stream.

Remember to call the *StoreAsync* method of the *DataWriter* class and the *CommitAsync* method of the *StorageStreamTransaction* to save to the file permanently and then to close the stream.



EXAM TIP

Remember to catch any exception using a *try/catch* block or *then* and *done* promises during input/output (I/O) operations.

To read some data from a file, the methods are straightforward:

```
var folder = Windows.Storage.KnownFolders.documentsLibrary;
folder.getFileAsync("Document.txt").then(function (doc)
{
    Windows.Storage.FileIO.readTextAsync(doc).then(function (content){
        // the content variable contains the text read from the file
    });
});
```

To read bytes from a file, you can use the *ReadBufferAsync* method:

```
var folder = Windows.Storage.KnownFolders.documentsLibrary;
folder.getFileAsync("Document.txt").then(function (doc)
{
    Windows.Storage.FileIO.readBufferAsync(doc).then(function (buffer){
        // the content variable contains the text read from the file
        var dataReader = Windows.Storage.Streams.DataReader.fromBuffer(buffer);
        var text = dataReader.readString(buffer.length);
    });
});
```

To read a stream from a file, use the following code:

```
var folder = Windows.Storage.KnownFolders.documentsLibrary;
folder.getFileAsync("Document.txt").then(function (doc)
{
    doc.openAsync(Windows.Storage.FileAccessMode.readWrite).then(readTheStreamFromDoc);
});
function readTheStreamFromDoc (stream) {
    var dataReader = new Windows.Storage.Streams.DataReader(stream);
    dataReader.loadAsync(stream.size).done(function (bytes) {
        var text = dataReader.readString(bytes);
        // Process the variable text that represents the content
        dataReader.close();
    });
}
```

Setting file extensions and associations

Windows enables an app to register to become the default handler for a certain file type. If the user chooses your app as the default handler for a certain file type, your app is activated every time that type of file is launched.

You should register for a file type only if you expect to handle all file launches for that type of file. If your app needs to use the file type only internally, you do not need to register to be the default handler. If you choose to register for a file type, it is important that you provide the user with the functionality that is expected when your app is activated for that file type. For example, a picture viewer app can register to display a .jpg file.

The first thing to do is declare the functionality in the application manifest, as shown in Figure 5-6.

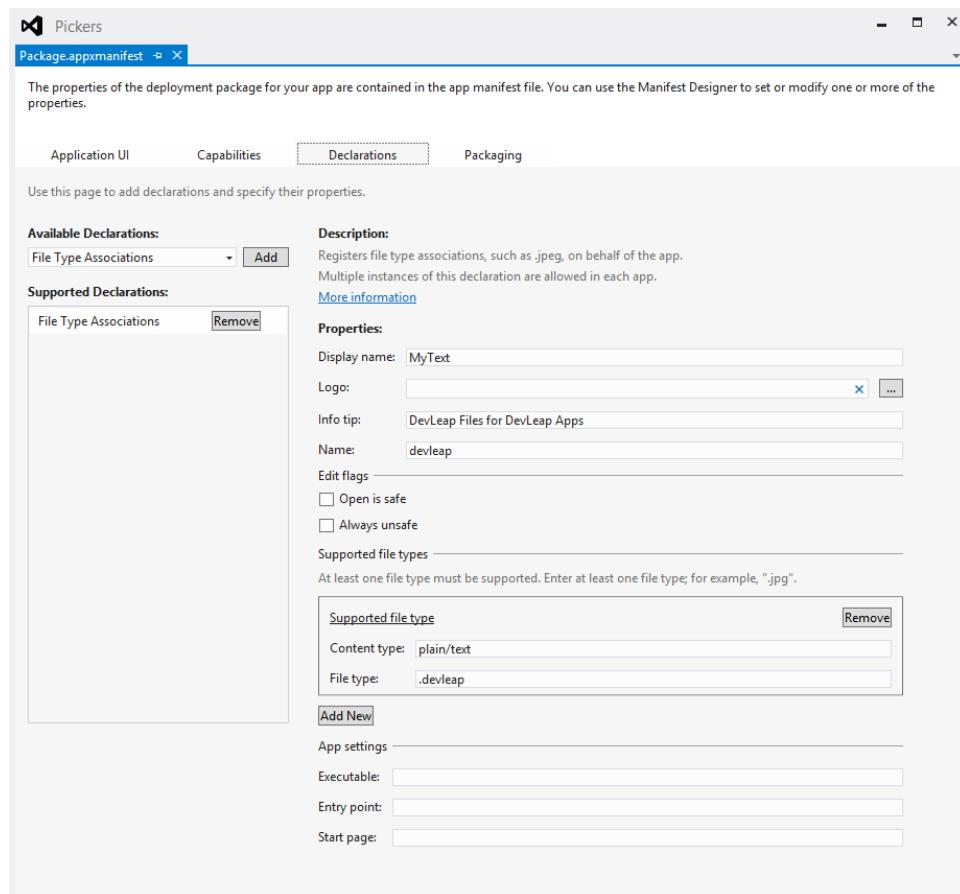


FIGURE 5-6 Application manifest with declarations for file associations and extensions

The corresponding XML settings are displayed in Listing 5-10.

LISTING 5-10 Application manifest with declarations for file associations and extensions

```
<?xml version="1.0" encoding="utf-8"?>
<Package xmlns="http://schemas.microsoft.com/appx/2010/manifest">
    <Identity Name="f512a0c2-b668-4a46-a026-b9090e60717b" Publisher="CN=Roberto"
        Version="1.0.0.0" />
    <Properties>
        <DisplayName>Pickers</DisplayName>
        <PublisherDisplayName>Roberto</PublisherDisplayName>
        <Logo>Assets\StoreLogo.png</Logo>
    </Properties>
    <Prerequisites>
        <OSMinVersion>6.2.1</OSMinVersion>
        <OSMaxVersionTested>6.2.1</OSMaxVersionTested>
    </Prerequisites>
    <Resources>
        <Resource Language="x-generate" />
    </Resources>
    <Applications>
        <Application Id="App" Executable="$targetnametoken$.exe" EntryPoint="Pickers.App">
            <VisualElements DisplayName="Pickers" Logo="Assets\Logo.png"
                SmallLogo="Assets\SmallLogo.png" Description="Pickers" ForegroundText="light"
                BackgroundColor="##464646">
                <DefaultTile ShowName="allLogos" />
                <SplashScreen Image="Assets\SplashScreen.png" />
            </VisualElements>
            <Extensions>
                <Extension Category="windows.fileTypeAssociation">
                    <FileTypeAssociation Name="devleap">
                        <DisplayName>MyText</DisplayName>
                        <InfoTip>DevLeap Files for DevLeap Apps</InfoTip>
                        <SupportedFileTypes>
                            <FileType ContentType="plain/text">.devleap</FileType>
                        </SupportedFileTypes>
                    </FileTypeAssociation>
                </Extension>
            </Extensions>
        </Application>
    </Applications>
    <Capabilities>
        <Capability Name="documentsLibrary" />
        <Capability Name="picturesLibrary" />
        <Capability Name="videosLibrary" />
        <Capability Name="internetClient" />
    </Capabilities>
</Package>
```

You need to provide the icon that the system will display on these particular custom files. To do this, add the following images to the application package. The icons must match the look of the app tile logo:

- Icon.targetsize-16.png
- Icon.targetsize-32.png
- Icon.targetsize-48.png

- Icon.targetsize-256.png
- smallTile-targetsize-16.png
- smallTile-targetsize-32.png
- smallTile-targetsize-48.png
- smallTile-targetsize-256.png

TIP TESTING IMAGES

Test all the images on a white background.

Implement the *OnFileActivated* method of the application overriding the base class method as in the following code:

Sample of JavaScript code

```
WinJS.Application.addEventListener("activated", onActivate, false);
function onActivate (eventArgs) {
    if (eventArgs.detail.kind ===
        Windows.ApplicationModel.Activation.ActivationKind.file)
    {
        // eventArgs.detail.files represents the collection of StorageFile
    }
}
```

Compressing files to save space

The *System.IO.Compression.FileSystem* assembly cannot be referenced from a Windows Store apps. You cannot compress and decompress files using the classic *ZipArchive* class. To save disk space, you can compress and decompress files by using the *GZipStream* class or the *DeflateStream* class. You can also leverage the *Compressor* and *Decompressor* classes.

You can use the following method to build your own compression and decompression utility:

```
var picker = new Windows.Storage.Pickers.FileOpenPicker;
picker.fileTypeFilter.append("*");
picker.pickSingleFileAsync().then(function (file) {
    try {
        if (file === null) {
            // No file selected by the user
        }
        var compressor;
        var algorithmName = getEnumerationValueName(
            Windows.Storage.Compression.CompressAlgorithm,
            compressAlgorithm, "default");
        var compressed = file.name + ".compressed";
        compressor = new WinJS.Compressor(compressed);
        compressor.compressAsync(file).then(function () {
            // Completed
            compressor.close();
        });
    }
});
```

```
    }, DisplayError);
  catch (e) {
    // Something Wrong
    DisplayError(e);
}
```

The same concept applies to the decompression. Use the following code as a reference:

```
var picker = new Windows.Storage.Pickers.FileOpenPicker;
picker.fileTypeFilter.append("*");
picker.pickSingleFileAsync().then(function (file) {
try {
  var decompressor = new WinJS.Decompressor(file.name);
  return decompressor.copyAsync(file.name + ".decompressed").then(function () {
    decompressor.close();
  }, DisplayError);
} catch (e) {
  DisplayError (e);
}
});
```



Thought experiment

Working with files

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

Your application needs to store a log file locally to store exception information.

Which kind of data access technique do you plan to use and why?

Objective summary

- Accessing files through pickers enables the user to choose files and folders explicitly during the process. The contract regulates data interchange from app to app.
- You can access files and folders programmatically in locations such as libraries, devices, and network paths.
- The code can query the file system for files and folders matching a search expression.
- After obtaining a reference to a file or folder, you can perform read and write operations on it as a stream.
- Windows allows an app to register to become the default handler for a certain file type.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the “Answers” section at the end of this chapter.

1. Which type of storage can you access with file pickers? (Choose all that apply.)
 - A. Local storage
 - B. Cloud storage on SkyDrive
 - C. Network storage
 - D. Windows Azure storage account
2. Is it possible to set an application as a file handler?
 - A. Yes, by declaring it in the application manifest.
 - B. Yes, but only for apps written in C# or VB.
 - C. Yes, but only for apps written in C++.
 - D. No, it is not possible.
3. Which class do you use to read a file programmatically?
 - A. *FileOpenPicker*
 - B. *StorageFolder*
 - C. *StorageFile*
 - D. *ApplicationData*

Objective 5.3: Secure application data

Communications over public networks, such as the Internet, can be easily intercepted by unauthorized third parties, who might be interested in reading or tampering with the content transmitted through the communication channel. You can use cryptography to protect data and communications over otherwise unsafe channels. In this objective, you learn about the most fundamental types and methods that help secure a Windows Store app.

NOTE .NET FRAMEWORK SECURITY

Discussing security, cryptography, or certificate management in the .NET Framework in detail is beyond the scope of this objective and is not covered on the exam. The purpose of this objective is to illustrate the most important classes and methods you can leverage in a Windows Store app to secure data and communications.

This objective covers how to:

- Encrypt data by using the *Windows.Security.Cryptography* namespace
- Enroll and request certificates
- Encrypt data by using certificates

Introducing the *Windows.Security.Cryptography* namespaces

The *Windows.Security.Cryptography* namespaces include types and methods that can be used for secure encoding and decoding of data, hashing, random numbers generation, conversions between byte arrays and buffers, and message authentication. These new libraries replace the .NET libraries included under the *System.Security.Cryptography* namespace, which performs the same operations in the classic .NET world.

Cryptography is used to achieve the following goals:

- **Authentication** Cryptography can be used to ensure that the communication actually originates from a certain source.
- **Confidentiality** Cryptography helps prevent data being read by unauthorized third parties; confidentiality is ensured by encryption algorithms.
- **Data integrity** Cryptography protects data from being tampered with by unauthorized parties.
- **Non-repudiation** As a consequence of authentication, confidentiality, and integrity, cryptography also prevents other parties from repudiating their own messages after they have been sent.

It is worth noticing that the Bureau of Industry and Security in the United States Department of Commerce regulates the export of technology that uses certain types of encryption. All apps listed in the Windows Store must comply with these laws and regulations because the app files can be stored in the United States.

MORE INFO EXPORT RESTRICTIONS

Further information on export restrictions on cryptography can be found at <http://msdn.microsoft.com/en-us/library/windows/apps/hh694069.aspx>.

Using hash algorithms

According to the Microsoft official documentation definition, “Hash algorithms transform binary values of arbitrary length to smaller binary values of a fixed length, known as hash values. A hash value is a numerical representation of a piece of data.” In other words, when you

apply a hash algorithm to a plain text message, you get a binary code known as a *message digest*, or just *digest*, representing the “fingerprint” that identifies the message content.

In fact, even the slightest change in the content of the message produces a different hash code. For example, if a single bit of a message is changed, a strong hash function might produce an output that differs by 50 percent. For this reason, hash algorithms can help protect message integrity. To ensure that a certain message has not been tampered with, you can send a message along with its corresponding hash code. The receiver applies the same algorithm used by the sender to the message to produce a new hash code and then compares the two hash codes to verify the integrity of the message.



EXAM TIP

Hash algorithms do not prevent someone from reading hashed content because messages are transmitted in plain text. Full security typically also requires digital signatures and encryption. (See the next sections for further details.)

Given the digest, you cannot go back to the original document. Hash algorithms are one-way algorithms.

Besides message authentication, cryptographic hash algorithms have many applications, such as indexing data in hash tables, uniquely identifying data and detecting data corruption.

To perform hash operations, you can leverage the *HashAlgorithmProvider* class, as shown in Listing 5-11.

LISTING 5-11 Hashing a plain text message through the *HashAlgorithmProvider* class

```
function hashMessage_click() {
    var message = "Plain text message to hash";
    var digestText = document.getElementById("digest");
    var cb = Windows.Security.Cryptography.CryptographicBuffer;

    var algorithmName = Windows.Security.Cryptography.Core
        .HashAlgorithmNames.sha512;

    var binaryMessage = cb.convertStringToBinary(
        message,
        Windows.Security.Cryptography.BinaryStringEncoding.utf8);

    var hashProvider = Windows.Security.Cryptography.Core
        .HashAlgorithmProvider.openAlgorithm(algorithmName);

    var digest = hashProvider.hashData(binaryMessage);

    if (digest.length != hashProvider.hashLength) {
        digestText
            .innerHTML = "<p>There was an error creating the hash</p>";
        return;
    }
    digestText.innerHTML = "<p>" + cb.encodeToBase64String(digest) + "</p>";
}
```

First, convert the original text message in its binary representation by calling the *ConvertStringToBinary* method, which is one of several methods provided by the *CryptographicBuffer* class to perform conversions between the base types involved in cryptographic operations (such as the *EncodeToString/DecodeToString* and the *EncodeToHexString/DecodeToHexString* methods). Second, choose the hash algorithm to use to produce the hash code. You can enumerate all the supported hash algorithms making use of the *HashAlgorithmNames* static class, as shown in Listing 5-11.

NOTE WINRT-SUPPORTED HASH ALGORITHMS

The first version of the Windows Runtime supports the MD5, SHA1, SHA256, SHA384, and SHA512 hash algorithms.

The next step is to get a reference to the hash algorithm provider, which encapsulates all the details of the inner mechanisms, by calling the *OpenAlgorithm* method and passing the name of the chosen algorithm (SHA512, in our sample). After you obtain the reference, you can hash the message simply by invoking the *HashData* method. The subsequent *if* statement checks that the length of the produced hash matches the length specified for the algorithm used for hashing (indicated by the *HashLength* property of the *HashAlgorithmProvider* instance).

Do not forget to add an event listener for the button click event, as shown in the following code snippet:

```
app.onloaded = function () {
    document.getElementById("btnHashMessage")
        .addEventListener("click", hashMessage_click);
};
```

To test this function, you can use the HTML code shown in Listing 5-12 as a reference for your default.html page.

LISTING 5-12 The HTML definition of the default.html file

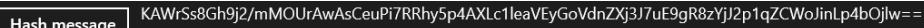
```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Crypto Sample</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- Crypto Sample references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
```

```
<button id="btnHashMessage">Hash message</button>
<div id="digest"></div>
</body>
</html>
```

Figure 5-7 shows the result of the hashing operation.



```
KAWrSs8Gh9j2/mMOUrAwAsCePi7RRhy5p4AXLc1leaVEyGoVdnZXj3J7uE9gR8zYj2p1qZCWoJinLp4bOjlw==
```

FIGURE 5-7 The generated hash code

To check whether two hash codes are the same (and therefore the message content has not been tampered with), you can leverage the *Compare* method exposed by the *CryptographicBuffer* class, which returns *true* when the two compared hashes match. The JavaScript code in Listing 5-13 shows how to check hash codes (changes are in bold).

LISTING 5-13 Comparing whether two hash codes match

```
function hashMessage_click() {
    var message = "Plain text message to hash";
    (code omitted)

    var reusableHash = hashProvider.createHash();
    reusableHash.append(binaryMessage);

    var otherDigest = reusableHash.getValueAndReset();

    if (!cb.compare(digest, otherDigest)) {
        digestText.innerHTML =
            "<p>CryptographicHash failed to generate the same hash data!</p>";

        return;
    }
    digestText.innerHTML = "<p>" + cb.encodeToBase64String(digest) + "</p>";
}
```

The *CreateHash* method of the *HashAlgorithmProvider* instance enables creation of a reusable hash container to hash the data through multiple calls, represented by the *CryptographicHash* class. You can pass the content that needs to be hashed via a call to the *Append* method, which accepts a buffer of data (in the form of a *Windows.Storage.Streams.IBuffer* object) as parameter and returns an instance of the *CryptographicHash* class.



EXAM TIP

After the hash code has been generated through a call to the *CreateHash* method, the hashed text can be retrieved through the *GetValueAndReset* method, which also cleans the *CryptographicHash* object after having retrieved the hash code. At this point, you can compare the two hash codes to check whether they match by calling the *Compare* method of the *CryptographicBuffer* class.

Generating random numbers and data

Random number generation represents an important step in many cryptographic operations, such as in the generation of cryptographic keys or passwords. The *CryptographicBuffer* class enables you to generate random numbers easily by using the *GenerateRandomNumber* method. The following code excerpt shows an example of its usage:

```
function generateRandomNumber_click(args) {
    var rnd = Windows.Security.Cryptography.CryptographicBuffer.generateRandomNumber();

    document.getElementById("randomNumber").innerHTML = rnd.toString();
}
```

The *GenerateRandomNumber* method returns a random 32-bit unsigned integer. (Remember, though, that this type is not compliant with the Common Language Specification.) You can use the unsigned integer to perform further cryptographic operations. You will see an example of its usage in the next section, titled “Encrypting messages with MAC algorithms.”

Besides random number generation, the *CryptographicBuffer* class also provides a *GenerateRandom* method that, as its name suggests, can be used to generate a buffer of random data. In this case, the only thing you have to provide is the length of the buffer in bytes, as shown in the following code snippet:

```
function generateRandomData_click(args) {

    var cb = Windows.Security.Cryptography.CryptographicBuffer;
    var length = 32;

    var rndData = cb.generateRandom(length);

    document.getElementById("randomNumber").innerHTML = cb.encodeToString(rndData);
}
```

Finally, update the *app.onloaded* event handler to add two more event listeners, as shown in the following code:

```
app.onloaded = function () {
    document.getElementById("btnHashMessage")
        .addEventListener("click", hashMessage_click);
    document.getElementById("btnGenerateRandomNumber")
        .addEventListener("click", generateRandomNumber_click);
    document.getElementById("btnGenerateRandomData")
        .addEventListener("click", generateRandomData_click);
};
```

To test these two methods, just add the lines of code shown in bold in Listing 5-14 to the code illustrated in Listing 5-12.

LISTING 5-14 Updating the default.html page to display generated random numbers and data

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Crypto Sample</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- Crypto Sample references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <button id="btnHashMessage">Hash message</button>
    <div id="digest"></div>
    <button id="btnGenerateRandomNumber">Generate random number</button>
    <div id="randomNumber"></div>
    <button id="btnGenerateRandomData">Generate random data</button>
    <div id="randomData"></div>
</body>
</html>
```

Figure 5-8 shows a hash message, a randomly generated number, and some randomly generated data.

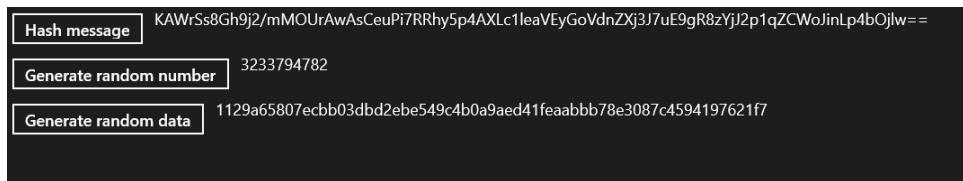


FIGURE 5-8 A hash, a random number, and random data

Encrypting messages with MAC algorithms

A special form of hash algorithms is represented by the message authentication code (MAC) algorithms, also known as *keyed hashing algorithms*. A MAC consists of a short set of data that is used not only to ensure the integrity of the transmitted message but also its authenticity. The integrity offered by a MAC algorithm depends on a secret key that both the sender and the receiver know (symmetric encryption). In the more common scenarios, the secret key is used in combination with a hash algorithm (hash-based message authentication code, or HMAC) to produce an encrypted hash code that can be decrypted only by those who possess the secret key. In other, less-common scenarios, the secret key is used to encrypt the whole message, and the last bits of the encrypted message are used as keyed hash code (even though no hash algorithms have been actually applied), whereas the rest of the data is discarded.

NOTE SYMMETRIC KEY ALGORITHMS

Symmetric key algorithms use the same cryptographic key for both encrypting and decrypting a message. The key, in practice, represents a shared secret between two or more subjects that can be used to encrypt and decrypt information that must be kept private. The fact that both parties must have access to the secret key represents one of the main drawbacks of symmetric key encryption, compared to public key encryption.

To encrypt a message by using a MAC algorithm, you can leverage the *MacAlgorithmProvider* class, which strictly resembles the *HashAlgorithmProvider* class illustrated in the preceding section. The main difference resides in the presence of a key that is used to encrypt the message, as shown in the JavaScript code in Listing 5-15.

LISTING 5-15 Encrypting a message using a MAC algorithm

```
(function () {
    "use strict";

    (code omitted)

    app.onloaded = function () {
        document.getElementById("btnGenerateMacSignature")
            .addEventListener("click", generateMacSignature_click);
        document.getElementById("btnVerifyMacSignature")
            .addEventListener("click", verifyMacSignature_click);
    };

    var macSignature;
    var message = "Plaintext message to sign";
    var key;

    function generateMacSignature_click(args)
    {
        var cb = Windows.Security.Cryptography.CryptographicBuffer;
        var macAlgorithmName = Windows.Security.Cryptography.Core
            .MacAlgorithmNames.hmacSha256;

        var macProvider = Windows.Security.Cryptography.Core
            .MacAlgorithmProvider.openAlgorithm(macAlgorithmName);
        key = cb.generateRandom(macProvider.macLength);
        var hmacKey = macProvider.createKey(key);

        var binaryMessage = cb.convertStringToBinary(
            message,
            Windows.Security.Cryptography.BinaryStringEncoding.utf8);

        macSignature = Windows.Security.Cryptography.Core
            .CryptographicEngine.sign(hmacKey, binaryMessage);

        document.getElementById("signature").innerHTML +=
            "<p>Signature: " + cb.encodeToString(macSignature) + "</p>";
    }
}
```

The first lines of code are quite similar to the hash sample illustrated in Listing 5-11. First, the code sets the name of the MAC algorithm you want to use by enumerating through the *MacAlgorithmNames* static class. It then obtains a reference to the MAC algorithm provider by calling the *OpenAlgorithm* method and passing the name of the algorithm to use (HmacSha256, in our sample).

NOTE MACALGORITHMPROVIDER

The *MacAlgorithmProvider* supports the following algorithm names: *AesCmac*, *HmacMd5*, *HmacSha1*, *HmacSha256*, *HmacSha384*, and *HmacSha512*.

The next step consists of creating the key that will be used to encrypt/decrypt the message. To generate the key, the code leverages the *GenerateRandom* method described in the “Generating random numbers and data” section to create a random buffer of data that is supplied to the *CreateKey* method of the MAC algorithm provider. The code then calls the *Sign* method exposed by the *CryptographicEngine* class to produce an encrypted hash code (signature) based on the provided key.

The authenticity of the keyed hash code can be verified by calling the *VerifySignature* method, which accepts the following parameters: the key used to encrypt the hash code, the message, and the signature to be verified. The method decrypts the signature using the provided key and then compares the hash codes. If they match, the recipient can be reasonably sure that the message comes from the sender, and it was not tampered with during transmission. Listing 5-16 shows an example of its usage.

LISTING 5-16 Verifying the authenticity of the secret key

```
function verifyMacSignature_click(args) {
    var cb = Windows.Security.Cryptography.CryptographicBuffer;

    var macAlgorithmName = Windows.Security.Cryptography.Core
        .MacAlgorithmNames.hmacSha256;
    var macProvider = Windows.Security.Cryptography.Core
        .MacAlgorithmProvider.openAlgorithm(macAlgorithmName);
    var hmacKey = macProvider.createKey(key);

    var binaryMessage = cb.convertStringToBinary(
        message,
        Windows.Security.Cryptography.BinaryStringEncoding.utf8);

    var isAuthenticated = Windows.Security.Cryptography.Core
        .CryptographicEngine.verifySignature(
            hmacKey,
            binaryMessage,
            macSignature);

    if (!isAuthenticated){
        document.getElementById("verification")
            .innerHTML += "The MAC signature does not match";
    }
}
```

```

        else
    {
        document.getElementById("verification")
            .innerHTML += "The MAC signature matches";
    }
}

```

The code starts with the same steps followed to sign the message, by obtaining a reference to the MAC algorithm provider and creating a *CryptographicKey* instance based on the same key used to encrypt the hash code. It then leverages the *VerifySignature* method to check whether the signature is valid by supplying it with the key, the message, and the associated signature.

To test this procedure, you can use the HTML code shown in Listing 5-17 as a reference.

LISTING 5-17 Complete definition of the default.html file used for this sample

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>MAC Signature Sample</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- MAC Signature Sample references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <button id="btnGenerateMacSignature">Generate MAC signature</button>
    <div id="signature"></div>
    <button id="btnVerifyMacSignature">Validate MAC signature</button>
    <div id="verification"></div>
</body>
</html>

```

If you run the app, generate a MAC signature, and then validate the generated signature, the result should resemble Figure 5-9.

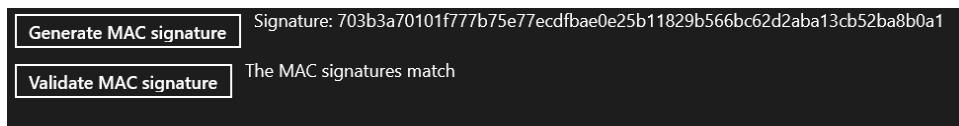


FIGURE 5-9 MAC signature generation and validation

Using digital signatures

Digital signatures are conceptually similar to message authentication codes. The difference is that, although the latter use a shared secret key (or symmetric key) to sign the message and prevent tampering, digital signatures use asymmetric keys to obtain the same result.

NOTE ASYMMETRIC ENCRYPTION

Asymmetric encryption uses a pair of keys mathematically linked to each other: a private key that must be kept secret and a public key that can be made publicly available. Data encrypted using the public key can be decrypted only with the private key, whereas data signed with the private key can be verified only with the public key. It is impossible to discover one of them without knowing the other.

To sign a message digitally, the sender first applies a hash algorithm to the message to create a digest; then the sender encrypts the digest with the private key to create a signature. Upon receiving the message and signature, the receiver decrypts the signature using the sender's public key to retrieve the digest, computes a new hash of the content, and compares the two hash codes. If the digests match, it means that the message comes from the owner of the private key and that the data has not been tampered with. Listing 5-18 shows an example of a digital signature's usage.

LISTING 5-18 Digitally signing a message

```
(function () {
    "use strict";

    (code omitted)

    app.onloaded = function () {
        document.getElementById("btnGenerateDigitalSignature")
            .attachEvent("onclick", generateDigitalSignature_click);
    };

    function generateDigitalSignature_click(args)
    {
        var messageBox = document.getElementById("digitalSignature");
        var keySize = 256;
        var message = "Plain text message to sign";

        var cb = Windows.Security.Cryptography.CryptographicBuffer;
        var asymmetricAlgorithmName = Windows.Security.Cryptography.Core
            .AsymmetricAlgorithmNames.ecdsaP256Sha256;
        var binaryMessage = cb.convertStringToBinary(
            message,
            Windows.Security.Cryptography.BinaryStringEncoding.utf8);
        var asymmetricKeyProvider = Windows.Security.Cryptography.Core
            .AsymmetricKeyAlgorithmProvider.openAlgorithm(asymmetricAlgorithmName);
```

```

var keyPair;
try {
    keyPair = asymmetricKeyProvider.createKeyPair(keySize);
}
catch (ex) {
    messageBox.innerHTML = "Invalid key size for the given algorithm";
    return;
}

var signature = Windows.Security.Cryptography.Core
    .CryptographicEngine.sign(keyPair, binaryMessage);

var publicKeyBuffer = keyPair.exportPublicKey();
var keyPairBuffer = keyPair.export();
var keyPublic = asymmetricKeyProvider
    .importPublicKey(publicKeyBuffer);

if (keyPublic.keySize != keyPair.keySize) {
    messageBox.innerHTML = "Importing public key failed";
    return;
}
keyPair = asymmetricKeyProvider.importKeyPair(keyPairBuffer);

if (keyPublic.keySize != keyPair.keySize) {
    messageBox.innerHTML = "Importing key pair failed";
    return;
}

var isAuthenticated = Windows.Security.Cryptography.Core.
    CryptographicEngine.verifySignature(keyPublic, binaryMessage, signature);

if (!isAuthenticated) {
    messageBox.innerHTML = "Signature verification failed!";
    return;
}
messageBox.innerHTML = "Signature was successfully verified: " +
    cb.encodeToBase64String(signature);
}
app.start();
})();

```

Compared to the code shown in Listing 5-15, the major difference resides in the use of a key pair to sign the message, instead of a shared secret key. To create a public/private key pair, the code uses the *CreateKeyPair* method of the *AsymmetricKeyProvider* class. In this case, the code does not use random data to generate the keys because, in asymmetric cryptography, the two keys are mathematically related.

NOTE ASYMMETRICKEYPROVIDER CLASS–SUPPORTED ALGORITHMS

The list of algorithms supported by the *AsymmetricKeyProvider* class is quite long. You can check the full list at <http://msdn.microsoft.com/en-us/library/windows/apps/windows.security.cryptography.core.asymmetricalgorithmnames.aspx>.

The private key is then used to sign the message digitally by calling the *CryptographicEngine.Sign* method (see Listings 5-15 and 5-18). After you create the encrypted signature with the provided keys, you can export the key pair or just the public portion of a public/private key pair into a buffer by calling, respectively, the *Export* and the *ExportPublicKey* methods of the *CryptographicKey* class (as shown in Listing 5-18 for illustrative purposes). Analogously, you can import a key pair or just the public key by leveraging the *ImportKeyPair* and the *ImportPublicKey* methods, respectively.

Finally, to verify the signature you can use the *VerifySignature* method, providing the imported public key, the message, and the digest encrypted using the private key.

You can use the HTML code in Listing 5-19 to test the flow.

LISTING 5-19 Complete HTML definition of the default.html file used to test the JavaScript code

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Digital Signature Sample</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- Digital Signature Sample references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <button id="btnGenerateDigitalSignature">Generate digital signature</button>
    <div id="digitalSignature"></div>
</body>
</html>
```

Figure 5-10 shows the digital signature generated by the code.

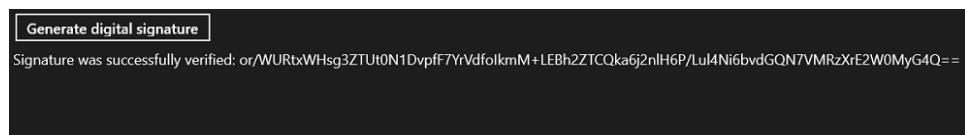


FIGURE 5-10 Digital signature

Enrolling and requesting certificates

As mentioned previously, asymmetric cryptography relies on a key pair made up by a public key and a private key. The key pair is used to encrypt and decrypt a message. Whereas the private key must be kept secret, the public key is usually embedded in a binary certificate.

A certificate is a signed data structure that binds a public key to a person, computer, or organization.

NOTE X.509 PUBLIC KEY INFRASTRUCTURE (PKI)

X.509 PKI is a standard for identifying the base requirements for using certificates. A certificate contains information about one subject, including the subject's public key. A certification authority (CA) issues certificates. All parties involved in a secure communication trust the CA and rely on it to verify the identities of the individuals, systems, or entities that represent the subject. The level of verification can be different based on the level of security required for a particular transaction. The following listing shows the definition of a X.509 certificate:

```
-----  
-- X.509 signed certificate  
-----  
  
SignedContent ::= SEQUENCE  
{  
    certificate      CertificateToBeSigned,  
    algorithm        Object Identifier,  
    signature        BITSTRING  
}  
  
-----  
-- X.509 certificate to be signed  
-----  
  
CertificateToBeSigned ::= SEQUENCE  
{  
    version          [0] CertificateVersion DEFAULT v1,  
    serialNumber     CertificateSerialNumber,  
    signature        AlgorithmIdentifier,  
    issuer           Name,  
    validity         Validity,  
    subject          Name,  
    subjectPublicKeyInfo SubjectPublicKeyInfo,  
    issuerUniqueIdentifier [1] IMPLICIT UniqueIdentifier OPTIONAL,  
    subjectUniqueIdentifier [2] IMPLICIT UniqueIdentifier OPTIONAL,  
    extensions       [3] Extensions OPTIONAL  
}
```

The `Windows.Security.Cryptography.Certificates` namespace contains types and methods that enable you to create certificate requests, as well as to install or import an issued certificate. Listing 5-20 shows the JavaScript code for creating a certificate request.

LISTING 5-20 Creating a certificate request

```
var request;
function createRequest() {

    try {
        var crp = Windows.Security.Cryptography.Certificates
            .CertificateRequestProperties();
        crp.friendlyName = "MyCertificate";
        crp.subject = "SampleCertificateRequest";
        crp.keyProtectionLevel = Windows.Security.Cryptography.Certificates
            .KeyProtectionLevel.noConsent;
        crp.keyUsages = Windows.Security.Cryptography.Certificates.
            EnrollKeyUsages.all;
        crp.exportable = Windows.Security.Cryptography.Certificates
            .ExportOption.exportable;
        crp.keySize = 2048;

        crp.keyStorageProviderName = Windows.Security.Cryptography.Certificates
            .KeyStorageProviderNames.softwareKeyStorageProvider;

        request = Windows.Security.Cryptography.Certificates
            .CertificateEnrollmentManager.createRequestAsync(crp).then(
            function (req) {
                document.getElementById("request").innerHTML = req;
                return request;
            },
            function (err) {
                document.getElementById("error")
                    .innerHTML = "createRequestAsync failed";
            });
    }
    catch (ex) {
        // handle the exception
    }
}
```

The first thing you must do is instantiate a `CertificateRequestProperties` object containing the properties of a certificate request. In particular, the `KeyProtectionLevel` enum indicates the level of protection and accepts one of the following values:

- **NoConsent** No strong key protection is required (this is the default value).
- **ConsentOnly** The user is notified through a dialog box when the private key is created or used.
- **ConsentWithPassword** The user is prompted to enter a password for the key when the key is created or when the key is used.

Figure 5-11 shows the dialog box asking for a password.

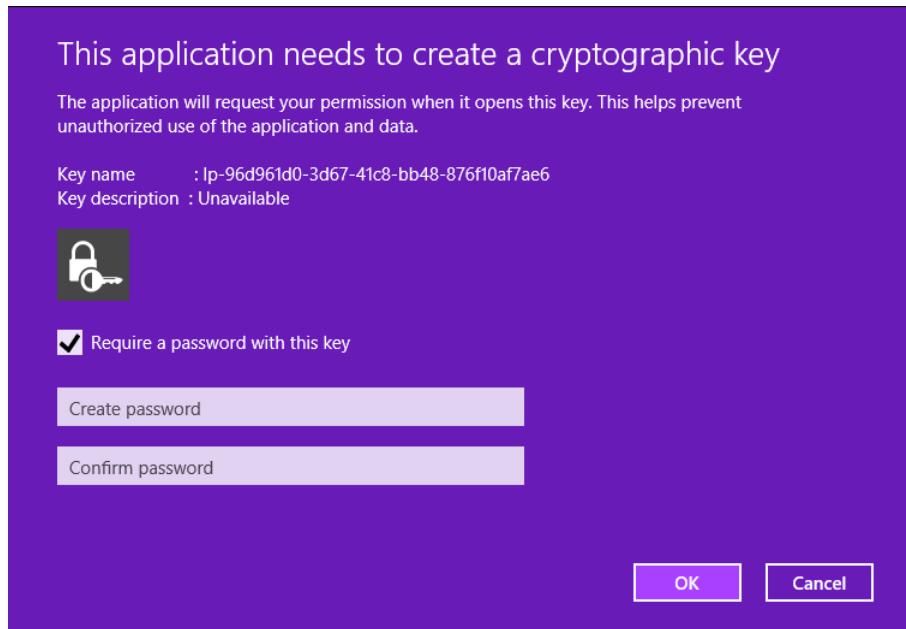


FIGURE 5-11 Dialog box requesting a password

The *KeyUsages* property (of type *EnrollKeyUsage*) indicates what kind of operation can be performed by the private key created for this certificate request. The possible values are:

- **None** No usage is specified for the key.
- **Decryption** The key can be used for decryption.
- **Signing** The key can be used for signing (this represents the default value).
- **KeyAgreement** The key can be used for secret agreement encryption.
- **All** The key can be used for decryption, signing, and key agreement.

The *Exportable* property (of type *ExportOption*) specifies whether the private key created for the request can be exported. (By default, the private key is not exportable for security reasons.) The *KeySize* property enables you to specify the size, in bits, of the private key to be generated. (For the RSA and DSA algorithms, the default value is 2,048 bits, whereas for elliptic curve cryptographic [ECC] algorithms, the key size will just be ignored.)

Another important property is represented by the *KeyStorageProviderName*, which indicates the key storage provider (KSP) that is used to generate the private key. There are three KSPs available for a Windows Store app:

- **PlatformKeyStorageProvider** Microsoft Platform Key Storage Provider
- **SmartcardKeyStorageProvider** Microsoft Smart Card Key Storage Provider
- **SoftwareKeyStorageProvider** Microsoft Software Key Storage Provider (the default value)

Finally, the *KeyAlgorithmName* property specifies the algorithm to be used to generate the public key. The default value is RSA.

The *CertificateEnrollmentManager* class also exposes an *InstallCertificateAsync* method that enables you to asynchronously install a certificate chain into the app container on the local computer. Listing 5-21 shows an example of its usage.

LISTING 5-21 Using the *InstallCertificateAsync* method to install a certificate

```
function installCertificate_click(args) {  
  
    var response = createRequest();  
  
    if (request === null || request === "") {  
        document.getElementById("error").innerHTML = "Create a request first";  
    }  
  
    try {  
        response = submitCertificateRequestAndGetResponse(request,  
            "http://www.contoso.org/");  
  
        if (response == null || response.length == 0) {  
            // Submit request succeeded but the returned response is empty.;  
        }  
  
        // Install the certificate  
        Windows.Security.Cryptography.Certificates.CertificateEnrollmentManager  
            .installCertificateAsync(  
                response,  
                Windows.Security.Cryptography.Certificates.InstallOptions.none).then(  
                    function () {  
                        document.getElementById("response")  
                            .innerHTML = "Response successfully installed";  
                    },  
                    function (err) {  
                        document.getElementById("response")  
                            .innerHTML = "Error installing the certificate";  
                    });  
    }  
    catch (ex) {  
        // handle the exception  
    }  
}  
  
function submitCertificateRequestAndGetResponse(request, uri) {  
    // Submit certificate request and get the response  
}
```

After creating a certificate request through the *CreateRequestAsync* method, the code invokes the *SubmitCertificateRequestAndGetResponseAsync* method, which basically creates an HTTP web request, submits the certificate request to the server indicated as second parameter, and then returns the certificate response coming from the server.

After the certificate is retrieved, the code calls the *InstallCertificateAsync* method and passes the encoded certificate as the first parameter, along with an *InstallOption* instance that specifies the certificate installation option (none, in this sample).

In Windows, issued certificates and pending or rejected requests to local computers and devices are stored in the Microsoft certificate store, which consists of the following logical stores:

- **Personal** Contains certificates associated with a private key controlled by the user or computer
- **Trusted Root Certification Authorities** Contains certificates from implicitly trusted CAs
- **Enterprise Trust** Contains certificate trust lists typically used to trust self-signed certificates from other organizations
- **Intermediate Certification Authorities** Contains certificates issued to subordinate CAs in the certification hierarchy
- **Active Directory User Object** Contains the user-object certificate or certificates published in Active Directory
- **Trusted Publishers** Contains certificates from trusted CAs
- **Untrusted Certificates** Contains certificates that have been explicitly identified as untrusted
- **Third-Party Root Certification Authorities** Contains trusted root certificates from CAs outside the internal certificate hierarchy
- **Trusted People** Contains certificates issued to users or entities that have been explicitly trusted
- **Other People** Contains certificates issued to users or entities that have been implicitly trusted
- **Certificate Enrollment Requests** Contains pending or rejected certificate requests

Certificates are normally stored in per-user, per-app container locations. A Windows Store app has write access to only its own certificate storage, and read access to only the certificates added by the app itself. An app can also have the right to read the local machine certificate stores. If the application adds certificates to any of its stores, these certificates cannot be read by other Windows Store apps. Because the certificate store is specific to the Windows Store app, when the app is uninstalled, any certificates specific to it are also removed.

As far as smart cards are concerned, certificates and keys contained on the card are automatically transferred to the user MY store and can be used by any application with full trust rights. To enable groups of principals to access groups of resources, you can leverage the Shared User Certificates capability to allow an app container process to access a specific resource. This capability grants an app container read access to the certificates and keys contained in the user MY store and the Smart Card Trusted Roots store, and it is typically used for financial or enterprise apps that require a smart card for authentication. The capability does not grant read access to the user REQUEST store.

Finally, to import a certificate from a Personal Information Exchange (PFX) message asynchronously, you can use the *ImportPfxDataAsync* method of the *CertificateEnrollmentManager* class. Listing 5-22 shows an example of its usage.

LISTING 5-22 Using the *ImportPfxDataAsync* method to import a certificate

```
function importCertificate_Click(args) {
    try {
        var pfxCertificate = Windows.ApplicationModel.Resources.ResourceLoader()
            .getString("MyCertificate");

        var password = "password";
        var friendlyName = "Pfx Certificate Sample";

        Windows.Security.Cryptography.Certificates
            .CertificateEnrollmentManager.importPfxDataAsync(
                pfxCertificate,
                password,
                ExportOption.NotExportable,
                KeyProtectionLevel.NoConsent,
                InstallOptions.None,
                friendlyName)
            .then(function () {
                // certificate successfully imported
            },
            function (err) {
                // handle error
            });
    }
    catch (ex) {
        // handle exception
    }
}
```

To import an issued certificate, it is not necessary for the certificate request to have been generated on the importing computer. The certificates included in the response do not need to be chained to trusted root certificates on the importing computer.

NOTE CERTIFICATE ENROLLMENT SAMPLE

For a more detailed example of certificate creation and enrollment, refer to <http://code.msdn.microsoft.com/windowsapps/Certificate-enrollment-84892e0c>.

Protecting your data with the *DataProtectionProvider* class

The *DataProtectionProvider* class (in the *Windows.Security.Cryptography.DataProtection* namespace) exposes methods that can help you to protect sensitive application data by asynchronously encrypting and decrypting static data or data streams.

The *DataProtectionProvider* class has two constructors: a default constructor with no parameters, and an overloaded constructor that accepts a string as a parameter describing the protection provider to use.

 **EXAM TIP**

The default constructor must be used before starting a decryption operation. Do not use this constructor to start an encryption operation; use the overloaded constructor instead.

NOTE THE ENTERPRISE AUTHENTICATION CAPABILITY

As stated in the official documentation, for security descriptors and security descriptor definition language (SDDL) strings, you must declare the enterprise authentication capability in the application manifest. The enterprise authentication capability is restricted to Windows Store apps built with company accounts and is subject to additional onboarding validation. You should avoid the enterprise authentication capability unless absolutely necessary.

The following code shows how to use the *ProtectAsync* method of the *DataProtectionProvider* to encrypt some static data:

```
var protectedBuffer = null;
function protectButton_click(args) {

    var descriptor = "LOCAL=user";
    var plainText = "Message to protect";
    var dpp = Windows.Security.Cryptography.DataProtection.
        DataProtectionProvider(descriptor);

    var binaryMessage = Windows.Security.Cryptography.CryptographicBuffer
        .convertStringToBinary(
            plainText,
            Windows.Security.Cryptography.BinaryStringEncoding.utf8);

    dpp.protectAsync(binaryMessage).then(
        function (result) {
            protectedBuffer = result;
        },
        function (err) {
            // handle error
        });
}

document.getElementById("encrypted").innerHTML = Windows.Security.Cryptography.
    CryptographicBuffer.encodeToBase64String(protectedBuffer);
}
```

To decrypt the protected data, you can leverage the *UnprotectAsync* method, which accepts an *IBuffer* object containing the encrypted message to unprotect, as shown in the following snippet:

```
function unprotectButton_click(args) {
    if (protectedBuffer !== null) {
        var dpp = Windows.Security.Cryptography.DataProtection.DataProtectionProvider();

        dpp.unprotectAsync(protectedBuffer).then(
            function (result) {
                document.getElementById("unprotected").innerHTML =
                    Windows.Security.Cryptography.CryptographicBuffer.
                    convertBinaryToString(
                        Windows.Security.Cryptography.BinaryStringEncoding.utf8,
                        result);
            },
            function (err) {
                // handle error
            });
    }
}
```

To test this code, you can use the HTML code in Listing 5-23 as a reference for your default page.

LISTING 5-23 Complete HTML definition for the default.html page

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Data Protection Sample</title>

    <!-- WinJS references -->
    <link href="/Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="/Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="/Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- Data Protection Sample references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <div id="plaintext">Message to protect</div>
    <button id="btnProtect">Protect Message</button>
    <div id="encrypted"></div>
    <button id="btnUnprotect">Unprotect Message</button>
    <div id="unprotected"></div>
</body>
</html>
```

If you run the application, the result should resemble Figure 5-12.

```

Message to protect

Protect Message

MIIB5QYJKoZIhvcNAQcDolB1jCCAdlCAQlxggF6oolBdgIBBDCCATgEggEGAQAAANCMnd8BFdERjHoAwE/Cl
+sBAAAAAYrgI0HgPvUmCw0rBsMQUIQAAAACAAAAAQZgAAAAEAACAAA AJ3s8n5ykiCZMw4Bm0ldim
dzAAAADX38UFdf5pb2R0//p2qkGeCjoTLPvdDmvY6rwL37Lts80pRzUTaKomJcUTbPMy1AAAAALoxOoPMa
+7tfkzAsBgkrBgEEAYI3SgEwHwYKKwYBBAGCN0oBCDARMA8wDQwFTE9DQuwMBHVzZXlwCwYJYIZIAWUD
+f7ptT505pME8CSqGSib3DQEHTAeBglghkgBZQMEAS4wEQQM07czCqfqmj0+h93TAgeEqgClrxPHHdGfs

Unprotect Message

Message to protect

```

FIGURE 5-12 A message first encrypted and then decrypted using the *DataProtectionProvider* class

You can also leverage the *ProtectStreamAsync* method to perform analogous operations on a stream of data. The method accepts two parameters: an *IInputStream* object representing the stream to protect and an *IOutputStream* object that contains the encrypted stream. Analogously, to decrypt a protected stream, you can call the *UnprotectStreamAsync* method, which also accepts two parameters: an *IInputStream* object that represents the stream to decrypt and an *IOutputStream* that contains the unprotected stream.

NOTE PASSWORDVAULT CLASS

To protect your users' passwords, you should leverage the dedicated *PasswordVault* class in the *Windows.Security.Credentials* namespace.



Thought experiment

Transmitting shared secret keys

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are developing a Windows Store app that needs to share a secret key with a third-party service. For symmetric key cryptography to work for online communications, the secret key must be securely shared with authorized communicating parties and protected from discovery and use by unauthorized parties.

1. Which of the techniques explained in this objective should you use to protect the confidentiality of the secret key over the Internet?
2. Which technique should you leverage to also protect the key's authenticity?

Objective summary

- Cryptography is used to secure your app and guarantee one or more of the following principles: authentication (the communication actually originates from a certain source), confidentiality (data being read by unauthorized third parties), data integrity (data being tampered with by unauthorized parties), nonrepudiation (other parties repudiating their own messages after they have been sent).
- You can leverage the *HashAlgorithmProvider* class and use one of the supported algorithms to protect message integrity. To ensure that a certain message has not been tampered with, send a message along with its corresponding hash code.
- Random generation of numbers and data represents an important step in many cryptographic operations, such as in generating cryptographic keys or passwords. To achieve these goals, you can use the *GenerateRandom* and *GenerateRandomNumber* methods exposed by the *CryptographicBuffer* class.
- Encrypt the hash code to ensure not only the integrity of the transmitted message but also its authenticity. Use the *MacAlgorithmProvider* class to encrypt the hash codes by using a shared secret key, or use the *CryptographicEngine.Sign* method to encrypt the hash code by using asymmetric cryptography.
- You can request a certificate from a specific URI by using the *CreateRequestAsync* method of the *CertificateEnrollmentManager* class. To install the certificate in the app container on the local computer, use the *InstallCertificateAsync* method, or use the *ImportPfxDataAsync* method to import a certificate from a Personal Information Exchange (PFX) message asynchronously.
- Protect sensitive data and streams used by your Windows Store app by encrypting them using the *DataProtectionProvider* class. To encrypt/decrypt static data, you can use the *ProtectAsync/UnprotectAsync* methods; use the *ProtectStreamAsync/UnprotectStreamAsync* methods to encrypt/decrypt data streams.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the “Answers” section at the end of this chapter.

1. Which of the following goals does a hash algorithm help protect?
 - A. Authentication
 - B. Data integrity
 - C. Confidentiality
 - D. All the above

2. Which method do you need to call to digitally sign (encrypt) a message using a key pair?
 - A. *HashData* method of the *HashAlgorithmProvider* class
 - B. *InstallCertificateAsync* method of the *CertificateEnrollmentManager* class
 - C. *ProtectAsync* method of the *DataProtectionProvider* class
 - D. *Sign* method of the *CryptographicEngine* class
3. When do you have to use the default constructor of the *DataProtectionProvider* class?
 - A. Before starting a decryption operation by calling the *UnprotectAsync/UnprotectStreamAsync* methods.
 - B. Before starting an encryption operation by calling the *ProtectAsync/ProtectStreamAsync* methods.
 - C. When you do not want to provide a specific security description and want to use the default descriptor instead.
 - D. The *DataProtectionProvider* class does not expose a default constructor.

Chapter summary

- Application data represents data that is bound to the application and is not useful outside of it. User settings for an application are application data.
- User data represents data that has meaning outside of the application. This type of data represents entities like customers, orders, and invoices, and is usually stored outside of the application in a database or some other form of storage.
- To access a file, you can use file pickers or folder pickers in which the user has an active part in the process, or you can rely on WinRT classes programmatically to save and retrieve text, stream, and binary data.
- Remember to define capabilities and declarations in the application manifest.
- Use one of the hash algorithms supported by the *Windows.Security.Cryptography* namespace to protect message integrity. To ensure not only the integrity of the transmitted message but also its authenticity, encrypt the hash code either by using a secret key or leveraging asymmetric cryptography. Generate random numbers and data for your keys and password, or request a certificate and use the key pair to digitally sign the hash code.
- Protect sensitive information managed by your Windows Store app by encrypting it.

Answers

This section contains the solutions to the thought experiments and answers to the lesson review questions in this chapter.

Objective 5.1: Thought experiment

You can use the local or roaming setting. The provided classes enable you to store application settings locally in the device or in live user profiles in the cloud. Settings in the cloud follow the user on each of his devices. For example, if you need to store the user's color preference, you can store it in the roaming profile; the user will see the same color on each of his devices. If you have to store the number of items to be displayed in a grid, this value can be stored locally because it can be different from device to device. Or you can store it in the roaming profile and override the value if the user changes it locally.

Objective 5.1: Review

1. Correct answers: A, D

- A. Correct:** User preferences are related to the application and have no meaning outside of it.
- B. Incorrect:** Entities are user data. They are usually stored outside of the application.
- C. Incorrect:** Not all application-related data is considered application data. Users and groups are considered user data.
- D. Correct:** Reference content like the list of cities for a weather application is bound to the application.

2. Correct answers: A, B

- A. Correct:** Use the *LocalSettings* properties to store application data locally.
- B. Correct:** Use the *RoamingSettings* properties to store application data in the user roaming profile.
- C. Incorrect:** The *ApplicationDataCreateDisposition* enum enables you to choose only the container creation methods.
- D. Incorrect:** The *ApplicationDataContainer* class enables you to store application data.

3. Correct answer: C

- A. Incorrect:** Windows Store apps can use the *StorageFolder* class to store data in files.
- B. Incorrect:** Windows Store apps can store data locally for application and user data.
- C. Correct:** The *ApplicationData* properties let you access the local, roaming, or temporary folder to store data and settings.
- D. Incorrect:** You can create files using the *LocalFolder* property of the *ApplicationData* class.

Objective 5.2: Thought experiment

After obtaining a reference to the file or folder, you can perform read and write operations on it. For example, you can create a file in a folder using the *CreateFileAsync* method and use the *then* promise to write some text to it as in the following code:

```
var localFolder = Windows.Storage.ApplicationData.current.localFolder;
localFolder.createFileAsync("log.txt",
    Windows.Storage.CreationCollisionOption.replaceExisting)
.then(function (logFile) {
    var formatter = new
        Windows.Globalization.DateTimeFormatting.DateTimeFormatter("longtime");
    var logDate = formatter.format(new Date());
    return Windows.Storage.FileIO.writeTextAsync(logFile, logDate);
}).done(null, function (err) {
    // Something wrong during save on log file
});
```

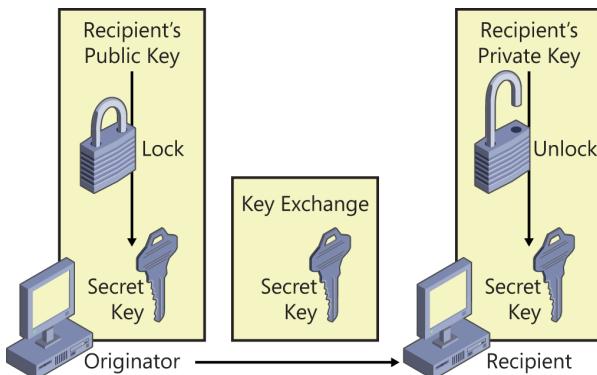
Remember, you need to define capabilities to perform operations on a system folder.

Objective 5.2: Review

1. **Correct answers:** A, B, C
 - A. **Correct:** You can access local storage.
 - B. **Correct:** You can access cloud storage on SkyDrive.
 - C. **Correct:** You can access network storage.
 - D. **Incorrect:** You cannot access the Windows Azure storage account with file pickers.
2. **Correct answer:** A
 - A. **Correct:** You have to define file extensions and associations in the application manifest.
 - B. **Incorrect:** Any Windows Store app can act as a file handler.
 - C. **Incorrect:** Any Windows Store app can act as a file handler.
 - D. **Incorrect:** You can associate an application as the handler for a particular extension.
3. **Correct answer:** C
 - A. **Incorrect:** The *FileOpenPicker* class does not let you access files.
 - B. **Incorrect:** The *StorageFolder* class does not let you access files.
 - C. **Correct:** The *StorageFile* class enables you to read a file programmatically.
 - D. **Incorrect:** The *ApplicationData* class does not let you access files.

Objective 5.3: Thought experiment

- One way to exchange the secret key over the Internet without compromising the security of the key is encrypting the secret key with the intended recipient's public key. Only the intended recipient can decrypt the secret key because it requires the use of the recipient's private key. Therefore, a third party who intercepts the encrypted, secret key cannot decrypt and use it. The following image illustrates the mechanism.



- Encrypting the secret key using the recipient's public key does not guarantee that the encrypted key comes from the intended sender. A malicious third party could replace the original key and encrypt it with the recipient's public key, for example. To address this issue, you could hash the secret key and then encrypt the hash key using the sender's private key. Only the corresponding sender's public key can decrypt the message. The recipient is guaranteed that the secret key comes from sender. The hash can also be used to verify data integrity.

Objective 5.3: Review

- Correct answer: B**
 - Incorrect:** Message authentication code (MAC) algorithms and digital signatures can help to ensure message authentication.
 - Correct:** A hash code obtained by applying a hash algorithm to binary content represents the fingerprint of that document and prevents unauthorized people from tampering with it.
 - Incorrect:** Hash algorithms only transform binary values of arbitrary length to smaller binary values of a fixed length. They do not encrypt the message content itself.
 - Incorrect:** A hash algorithm helps to protect authentication, data integrity, and confidentiality.

2. Correct answer: D

- A. **Incorrect:** The *HashData* method of the *HashAlgorithmProvider* class is used to generate a hash code.
- B. **Incorrect:** The *InstallCertificateAsync* method of the *CertificateEnrollmentManager* class is used to install a certificate in the app container.
- C. **Incorrect:** The *ProtectAsync* method of the *DataProtectionProvider* class is used to encrypt static data.
- D. **Correct:** The *Sign* method of the *CryptographicEngine* class digitally signs (encrypts) a message with the provided key pair.

3. Correct answer: A

- A. **Correct:** You must create a *DataProtectionProvider* object using the default constructor before starting a decryption operation through the *UnprotectAsync*/*UnprotectStreamAsync* methods.
- B. **Incorrect:** Before starting an encryption operation by calling the *ProtectAsync*/*ProtectStreamAsync* methods, you have to use the overloaded constructor. The constructor accepts as its only parameter a string indicating the security descriptor to be used.
- C. **Incorrect:** The overloaded constructor must be used before starting an encryption operation. The constructor accepts as its only parameter a string indicating the security descriptor to be used. Providing the descriptor is not optional.
- D. **Incorrect:** The *DataProtectionProvider* exposes a default constructor that must be used to perform decryption operations.

CHAPTER 6

Prepare for a solution deployment

This chapter begins by showing you how to design and implement trial functionality in your app, handle licensing scenarios, and perform in-app purchases from your Windows Store app. From there you learn how to handle errors and exceptions in a Windows Store app, with a focus on device capability errors and exceptions thrown during asynchronous operations. The chapter also explores designing and implementing a test strategy for your app, which includes the peculiarities of the test framework for Windows Store apps and the Unit Test Library project template. Finally, you learn how to test your app's performance, trace and collect detailed information about your app's behaviors, and generate reports and other analytical information.

Objectives in this chapter:

- Objective 6.1: Design and implement trial functionality in an app
- Objective 6.2: Design for error handling
- Objective 6.3: Design and implement a test strategy
- Objective 6.4: Design a diagnostics and monitoring strategy

Objective 6.1: Design and implement trial functionality in an app

In this section, you design and implement trial functionality in your Windows Store app, such as timed trials and feature-based trials. You also learn how to handle in-app purchases to offer your customers extra features and products.

This objective covers how to:

- Set up a timed trial
- Set up a feature-based trial
- Set up in-app purchases
- Transition an app from trial to full

Choosing the right business model for your app

One of the most critical steps in publishing a Windows Store app is determining the right type of license. In the Selling details section of the Windows Store Dashboard, shown in Figure 6-1, you can enter the price of the app, select a trial period, and choose the countries/regions where you want to sell the app.

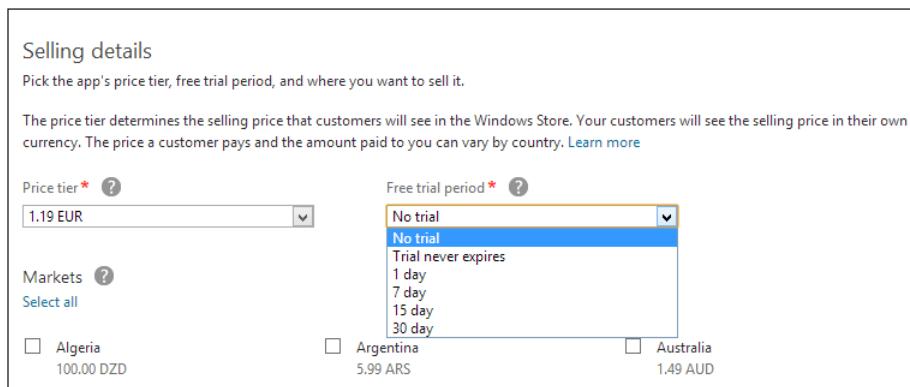


FIGURE 6-1 Choosing app selling details in the Windows Store Dashboard

The simplest, but probably the least effective, possibility is selling your entire application for a certain price. Users can decide whether your app is worth the price only by reading the information and the reviews published on your app's dedicated page in the Windows Store. This approach corresponds to the No trial option shown in Figure 6-1.

A second possibility is to let your customers download and try your app before paying for it. One option is called a *timed trial*, in which you provide a trial version of your app with all the features of the paid version. When the trial period expires, the app simply stops working. In more refined scenarios, the app still works when the trial period ends, but it can ask the users to buy the full license at regular intervals or display commercial ads until the user purchases the full version. In the drop-down list shown in Figure 6-1, you have to choose among four different deadlines for a timed trial: 1, 7, 15, and 30 days.

Another option is the *feature-based trial*, which enables potential customers to access only a subset of your app's functionalities or content, unless they decide to buy the full version of the app. Imagine, for example, a game that lets you play only the first levels, or a photo enhancement app that allows you to use only a limited set of filters and effects. In this scenario, the trial period lasts indefinitely and corresponds to the Trial never expires option shown in Figure 6-1.

Figure 6-2 shows the left panel of a feature trial game that enables you to play the first few levels for an indefinite period, but if you want to access the entire content, you must purchase the full version of the app in the Windows Store.

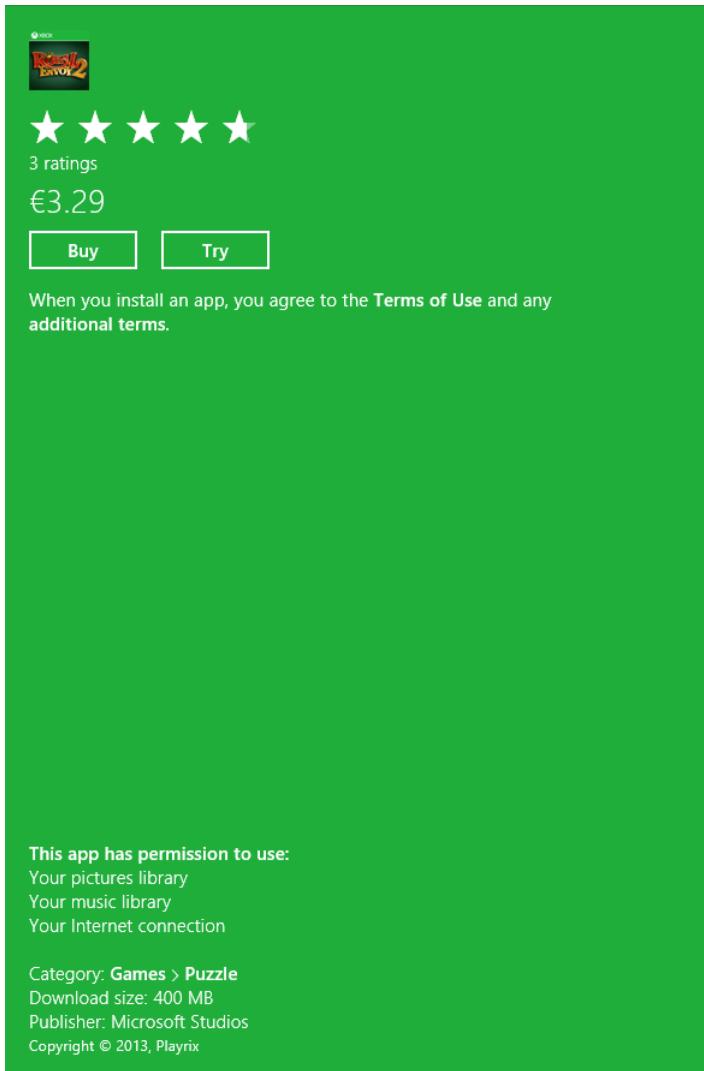


FIGURE 6-2 Buy and Try buttons in a feature-based game

Finally, you can decide to distribute a basic version of your app but give your customers the option to extend it by purchasing advanced functionalities or additional content. This distribution method is referred to as *in-app purchase*. In this case, the app is split into multiple modules that can be purchased separately.

In the Windows Store, you can define which features or products can be purchased by your customers. For each feature or product, you have to pick a price and decide the product lifetime. The feature lifetime describes how long a customer can use the purchased feature. After the time expires, the feature stops working and it must be purchased again. Figure 6-3 shows the various options.

NOTE IN-APP PURCHASES

In-app purchases cannot be offered during a trial version of an app.

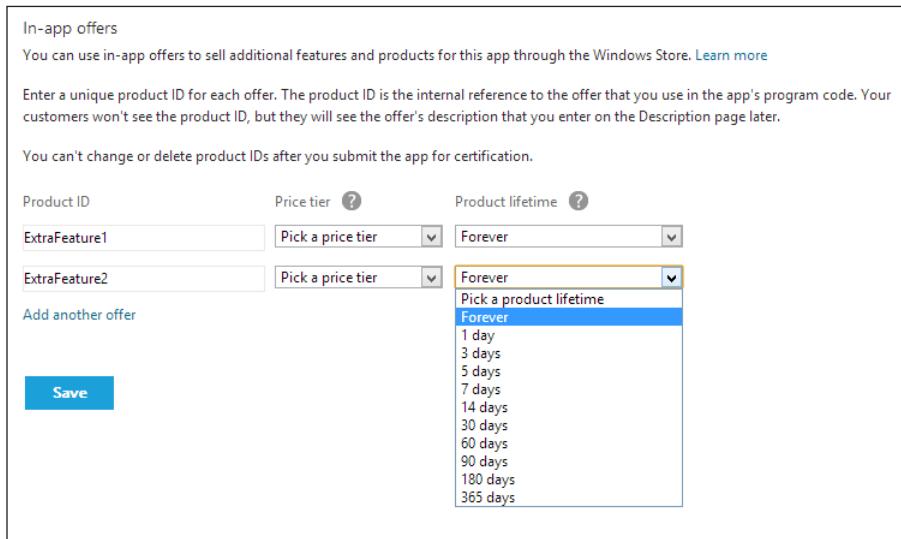


FIGURE 6-3 In-app offers in the Windows Store Dashboard

Remember to implement all the features and products you want to offer to your customers through in-app purchases before publishing the app in the Windows Store. If you want to add new features or products after your app has been published, you need to submit an updated version of your app to the Windows Store.

Exploring the licensing state of your app

You can use the licensing application programming interfaces (APIs) provided by the `Windows.ApplicationModel.Store` namespace to determine the license state of an app or an in-app purchase feature. The licensing APIs enable you to:

- Check the current license status of an app.
- Check the expiration date of a trial period.
- Check whether an app's feature has been purchased through an in-app purchase.
- Perform an in-app purchase.

The `LicenseInformation` property exposed by the `CurrentApp` class enables you to access information about the current license state of your app and of other products or features that are enabled when the customer makes an in-app purchase. In particular, the `LicenseInformation` class exposes the following read-only properties:

- **IsActive** Describes the current license state of this app. A value of *true* indicates a valid license, regardless of whether the app is in trial mode. A value of *false* indicates that the app's license state is invalid because the license is missing, expired, or has been revoked.
- **IsTrial** Indicates whether an app is in trial mode. A value of *false* means that a full version of the app has been bought by the user, whereas a value of *true* indicates the app is still in trial mode. It is important to understand that this property returns *true* even if the trial period is expired, so you should always check the *IsTrial* property with the *IsActive* property.
- **ExpirationDate** Indicates the expiration date for the trial. The date must be expressed in the ISO 8601 format, which is *yyyy-mm-ddThh:mm:ss.ssZ*. For example, the date 2014-06-19T09:00:00.00Z means that the trial will expire on June 19, 2014, at 9:00 A.M.
- **ProductLicenses** Contains the list of licenses for the app's features that can be bought through an in-app purchase.

Because the *CurrentApp* class contains data and information retrieved from the Windows Store, you can access them only if your app has been published in the Windows Store. Even in that case, it would be hard to test different behaviors of your app in the local environment by using the *CurrentApp* class because you would be working against the actual Windows Store. When dealing with trials and in-app purchases, you have to use the *CurrentAppSimulator* class, which defines methods and properties that mimic those exposed by the *CurrentApp* class, but in a simulated environment. You can use the methods and properties of the *CurrentAppSimulator* class to get simulated license information (such as the app's ID and license metadata) for testing the app's behaviors in your development environment.



EXAM TIP

Remember to replace the *CurrentAppSimulator* class with the *CurrentApp* class before submitting the app to the Windows Store; otherwise, the app does not pass the certification process.

Listing 6-1 shows how to retrieve a simulated license state via the *CurrentAppSimulator* object to display on the screen.

LISTING 6-1 Retrieving a simulated license state

```
app.onloaded = function () {
    displayLicenseInfo();
};

var currentApp = Windows.ApplicationModel.Store.CurrentAppSimulator;

function displayLicenseInfo() {
    try {
        var licenseInfo = currentApp.licenseInformation;
```

```

        if (licenseInfo.isActive) {
            if (licenseInfo.isTrial) {
                licenseState.innerHTML = "<p>License current status: Trial license</p>";

                var longDateFormat = Windows.Globalization
                    .DateTimeFormatting.DateTimeFormatter("shortdate");

                var remainingDays =
                    (licenseInfo.expirationDate - new Date()) / 86400000;

                licenseRemainingDays.innerHTML = "<p>Expiration date: " +
                    longDateFormat.format(licenseInfo.expirationDate) +
                    " - Remaining days: " + Math.round(remainingDays) + "</p>";
            }
            else {
                licenseState.innerHTML = "<p>License current status: Full license</p>";
                licenseRemainingDays.innerHTML = "<p>No expiration</p>";
            }
        }
        else {
            licenseState.innerHTML = "<p>License current status:
                license is expired. Please buy the app!</p>";
        }
    } catch (e) {
        licenseState.innerHTML =
            "<p>Listing information unavailable. Exception: " + e.message + "</p>";
    }
}

```

The following code shows the HTML definition of the default page that you can use as a reference:

Sample of HTML code

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Trial Sample in JavaScript</title>

    <!-- WinJS references -->
    <link href="//Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="//Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="//Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- TrialSampleJS references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <div id="licenseState"></div>
    <div id="licenseRemainingDays"></div>
</body>
</html>

```

If you execute the application, the result should look similar to Figure 6-4.

```
License current status: Trial license  
Expiration date: 12/31/9999 - Remaining days: 2917106
```

FIGURE 6-4 The sample app displaying the license status, expiration date, and remaining days

The information comes from the WindowsStoreProxy.xml file. When using the *CurrentAppSimulator* class, the app's initial license state is defined in the WindowsStoreProxy.xml file that is located in the %userprofile%\appdata\local\packages\<package-moniker>\localstate\microsoft\Windows Store\Apidata folder.

Listing 6-2 shows the autogenerated WindowsStoreProxy.xml file of an app deployed on the local machine.

LISTING 6-2 WindowsStoreProxy.xml file

```
<?xml version="1.0" encoding="utf-16" ?>  
<CurrentApp>  
    <ListingInformation>  
        <App>  
            <AppId>00000000-0000-0000-0000-000000000000</AppId>  
            <LinkUri>  
                http://apps.microsoft.com/webpdp/app/00000000-0000-0000-0000-000000000000  
            </LinkUri>  
            <CurrentMarket>en-US</CurrentMarket>  
            <AgeRating>3</AgeRating>  
            <MarketData xml:lang="en-us">  
                <Name>AppName</Name>  
                <Description>AppDescription</Description>  
                <Price>1.00</Price>  
                <CurrencySymbol>$</CurrencySymbol>  
                <CurrencyCode>USD</CurrencyCode>  
            </MarketData>  
        </App>  
        <Product ProductId="1" LicenseDuration="0">  
            <MarketData xml:lang="en-us">  
                <Name>Product1Name</Name>  
                <Price>1.00</Price>  
                <CurrencySymbol>$</CurrencySymbol>  
                <CurrencyCode>USD</CurrencyCode>  
            </MarketData>  
        </Product>  
    </ListingInformation>  
    <LicenseInformation>  
        <App>  
            <IsActive>true</IsActive>  
            <IsTrial>true</IsTrial>  
        </App>  
        <Product ProductId="1">  
            <IsActive>true</IsActive>  
        </Product>  
    </LicenseInformation>  
</CurrentApp>
```

Notice that the first section, <ListingInformation>, includes general information about the app, as well as about additional features and products that can be bought separately. Some of this information can be accessed through the corresponding properties of the *CurrentApp* class (or the *CurrentAppSimulator* class, in a simulated environment), such as the *AppId* property, which represents the app's unique ID in the Windows Store, and the *Link* property, which represents the link to the listing page in the store.

The other information contained in the <ListingInformation> section includes the name of the app in the store, the age rating, the current market, the price, and so on. This information can be retrieved from the Windows Store by leveraging the *LoadListingInformationAsync* method of the *CurrentApp* class (or *CurrentAppSimulator*, if you are testing your app in a simulated environment), which returns an instance of the *ListingInformation* class.

NOTE WINDOWSSTOREPROXY.XML FILE

The WindowsStoreProxy.xml file is normally created the first time your application tries to access the *CurrentAppSimulator.LicenseInformation* property.

The JavaScript code in Listing 6-3 displays some of the listing information on the screen (changes to Listing 6-1 are in bold).

LISTING 6-3 Displaying listing information

```
app.onloaded = function () {
    displayListingInfo();
    displayLicenseInfo();
};

var currentApp = Windows.ApplicationModel.Store.CurrentAppSimulator;

function displayListingInfo() {

    appId.innerHTML = "<p>App ID: " + currentApp.appId + "</p>";
    storeLink.innerHTML = "<p>Store Link: " + currentApp.linkUri.absoluteUri + "</p>";

    var listingInfo = currentApp.loadListingInformationAsync().done(
        function (listing) {
            appName.innerHTML =
                "<p>App's Name on the Store: " + listing.name + "</p>";
        },
        function () {
            appName.innerHTML = "<p>Listing information unavailable.</p>";
        }
    );
}
```

To display the app listing information on the screen, modify the HTML definition of the default page as follows (changes are in bold):

Sample of HTML code

```
<body>
    <div id="appId"></div>
    <div id="storeLink"></div>
    <div id="appName"></div>
    <div id="licenseState"></div>
    <div id="licenseRemainingDays"></div>
</body>
```

If you launch the application, the results should resemble Figure 6-5.

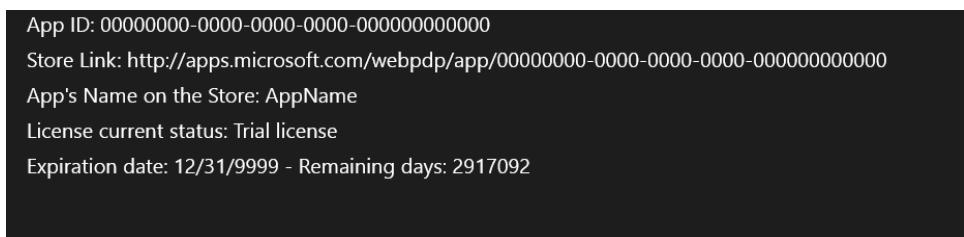


FIGURE 6-5 Displaying app listing information

The *LicenseInformation* section contains, as its name suggests, specific information about the license state of the app (in the *App* element) and of other products or features that are enabled when the customer makes an in-app purchase (in the *Product* element). As mentioned previously, this information can be retrieved from the store by accessing the *LicenseInformation* object exposed by the *CurrentApp* (or *CurrentAppSimulator*) class.



EXAM TIP

It is important to understand that any modification that alters the license state does not affect the `WindowsStoreProxy.xml` file, only the object in memory. This means that the next time you launch the app, you will find that the content of the original file has not changed.

Now that you know where the initial licensing information comes from, the next question is why the expiration date corresponds to the maximum value of a *Date* object (more precisely, the value of this constant is equivalent to December 31, 9999, exactly one 100-nano-second tick before 00:00:00, January 1, 10000). The answer is that the default Extensible Markup Language (XML) definition does not indicate any expiration date. In this case, the *CurrentAppSimulator* object will assume that the trial will never expire (as in a feature-based trial).

MORE INFO DATE OBJECT

In the Windows Runtime (WinRT), a JavaScript *Date* object represents a projection of an inner type, that is, an instance of the *Windows.Foundation.DateTime* structure. None of the supported languages uses this type directly. Each of them exposes a projection of it (for example, in C# this structure is projected as a *System.DateTimeOffset* type), and handles conversions and date ranges transparently.

Using custom license information

To test your app under different licensing options, you can write your own XML definition file and provide it to the *CurrentAppSimulator* object by leveraging the *ReloadSimulatorAsync* static method. Listing 6-4 shows the custom licensing information used for this sample.

LISTING 6-4 The timed-trial.xml custom file definition

```
<?xml version="1.0" encoding="utf-16"?>
<CurrentApp>
    <ListingInformation>
        <App>
            <AppId>01234567-1234-1234-0123456789AB</AppId>
            <LinkUri>
                http://apps.windows.microsoft.com/app/2B14D306-D8F8-4066-A45B-0FB3464C67F2
            </LinkUri>
            <CurrentMarket>en-US</CurrentMarket>
            <AgeRating>5</AgeRating>
            <MarketData xml:lang="en-us">
                <Name>Basic Timed Trial</Name>
                <Description>Basic Timed trial sample</Description>
                <Price>0.99</Price>
                <CurrencySymbol>$</CurrencySymbol>
            </MarketData>
        </App>
    </ListingInformation>
    <LicenseInformation>
        <App>
            <IsActive>true</IsActive>
            <IsTrial>true</IsTrial>
            <ExpirationDate>2014-06-19T09:00:00.00Z</ExpirationDate>
        </App>
    </LicenseInformation>
</CurrentApp>
```

The *ReloadSimulatorAsync* method accepts a *StorageFile* object containing the XML definition of the simulated license, which is used by the simulator. Listing 6-5 shows an example of its usage (changes to Listing 6-3 in bold).

LISTING 6-5 Providing the custom license defined in the timed-trial.xml file to the *CurrentAppSimulator* object

```
app.onloaded = function () {
    loadCustomSimulator();
    //displayLicenseInfo();
    //displayListingInfo();
};

var currentApp = Windows.ApplicationModel.Store.CurrentAppSimulator;

function loadCustomSimulator() {

    var currentApp = Windows.ApplicationModel.Store.CurrentAppSimulator;
    Windows.ApplicationModel.Package.current.installedLocation
        .getFolderAsync("data")
        .then(
            function (folder) {
                return folder.getFileAsync("timed-trial.xml");
            }
        )
        .then(
            function (file) {
                currentApp.licenseInformation.onlicensechanged = reloadLicense;
                currentApp.reloadSimulatorAsync(file);
            }
        )
        .done(
            null,
            function (err) {
                // handle error
            });
}

function reloadLicense() {
    displayLicenseInfo();
    displayListingInfo();
}
```

When the simulator has finished loading the custom license, you need to update the information displayed on the screen. The code illustrated in Listing 6-5 leverages the *LicenseChanged* event of the *LicenseInformation* object. This event is raised whenever the status of the app's license changes.

Figure 6-6 shows the new license information that displays on the screen.

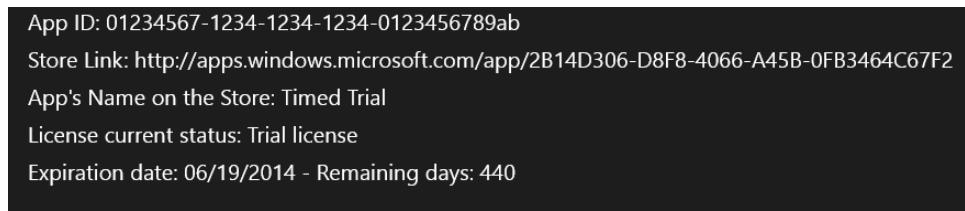


FIGURE 6-6 Updated license information

Purchasing an app

After you enable trial functionality in your app, the next step is to let customers buy the full version. To accomplish this task, the *CurrentApp* (or *CurrentAppSimulator*) class exposes the *RequestAppPurchaseAsync* static method. This method creates the async operation that enables the user to purchase (or simulate a purchase, in the case of the *CurrentAppSimulator* object) a full license for the current app.

The *RequestAppPurchaseAsync* method accepts a Boolean value as a parameter to indicate whether the method should return a string representing the receipt for the purchase, as shown in the following code excerpt. You learn how to handle purchase receipts in the “Retrieving and validating the receipts for your purchases” section. For now, use the simplest version of this method by passing *false* as parameter. (Changes to Listing 6-5 code are in bold.)

Sample of JavaScript code

```
app.onloaded = function () {
    buyButton.addEventListener("click", buyButton_click, false);
    loadCustomSimulator();
};

var currentApp = Windows.ApplicationModel.Store.CurrentAppSimulator;

function buyButton_click(args) {
    currentApp.requestAppPurchaseAsync(false).done(
        function () {
            // Notify user that the purchase went fine
        }, function (err) {
            purchaseMessage.innerHTML = "<p>Unable to buy: " + err + "</p>";
        });
}
```

In the HTML page, just add the line of code highlighted in bold:

Sample of HTML code

```
<body>
    <div id="appId"></div>
    <div id="storeLink"></div>
    <div id="appName"></div>
    <div id="licenseState"></div>
    <div id="licenseRemainingDays"></div>
    <button id="buyButton">Buy app</button>
    <div id="purchaseMessage"></div>
</body>
</html>
```

If you launch the application and try to buy the app, the *CurrentAppSimulator* displays a Windows Store dialog box (see Figure 6-7) prompting you to select an error code to return. You can choose the result you want to simulate for the current transaction to test whether your code reacts properly to all possible responses, without purchasing the app from the Windows Store. Leave the default value (S_OK) and click Continue to simulate a successful transaction.

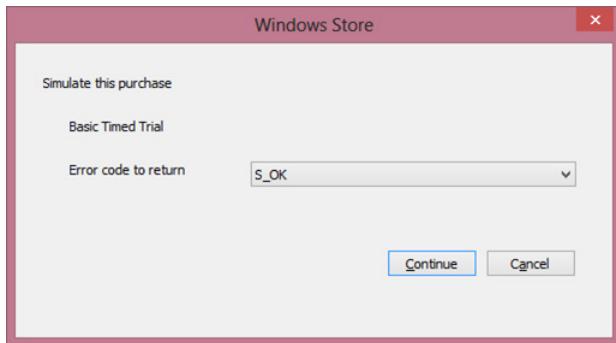


FIGURE 6-7 The Windows Store dialog box prompting for an error code

Figure 6-8 shows the updated license state after the user has purchased the full version.

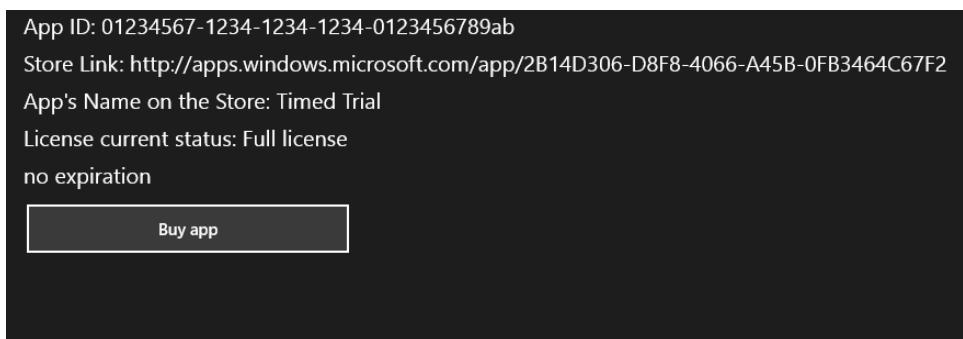


FIGURE 6-8 Updated license information after purchase

NOTE LICENSE STATE MODIFICATION

Remember that any modification to the license state does not affect the XML file supplied to the *CurrentAppSimulator* object, only the object in memory.

So far, you have learned about a timed trial. But, what happens if you want to implement a feature-based trial? Depending on the type of trial, the semantics of the *IsActive* and *IsTrial* properties might be slightly different.

If you chose a timed trial, the *IsActive* property tells you if the trial period has already expired (or, in rare cases, if the license is missing or has been revoked). If the property returns *false*, it means the evaluation period is expired and that consequently your app (in the simplest scenario) should stop working. If the property returns *true*, it means the evaluation period is not over yet and the app can keep working normally.



EXAM TIP

You should check the *IsTrial* property to verify whether the customer has bought the full version of the app or if it is still in trial mode. If in trial mode, you should let your customer know the number of days left before the app stops working. However, remember that, as mentioned in the previous section, the *IsTrial* property returns *true* even if the trial period is expired.

The following code uses a pattern similar to the timed-trial sample in Listing 6-1.

```
if (licenseInfo.isActive) {
    if (licenseInfo.isTrial) {
        // Your app is still in trial mode.
        // Remember to let your customers know the remaining time
        // before the app stops working.
    } else {
        // The user has bought the full license version of your app.
    }
} else {
    // Your app's license is expired (or the license is missing or has been revoked).
    // It means that your app should stop working.
    // Remember to invite your customers to buy the full version of the app
}
```

On the contrary, if you offer a feature-based trial, you need to modify the pattern followed in Listing 6-1 because, in this case, there is no expiration date that needs to be checked. In a feature-based scenario, the *IsActive* property should return *true* (unless something has gone wrong; remember to handle this scenario as well). In this type of trial, you want to know whether the user has bought the full version of the app to decide what features to enable. This code follows this type of pattern:

```
if (licenseInfo.isActive) {
    if (licenseInfo.isTrial) {
        // Your app is in trial mode. Enable only the features available in trial mode.
    } else {
        // The user has bought the full license for your app.
        // You can now enable all the features.
    }
} else {
    // something went wrong: the license is missing or revoked
}
```

Handling errors

When you use the *CurrentApp* class in an application, invoking any methods that access the Windows Store API can result in an exception. For this reason, any call to one of these methods should be wrapped in a *try/catch* block, and you have to be sure that your app is capable of handling these errors gracefully.

You learn how to set up different test strategies in the Objective 6.2, “Design for error handling,” section. For now, keep in mind that in the local environment, you can leverage the *CurrentAppSimulator* class to ensure your code covers the most common failure scenarios, at a minimum. There are two ways to achieve the same result.

The simplest method is to take advantage of the Windows Store dialog box shown in Figure 6-7 to return different error codes. However, this procedure can be quite time-consuming, and, more importantly, it cannot be used in an automated test strategy. The *CurrentAppSimulator* object gives you another way to achieve this objective. When supplying the *CurrentAppSimulator* object with a custom XML definition for the license state, you can leverage the *Simulation* element to specify what error code you want each method to return. Listing 6-6 shows a revised version of the *timed-trial.xml* supplied to the *CurrentAppSimulator* class (changes are highlighted in bold).

LISTING 6-6 Retrieving a simulated license state

```
<?xml version="1.0" encoding="utf-16"?>
<CurrentApp>
    <ListingInformation>
        <App>
            <AppId>01234567-1234-1234-0123456789AB</AppId>
            <LinkUri>
                http://apps.windows.microsoft.com/app/2B14D306-D8F8-4066-A45B-0FB3464C67F2
            </LinkUri>
            <CurrentMarket>en-US</CurrentMarket>
            <AgeRating>5</AgeRating>
            <MarketData xml:lang="en-us">
                <Name>Basic Timed Trial</Name>
                <Description>Basic Timed trial sample</Description>
                <Price>0.99</Price>
                <CurrencySymbol>$</CurrencySymbol>
            </MarketData>
        </App>
    </ListingInformation>
    <LicenseInformation>
        <App>
            <IsActive>true</IsActive>
            <IsTrial>true</IsTrial>
            <ExpirationDate>2014-06-19T09:00:00.00Z</ExpirationDate>
        </App>
    </LicenseInformation>
    <Simulation SimulationMode="Automatic">
        <DefaultResponse MethodName="RequestAppPurchaseAsync_GetResult" HResult="E_FAIL"/>
    </Simulation>
</CurrentApp>
```

The *Simulation* element describes how to handle the various method calls. When the *SimulationMode* attribute is set to *Automatic*, the method specified in the *MethodName* attribute automatically returns the error code indicated in the *HResult* attribute.

NOTE <SIMULATION> SECTION

If you add a `<Simulation>` section to the custom XML license definition and launch the app, the Windows Store dialog box shown in Figure 6-7 does not appear. In this case, you have already defined the response code to be returned for the method indicated in the *MethodName* attribute.

Unfortunately, this strategy has significant drawbacks because the *MethodName* attribute supports only the following three methods: `RequestAppPurchaseAsync_GetResult`, `RequestProductPurchaseAsync_GetResult`, and `LoadListingInformationAsync_GetResult`. (The `GetResult` suffix indicates a call to the `GetResults` method exposed by the `IAsyncOperation<TResult>` object to retrieve the result of the asynchronous operation). For example, if you use the XML definition in Listing 6-6, any call to the `RequestAppPurchaseAsync` method throws an exception with an `E_Fail` error code.

This strategy can be used when running automated test cases. In fact, you can leverage the `ReloadSimulatorAsync` method to supply different metadata definitions to your app. For example, you could prepare multiple XML files with different metadata and then feed them to the `CurrentAppSimulator` object to simulate the most common scenarios.

Setting up in-app purchases

You can offer products and features from within your app that your customers can buy. You must first design the app's code to treat these features and products as separate modules so they can be incorporated into your licensing model without effort. Then you can indicate which features or products you want to offer in the in-app section of your Windows Store Dashboard.

MORE INFO IN-APP PURCHASES

Deciding what features or products might be worth selling separately to your customers is just the first step. You must design the code of your application so that you can actually treat these features and products as separate modules, so they can be incorporated into your licensing model without effort. You can find some useful tips on this topic at <http://msdn.microsoft.com/en-us/library/windows/apps/hh694065.aspx>.

To begin, you must add a new custom license definition to be supplied to the `CurrentAppSimulator` that includes information about the features and products that can be purchased. It enables you to test the app's behavior in the local environment. Listing 6-7 shows a custom XML license definition to test the in-app purchase feature. For the sake of simplicity, only one extra feature is added.

LISTING 6-7 The in-app-purchase.xml custom file definition to test the in-app purchase feature

```
<?xml version="1.0" encoding="utf-16"?>
<CurrentApp>
    <ListingInformation>
        <App>
            <AppId>01234567-1234-1234-0123456789AB</AppId>
            <LinkUri>
                http://apps.windows.microsoft.com/app/2B14D306-D8F8-4066-A45B-0FB3464C67F2
            </LinkUri>
            <CurrentMarket>en-US</CurrentMarket>
            <AgeRating>5</AgeRating>
            <MarketData xml:lang="en-us">
                <Name>In-app Purchase Sample</Name>
                <Description>In-app Sample</Description>
                <Price>0.99</Price>
                <CurrencySymbol>$</CurrencySymbol>
            </MarketData>
        </App>
        <Product ProductId="feature1">
            <MarketData xml:lang="en-us">
                <Name>Advanced Feature</Name>
                <Price>0.99</Price>
                <CurrencySymbol>$</CurrencySymbol>
                <CurrencyCode>USD</CurrencyCode>
            </MarketData>
        </Product>
    </ListingInformation>
    <LicenseInformation>
        <App>
            <IsActive>true</IsActive>
            <IsTrial>false</IsTrial>
        </App>
        <Product ProductId="feature1">
            <IsActive>false</IsActive>
        </Product>
    </LicenseInformation>
</CurrentApp>
```

The *IsTrial* element of the *<App>* section is set to *false* because the in-app purchase feature is not compatible with trial mode. In addition, the *IsActive* property of the offered product is set to *false*, which means the user has not yet purchased the extra feature or product.

After you provide the new custom license definition to the *CurrentAppSimulator* object, you can retrieve the information contained in the *Product* element within the *ListingInformation* section, such as the feature's name and price, by leveraging the *LoadListingInformationAsync* method. This method returns a read-only dictionary containing all the features and products that can be purchased by the customer. Listing 6-8 shows how to retrieve the listing information concerning the offered feature.

LISTING 6-8 Retrieving listing information of a feature

```
function displayProductListingInfo() {  
  
    var listingInfo = currentApp.loadListingInformationAsync().done(  
        function (listing) {  
            featureName.innerHTML = "Feature name: " + listing.name;  
            featurePrice.innerHTML = "You can buy this feature for " +  
                listing.formattedPrice;  
        },  
        function (err) {  
            featurePurchaseError.innerHTML = "Unable to show feature info: " + err;  
        }  
    );  
}
```

You can also display the licensing information of the offered feature, as shown in Listing 6-9, in which the code retrieves the *ProductLicense* for the current feature and displays the state of the license.

LISTING 6-9 Displaying licensing information of a feature

```
function displayProductLicenseInfo() {  
    try {  
        var productLicense = currentApp.licenseInformation.productLicenses  
            .lookup("feature1");  
        if (!productLicense.isActive)  
            featureLicenseStatus.innerHTML = "Advanced feature license status: inactive.  
                You cannot use this feature";  
        else {  
            featureLicenseStatus.innerHTML = "Advanced feature license status: active.  
                You can now use this feature";  
            document.getElementById("featurePrice").style.visibility = "collapse";  
        }  
    }  
    catch (ex) {  
        featurePurchaseError.innerHTML = "Unable to show product information";  
    }  
}
```

Do not forget to modify the *LoadCustomSimulator* method to load the proper file (in-app-purchase.xml) and the *ReloadLicense* method to display the new information, as shown in Listing 6-10 (changes are highlighted in bold).

LISTING 6-10 Modifying methods to display new information

```
function loadCustomSimulator() {  
  
    var currentApp = Windows.ApplicationModel.Store.CurrentAppSimulator;  
    Windows.ApplicationModel.Package.current.installedLocation  
        .getFolderAsync("data")  
        .then(  
            function (folder) {  
                return folder.getFileAsync("in-app-purchase.xml");  
            })
```

```

        .then(
            function (file) {
                currentApp.licenseInformation.onlicensechanged = reloadLicense;
                currentApp.reloadSimulatorAsync(file);
            }
        .done(
            null,
            function (err) {
                // handle error
            });
    }

function reloadLicense() {
    displayLicenseInfo();
    displayListingInfo();
    displayProductListingInfo();
    displayProductLicenseInfo();
}

```

Now that you have all the information about the features and products offered through this app, you can implement the in-app purchase feature. The *RequestProductPurchaseAsync* method accepts two parameters: a string representing the ID of the feature to be purchased (which must match the name of the product added in the Windows Store Dashboard) and a Boolean value indicating whether to return a receipt for the purchase. The following code excerpt shows the complete code for this method:

Sample of JavaScript code

```

app.onloaded = function () {
    buyButton.addEventListener("click", buyButton_click, false);
    buyProductButton.addEventListener("click", buyProduct_Click, false);
    loadCustomSimulator();
};

(code omitted)

function buyProduct_Click(args) {

    currentApp.requestProductPurchaseAsync("feature1", false).done(
        function () {
            // notify user that the purchase went fine
        },
        function (err) {
            featurePurchaseMessage.innerHTML = "Unable to purchase the feature.";
        });
}

```

You can modify the Extensible Application Markup Language (XAML) code of the page, as follows, to display the new information:

Sample of HTML code

```
<body>
    <div id="appId"></div>
    <div id="storeLink"></div>
    <div id="appName"></div>
    <div id="licenseState"></div>
    <div id="licenseRemainingDays"></div>
    <button id="buyButton">Buy app</button>
    <div id="purchaseResult"></div>
    <div id="featureName"></div>
    <div id="featureLicenseStatus"></div>
    <div id="featurePrice"></div>
    <button id="buyProductButton">Buy advanced feature</button>
    <div id="featurePurchaseMessage"></div>
</body>
```

Running the app displays a screen similar to Figure 6-9. The newly added information is indicated by a callout. The state of the feature's license is set to inactive, which means you do not have access to this feature unless you purchase it.

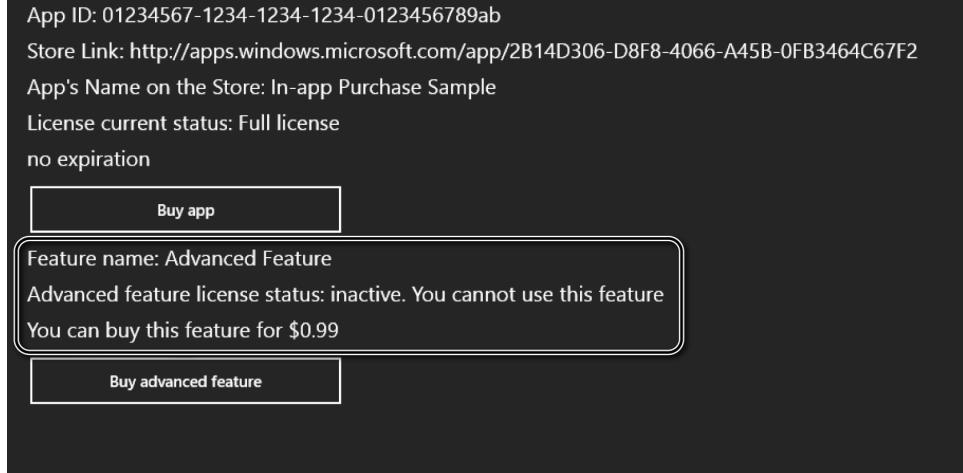


FIGURE 6-9 Information about the advanced feature

To step through the transaction, click the Buy advanced feature button. In the Windows Store dialog box, select the S_OK code in the simulator to simulate that the transaction has succeeded. The screen should display the updated license state for the newly purchased advanced feature, as shown in Figure 6-10.



FIGURE 6-10 Updated license information for the advanced feature

Retrieving and validating the receipts for your purchases

Sometimes, you might need to check the receipt of a transaction to determine whether the user bought the full version of your app or a specific feature or product. The *Windows.ApplicationModel.Store* namespace supports two ways of getting a receipt:

- You can ask for a receipt when you purchase an app. To do that, you only need to pass *true* to the *RequestProductPurchaseAsync* method (as the only parameter) or to the *RequestAppPurchaseAsync* method (as a second parameter).
- You can ask for a receipt at any time by calling the *GetAppReceiptAsync* and *GetProductReceiptAsync* methods.

The receipt for an app or product purchase consists of an XML fragment that contains all the information regarding the transaction, such as the receipt ID, the app ID, the date of the purchase, and the type of license purchased. Listing 6-11 shows an example of an app purchase receipt.

LISTING 6-11 An in-app purchase receipt in XML

```
<?xml version="1.0" encoding="utf-8" ?>
<Receipt Version="1.0" ReceiptDate="2013-04-09T08:34:52Z" CertificateId=""
  ReceiptDeviceId="db2e1812-c704-4ce9-82b0-dc0910476139">
<AppReceipt Id="843f7847-faf4-49bc-be5c-ae818216ad57"
  AppId="b1cd9500-216a-496a-b377-d8343aa36f32_16p6bmm8bp4fr"
  PurchaseDate="2013-04-09T08:34:45Z" LicenseType="Full" />
<ProductReceipt Id="673f1d2a-2518-41e4-883f-b0cb4d2d8f09"
  AppId="b1cd9500-216a-496a-b377-d8343aa36f32_16p6bmm8bp4fr" ProductId="feature1"
  PurchaseDate="2013-04-09T08:34:47Z" ProductType="Durable" />
<Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
  <SignedInfo>
    <CanonicalizationMethod Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#" />
```

```

<SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256" />
<Reference URI="">
    <Transforms>
        <Transform
            Algorithm="http://www.w3.org/2000/09/xmldsig#enveloped-signature" />
    </Transforms>
    <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256" />
    <DigestValue>cdiU06eD8X/w1aGHeaGCG9w/kWZ8I099rw4mmPpvU=</DigestValue>
</Reference>
</SignedInfo>
<SignatureValue>
    SjR1xS/2r2P6ZdgaR9bwUsa6ZItYYFpKLJZrnAa3zkMy1biWjh9oZGGng2p6... (omitted)
</SignatureValue>
</Signature>
</Receipt>

```

The receipt fragment uses the following elements and attributes:

- **Receipt** This element contains information about apps and in-app purchases. Its attributes list includes:
 - **CertificateId** The certificate thumbprint used to sign the receipt.
 - **ReceiptDate** When the receipt was signed and downloaded.
 - **ReceiptDeviceId** Identifies the device used to request the receipt.
- **AppReceipt** This element includes all the information about the purchase of the app. Its attributes are:
 - **Id** The unique identifier of the purchase.
 - **AppId** A unique name assigned by the operating system to identify the package. Corresponds to the package family name. The *AppId* name is displayed in the Packaging section of the Package.appxmanifest file.
 - **LicenseType** The type of license purchased by the user.
 - **PurchaseDate** When the app was bought.
- **ProductReceipt** This element contains information about in-app purchases. Its attributes include:
 - **AppId** The app ID (see the previous list).
 - **ProductId** The product ID.
 - **PurchaseDate** When the product was purchased.
 - **ProductType** The duration of the license for the product. In theory, *ProductType* can assume two values: *Consumable*, meaning that the purchased product can be used and then can be purchased again, such as credits that can be used to buy items in a game, or *Durable*, meaning that the purchased product cannot be consumed and will last forever. However, the first option does not apply to Windows Store apps (only Windows Phone apps). Remember, though, that a product/feature can have an expiration date.

- **Signature** This element contains the digital signature for the receipt you can use to validate the receipt and make sure no tampering took place. (See Chapter 5, Objective 5.3, “Secure application data”, for further details.)

MORE INFO RECEIPT AUTHENTICITY VALIDATION

For details about receipt authenticity validation, visit <http://msdn.microsoft.com/en-us/library/windows/apps/jj649137.aspx>.



Thought experiment

Making the purchased feature available on different Windows 8 devices

In this thought experiment, apply what you’ve learned about this objective. You can find answers to these questions in the “Answers” section at the end of this chapter.

You have developed a Windows Store game that enables the users to play a few levels for free; then they have to buy additional levels in exchange for a small fee.

Because the app can be installed on up to five devices, what safe mechanism could you leverage to ensure that, after the extra content has been purchased, the user can play the new levels on a different Windows 8 device?

Objective summary

- Choose the business model that best suits your needs and use the *CurrentAppSimulator* to test your app’s behavior. Remember to replace any instance of *CurrentAppSimulator* with the *CurrentApp* instance before publishing your app in the Windows Store.
- Check the state of your app’s license by leveraging the *LicenseInformation* property of the *CurrentApp* class. Check the *IsActive* and *IsTrial* properties and act accordingly.
- Let your customers purchase your app and other features and products by using the *RequestAppPurchaseAsync* and *RequestProductPurchaseAsync* methods.
- Retrieve purchase receipts by calling *GetAppReceiptAsync* and *GetProductReceiptAsync*.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the “Answers” section at the end of this chapter.

1. What class should you use to test your app's behaviors in the local environment when dealing with license states and in-app purchases?

 - A. Always use the *CurrentApp* class, regardless of whether your app was published in the Windows Store.
 - B. Use the *CurrentApp* class, but only if your app was not published in the Windows Store yet.
 - C. Use the *CurrentAppSimulator* to test your app locally and then replace any reference to this class with the *CurrentApp* class before publishing your app.
 - D. There is no way to test your app's behaviors before it is published in the Windows Store.
2. In a feature-based trial, what does it mean when the *IsActive* property of the *LicensingInformation* class returns *false*?

 - A. The trial period has expired.
 - B. The app's license has been revoked or is missing.
 - C. The user has purchased the full version of the app.
 - D. The user can no longer purchase the app.
3. What is the meaning of the Boolean parameter passed to the *RequestAppPurchaseAsync* method when purchasing a trial app?

 - A. It indicates whether the app is performing a real transaction or a simulated purchase.
 - B. It indicates whether the app can be purchased for free.
 - C. It indicates whether your customer is purchasing all extra features and products, along with the full license version of the app.
 - D. It indicates whether you want the method to return the receipt for the app purchase.

Objective 6.2: Design for error handling

In this objective, you learn how to handle errors and exceptions in Windows Store apps written in HTML/JavaScript. In particular, you see how to deal with device capability errors and how to use promises to handle errors.

This objective covers:

- Design the app so that errors and exceptions never reach the user
- Handle device capability errors
- Handle promises errors

Designing the app so that errors and exceptions never reach the user

In all Windows Store apps, it is important that errors and exceptions are handled so they never reach the user directly. For Windows Store apps written in C# or Visual Basic, the common language runtime (CLR) provides a model for notifying errors in a uniform way. This model states that any operation should indicate its failure by throwing exceptions at runtime. These exceptions propagate up the chain of subsequent calls, until they are caught by a *try/catch/finally* block or go otherwise unhandled.

Most of these concepts also apply to Windows Store apps written in JavaScript. For example, when runtime errors occur, *Error* objects are thrown as exceptions. These exceptions can be caught and handled by *try/catch/finally* statements. It also means that most of the guidelines and best practices normally followed when using fully object-oriented languages, such as C# or VB, also apply to Windows Store apps written in JavaScript. The JavaScript apps need a few adjustments due to the particular nature of the language, especially when dealing with promises.

For example, it is important that you set up a *try/catch* block wherever there's a chance the app might not function as expected. This behavior can occur, for example, when accessing a resource that might be unavailable or missing, or when relying on types and functions over which you have no control, such as third-party libraries, files, remote services, and so on. You should also use a *finally* block whenever you need to release any resource and perform any necessary clean-up before exiting.

The following code shows a basic example of how exceptions are handled in Windows Store apps:

Sample of JavaScript code

```
app.onloaded = function () {
    mainMethod();
}

function mainMethod() {
    try {
        doSomeWork(10);
    } catch (e) {
        Debug.WriteLine(e.message);
    }
}
```

```
function doSomeWork(name) {  
    var result = name.toUpperCase();  
    return result;  
}
```

The *doSomeWork* method raises an exception when trying to convert the integer value provided as a parameter in an uppercased string. The exception is caught and handled by the *try/catch* block placed around the caller's body. You can also check the type of exception raised at runtime by using the *instanceOf* statement to decide the right strategy to handle specific types of exceptions, as shown in the following code excerpt.

```
function MainMethod() {  
    try {  
        sampleText.innerHTML = "<p>" + doSomeWork(0) + "</p>";  
    } catch (e) {  
        if (e instanceof TypeError) {  
            // do something  
        }  
        else {  
            // do something else  
        }  
    }  
}
```

NOTE JAVASCRIPT ERROR TYPES

The *TypeError* in the preceding code occurs when the actual type of an operand does not match the expected type. Other important types are *ConversionError*, which occurs whenever the attempt to convert an object into a different object fails; *RangeError*, which occurs when an argument that exceeded its maximum range is provided to a function; *ReferenceError*, which occurs when, for example, an expected reference is null; and *URIError*, which occurs when the code uses an invalid uniform resource indicator (URI), generally due to encoding or decoding issues.

In your methods, you should prefer throwing an exception to indicate that something went wrong instead of returning error codes or other messages. Error codes or null values do not continue to propagate through the system; thus they might go unnoticed. However, you can return *null* for common error cases (for example, a *getCustomerById* method that returns *null* when the customer is not found). The following code throws an exception when the provided parameter is not a *string* type:

```
function doSomeWork(name) {  
    if (typeof name != "string") {  
        throw new TypeError("Invalid parameter.");  
    }  
    var result = name.toUpperCase();  
    return result;  
}
```

Remember to clean up any intermediate results when throwing an exception. It should be safe for callers to assume that, when an exception is thrown from a method, the state of the object involved was not affected.

You can also create your own exception objects, either “deriving” them from the *Error* class or just from scratch. Listing 6-12 shows an example of the second scenario, in which a custom object is thrown by the *doSomeWork* method when an invalid parameter is provided.

LISTING 6-12 Using your own exception types

```
function MyCustomError(code, message) {
    this.code = code;
    this.message = message;
}

function mainMethod() {
    try {
        doSomeWork(0);
    } catch (e) {
        if (e instanceof MyCustomError) {
            // do something
        }
        else {
            // do something else
        }
    }
}

function doSomeWork(name) {
    if (typeof name != "string") {
        throw new MyCustomError("001", "Invalid parameter");
    }
    var result = name.toUpperCase();
    return result;
}
```

Windows Library for JavaScript (WinJS) provides a way to catch any exception that might slip unhandled through your code. (There are actually two ways, but because the second one is specifically designed to deal with promises, it will be discussed in the next section.) More precisely, the *WinJS.Application* object, which encapsulates application-level functionalities such as activation, storage, and application events, exposes a special event called *error*, which is raised any time an unhandled error has been raised by the code. Before learning how to handle this event from code, it might be interesting to see what happens when an unhandled exception is raised while debugging your app.

The next code excerpt shows how the *base.js* file, included in WinJS, handles this event.

(code omitted)

```
error: [
    function Application_errorHandler(e, handled) {
        if (handled) {
            return;
        }
    }
]
```

```

        (...)

        WinJS.Application._terminateApp(terminateData, e);
    }

}

[,  

 (code omitted)

var terminateAppHandler = function (data, e) {
    (code omitted)
    debugger;
    MSApp.terminateApp(data);
};
```

In the event handler, if the handled variable (of type *bool*) is *true*, the error has been handled by the code, whereas a value of *false* causes the debugger to pause (through the `debugger` statement) and then terminates the application.

If you want to handle the event yourself, you should subscribe to the *error* event as soon as possible in the app life cycle. (The `default.js` file should be a good place to start.) The following code excerpt shows an example of its usage:

```

var app = WinJS.Application;

app.onerror = errorListener;

function errorListener(err) {
    new Windows.UI.Popups.MessageDialog(err.detail.errCode).showAsync();
    return true;
}
```

When an unhandled error exception is raised, the event handler simply displays a message dialog to the user. Then, by returning *true*, the function signals that the exception has been handled by the code and that the app does not need to be terminated. If you omit the return statement (or if you explicitly return *false*, which has the same effect), the app will be terminated because the error is still considered unhandled.

However, before returning *true*, consider that there is no way to know whether keeping the application up and running is safe. Because you do not know where that exception comes from (otherwise you would have handled it in its appropriate place), parts of your app might be in an inconsistent state when the exception is raised. In other words, you can never be sure about what kind of error you are actually dealing with. For this reason, you should be careful when returning *true* to avoid termination. Instead, you should use the *Error* event handler to perform certain actions, such as logging the exception for tracking purposes or saving temporary data in local storage, before exiting the application.

NOTE THE `WINDOW.ONERROR` EVENT

As an alternative to `WinJS.Application.error`, you can use the classic `window.onerror` JavaScript event to catch unhandled exceptions raised during the execution of code. The two events work in a similar way. However, the `window.onerror` event is fired before the `WinJS.Application.error` event.

So far, we have been dealing with synchronous exceptions, that is, exceptions raised during the execution of synchronous code. However, most of the WinRT APIs are exposed through asynchronous methods, that is, through promises. Handling exceptions in asynchronous calls, especially when chaining together multiple promises, can easily become a challenging task. In these kinds of scenarios, you have to rely on mechanisms other than `try/catch` blocks to handle exceptions raised during `async` calls. (This is unlike C# and Visual Basic, in which the `async/await` keywords introduced with .NET Framework 4.5 have greatly simplified handling asynchronous exceptions.) For this reason, in the next section, we will dig a little deeper into the internal mechanisms of promises to see the proper way to handle errors raised during the execution of asynchronous code.

Handling promise errors

In Chapter 4, Objective 4.1, “Design for and implement UI responsiveness,” you learn that a promise is nothing more than an object that gives you a value sometime in the future. You don’t know exactly when that value will be available, but you can schedule code to be executed when your promise object has completed its operations. Compared to traditional callbacks, the advantage of using promises is that they provide a standard interface for dealing with asynchronous code.

In the case of a callback, the method signature defines which callback methods you can attach, but the number and order of arguments can be different for each callback. A promise exposes standard methods, such as `done` and `then`, that you can call on each and every promise object. This provides a consistent way of working with promises.

You have also learned in Chapter 4 that there is an important difference between `done` and `then` with respect to unhandled exceptions.



EXAM TIP

Because a `then` method returns a promise, the exception is silently captured as part of the state of the promise. Other chained `then` or `done` methods can then access this promise state and handle the exception. It also means that exceptions raised during the execution of a `then` method body will never reach the surface, and you won’t know that something went wrong during the performed asynchronous operations.

Let's review an example:

```
function getFile(folderName, fileName) {
    Windows.ApplicationModel.Package.current.installedLocation
        .getFolderAsync(folderName)
        .then(function (folder) {
            return folder.getFileAsync(fileName);
        })
        .then(function (file) {
            // do something with the file
        });
}
```

In this scenario, if an error occurs in one of the promises along the chain, the returning value will be another promise object with an error. For example, if you pass the wrong folder name to the *GetFolderAsync* method, the exception will be stored in the state of the first returned promise. The state of a promise can assume one of the values of the *Windows.Foundation.AsyncStatus* enum:

- **Canceled** The operation has been canceled.
- **Completed** The operation has been completed.
- **Error** The operation has encountered an error.
- **Started** The operation has been started.

When a promise with an error state is returned, the code checks whether an error function (also known as a fallback function) is provided in the next *then* method in the chain. Because the function is not present, a new promise with an error state is created and passed to the second *then* statement, and so on, each time passing the same line of code of the WinJS library, which creates a new promise in an error state with the specified value. The following line of code shows how the WinJS library first uses the *WinJS.Promise.wrapError* method to wrap a non-promise error value in a promise, and then returns the promise:

```
return WinJS.Promise.wrapError(cleanup(r));
```

If none of the chained *then* methods provides an error function, the exception will be simply lost. This way, the user doesn't know that something did not work properly, at least not at the time. This might sound like a good thing, but the problem is that your application code has no way to determine if everything worked as expected. Even worse, this can lead your app to an inconsistent state.

Because *then* and *done* accept the same three parameters (that is, a completed function to update the result after the promise has been fulfilled, an error function for handling errors, and a progress function to keep track of the progress), one option is to provide an error function to the last *then* method in the chain. This way, you can handle errors raised during the execution of the async operations at the end of the chain, as shown in the following code:

```
function getFile(folderName, fileName) {
    Windows.ApplicationModel.Package.current.installedLocation
        .getFolderAsync(folderName)
        .then(function (folder) {
```

```

        return folder.getFileAsync(fileName);
    })
    .then(function (file) {
        // do something with the file
    },
    function (err) {
        // handle error
    });
}

```

The problem with this code is that if an error occurs in the callback of the last async operation in the chain (the one dealing with the file, in the sample code), the exception will be lost.

This is why you should close the chain of promises with a *done* instead of *then*, as shown in the following snippet:

```

function getFile(folderName, fileName) {
    Windows.ApplicationModel.Package.current.installedLocation
        .getFolderAsync(folderName)
        .then(function (folder) {
            return folder.getFileAsync(fileName);
        })
        .done(function (file) {
            // do something with the file
        });
}

```

With a *done* clause at the end of a promise, if an error occurs during any of the performed async operations and no error function has been explicitly provided, *done* throws an exception. (You will see how to handle it in a moment.) The exception eventually bubbles up, giving you the opportunity to handle it. On the other hand, if you provide the *done* method with an error function, the error will be directly handled in the body of the fallback, so no exception will be raised. This is an important point to understand for the 70-482 exam.

The following snippet shows the *done* method providing an error function:

```

function getFile(folderName, fileName) {
    Windows.ApplicationModel.Package.current.installedLocation
        .getFolderAsync(folderName)
        .then(function (folder) {
            return folder.getFileAsync(fileName);
        })
        .done(function (file) {
            // do something with the file
        },
        function (err) {
            // handle error
        });
}

```

If something goes wrong during the execution of the *done* method (either in the completed function, the error function, or the progress function), an exception is *always* thrown, regardless of the fact that an error function has been provided. A *then* method would have “swallowed” the exception, as already mentioned.

Besides chaining promises by appending *then/done* methods at the end of multiple async calls, you can also nest promises, that is, call another async API in the callback of the previous one, concatenating them through a chain of *done* calls. For example, in the “Using custom license information” section of this chapter, we could have written the *LoadCustomSimulator* method using nested promises, as follows:

```
function loadCustomSimulator() {
    Windows.ApplicationModel.Package.current.installedLocation
        .getFolderAsync("data")
        .done(function (folder) {
            folder.getFileAsync("timed-trial.xml")
                .done(function (file) {
                    currentApp.licenseInformation.onlicensechanged = reloadLicense;
                    currentApp.reloadSimulatorAsync(file).done(
                        null,
                        function (err) {
                            //handle error
                        });
                }, function (err) {
                    //handle error
                });
            }, function (err) {
                // handle error
            });
        }, function (err) {
            // done
        });
}
```

The first *done* contains a call to another async method, *GetFileAsync*, and in the callback of this method, the code calls another async API, *ReloadSimulatorAsync*. The advantage of nesting promises is that each callback can access any variable declared before it. However, unless you want the error to bubble up, you must provide each *done* method with its own error handling function, and sometimes this is just extra work. Therefore, you should prefer flat promise chains to nested ones whenever possible.

To recap, when chaining multiple promises, you should always close the chain with a final call to the *done* method. Unlike *then*, *done* tends not to hide exceptions raised during the execution of asynchronous operations unless you provide *done* with an explicit error function. This way, you’ll know that something went wrong during the operations, and you’ll have the opportunity to handle the resulting exceptions. However, you must be aware that, when *done* throws an exception, it won’t be caught by a *try/catch* block placed around the chained promises, because the exception has been raised during an async operation. To handle this kind of exception, WinJS provides a specific *WinJS.Promise.error* event that, similar to the *WinJS.Application.error* event, can be used to handle promise errors (usually for logging purposes):

```
WinJS.Promise.onerror = promiseErrorHandler;
function promiseErrorHandler(err) {
    var ex = err.detail.exception;
    var promise = err.detail.promise;
}
```

The error event provides general error information, such as the exception, the promise in which it occurred, and the state of the promise (which is always error).

Handling device capability errors

Some of the capabilities declared in the application manifest file, such as microphone, camera, and location providers, are considered “sensitive devices” because they can reveal personal information that the user might want to be private (such her current location). The first time your Windows Store app needs to access a sensitive device declared in the application manifest, users have to grant their permission before the app can start using the corresponding feature. Figure 6-11 shows an app asking the user’s permission to use the webcam.

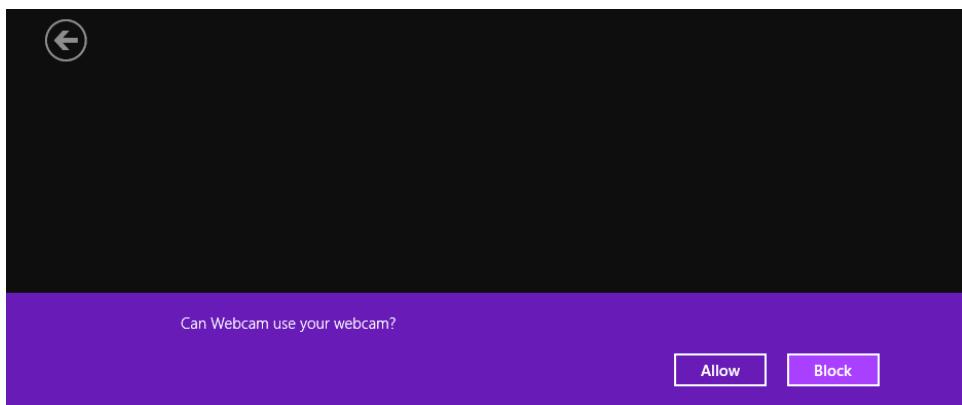


FIGURE 6-11 Asking permission to use a webcam

After the permission has been granted, a user can revoke it at any time by modifying the Privacy settings in the Permissions flyout, activated through the Settings charm. See Figure 6-12.

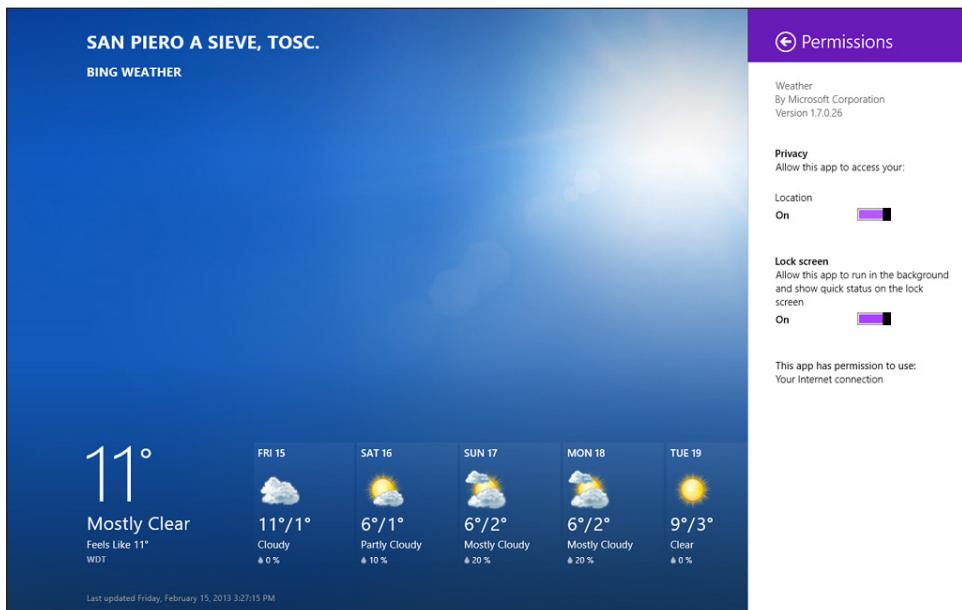


FIGURE 6-12 Privacy settings

NOTE SENSITIVE DEVICE USE

Permissions for Windows Store apps to use sensitive devices are specific for each app on a per-user basis.

To provide a good user experience when your app needs to access sensitive devices, you need to handle the errors that can occur when trying to access a disabled device capability. Some reasons your app might not be able to access a sensitive device are:

- The user does not grant access to the device capability in the dialog box.
- The user revoked the permission to access the device in the Settings charm.
- The device capability is not present on the system.

However, not all WinRT APIs behave in the same manner when trying to access capabilities for which the app does not have permission.

For example, if your app uses the *Windows.Media.Capture.MediaCapture* class to preview or capture photos, video, or audio, and the device is not present on the system, any call to the *InitializeAsync* method of the *MediaCapture* instance raises an exception. The same thing occurs if the device is present but the user did not grant the app permission to access the device capability (or the permission was revoked). In both cases, you have to handle the errors gracefully, informing the user that the device capability is not available on the system or that it cannot be used until permission is granted through the Settings charm. Listing 6-13 shows how to handle a simple scenario.

LISTING 6-13 Handling media capture errors

```
mediaCapture = new Windows.Media.Capture.MediaCapture();
var deviceInfo = Windows.Devices.Enumeration.DeviceInformation
    .findAllAsync(Windows.Devices.Enumeration.DeviceClass.videoCapture)
    .done(function (devices) {
        if (devices.length > 0) {
            var cameraSettings = new Windows.Media.Capture
                .MediaCaptureInitializationSettings();
            cameraSettings.videoDeviceId = devices[0].id;
            mediaCapture.initializeAsync(cameraSettings)
                .done(null, function (err) {
                    errorMessage.innerHTML = "Something went wrong when
                        initializing the camera. Please check the permissions.";
                });
        }
        else {
            errorMessage.innerHTML = "No camera device on the system";
        }
    }, function (err) {
        // handle error
});
```

Before trying to access a media capture device, the code searches for all available devices, looking for a video capture device (see Chapter 2, Objective 2.3, “Enumerate and discover device capabilities”). If something goes wrong when enumerating devices, the outer *done* method handles the error. If a video capture device is found, the code tries to initialize the device by calling the *InitializeAsync* method. If an exception is raised (for example, because the user did not grant the permission to access the camera), the error function displays a warning to the user.

On the contrary, if you use the *CaptureCameraUI* class and the webcam capability is turned off, invoking the *CaptureFileAsync* method does not raise an error. Instead, the Camera Capture UI displays a message indicating that the webcam capability is turned off and needs user intervention to be enabled, as shown in Figure 6-13.

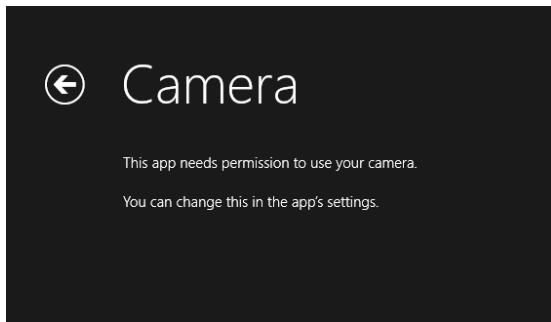


FIGURE 6-13 Message regarding permission to use camera

A different message displays if you do not have a camera connected to your PC, as shown in Figure 6-14.

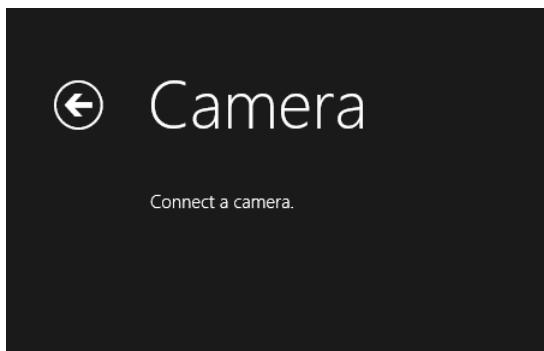


FIGURE 6-14 Message displayed when no camera is connected

Listing 6-14 shows an example of the *CaptureFileAsync* method. Because you do not need to check for permission in this case, the resulting code is simple.

LISTING 6-14 Using the *CaptureFileAsync* method

```
function takePicture_click(args) {
    var camera = new Windows.Media.Capture.CameraCaptureUI();

    camera.videoSettings.format = Windows.Media.Capture
        .CameraCaptureUIPhotoFormat.jpeg;

    camera.captureFileAsync(Windows.Media.Capture.CameraCaptureUIMode.photo)
        .done(function (file) {
            if (file) {
                // Show the picture to the user
            } else {
                // No picture captured
            }
        }, function (err) {
            // something went wrong
});}
```

Although the *CameraCaptureUI.CaptureFileAsync* does not throw any exceptions when the permission is missing, you should still provide an explicit function to handle the error in the *done* clause to handle any errors gracefully. That's because, permissions aside, something else can go wrong, such as an unpredictable malfunction of the webcam device.

MORE INFO FIRST CALL TO A DEVICE CAPABILITY

When dealing with device capabilities, remember that the first call to a device capability should be done from the UI thread so that the permission dialog can be shown to the user. Otherwise, the user cannot grant permission for access to the device capability. For this reason, you should never use a background task for the first use of the device.

Finally, in case of device capabilities that do not represent the main feature of the app, the official Microsoft guidelines suggest displaying the error message only when the user attempts to use the feature. This means you should not display an error message for a device capability that has not yet been requested by the user. Besides, the user should be aware of the loss of functionality, so try to make the message clearly visible to the user.



Thought experiment

Handling exceptions and security

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are developing a Windows Store app that handles sensitive user information, such as Social Security number, home banking account information, and other data that should remain confidential. This information can be exposed by incautious error-handling strategies, such as displaying the inner message or the stack trace of an exception on the screen.

What precaution could you adopt when implementing your error-handling strategy to ensure that your application does not leak sensitive information?

Objective summary

- Set up a *try/catch* block wherever you are trying to access a resource that might be unavailable or missing, or you are relying on types and functions over which you have no control. Use the *finally* block to release any unmanaged resources and perform necessary clean-up as soon as possible.
- Remember that, when an error occurs during the execution of asynchronous operations, a *then* method does not let it propagate outside the function, whereas a *done* method always throws an exception if an error function has not been provided.
- The *WinJS.Application.error* event handles exceptions that have not been handled by the application code, whereas the *WinJS.Promise.error* event can be subscribed to handle errors in promises. Use the corresponding event handler to perform actions such as logging the exception for tracking purposes or saving temporary data in local storage, before exiting the application. Return *true* to indicate that the exception should not be processed any further, but be aware that the unhandled exception might have left the application in an inconsistent state.
- A user must grant permission before a Windows Store app can access a sensitive device capability for the first time. Because your app might not be able to access a sensitive device, you need to handle the errors that can occur when trying to access a disabled or missing device capability.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the "Answers" section at the end of this chapter.

1. When is the *WinJS.Application.error* event raised by the framework?
 - A. Every time an exception is caught in a *try/catch* block
 - B. When a call to the *then* method does not provide any error-handling function
 - C. Any time an error occurs, no matter whether it is handled or not by the application code
 - D. When an exception is raised and there is no longer any possibility for the application code to catch the exception
2. What happens if, in a chain of *then* promises, an error occurs during the execution of the last *then* method in the chain?
 - A. If the last *then* method provides an error-handling function, the exception is stored in the state of the first returned promise, otherwise it goes unhandled.
 - B. An exception is thrown anyway, regardless of whether an error-handling function was provided.
 - C. The exception is stored in the state of the promise and doesn't reach the surface.
 - D. The *WinJS.Promise.error* event is raised, regardless of whether the exception was handled.
3. What happens when an application tries to invoke the *Windows.Media.Capture.MediaCapture.InitializeAsync* method after the user has revoked her permission to use the device?
 - A. An exception is raised.
 - B. A dialog box appears, prompting the user to grant permission to access the camera.
 - C. The camera simply won't work.
 - D. An error message is displayed on the screen.

Objective 6.3: Design and implement a test strategy

In this objective, you learn some introductory principles for testing your Windows Store app. You find out how the test framework for Windows Store apps differs from the classic Microsoft unit test framework for .NET applications.

At the time of this writing, Windows Store apps using JavaScript do not support the unit test framework in Microsoft Visual Studio 2012. Therefore, in this section, you learn how to

use the unit test framework to test business components, which you will probably write in C# (or VB.NET), leveraging its object-oriented capabilities.

This objective covers how to:

- Design a functional test plan
- Implement a coded UI test
- Design a reliability test plan, including performance testing, stress testing, scalability testing, and duration testing

Understanding functional testing vs. unit testing

Testing is a fundamental part of any software development process, including Windows Store apps. Testing is not just executing the code that you have written (whether it's just a small piece, one or more components, or the entire app) with the intent of finding bugs. It is also a process of verifying that your app meets all of the functional and nonfunctional requirements, and that it behaves as expected. One of the most commonly accepted distinctions between test methodologies is the one between functional testing and unit testing.

Functional testing

An application's functional requirements are the tasks the software is supposed to perform. Functional testing is a generic category that includes all tests that verify that the functional requirements of the software are working properly from a user's perspective. Functional testing requires prior identification of the functions (or tasks) that the software is supposed to perform and of the expected outcomes based on the functions' specifications.

For example, in a hotel booking application, a user asks to reserve a room for a specific night. A functional test might verify that the app actually sends the request to a remote service, the service processes the request and then sends a response back to the app, informing the user whether the booking operation succeeded. The app should be able to handle the scenario successfully and any issues that could affect the execution flow, such as when the Internet connection is not available, when someone else has reserved the same room, and so on.

MORE INFO NONFUNCTIONAL TESTING

Nonfunctional testing includes all tests performed on nonfunctional requirements, that is, every requirement that depends on external properties of the system, such as its performances under stress, security, and scalability. Examples of nonfunctional testing are security testing, usability testing, stress testing, and load testing. For further information about nonfunctional testing, you can start with the Wikipedia definition at http://en.wikipedia.org/wiki/Non-functional_testing.

There are three general approaches to functional testing:

- **Bottom-up approach** Integrates and tests lower-level components first. After all lower-level modules have been tested and their outcome verified, it is the turn of the higher-level components. The process is repeated until the components at the top of the hierarchy have been tested.
- **Top-down approach** Follows the opposite direction, where the higher-level components are integrated and tested first and then each branch of the flow is systematically tested until all lower-level components involved in the use case are tested.
- **Combination** Combines top-down testing with bottom-up testing to leverage the advantages of both the approaches.

INTEGRATION TESTING

A particular type of functional testing is represented by integration testing, whose purpose is to verify that the various components and resources of an application work together as expected. In this type of testing, software modules are coupled to form composite aggregates that can be tested to verify how the different parts integrate as a whole. Unlike unit testing, in which specific classes and methods are tested after they have been isolated and their dependencies replaced with stubs and mock objects, integration testing tests all code between the method under test and the lower-level components upon which the application was built. However, despite their differences, both unit and integration tests can usually be created using the same test framework. For example, in Visual Studio 2012, you can leverage the same Unit Test Library template for Windows Store apps (application or class libraries) to perform both unit and integration testing.

CODED UI TESTING

Another type of functional testing is UI testing, which enables you to test the application by driving it through the user interface. Conceptually, this is nothing different from manually testing the entire application by pressing F5 and then interacting with the elements of the user interface. However, Visual Studio 2012 enables you to automate this kind of test by recording the sequence of actions performed on the user interface and then generating the code to automatically repeat the same steps and verify the outcome (hence the name "coded UI testing," or CUIT). Unfortunately, Windows Store apps do not support this feature at the time of this writing. (You can refer to <http://support.microsoft.com/kb/2858781/en-us> for further clarification.) The only way to test your application is through manual interaction with the user interface. To keep track of manual tests, you can use the Microsoft Test Manager to describe them and then associate the test results to the build in Visual Studio Team Foundation Server.

NOTE REMOTE DEBUGGING

When manually testing the user interface of a Windows Store app, the Visual Studio remote debugger enables you to run, debug, and test an app remotely from a second computer running Visual Studio. Running on a remote device can be especially effective when the Visual Studio computer does not support functionality specific to Windows Store apps, such as touch, geolocation, and physical orientation. For further information about this topic, visit [http://msdn.microsoft.com/en-us/library/windows/apps/hh441469\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/windows/apps/hh441469(v=vs.110).aspx).

Unit testing

Functional testing, and especially integration testing, should not be confused with unit testing. In unit testing, you take the *smallest* piece of testable software in the application, *isolate* it from the rest of the code, and determine whether it behaves as expected by testing it as a *single unit*. Isolating the class or method under test requires you to remove external dependencies, that is, objects or components that the unit of code under test interacts with, but over which you have no control (such as file systems, databases, and remote services). To do that, you can replace external dependencies with stubs and mock objects.

Isolating a piece of software for unit testing involves adopting a design that encourages separation of concerns and coding against interfaces rather than concrete implementations. Isolation also means that unit tests should not depend on each other to succeed or fail or require to be executed in a specific order.

Unit testing can also be used for regression testing. Any time you have to modify your code, you can run the existing unit tests against the modified code to verify whether the changes affected the rest of the code. In this way, you can modify your code more easily because, when you add new features to your software, regression tests greatly reduce the risk of introducing new bugs.

NOTE UNIT TESTING AND SOFTWARE DEVELOPMENT METHODOLOGIES

Discussing the philosophies behind the notion of unit testing and enumerating the differences with other test strategies, as well as discussing the different methodologies of software development (such as test-driven and model-driven development), is beyond the scope of this book. If you are interested in design patterns, methodologies, and tools for testing, you can start by visiting <http://msdn.microsoft.com/en-us/library/bb969130.aspx>.

Implementing a test project for a Windows Store app

As of this writing, Visual Studio 2012 does not support test projects or the unit test framework for Windows Store apps written in JavaScript. However, you probably won't implement business components or web services clients directly by using a language such as JavaScript. You will probably choose a more sophisticated language such as C# or even VB.NET for business logic, complex data access components, and other strategic pieces of code. You will also likely create a Windows Metadata (WinMD) library for reusable components. As you learned in Chapter 1, JavaScript cannot be used to create such libraries.

Even if you don't feel confident with object-oriented languages, you might need to create the building blocks of the application using an object-oriented language such as C#, while leveraging HTML/JavaScript for the UI. For these reasons, this section includes an introduction to the test framework available for testing your app's logic and/or WinMD libraries. If you aren't familiar with C# code, don't worry. The sample presented in this section leverages the unit test framework for simple C# methods.



EXAM TIP

Windows Store apps do not use the classic Visual Studio unit test framework for .NET applications defined in the *Microsoft.VisualStudio.TestTools.UnitTesting* namespace. Instead, they use a framework specifically designed to work with Windows Store apps that resides in the *Microsoft.VisualStudio.TestPlatform.UnitTestFramework* namespace.

The *Microsoft.VisualStudio.TestPlatform.UnitTestFramework* namespace contains types and methods that provide specific support for testing Windows Store apps and relies on the same WinRT engine of a Windows Store app. In fact, the test code executes in the same sandboxed environment of a Windows Store app. Therefore, not only can you call the WinRT APIs from the test code, but a unit test project for Windows Store apps also includes a Package.appxmanifest file, the same kind of application manifest you would expect in a Windows Store app project. To test how your app interacts with certain resources or devices (such as user libraries, a webcam, or a microphone), you have to declare the corresponding capability in the application manifest of your unit test project, as you would in any Windows Store project.

There are other differences between the traditional test framework for .NET applications and the one specific for Windows Store apps. For example, in the newer framework, the *ExpectedExceptionAttribute* attribute is no longer supported. Instead, the framework introduces a generic method in the *Assert* class, named *ThrowsException<TException>*. This method explicitly states that raising an exception is the behavior you actually expect from the tested method. Another difference is the new *UITestMethodAttribute* attribute, which enables you to run unit tests on the main UI thread, without the need to use a *Dispatcher* for marshaling. Finally, the unit test framework for Windows Store apps provides basic support for data-driven testing, a particular form of functional testing where test input and output values are separated from the code. These new features are further discussed in this section.

NOTE VISUAL STUDIO 2012 UPDATE 2

Both the *ThrowsException* method and the *UITestMethodAttribute* attribute were introduced with Visual Studio 2012 Update 2. You can download the update from <http://www.microsoft.com/visualstudio/eng/visual-studio-update>.

Finally, the test framework for Windows Store apps does not include the Microsoft Fake framework shipped with Visual Studio 2012. For this reason, if you want to isolate your code by using stubs and fake objects, you must write your own code or use third-party libraries.

The first thing to do to implement tests for your app is to add a new Unit Test Library project template to the solution that hosts your Windows Store app, as shown in Figure 6-15. You must also add a reference to the Windows Store project that you want to test.

NOTE TEST CODE IN THE ASSEMBLY OF A DISTRIBUTED APP

You should never implement tests of any sort in the same project of your app. Including test code in the assembly that will be distributed via the Windows Store occupies valuable disk space, increases the download size of the app, and consumes bandwidth. It also makes future releases much harder to manage because it would inevitably lead to many different versions not only of your app but also of your test code, with inevitable impacts on any continuous integration strategies.

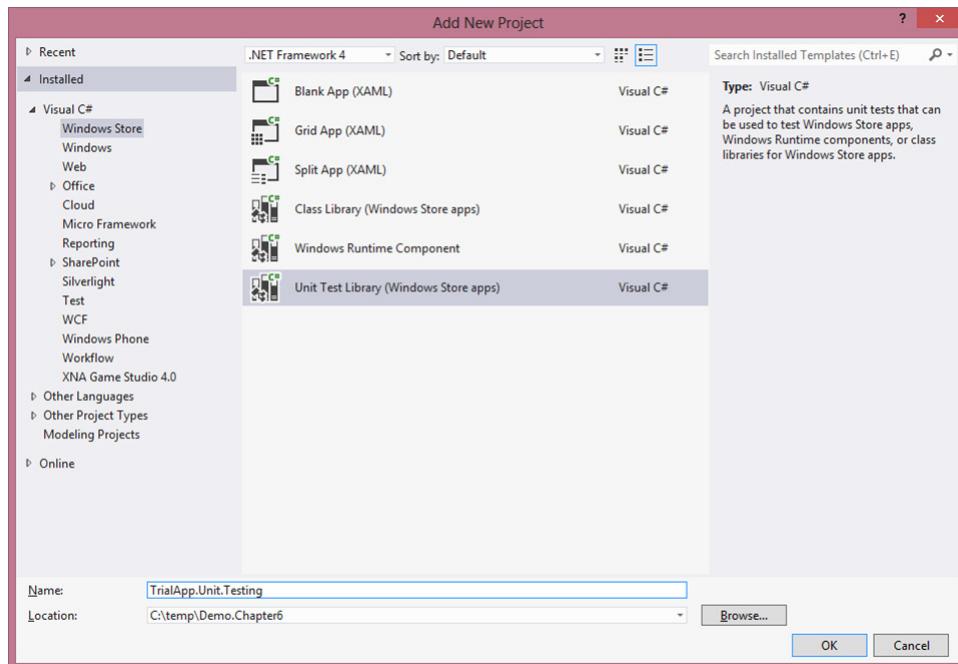


FIGURE 6-15 The Unit Test Library (Windows Store apps) template in Visual Studio 2012

Before you begin coding your first test, you must add a new class to test in your Windows Store project. The following code excerpt shows the complete definition of a sample *Utility* class that contains a trivial method that needs to be tested:

Sample of C# code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SimpleApplication
{
    public class Utility
    {
        public Double Divide(Double a, Double b)
        {
            return a / b;
        }
    }
}
```

In this example, the *Utility* class contains a single method named *Divide* that takes two doubles as parameters and returns the result of a simple division operation.

Now we can start to test this method. The first thing to be tested is that when you pass two doubles to the method under test, it returns the actual result of the performed operation. Listing 6-15 shows a possible implementation for this test.

LISTING 6-15 Unit testing the *Divide* method

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace SimpleApplication.Unit.Test
{
    [TestClass]
    public class UtilityTest
    {
        [TestMethod]
        public void Divide_TwoDoubles_ReturnsDouble()
        {
            // Arrange
            var expected = 2.4; // expected result
            var utility = new Utility(); // set up fixture

            // Act
            var actual = utility.Divide(4.8, 2.0); // actual result
        }
    }
}
```

```
// Assert
Assert.AreEqual(expected, actual); // verify condition
}
}
}
```

NOTE NAMING CONVENTIONS

If you are familiar with unit testing, you might have noticed that Listing 6-15 uses the traditional naming convention according to which class names follow the *<ClassUnderTest>Test* pattern. (This sample tests a class named *Utility*, so all the tests that concern this class are grouped into a unit test file named *UtilityTest*.) Single tests are named based on the following pattern: *<MethodUnderTest>_<StateUnderTest>_<ExpectedBehavior>*. *StateUnderTest* indicates the conditions under which the method is being tested (for example, whether you are passing the right parameters or invalid values). *ExpectedBehavior* enables you to state what behavior you expect from the method under the provided conditions (for example, to return a certain value or throw an exception of some sort). No matter what naming convention you choose for your test project, remember to follow the same standard for all tests, and be clear about what each test state is and what behavior you expect.

As in the classic test framework for .NET apps, the *TestClass* attribute is used to identify the groups of unit tests to run, whereas the *TestMethod* attribute indicates each unit test that needs to be run. In the sample, the test verifies that, provided two doubles as parameters, the method under test returns the right result. The body of the method also illustrates the so-called Three A's rule: you set up the object to be tested (*arrange*), exercise the method under test (*act*), and then make claims about the object (*assert*).

The *Assert* class contains methods to verify conditions in unit tests using *true/false* propositions. For example, the *AreEqual* method used in Listing 6-15 verifies whether two values are equal through a call to the *Object.Equals* method under the hood. The sample asserts that, given the provided parameters, the method should return a specific value, which is represented by the *expected* variable. If it does, the assertion would be *true* and the test would pass. (To verify whether two objects point to the same reference, you have to use one of the *AreSame* overload methods.)

If you run the unit test, the Test Explorer window in Visual Studio 2012 indicates that your app passed the test, indicated by a green check mark. See Figure 6-16.

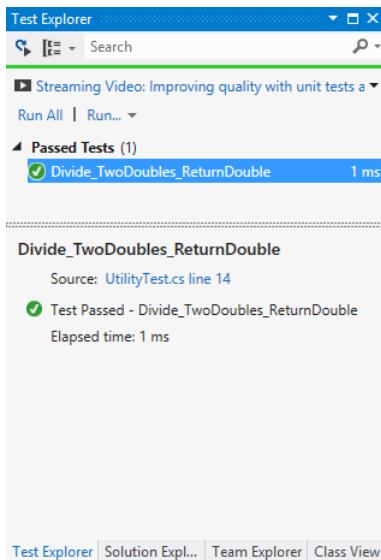


FIGURE 6-16 Unit test results

What happens if you pass different values to the method, such as a zero (0) as the second parameter? When you try to divide a double by zero, instead of the classic *DividedByZeroException*, you receive a particular value: *Infinity* (in two flavors: *PositiveInfinity* and *NegativeInfinity*). Therefore, in a second test, you can verify this behavior by calling the *IsInfinite* method of the *Double* class and check whether this method returns *true* by leveraging a different assert. The following code excerpt shows a possible unit test definition that uses the *Assert.IsTrue* method instead of the *Assert.AreEqual* method:

```
[TestMethod]
public void Divide_ZeroAsSecondParameter>ReturnsInfinite()
{
    // Arrange
    var utility = new Utility();

    // Act
    var actual = utility.Divide(4.8, 0.0);

    // Assert
    Assert.IsTrue(Double.IsInfinite(actual));
}
```

Besides verifying that a method behaves correctly when supplied with the right parameters, you also have to test what happens when you provide the wrong parameters or when the state of the object under test is not what it should be. For example, suppose that you are expecting a certain method to throw an exception of type *ArgumentNullException* when supplied with a *null* value as parameter. To test this behavior, you can use another type of assertion: the *ThrowsException<TException>* method, where *TException* indicates the type of exception expected. The following code shows an example of its usage:

```
[TestMethod]
public void GetCustomerName_EmptyId_ThrowsArgumentNullException()
{
    // Arrange
    var biz = new Biz();

    // Act & Assert
    Assert.ThrowsException<ArgumentNullException>(() => biz.GetCustomerName(""));
}
```

As mentioned previously, Visual Studio 2012 Update 2 introduced a new attribute to run code on the UI thread. When you create a unit test project for a Windows store app, tests marked with the *TestMethod* attribute do not run on the UI thread. Before the introduction of this new attribute, if you wanted to perform an action on the UI thread, you had to use a dispatcher. Now, all you have to do is to mark your test with the new *UITestMethod* attribute, as shown in the following code snippet:

```
[UITestMethod]
public void Some_UI_Test()
{
    //Some code to execute on UI Thread
}
```



EXAM TIP

Following the unit testing philosophy, each test needs to start in a well-known condition and clean up things at the end. In this way, the test is not affected by other tests or external conditions.

You can centralize the code for setting up all the things that must be in place before running the tests (also known as a “test fixture,” or just “fixture”), such as creating fake objects to supply to the methods under test or opening a connection to a database, by using the *ClassInitialize* and *TestInitialize* attributes. After the fixture is used, it can be torn down by leveraging the *ClassCleanup* and *TestCleanup* attributes, respectively.

The difference between these two kinds of attributes is that methods with the *ClassInitialize* and *ClassCleanup* attributes are called, respectively, before and after *all* the tests in a test battery are run. On the contrary, methods marked with the *TestInitialize* and *TestCleanup* attributes are called, respectively, before and after *each* unit test. Therefore, if you need to share a fixture among different unit tests, you can set up the fixture in a method marked with the *ClassInitialize* attribute, share it among different tests, and then tear down the fixture in the *ClassCleanup* method. If you want to re-create a fresh fixture for each single test (which represents the best case scenario, because it avoids test coupling), you can mark the method with the *TestInitialize* attribute, while the method responsible for tearing down the resource after each test will be marked with the *TestCleanup* attribute.

NOTE ASSEMBLYINITIALIZE/ASSEMBLYCLEANUP ATTRIBUTES

You can also use the *AssemblyInitialize*/*AssemblyCleanup* attributes to identify the methods that contain code to be used before and after *all* tests included in the assembly have run.

The following code snippet shows an example of the *TestInitialize/TestCleanup* pattern, in which a simple object is created before each test and then destroyed:

```
[TestInitialize]
public void TestSetup()
{
    // Set up a fresh fixture that will be used by each unit test
}

[TestMethod]
public void Test_Using_Fresh_Fixture()
{
    // Use the fixture
}

[TestCleanup]
public void TestTearDown()
{
    // Tear down the fixture before moving to the next unit test
}
```

Finally, as mentioned previously, the unit test framework for Windows Store apps introduces a basic support for lightweight data-driven testing, a particular test strategy that enables executing the same test method with different input values. The next code excerpt shows an example of its usage:

```
[DataTestMethod]
[DataRow("25892e17-80f6-715f-9c65-7395632f0223", "Customer #1")]
[DataRow("a53e98e4-0197-3513-be6d-49836e406aaa", "Customer #2")]
[DataRow("f2s34824-3153-2524-523d-29386e4s6as1", "Customer #3")]
public void GetCustomerName_RightID_ReturnExpected(String id, String customerName)
{
    var biz = new Biz();

    var actualCustomer = biz.GetCustomerName(id);

    Assert.AreEqual(customerName, actualCustomer);
}
```

This particular test strategy is based on two new attributes: *DataTestMethod* (which takes the place of the standard *TestMethod* attribute) and *DataRow*. In addition, the signature of the test method is different because it now accepts (in our sample) two strings as parameters. The *DataRow* attributes define the data set that will be passed to the test method as parameters. Then the test method, along with the associated setup and teardown methods, is executed several times, each time using the values provided by a different attribute. The first value is a string representing the customer ID, in our sample, which corresponds to the first parameter

accepted by the test method and represents the value that will be supplied to the method under test (*GetCustomerName*). The second value passed to the test method represents the return value that we expect the method under test will return.



Thought experiment

Checking the user's permission

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You have the following method that needs to be tested:

```
public async Task<String> LoadDocumentSample(String fileName)
{
    var folder = KnownFolders.DocumentsLibrary;
    var file = await folder.GetFileAsync(fileName);
    var text = await Windows.Storage.FileIO.ReadTextAsync(file,
        Windows.Storage.Streams.UnicodeEncoding.Utf8);
    return text;
}
```

As the first test, you want to be sure that if you provide the name of an existing text file in the user's Documents library, the method returns the actual content of the file (just "some content," in this example). The following code shows the implementation of your first integration test:

```
[TestMethod]
public void SaveDocumentSample_ExistingFileName_
    ReturnsExpectedText()
{
    var utility = new Utility();
    var t = utility.LoadDocumentSample("document.txt");
    Assert.IsTrue(t.Result == "some content");
}
```

You run the test but the test fails due to an *AggregateException* raised during the execution of the asynchronous operation. What is the first thing you should check in your test project to solve the problem?

Objective summary

- When developing a Windows Store app, consider adopting a functional test plan to verify that all the functional requirements of the software are working properly from the user's perspective.
- Adopt a unit test plan to verify whether the single units of software, isolated from any external dependency, behave as expected.

- Unit testing is highly useful for business components, data access components, and web service agents. These components are most often written in an object-oriented programming language, and are used from a UI written in HTML/JavaScript.
- The unit test framework for Windows Store apps is in the *Microsoft.VisualStudio.TestTools.UnitTesting* namespace, which is different than the “classic” test framework for .NET applications. This new namespace contains types and methods that provide specific support for testing Windows Store apps and relies on the same WinRT engine of a Windows Store app.
- In the unit test framework for Windows Store app, the *ExpectedException* attribute is no longer supported. Use the *Assert.ThrowsException<TException>* method instead.
- Implement your data-driven test by using the *DataTestMethod* attribute.
- When implementing a unit test, remember to follow the Three A’s rule: set up the object to be tested (arrange), exercise the method under test (act), and then make claims about the object (assert).

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the “Answers” section at the end of this chapter.

1. Which of the following statements is incorrect when referring to unit testing?
 - A. Small units of software are tested after they are isolated from external dependencies.
 - B. Tests should never depend on each other to succeed or fail, nor should they require to be executed in a specific order.
 - C. Unit testing encourages separation of concerns and coding against interfaces.
 - D. Different components and resources of an application are tested together to verify they work as expected.
2. When are methods marked with the *TestInitialize* and *TestCleanup* attribute executed?
 - A. Before and after all tests in the assembly are executed
 - B. Before and after all tests belonging to the same class are executed
 - C. Before and after each test is executed
 - D. Before and after all the tests marked with the *DataTestMethod* attribute are executed

3. Which of the following features is not supported in the unit test framework for Windows Store apps?
 - A. `Assert.ThrowsException<TException>` method
 - B. `TestMethod` attribute
 - C. `DataTestMethod` attribute
 - D. `ExpectedException` attribute

Objective 6.4: Design a diagnostics and monitoring strategy

In this section, you learn how to profile your Windows Store app by using the analysis tools provided by Visual Studio 2012 to detect performance issues that could affect your app. You also learn how to use Event Tracing for Windows (ETW) to log the most significant events your app will encounter during its life cycle and how to implement different strategies for logging those events. Finally, you explore the Quality reports provided by the Windows Store, which identify the most common issues affecting your app.

This objective covers how to:

- Design profiling, tracing, performance counters, audit trails (events and information), and usage reporting
- Decide where to log events (local vs. centralized reporting)

Profiling a Windows Store app and collecting performance counters

A Windows Store app is essentially a client app that executes in a sandboxed and highly responsive environment. In some scenarios, however, you (or, in the worst case, your customers) might find that your app is performing slowly. The reasons behind poor performance can vary: inefficient code that slows down your app, calls to third-party libraries or remote services that require too much time to complete, complex calculations that consume too much CPU, and so on.



EXAM TIP

Visual Studio 2012 profiling tools for Windows Store apps enable you to observe and record metrics about the behavior of your app by using a sampling method that collects information from the CPU call stack at regular intervals, navigating through the execution paths of your code and evaluating the cost of each function. It's also important to note that a Windows Store app does not support gathering performance counters programmatically, as you can do in a traditional .NET application. You have to rely on the provided profiling tools only.

Visual Studio provides a suite of tools for debugging, profiling, testing, and collecting information about Windows Store apps and the overall execution environment. The profiling tools enable you to observe and record performance counters about the behavior of your app by using a sampling method that collects information from the CPU call stack at regular intervals, navigating through the execution paths of your code and evaluating the cost of each function. In other words, a sampling tool collects statistical data about the work performed by an application, taking snapshots of the system at regular intervals, without modifying any code.

However, many profiling tools and options are not available in Visual Studio 2012 for Windows Store apps, as of this writing. The following list refers to the unsupported features for Windows Store apps written in JavaScript, but most limitations also apply, with some differences, to Windows Store apps written in other languages:

- **Instrumentation profiling** There are two main profiling techniques: sampling and instrumentation. Sampling is the only option available in a Windows Store app, although with some limitations. (Sampling options such as setting the sampling event and timing interval, or collecting additional performance counter data, are not available in a Windows Store app.) On the other hand, an instrumentation profiler follows a more invasive approach because it injects specific tracing code at the beginning and at the end of each function. This tracing code (also known as “tracing markers” or “probes”) enables the profiling tool to record each time the execution flow enters and exits an instrumented function.
- **Concurrency profiling** This profiling method collects detailed information from the call stack each time competing threads are forced to wait for access to a shared resource (resource contention). It also provides useful information about how an application interacts with the overall environment, enabling you to identify performance bottlenecks, synchronization issues, and so on.
- **.NET memory profiling** This profiling method collects detailed information about memory allocation and garbage collection. Note that Visual Studio 2012 Update 1 introduced the Memory Analysis tool and a new tool for analyzing UI responsiveness for Windows Store apps using JavaScript. These tools help developers collect data about memory usage and detect memory leaks and are described in the next section.

- **Tier interaction profiling (TIP)** This profiling method collects information about ADO.NET function calls to a SQL Server database. This profiling is not available for Windows Store apps because an app does not have access to the *System.Data** namespaces and has to rely on remote services (or local storage) to consume data.

Now that you understand which features and options are not available for a Windows Store app, let's examine what you can use to identify possible performance issues in a Windows Store app using JavaScript.

MORE INFO WINDOWS PERFORMANCE TOOLKIT (WPT)

If you want to collect broader information about your app's performance, you can use the Windows Performance Toolkit (WPT). This tool contains performance analysis tools that are useful to a broad audience, including general application developers, driver developers, hardware manufacturers, and system builders. These tools are designed for measuring and analyzing system and application performance on Windows 8. You can download the toolkit from <http://msdn.microsoft.com/en-us/performance/cc825801.aspx>. Discussions about this tool, as well as about any other third-party tool for performance analysis, are beyond the scope of this book.

To help you understand the profiling tools available in Visual Studio 2012, let's analyze a simple two-layered application that includes a WinMD component written in C#, whose only purpose is to retrieve a list of customers and return them to the UI.

The following snippet shows a simple HTML page you can use as a reference for testing purposes:

Sample of HTML code

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>Profiling Sample JS</title>

    <!-- WinJS references -->
    <link href="/Microsoft.WinJS.1.0/css/ui-dark.css" rel="stylesheet" />
    <script src="/Microsoft.WinJS.1.0/js/base.js"></script>
    <script src="/Microsoft.WinJS.1.0/js/ui.js"></script>

    <!-- ProfilingSampleJS references -->
    <link href="/css/default.css" rel="stylesheet" />
    <script src="/js/default.js"></script>
</head>
<body>
    <button id="btnGetCustomers">Get Customers</button>
    <div id="customerList"></div>
</body>
</html>
```

In the button click's event handler, the code retrieves the collection of customers by calling a fake business layer (wrapped in a WinMD component) and displays it on the screen, as shown in the following code excerpt:

Sample of JavaScript code

```
app.onload = function (args) {
    btnGetCustomers.addEventListener("click", getCustomers_click);
}

function getCustomers_click(args) {

    var biz = new WinMD.Biz.FakeBiz();
    var customerList = biz.getCustomers();
    var count = customerList.length;

    for (var i = 0; i < count; i++) {
        displayCustomer(customerList[i]);
    }
}

function displayCustomer(customer) {
    customerList.innerHTML += "<p>" + customer.name + "</p>";
}
```

NOTE EXAMPLE CODE

You would never use this code in a real app. The code freezes the user interface until the list of customers has been retrieved. The sole purpose of this code is to illustrate usage of the profiling tools in Visual Studio 2012.

The *FakeBiz* class is a business component that, in this sample, is written in C# to present an architecture similar to a real application. The business layer component, wrapped in a WinMD library, contains only the *GetCustomers* method that retrieves the collection of customers by simulating a time-consuming call to a remote service through an empty loop. Listing 6-16 shows the complete C# code for the *FakeBiz* class.

LISTING 6-16 Complete code for the *FakeBiz* class

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace WinMD.Biz
{
    public sealed class FakeBiz
    {
        public IList<Person> GetCustomers()
        {
            return this.SimulateRemoteServiceCall();
        }
    }
}
```

```

private List<Person> SimulateRemoteServiceCall()
{
    for (int i = 0; i < Int32.MaxValue; i++)
    {
        // code omitted
    }

    return new List<Person>()
    {
        new Person() { Name = "Roberto Brunetti" },
        new Person() { Name = "Vanni Boncinelli" },
        new Person() { Name = "Luca Regnicoli" },
        new Person() { Name = "Katia Egiziano" },
        new Person() { Name = "Paolo Pialorsi" },
        new Person() { Name = "Marco Russo" },
    };
}

public sealed class Person
{
    public String Name { get; set; }
}
}

```

To analyze the code in Visual Studio 2012, click the **Start Performance Analysis** item in the Debug menu, as shown in Figure 6-17. Visual Studio starts the profiling tools and launches the application.

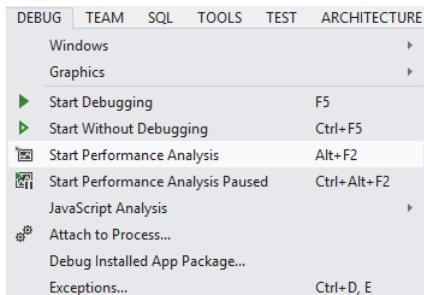


FIGURE 6-17 Selecting Start Performance Analysis in the Debug menu

NOTE PROFILING WINMD COMPONENTS

To collect performance data for C#/Visual Basic/C++ components of a JavaScript app, in the Debugger Type list on the Debugging property page of the JavaScript project, choose any item except Script Only.

In the app's default page, if you click the Get Customers button and wait until the collection of customers appears on the screen, the profiler records the operations the code performs under the hood. At the end of the process, you can go back to Visual Studio and click the Stop profiling link shown in Figure 6-18.

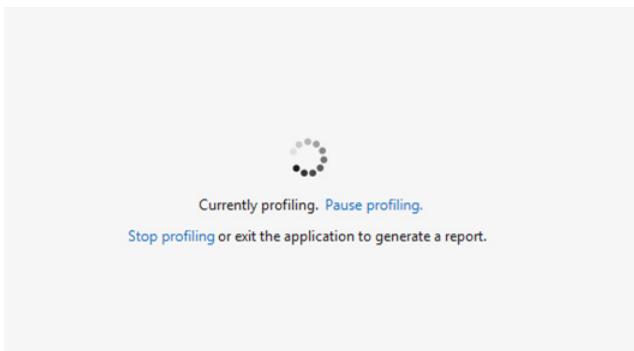


FIGURE 6-18 The Stop profiling link

After you stop the profiling analysis, Visual Studio 2012 generates a report containing all information collected during the sampling. Figure 6-19 shows the generated report.

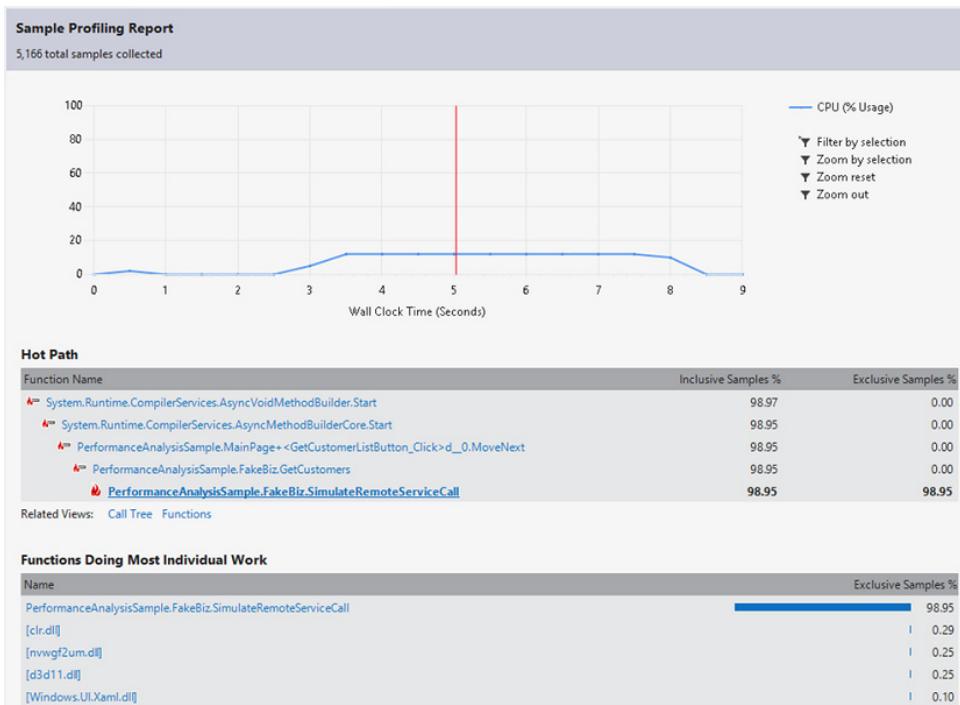


FIGURE 6-19 A profile analysis report

The report contains data collected during the execution of your app. (The data displayed in your report might differ.) The upper part of the screen is occupied by a chart showing the percentage of CPU usage during the execution of the app. You can select a specific area in the chart, for example, a spike indicating a burst in CPU usage, to zoom or filter the samples collected within a specific interval. Just below the CPU usage chart, the Hot Path shows the most expensive call paths in terms of CPU usage. These hot paths are highlighted with a flame-shaped icon by the function name. Each function call included in the path has two indicators: inclusive samples and exclusive samples.



EXAM TIP

The inclusive sample percentage (or inclusive time) indicates the CPU time spent to complete a specific function, including the time spent waiting for any other function that it might call. The exclusive samples percentage (or exclusive time) does not take into account the time spent waiting for other functions called from the current one to complete, but simply indicates the amount of time consumed by the inner work of a function.

In this sample, the *SimulateRemoteServiceCall* method shows an impressive 98.95 percent value in both indicators (inclusive and exclusive CPU usage), which means that almost all the CPU time consumed by the app has been used by this method. All the other functions show an inclusive time of 98.95 percent and an exclusive time approximately close to zero. That means that all they did was wait for the *SimulateRemoteServiceCall* method of the WinMD component to complete.

The Function Doing Most Individual Work section shows which individual functions have consumed most of the CPU time. In this case, it shows only the exclusive time consumed by each function, that is, without considering the time spent waiting for other functions to complete. You can navigate through the collected data by using more specific views, such as the Call Tree view illustrated in Figure 6-20, which includes more information about the sampled execution paths.

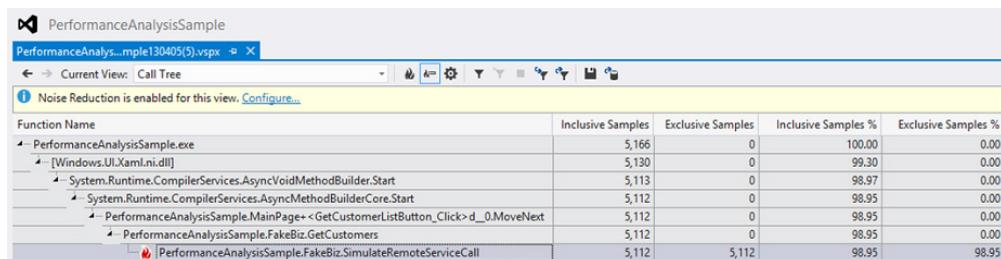


FIGURE 6-20 Call Tree view

If you click a function name in one of the preceding views, you can get more details about that function. Figure 6-21 shows the Function Details view for the *FakeBiz.GetCustomers* method.

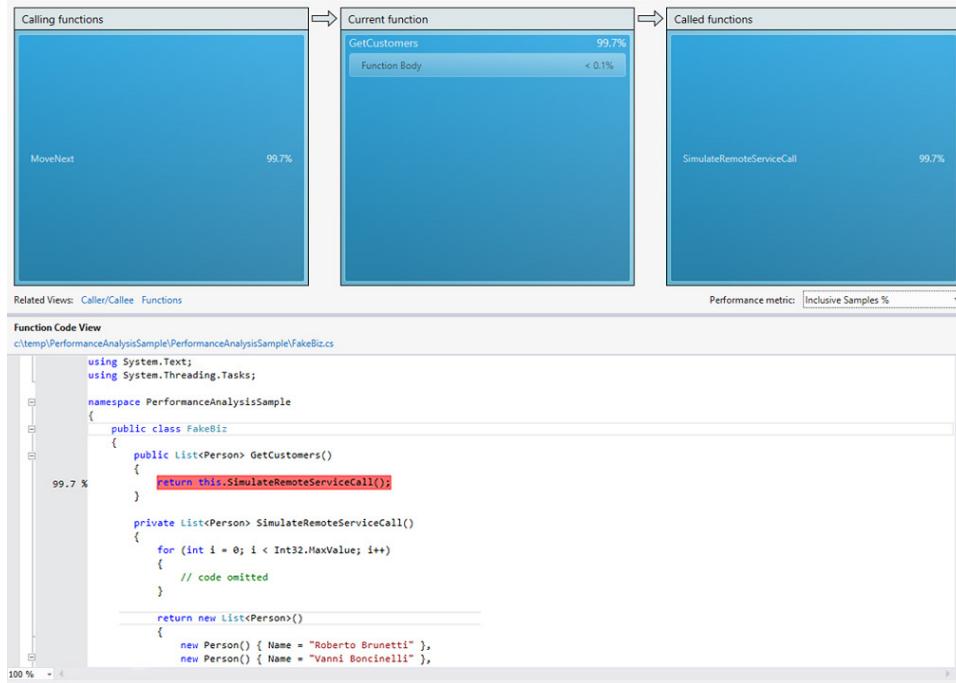


FIGURE 6-21 Function Details view

The upper part of the screen displays the inclusive time allocated along the path that includes the current function. The bars illustrate the relationships among the currently selected function (in this case, the `GetCustomers` method, which corresponds to the Current function bar), the calling functions that executed the selected function (shown in the Calling functions bar), and the selected function and the functions that were called by it (the `SimulateRemoteServiceCall` method, which corresponds to the Called functions bar). The size of each bar indicates the relative amount of the total execution time that the corresponding function has consumed.

The Called functions bar indicates that 99.7 percent of the CPU time has been consumed by the `SimulateRemoteServiceCall` method invoked inside the `GetCustomers` method, while the body of the current method itself consumed less than 0.1 percent of the CPU time to accomplish its internal work.

The Function Code View pane in the lower part of the screen shows the lines of code that caused the bottleneck (highlighted). If you click the `SimulateRemoteServiceCall` method, the Function Code View pane displays additional details that can help you isolate the problem. In Figure 6-22, the bottleneck represented by the `for` block is highlighted and indicates the CPU consumed by each line of code.

```

private List<Person> SimulateRemoteServiceCall()
{
    for (int i = 0; i < Int32.MaxValue; i++)
    {
        // code omitted
    }

    return new List<Person>()
    {
        new Person() { Name = "Roberto Requetti" }
    };
}

```

FIGURE 6-22 The for block

Another useful view, especially when your app uses a multitier architecture or third-party libraries, is the Modules view, which displays details of the profiling data grouped by modules. Figure 6-23 shows some details of the Modules view for the sample application.

Name	Inclusive Samples	Exclusive Samples	Inclusive Samples %	Exclusive Samples %
PerformanceAnalysisSample.exe	5,132	5,112	99.34	98.95
PerformanceAnalysisSample.FakeBiz.SimulateRemoteServiceCall	5,112	5,112	98.95	98.95
PerformanceAnalysisSample.App..ctor	1	0	0.02	0.00
PerformanceAnalysisSample.App.GetXamType	4	0	0.08	0.00
PerformanceAnalysisSample.App.OnLaunched	12	0	0.23	0.00
PerformanceAnalysisSample.FakeBiz.GetCustomers	5,112	0	98.95	0.00
PerformanceAnalysisSample.MainPage..ctor	6	0	0.12	0.00
PerformanceAnalysisSample.MainPage.Connect	1	0	0.02	0.00
PerformanceAnalysisSample.MainPage.GetCustomerListButton_Click	1	0	0.02	0.00
PerformanceAnalysisSample.MainPage.InitializeComponent	4	0	0.08	0.00
PerformanceAnalysisSample.MainPage.<GetCustomerListButton_Click>d_0.MoveNext	5,112	0	98.95	0.00
PerformanceAnalysisSample.PerformanceAnalysisSample_XamlTypeInfo.XamlTypeInfoProvider.Activate_0_MainPage	6	0	0.12	0.00
PerformanceAnalysisSample.PerformanceAnalysisSample_XamlTypeInfo.XamlTypeInfoProvider.CreateXamlType	1	0	0.02	0.00
PerformanceAnalysisSample.PerformanceAnalysisSample_XamlTypeInfo.XamlTypeInfoProvider.GetXamlTypeByName	1	0	0.02	0.00
PerformanceAnalysisSample.PerformanceAnalysisSample_XamlTypeInfo.XamlTypeInfoProvider.GetXamlTypeByType	1	0	0.02	0.00
PerformanceAnalysisSample.PerformanceAnalysisSample_XamlTypeInfo.XamlUserType.ActivateInstance	6	0	0.12	0.00
PerformanceAnalysisSample.Program..<Main>b_0	1	0	0.02	0.00
PerformanceAnalysisSample.Program.Main	5	0	0.10	0.00
System.Runtime.CompilerServices.AsyncMethodBuilderCore.Start	5,112	0	98.95	0.00
System.Runtime.CompilerServices.AsyncVoidMethodBuilder.Start	5,113	0	98.97	0.00
clr.dll	15	15	0.29	0.29
d3d11.dll	13	13	0.25	0.25
nvwgf2um.dll	31	13	0.60	0.25
Windows.UI.Xaml.dll	14	5	0.27	0.10
ntdll.dll	4	4	0.08	0.08
mscorlib.ni.dll	3	2	0.06	0.04
kernel32.dll	1	1	0.02	0.02
System.ni.dll	1	1	0.02	0.02
Windows.UI.Xaml.Lni.dll	5,135	0	99.40	0.00

FIGURE 6-23 Modules view

Using JavaScript analysis tools

Visual Studio 2012 Update 1 introduced two tools that specifically address Windows Store apps using JavaScript: the Memory Analysis tool, which collects data about memory usage, and the UI Responsiveness Profiler tool, which helps developers improve overall UI responsiveness and fluidity. Figure 6-24 shows the two tools in the Visual Studio 2012 Debug menu.

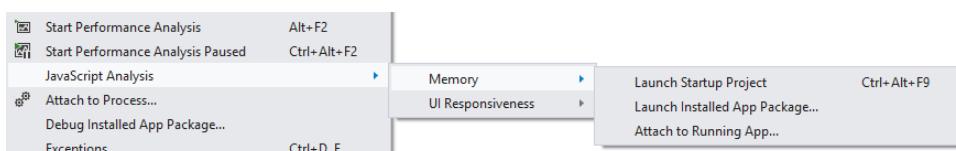


FIGURE 6-24 Menu items for the Memory Analysis and UI Responsiveness Profiler tools

Both tools present three initial options to the user:

- **Launch Startup Project** The tool begins analyzing the startup project of the current Visual Studio solution.
- **Launch Installed App Package** The tool prompts you to select an app already installed on your machine. Then Visual Studio launches the app and begins the diagnostic session. (This option is not supported when running the app on a remote machine.)
- **Attach to Running App** Using this option, you can attach an already running app, and then Visual Studio starts the diagnostic session. (This option is not supported when running the app on a remote machine.)

Memory Analysis tool

When the JavaScript memory analyzer starts, the Summary view provides a real-time memory usage graph for the app being executed, as shown in Figure 6-25.

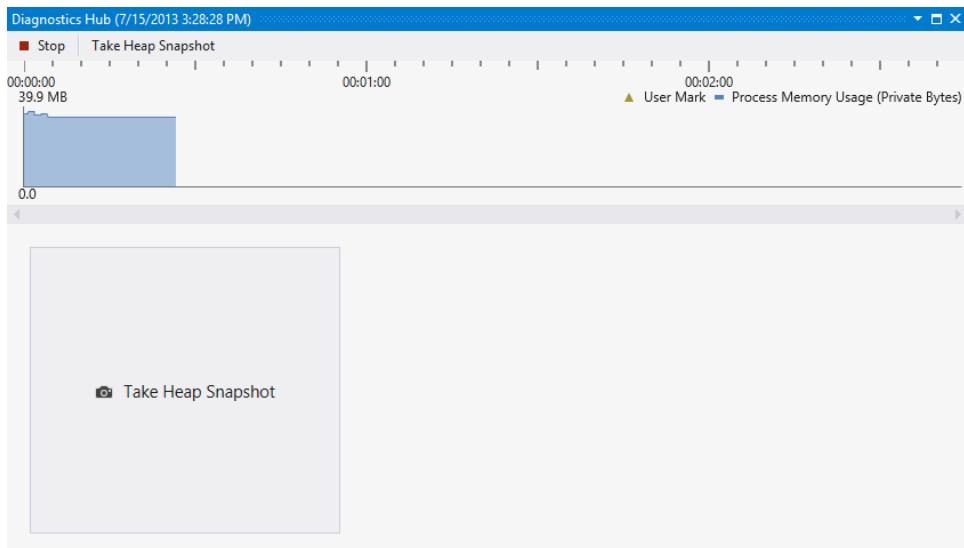


FIGURE 6-25 Memory usage graph

The graph shows the process memory usage during the app's life cycle, including native memory and JavaScript heap. Consider, though, that some of the memory shown in the graph is allocated by the JavaScript runtime and cannot be avoided.

During the execution of the app, you can take a snapshot of the heap, that is, of the current state of the memory, by clicking the Take Heap Snapshot button. Each snapshot appears in the Summary view as a distinct square box providing general information about the heap size at the time the snapshot was taken, and the number of objects created by your app. In Figure 6-26, the Summary view includes two different snapshots, taken a few seconds apart.

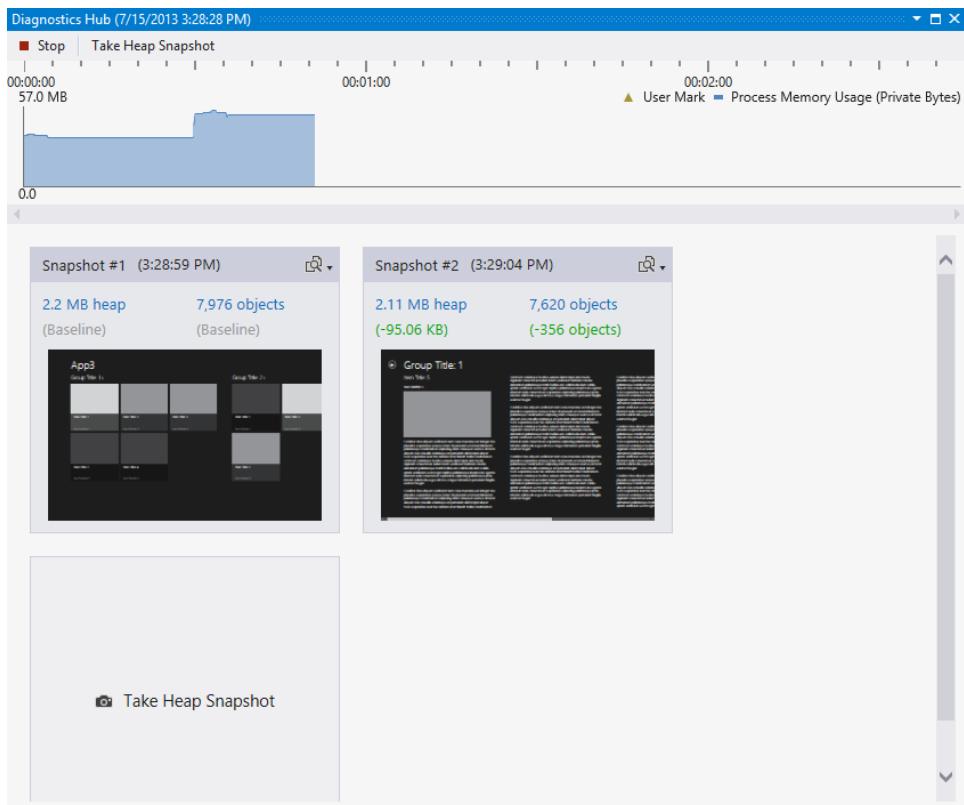


FIGURE 6-26 Two snapshots in Summary view

If you take multiple snapshots, each subsequent snapshot also provides additional information by comparing its data with those collected by the previous snapshots.

NOTE GARBAGE COLLECTOR

Before taking a snapshot, the Memory Analysis tool forces the garbage collector to run, ensuring consistency of data between two or more runs.

You can access detailed information about memory usage by clicking one of the snapshots. The information is grouped into the following categories:

- **Dominators** Shows all objects in the heap
- **Types** Shows the instance count and total size of objects, grouped by object type
- **Roots** Shows a tree of objects from the root objects through child references
- **DOM** Shows all markup elements from the Document Object Model (DOM)
- **WinRT** Shows all Windows managed and unmanaged objects that are referenced from the JavaScript app

NOTE SNAPSHOT DETAIL VIEW AND RETAINED SIZE

By default, all information displayed in the Snapshot Detail view is sorted by retained size, which is the sum of the object size plus the size of its child objects. In this way, the objects that consume the most memory are listed first.

Figure 6-27 shows an example of the Snapshot Detail view indicating, for each type, the size, the retained size, and the number of instances currently on the heap.

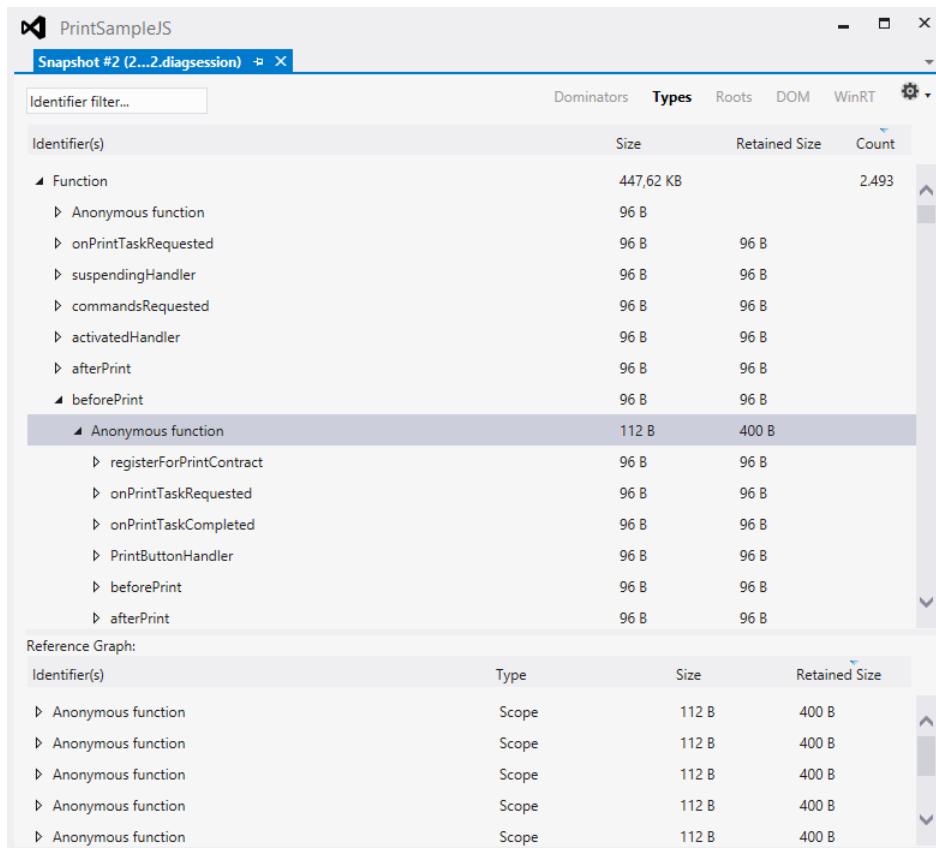


FIGURE 6-27 Snapshot Detail view

MORE INFO SNAPSOTS OF MEMORY HEAP

It is also possible to take snapshots of the memory heap programmatically. You can use the `console.takeHeapSnapshot` method to take a snapshot that appears on the Summary view of the tool, and use the `console.profileMark` method to display a profile mark in the memory graph while the diagnostic session is running. The following code excerpt shows the usage of the first method:

```
if (console && console.takeHeapSnapshot) {  
    console.takeHeapSnapshot();  
}
```

UI Responsiveness Profiler tool

The other JavaScript tool shipped with Visual Studio 2012 Update 1 is the UI Responsiveness Profiler. This tool is useful for isolating and resolving performance issues affecting UI responsiveness, such as synchronous JavaScript code, excessive Cascading Style Sheets (CSS) parsing or calculation, processor-intensive code, low frame rate, and so on.

Figure 6-28 shows the UI Responsiveness Profiler full view.

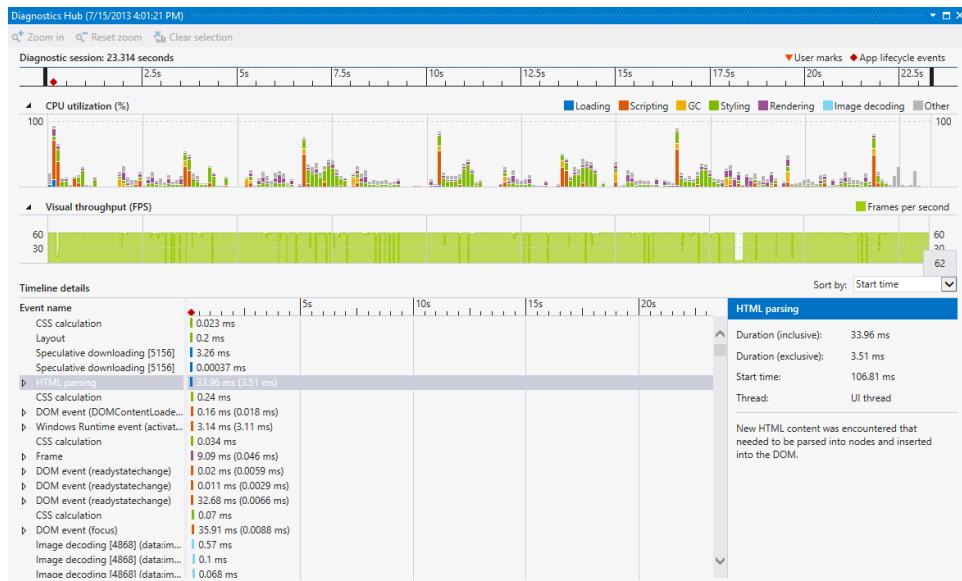


FIGURE 6-28 UI Responsiveness Profiler full view

The tool uses different colors to indicate distinct categories of events that are considered when generating the final report. The event categories are:

- **Loading** Indicates time spent retrieving app resources and parsing HTML and CSS when the app first loads. This can include network requests.
- **Scripting** Indicates time spent parsing and running JavaScript. This includes DOM events, timers, script evaluation, and animation. It also includes both user code and library code.
- **GC** Indicates time spent on garbage collection.
- **Styling** Indicates time spent parsing CSS and calculating element presentation and layout.
- **Rendering** Indicates time spent painting the screen.
- **Image decoding** Indicates time spent decompressing and decoding images.
- **Other** Indicates miscellaneous work on the UI thread, such as infrastructure work. This category is non-actionable work and is included because it affects CPU utilization values.

The diagnostic session timeline graph, in the upper part of the screen, provides a timeline of the app life-cycle events that occurred in a given interval. You can also isolate specific portions of the timeline to highlight specific moments of your app's life cycle.

The CPU utilization graph, just below the diagnostic session timeline, enables collecting information about the app's CPU consumption. This helps you identify the moments in which there is excessive CPU activity. The bars in the graph indicate peaks in CPU usage, and use different colors to represent different event categories. Figure 6-29 shows a detailed view of the CPU utilization graph.

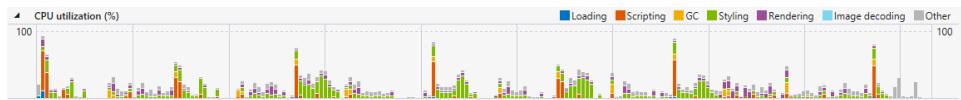


FIGURE 6-29 CPU utilization graph

The visual throughput graph shows the frames per second (FPS) for the app, enabling you to identify periods of time in which the frame rate has dropped. (Keep in mind that the reported FPS value might be different from the actual frame rate of the app.) Figure 6-30 shows a detailed view of the graph. The blank areas indicate many drops in the app's frame rate.



FIGURE 6-30 Frames per second in visual throughput graph

Finally, the timeline details graph, in the lower panel of the UI Responsiveness Profiler, provides information about the events that consumed the most CPU time during a selected

time interval. This graph can help you determine what triggered a particular event, and, for some events, how the event maps back to source code. Figure 6-31 shows a detailed view of the timeline details graph.

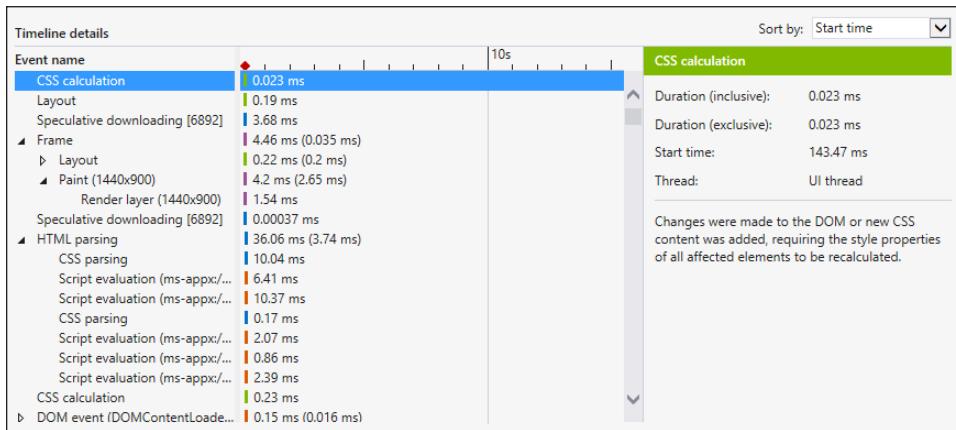


FIGURE 6-31 Timeline details graph, detailed view

The column on the right side of the screen includes some detailed information regarding a specific event, such as duration (inclusive and exclusive), start time, which thread was responsible for performing the operation, and some additional information about the operation.

MORE INFO UI RESPONSIVENESS PROFILER

You can find more detailed information about the UI Responsiveness Profiler at [http://msdn.microsoft.com/en-us/library/windows/apps/dn194502\(v=vs.120\).aspx](http://msdn.microsoft.com/en-us/library/windows/apps/dn194502(v=vs.120).aspx).

Logging events in a Windows Store app written in JavaScript

Windows Store apps written in C#/Visual Basic can use types and methods that enable capturing strongly-typed events for logging purposes. (This practice is also known as creating an “audit trail” or “audit log.”) However, at the time of this writing, a Windows Store app written in JavaScript cannot leverage the Event Tracing for Windows (ETW) mechanisms (such as event listeners) to keep track of any significant event encountered by your app.

That means you have to provide your own implementation of a logging system if you want to trace your app’s behavior and register the most relevant events encountered during your app’s life cycle. In the next section, you will learn how to take advantage of the entry point offered by the *WinJS.log* function for your custom logging system. In the “Tracing and event logging for WinMD components” section, you will find out how to leverage the ETW mechanisms from a WinMD component written in C#.

Event logging according to the WinJS library

By default, the WinJS library provides an empty logging mechanism, represented by the *WinJS.log* function. You can call the *WinJS.Utilities.startLog* function to log to the JavaScript console, which can be useful during application debugging. You can also provide your own implementation of this method to perform more sophisticated operations, such as sending the logged operations or errors to a remote web service, emailing reports, and so on.

Let's start with the simplest scenario, which enables you to log information that helps you debug your app locally. As mentioned previously, before starting the log operations, you have to initialize the *WinJS.log* object through a call to the *WinJS.Utilities.startLog* method, as shown in the following line of JavaScript code:

```
WinJS.Utilities.startLog();
```

After the log has been initialized, you can start logging any relevant event that occurred during your app's life cycle by using the following line of code:

```
WinJS.log("my message", "info", "custom");
```

The *log* function receives three strings as parameters: the message to log, one or more tags to categorize the message, and the type of event being logged.



EXAM TIP

The last line of code is a little unsafe, at least in situations where you cannot assume that the log has been initialized. An alternative is to create a global variable to track whether the log has been initialized. Another, more elegant, option is to check whether the log object exists before using it, as follows:

```
WinJS.log && WinJS.log("my message", "info", "custom");
```

After the code executes, you can inspect the content of the log in the Visual Studio 2012 JavaScript Console window, as shown in Figure 6-32.

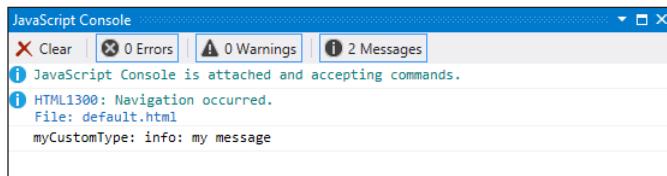


FIGURE 6-32 JavaScript Console window

This use of the *log* function can be useful when debugging your app locally, but it does not help track events after your app has been published in the Windows Store. However, it offers an interesting entry point for your custom implementation of a logging system. Using the *WinJS.Utilities.startLog* method, you can specify additional parameters that enable you to filter which events need to be logged. For example, consider the following code excerpt:

```
WinJS.Utilities.startLog({ type: "myCustomType", action: myCustomLog });
```

The *type* parameter is mandatory and indicates the type of events you want to log. The *action* parameter lets you specify which action should be performed when such an event has occurred. For example, when the following code is executed:

```
WinJS.log("my message", "custom", "myCustomType");
```

The *myCustomLog* function is called, receiving the three parameters: *message*, *tags*, and *type*. You can use this entry point to implement a custom logging system, as shown in the following code snippet:

```
function myCustomLog(message, tag, type) {  
    // your custom log implementation  
    // you can save the file in the local storage and/or send it to a remote service  
}
```

In this way, you can implement different strategies for different kinds of events. For example, you might decide to implement a log mechanism that sends only the most critical events occurred during your app's life cycle to a remote service, while using the default mechanism for less-relevant events.

However, if you want to achieve more fine-grained control over logging operations, without the limits of the standard mechanism, the best way is to provide your own implementation of the *log* function, as follows:

```
WinJS.log = mylogImplementation;  
  
(code omitted)  
  
function mylogImplementation(message, tags, type) {  
    // your log implementation  
}  
  
(code omitted)  
  
WinJS.log("message", "info", "type");
```

In this case, you do not need a prior call to the *WinJS.Utilities.startLog* method before using the log function.

Tracing and event logging for WinMD components

As mentioned previously, it is likely you will encapsulate application logic in a WinMD component written in C# or Visual Basic when developing a Windows Store app with some business logic. Although the *System.Diagnostics.Tracing* namespace provides types and members that enable you to create strongly-typed events to be captured by ETW, the Windows Runtime supports only a subset of the types available to a .NET application. For example, in a Windows Store app, the Windows Runtime does not support the *TraceListener* class (and its derived types, such as *DefaultTraceListener*, *TextWriterTraceListener*, and *EventLogTraceListener*). In

this section, you explore how to create, trace, and log events that enhance the overall quality of your software by keeping track of errors and other significant events encountered during your app's life cycle.

The first thing you need to do is provide your own *EventSource* implementation. The *EventSource* class provides the ability to create different event types that need to be traced. The C# code excerpt in Listing 6-17 shows an example of its implementation.

LISTING 6-17 Implementing the *EventSource* class

```
using System;
using System.Collections.Generic;
using System.Diagnostics.Tracing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SimpleEventTracing
{
    public class MyCustomEventSource : EventSource
    {

        [Event(1, Level = EventLevel.LogAlways)]
        public void WriteDebug(string message)
        {
            this.WriteEvent(1, message);
        }

        [Event(2, Level = EventLevel.Informational)]
        public void WriteInfo(string message)
        {
            this.WriteEvent(2, message);
        }

        [Event(3, Level = EventLevel.Warning)]
        public void WriteWarning(string message)
        {
            this.WriteEvent(3, message);
        }

        [Event(4, Level = EventLevel.Error)]
        public void WriteError(string message)
        {
            this.WriteEvent(4, message);
        }

        [Event(5, Level = EventLevel.Critical)]
        public void WriteCritical(string message)
        {
            this.WriteEvent(5, message);
        }
    }
}
```

Each event is marked with an *EventAttribute* attribute that enables you to specify additional event information. The information can be used by the event listener to filter the event to be traced. The information that can be specified for each event is:

- **EventId** Represents the identifier for the event
- **Keywords** Specifies the keywords associated with an event as defined in the *EventKeywords* enum
- **Level** Indicates the level of the event; it can assume one of the following values (defined in the *EventLevel* enum):
 - **LogAlways** Is typically used by the listener to indicate that no filtering will be performed on events
 - **Critical** Corresponds to a critical error, which has caused a major failure
 - **Error** Indicates a standard error
 - **Warning** Represents a warning event
 - **Informational** Provides additional information about events that are not considered errors, such as the progress state of an application
 - **Verbose** Adds lengthy events or messages
- **Message** Specifies the message for the event
- **Opcode** Indicates the operation code for the event
- **Task** Specifies the task for the event
- **TypeId** When implemented in a derived class, represents a unique identifier for this attribute
- **Version** Indicates the version of the event

Because the same event source instance can be shared throughout the entire application, you might want to use a singleton to get a reference to the custom event source, as shown in Listing 6-18. Remember, however, that if you are planning to use multiple *EventSource* instances instead of sharing the same instance throughout the application, you should call the *Dispose* method when you no longer need to work with a specific instance.

LISTING 6-18 Using the singleton pattern to get a reference to the custom event source

```
using System;
using System.Collections.Generic;
using System.Diagnostics.Tracing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SimpleEventTracing
{
    public static class EventSourceFactory
    {
        private static MyCustomEventSource _eventSource;
```

```

public static MyCustomEventSource EventSource
{
    get
    {
        if (_eventSource == null)
        {
            _eventSource = new MyCustomEventSource();
        }

        return _eventSource;
    }
}
}
}

```

After you decide what events to trace, you must implement an event listener. Because the *EventListener* class is marked as abstract, you must derive your custom listener and provide your own implementation of the *OnEventWritten* abstract method that will receive the callback for the traced events. You can also define multiple event listeners. Each listener is logically independent of other listeners, which means you can use different listeners to trace different event types. For example, a listener might trace only the most severe errors that need to be immediately sent to the cloud or to a remote service for further analysis, whereas another listener might take care of less-critical events that can be logged in local storage and collected at a later time. Listing 6-19 shows the skeleton of two implementations of the *EventListener* abstract class used to log different kinds of events.

LISTING 6-19 Two implementations of the *EventListener* abstract class

```

using System;
using System.Collections.Generic;
using System.Diagnostics.Tracing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SimpleEventTracing
{
    public class RemoteEventListener : EventListener
    {
        protected override void OnEventWritten(EventWrittenEventArgs eventData)
        {
            // log the event via a remote service or in the cloud
        }
    }

    public class LocalStorageEventListener : EventListener
    {
        protected override void OnEventWritten(EventWrittenEventArgs eventData)
        {
            // log the event in the local storage;
        }
    }
}

```

The *OnEventWritten* method is called whenever the *WriteEvent* method of the associated *EventSource* has been invoked. The *OnEventWritten* method receives an instance of the *EventWrittenEventArgs* class as a unique parameter, which contains all the information concerning the event written by the event source. The following code snippet shows how an event listener can filter the set of events to trace, based on the event attributes defined in the *EventSource* associated with the listener:

```
private void InitializeEventListener()
{
    EventListener remoteListener = new RemoteEventListener();
    remoteListener.EnableEvents(EventSourceFactory.EventSource, EventLevel.Error |
        EventLevel.Critical);

    EventListener storageListener = new LocalStorageEventListener();
    storageListener.EnableEvents(EventSourceFactory.EventSource,
        EventLevel.Informational | EventLevel.Verbose | EventLevel.Warning);
}
```

In this sample, the first listener accepts only the most critical events that need to be addressed as soon as possible, whereas the second listener takes care of those events that do not represent an actual error but nonetheless can offer important information that you might want to store for future action.

You can log events by invoking the *Write** method that best suits your needs, as shown in the following line of code:

```
EventSourceFactory.EventSource.WriteCritical(ex.Message);
```

Using the Windows Store reports to improve the quality of your app

After you publish your app in the Windows Store, the store provides you with two types of information:

- **Analytics** Refers to data collected from the Windows Store. Analytics data includes information such as app listing views, downloads, and customer ratings and reviews. This data provides you with a considerable amount of information that can help you improve app sales and revenues. Collecting analytics data cannot be disabled.
- **Telemetry** Refers to data collected when your app is running on a user's device and provides information about how often it has been launched, how long it has been running, and whether it has experienced crashes, unresponsiveness (hangs), or JavaScript exceptions. From a developer's perspective, telemetry data represents a highly valuable source of information that can help you improve the reliability of your app. You can enable or disable telemetry data collection from your Profile section in the Windows Store Dashboard, as shown in Figure 6-33.

FIGURE 6-33 The Profile section of the Account info screen

Analytics information concerning your app are provided through Adoption reports, which include various analytics data that can help you understand the download trends for your app and review customer feedback. These reports also include useful information about the conversion rate of your app (from trial to full version) and in-app purchases. Figure 6-34 shows an example of a Downloads report that helps you track the download trends of your app and understand how you are performing with respect to your competitors.

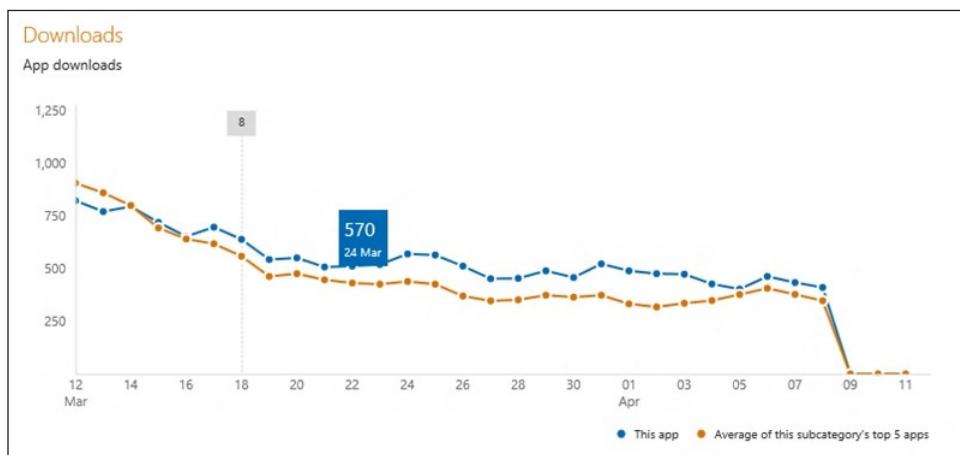


FIGURE 6-34 A Downloads report available from the Windows Store Dashboard

Source: <http://blogs.msdn.com/b/windowsstore/archive/2012/05/10/making-customer-focused-decisions-with-adoption-reports.aspx>

Telemetry data, on the other hand, is collected and summarized through Quality reports, which measure your app's reliability. To access Quality reports for your app, go to the app's summary page and click the Quality link in the left pane.



EXAM TIP

Quality reports measure the failure rate of your app, that is, the number of unexpected errors experienced by your customers due to a crash, an unresponsive app, or an unhandled exception (for apps written in Javascript).

When you click the Details link, you receive detailed information about the different errors encountered by your app. Figure 6-35 shows a Quality report summary.

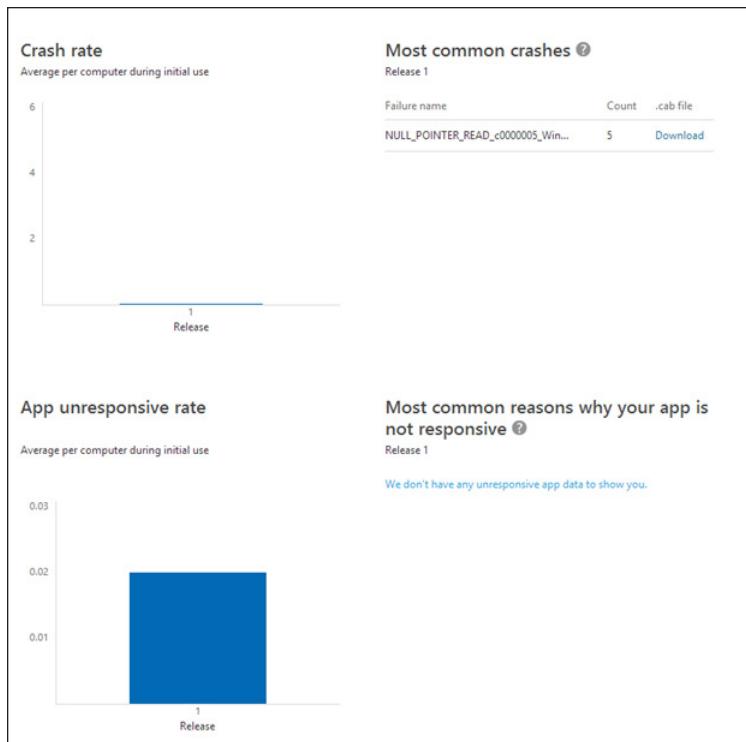


FIGURE 6-35 The Quality report summary

You can use this information to improve the quality of your app by identifying (and then fixing) the most common errors faced by your customers. You can also compare the quality of the different versions of your app published to the Windows Store.

According to Microsoft documentation, failure rates are computed for each failure type: crashes, unresponsiveness rate, and unhandled JavaScript exceptions. The data for calculating the failure rate is collected from a random sample of machines—called a quality panel—on which your app is used. Microsoft considers a quality panel of at least 500 machines to be an adequate sample size for calculating failure rates.

In the Most Common Crashes section, you can find the top five crashes that have affected your app. Each crash's name is uniquely identified by its name. Each name provides the following information: the problem that originated the crash, the error code and the debug symbols.

Next to the error, you can find a link to the cab file containing the process dump associated with that specific failure. A dump file represents a snapshot of an app that shows the process being executed and the modules that were loaded when the snapshot was taken. By analyzing the process dump, you can get the stack traces and other details regarding the error. You can open the dump file with Visual Studio 2012 or with an external tool such as the WinDbg.exe tool to analyze the collected data.

NOTE DUMP FILES

For further details on using dump files to debug app crashes and unresponsiveness in Visual Studio 2012, visit <http://msdn.microsoft.com/en-us/library/vstudio/d5zhxt22.aspx>.



Thought experiment

Solving a performance issue

In this thought experiment, apply what you've learned about this objective. You can find answers to these questions in the "Answers" section at the end of this chapter.

You are almost finished developing an application. Your unit tests and functional tests indicate that everything is fine. The only thing you have to resolve are some performance issues during the call to a remote web service.

How do you proceed to analyze the application, and which tools do you use?

Objective summary

- Visual Studio 2012 profiling tools for Windows Store apps enable you to observe and record metrics about the behavior of your app. They work by using a sampling method that collects information from the CPU call stack at regular intervals, navigating through the execution paths of your code and evaluating the cost of each function.
- Take advantage of the new tools introduced by Visual Studio 2012 Update 1 for Windows Store apps using HTML/JavaScript. The Memory Analysis tool collects data about memory usage, and the UI Responsiveness Profiler tool helps developers to improve the overall UI responsiveness and fluidity.
- Check the Quality reports provided by the Windows Store to discover the failure rate and the most common issues of your app.

Objective review

Answer the following questions to test your knowledge of the information in this objective. You can find the answers to these questions and explanations of why each answer choice is correct or incorrect in the “Answers” section at the end of this chapter.

1. Which of the following statements is incorrect?
 - A. The performance analysis tool in Visual Studio 2012 takes snapshots of the system at regular intervals, keeping track of the calls being executed and evaluating their execution cost.
 - B. The inclusive sample percentage (or inclusive time) in the Visual Studio 2012 profiling report indicates the CPU time spent to complete a specific function, including the time spent waiting for any other function it might call.
 - C. The exclusive samples percentage (or exclusive time) in the Visual Studio 2012 profiling report does not account for time spent waiting for other functions called from the current one to complete; it only indicates the amount of time consumed by the inner work of a function.
 - D. For Windows Store apps written in HTML/JavaScript, Visual Studio 2012 does not provide a tool to inspect the memory consumption and/or the number and hierarchy of objects currently in the memory heap.
2. When tracing a WinMD component written in C#, which class provides the ability to create different types of events that need to be traced?
 - A. *EventSource* class
 - B. *EventListener* class
 - C. *EventWrittenEventArgs* class
 - D. *TraceListener* class
3. In the Quality reports provided by the Windows Store, how is the quality of your app measured?
 - A. By user reviews and ratings
 - B. By number of downloads
 - C. By number of hours that users have spent using the application
 - D. By the failure rate of the app, based on a random sample of machines on which your app is used

Chapter summary

- Choose the business model that best suits your needs. Use the *CurrentAppSimulator* to test your app's behavior locally in a simulated scenario before publishing the app in the Windows Store. Before publishing, replace any instance of the *CurrentAppSimulator* class with the "real" *CurrentApp*.
- Set up a *try/catch/finally* block every time you are trying to access a resource that might be unavailable or missing, or when you are relying on types and functions over which you do not have any control.
- Use the *WinJS.Application.error* and *WinJS.Promise.error* events as a last resort to catch exceptions that have not been handled by the application code.
- The first time your Windows Store app needs to access a sensitive device, users are asked to grant their permission before the app can use the corresponding feature. Because your app might not be able to access a sensitive device, you need to handle the errors that can occur when trying to access a disabled device capability.
- When an error occurs during the execution of asynchronous operations, a *then* method does not allow it to propagate outside the function, whereas a *done* method always throws an exception if an error function has not been provided.
- A functional test plan verifies that all functional requirements of the software are working properly from the user's perspective. A unit test plan verifies whether single units of software, isolated from external dependencies, behave as expected.
- When implementing a unit test, remember to follow the Three A's rule: Set up the object to be tested (*arrange*), exercise the method under test (*act*), and then make claims about the object (*assert*).
- Visual Studio 2012 profiling tools for Windows Store apps let you observe and record metrics about the behavior of your app. They collect information from the CPU call stack at regular intervals, navigate through the execution paths of your code, and evaluate the cost of each function.
- The Memory Analysis tool and the UI Responsiveness Profiler tool were introduced in Visual Studio 2012 Update 1 for Windows Store apps using HTML/JavaScript. The Memory Analysis tool collects data about memory usage. The UI Responsiveness Profiler tool helps you improve your app's overall UI responsiveness and fluidity.
- Windows Store Quality reports measure the failure rate of your app and the most common issues experienced when using your app.

Answers

This section contains the solutions to the thought experiments and answers to the lesson review questions in this chapter.

Objective 6.1: Thought experiment

The app can be installed on up to five devices. After extra content has been purchased, you want to make sure the user can play the new levels on a different Windows 8 device. To do so, you can leverage one of the APIs exposed by the Windows Runtime to retrieve the purchase's receipt (such as the *GetAppReceiptAsync* and the *GetProductReceiptAsync* methods) and use it to validate the user's purchase on a different machine.

When the user launches the app from a different device, you can ask the Windows Store for the receipt, and validate it against your web server or service to ensure it has not been tampered with (using the digital signature information shipped with the receipt, as shown in Listing 6-11). You can then enable the extra content purchased by the user on a different device. The complete flow to validate the purchase's receipt can be found at <http://msdn.microsoft.com/en-us/library/windows/apps/jj649137.aspx>.

Objective 6.1: Review

1. Correct answer: C

- A. Incorrect:** You can use the *CurrentApp* class only if your app has been published in the Windows Store.
- B. Incorrect:** You can use the *CurrentApp* class only if your app has been published in the Windows Store.
- C. Correct:** You can use the *CurrentAppSimulator* to test your app locally. Remember to replace any reference to this class with the *CurrentApp* class before publishing your app, otherwise it will not pass the certification process.
- D. Incorrect:** You can use the *CurrentAppSimulator* to test your app locally. If your app has not been published in the Windows Store yet. The *CurrentAppSimulator* class is your only option.

2. Correct answer: B

- A. Incorrect:** There is no evaluation period in a feature-based trial. The user purchases the app to enable all the available features.
- B. Correct:** In a feature-based trial, the *IsActive* property returning *false* means that your app's license has been revoked or is missing.
- C. Incorrect:** When the user purchases the full version of the app, the *IsTrial* property should return *false*.
- D. Incorrect:** There is no evaluation period in a feature-based trial. The user can purchase the app at any time.

3. Correct answer: D

- A. Incorrect:** Whether the purchase is real or simulated depends on the class you are using (*CurrentApp* or *CurrentAppSimulator*), not on the value provided to the *RequestAppPurchaseAsync* method.
- B. Incorrect:** The price of the app and other features does not depend on the *RequestAppPurchaseAsync*.
- C. Incorrect:** To buy extra features or products offered through an in-app purchase, you have to leverage the *RequestProductPurchaseAsync*.
- D. Correct:** It indicates whether you want the method to return the receipt for the app purchase.

Objective 6.2: Thought experiment

A well-planned exception-handling strategy represents an important part of your app design and implementation. It can reduce the risks of exposing error messages containing sensitive information.

For example, you should always catch any exception that could potentially contain sensitive information, such as connection strings, URLs, login information, passwords, and so on. Replace them with custom exceptions that no longer contain this information before logging or rethrowing them, or before displaying the error message to the user. (This strategy is also known as exception shielding.)

Objective 6.2: Review

1. Correct answer: D

- A. Incorrect:** When caught in a *try/catch* block, the exception is handled.
- B. Incorrect:** When an exception is raised within a *then* method, the exception is stored in the promise state and won't propagate outside the promise.
- C. Incorrect:** The *WinJS.Application.error* event is raised only when there is no longer any possibility for the application code to catch the exception.
- D. Correct:** When an exception is raised and there is no longer any possibility for the application code to catch the exception, the *WinJS.Application.error* event is raised.

2. Correct answer: C

- A. Incorrect:** When an error occurs during the execution of a *then* method, the exception will be stored in the state of the promise and won't reach the surface.
- B. Incorrect:** When an error occurs during the execution of a *then* method, the exception will be stored in the state of the promise and won't reach the surface.

- C. **Correct:** The exception will be stored in the state of the promise and won't reach the surface.
 - D. **Incorrect:** Because the exception is stored in the state of the promise, the *WinJS.Promise.error* event will not be raised.
3. **Correct answer:** A
- A. **Correct:** An exception is raised when an app tries to invoke the *Windows.Media.Capture.MediaCapture.InitializeAsync* method after the user has revoked her permission to use the device.
 - B. **Incorrect:** The dialog box asking the user to grant permission to access the camera appears the first time the user launches the app.
 - C. **Incorrect:** An exception is raised.
 - D. **Incorrect:** An exception is raised.

Objective 6.3: Thought experiment

The reason the method under test is more likely to fail is that it is trying to access the user's Documents library. Before your app can programmatically access any user libraries, you need to add the corresponding declaration in the *Package.appxmanifest* file that is included in your unit testing project.

Your test code is subject to the same rules that apply to your Windows Store app. Therefore, any time you try to access a resource for which your app has to declare the corresponding capability in the app manifest (such as a user library or webcam, for example), you need to modify the *Package.appxmanifest* file of your Windows Store app (so it can use those resources) and the app manifest located inside your unit test project for Windows Store apps.

In the thought experiment example, to make the test pass, you have to declare the Documents Library capability and add one or more file associations on the Declarations tab.

Objective 6.3: Review

1. **Correct answer:** D
- A. **Incorrect:** The statement is correct.
 - B. **Incorrect:** The statement is correct.
 - C. **Incorrect:** The statement is correct.
 - D. **Correct:** The statement is incorrect. It refers to integration testing, in which different components and resources of an application are tested together to verify they work as expected.

2. Correct answer: C

- A. Incorrect:** Methods marked with the *AssemblyInitialize/AssemblyCleanup* attributes are invoked before and after all tests in the assembly are run.
- B. Incorrect:** Methods marked with the *ClassInitialize/ClassCleanup* attributes are invoked before and after all tests of the same test battery are run.
- C. Correct:** Methods marked with the *TestInitialize/TestCleanup* attributes are invoked before and after the execution of each single unit test.
- D. Incorrect:** The *DataTestMethod* attribute is used for data-driven testing.

3. Correct answer: D

- A. Incorrect:** The *Assert.ThrowsException<TException>* method is used in the test framework for Windows Store apps when the expected behavior of the method under test is to throw an exception.
- B. Incorrect:** The *TestMethod* attribute works in both frameworks.
- C. Incorrect:** The *DataTestMethod* attribute is supported only by the unit test framework for Windows Store apps.
- D. Correct:** The *ExpectedException* attribute is not supported in the unit test framework for Windows Store apps. You need to use the *Assert.ThrowsException<TException>* method instead.

Objective 6.4: Thought experiment

To find the bottleneck, you should first profile the application at a high level using the sampling method. (Remember that instrumentation is unsupported for Windows Store apps.) The profiling tool enables you to identify the functions that take the longest time to execute. Be sure to analyze both the exclusive execution time and the inclusive execution time.

After you identify the components/functions that take too much time to execute, you can try to optimize their code. If you can't optimize the code (because the code calls a remote web service that provides slow replies, for example), use asynchronous calls to perform web requests in parallel and/or do not block the user interface thread.

If the issue persists, verify the application using the Memory Analysis tool to analyze memory pressure. Sometimes your app simply requests too much data from remote resources. For example, your app requests all orders for a particular customer and related objects (such as order details) from a remote web service when you do not need them. In this case, you can use the Memory Analysis tool to explore the number of objects (and their hierarchy) you pushed into memory, and how many times the garbage collector is performing its duty.

The last thing to do is to verify the responsiveness of the application using the UI Responsiveness Profiler tool. It is important to verify the UI at the end as the last step of the analysis. If you call a web service that provides a reply in four seconds, for example, it is impossible for the UI to present that data in less time. The same applies to memory pressure. If you bind too

many objects to a control in the UI, the performance problem does not depend on the binding mechanism but rather the underlying code.

Objective 6.4: Review

1. Correct answer: D

- A. Incorrect:** The statement is correct.
- B. Incorrect:** The statement is correct.
- C. Incorrect:** The statement is correct.
- D. Correct:** The statement is incorrect. For Windows Store apps written in HTML/JavaScript, Visual Studio 2012 provides the Memory Analysis tool to inspect memory consumption and/or the number and hierarchy of objects currently in the memory heap.

2. Correct answer: A

- A. Correct:** The *EventSource* class provides the ability to create events for event tracking for Windows.
- B. Incorrect:** The *EventListener* class provides methods for enabling and disabling events from event sources.
- C. Incorrect:** The *EventWrittenEventArgs* class provides data for the *EventListener.OnEventWritten* callback.
- D. Incorrect:** The *TraceListener* class is not supported by the Windows Runtime.

3. Correct answer: D

- A. Incorrect:** User reviews and ratings are not included in Quality reports. User reviews and ratings data is summarized in the Ratings reports.
- B. Incorrect:** The number of downloads is not taken into account by Quality reports. Download data is summarized in the Downloads reports.
- C. Incorrect:** The number of hours spent using the application is not taken into account by Quality reports. Data about time spent by users with your app is summarized in the Usage reports.
- D. Correct:** The Quality reports provided by the Windows Store measure the app's failure rate, based on a random sample of devices on which your app is used.

Index

Numbers

400 Bad Request HTTP status code, 170

A

AAC (Advanced Audio Codec) audio profile, 74
Abandoned value (*Completion* property), 131
Aborted value (*DeviceWatcherStatus* enum), 116
AccelerationReading class, 83
Accelerometer class, 80
accelerometer sensor, 80–84
accessing
 sensors, 80–96
 accelerometer, 80–84
 compass, 87–88
 gyrometer, 85–86
 inclinometer, 92–94
 light, 95–96
 orientation, 89–92
AccessToken property, 167
activating
 file pickers, 266
 transitions, JavaScript, 200–202
Active Directory User Object store (Microsoft certificate store), 295
AddCondition function, 5
add/delete from list animations, 208–210
Added event (*DeviceWatcher* class), 112
addEventListerner event, 217
adding
 animations, 204–206
 transitions, 197–201
Add Reference dialog box, 47
Adoption reports, 378
Advanced Audio Codec (AAC) audio profile, 74

Advanced Query Syntax (AQS) string, 109
algorithms
 hash, 279–282
 MAC, 284–287
 symmetric key, 285
All value (*DeviceClass* enum), 108
alternate option (*animation-direction* property), 205
alternate-reverse option (*animation-direction* property), 205
always-connected experience, 27
ambient lighting, 95
analysis tools, JavaScript, 365–371
analytical data, 377
angular velocity, gyrometer sensor, 85–86
animation-delay property, 206
animation-direction property, 205
animation-duration property, 205
animation-fill-mode property, 206
animation-iteration-count property, 205
animation library, 206–211
animation-name property, 205
animation-play-state property, 206
animations, 195–212
 CSS3 transitions, 196–203
 activating transitions with JavaScript, 200–202
 adding/configuring transitions, 197–201
 UI enhancements
 animation library, 206–211
 creating/customizing animations, 203–206
 HTML5 canvas element, 211–212
 animation-timing-function property, 204
 anonymous method, 189
APIs, caching app data, 249–256
APIs (application programming interfaces)
 licensing, 310–316
 media capture, 57–79
 CameraCaptureUI API, 58–68
 MediaCaptureUI API, 67–77

- app data
 - caching, 248–255
 - ESE (Extensible Storage Engine), 257–258
 - IndexewdDB technology, 257
 - local storage, 249–252
 - defined, 247–248
 - security, 278–299
 - certificate enrollment and requests, 290–296
 - DataProtectionProvider class, 296–299
 - digital signatures, 288–290
 - hash algorithms, 279–282
 - MAC algorithms, 284–287
 - random number generation, 283–284
 - Windows.Security.Cryptography
 - namespaces, 279
 - understanding, 247–248
- Append method, 139, 282
- AppId attribute, 328
- AppId property, 314
- application data APIs, caching app data, 249–256
- ApplicationData class, 249
- ApplicationDataCompositeValue instance, 253
- ApplicationDataCreateDisposition enumeration, 250
- application language list, 237
- application manifest, declaring background task
 - usage, 5–6
- ApplicationModel.Background namespace, 32
- application programming interfaces. *See APIs*
- Application UI tab (App Manifest Designer), 237
- apply method, 225
- App Manifest Designer (Visual Studio), 7–8
 - Application UI tab, 237
 - background taskApp settings, 32
 - Badge and wide logo definition, 30–31
 - enabling transfer operations in background, 23–24
 - Location capability enabled, 97
 - webcam capability, 61–62
- app.onloaded event handler, 283
- AppReceipt element, 328
- apps (applications)
 - accessing sensors, 80–96
 - accelerometer, 80–84
 - compass, 87–88
 - gyrometer, 85–86
 - inclinometer, 92–94
 - light, 95–96
 - orientation, 89–92
- development
 - background tasks, 1–8
 - consuming background tasks, 10–36
 - integrating WinMD components, 38–50
- enhancements
 - animations and transitions, 195–212
 - custom controls, 213–225
 - globalization and localization, 228–239
 - responsiveness, 181–194
- implementing printing, 125–142
 - choosing options to display in preview window, 139–140
 - creating user interface, 132–133
 - custom print templates, 133–136
 - in-app printing, 142
 - PrintTask events, 131–132
 - PrintTaskOptions class, 136–138
 - reacting to print option changes, 140–142
 - registering apps for Print contract, 126–130
- PlayTo feature, 144–161
 - PlayTo contract, 144–147
 - PlayTo source applications, 149–155
 - registering apps as PlayTo receiver, 155–161
 - testing sample code, 147–149
- security, 278–299
 - certificate enrollment and requests, 290–296
 - DataProtectionProvider class, 296–299
 - digital signatures, 288–290
 - hash algorithms, 279–282
 - MAC algorithms, 284–287
 - random number generation, 283–284
- Windows.Security.Cryptography namespaces, 279
- solution deployment
 - diagnostics and monitoring strategies, 357–380
 - error handling, 330–342
 - testing strategies, 344–355
 - trial functionality, 307–329
- WNS (Windows Push Notification Service), 163–172
 - requesting/creating notification channels, 163–165
 - sending notifications to clients, 165–171
- AQS (Advanced Query Syntax) string, 109
- architecture, Windows Runtime, 40–41
- AreEqual method, 351
- aria property, 214
- AssemblyCleanup attribute, 354
- AssemblyInitialize attribute, 354

Assert.AreEqual method, 352
 Assert class, 348
 Assert.IsTrue method, 352
 associations (files), 274–276
 asymmetric encryption, 288
 AsymmetricKeyProvider class, 289
 asynchronous operations, 182–183
 attachAsync method, 26
 Attach to Running App option (JavaScript analysis tools), 366
 attributes, 353
 AppId, 328
 AssemblyCleanup, 354
 AssemblyInitialize, 354
 CertificateId, 328
 ClassCleanup, 353
 ClassInitialize, 353
 CSS, HTML, 133–136
 DataRow, 354
 DataTestMethod, 354
 EventAttribute, 375
 ExpectedExceptionAttribute, 348
 Id, 328
 LicenseType, 328
 MethodNam, 322
 ProductId, 328
 ProductType, 328
 PurchaseDate, 328
 ReceiptDate, 328
 ReceiptDeviceId, 328
 Signature, 329
 SimulationMode, 321
 TestClass, 351
 TestCleanup, 353
 TestMethod, 353
 UITestMethodAttribute, 348
 audio
 CameraCaptureUI API, 58–68
 capturing from the microphone, 76
 AudioCapture value (DeviceClass enum), 108
 AudioDeviceId property, 73
 AudioRender value (DeviceClass enum), 108
 audit trails, 371
 authentication, 279

B

BackgroundAccessStatus enumeration, 33
 background color, CSS3 transitions, 196–197
 BackgroundCompletedEventArgs object, 13
 BackgroundDownloader class, 23
 BackgroundExecutionManager class, 32
 BackgroundTaskBuilder object, 3
 BackgroundTaskCompletedEventArgs object, 13
 BackgroundTaskRegistration object, 7
 background tasks
 consuming, 10–36
 canceling tasks, 16–19
 debugging tasks, 20–21
 keeping communication channels open, 27–36
 progressing through and completing tasks, 12–15
 task constraints, 15–16
 task usage, 22
 transferring data in the background, 22–27
 triggers and conditions, 10–12
 updating tasks, 19–20
 creating, 1–8
 declaring background task usage, 5–7
 enumeration of registered tasks, 7–8
 using deferrals with tasks, 8
 BackgroundTransferCostPolicy, 23
 BackgroundTransfer APIs, 23
 BackgroundUploader class, 23
 Badge Logo reference, 30
 badge updates, 166
 Binding option (PrintTastOptions class), 136
 binding to custom controls, 220–221
 Bing Maps Geocode service, 100
 Blank App (XAML) template, 47
 bottom-up approach (functional testing), 346
 business model selection, trial functionality, 308–310
 buttons
 Buy, 309
 Capture Photo, 60
 Take Heap Snapshot, 366
 Try, 309
 Buy button, 309

C

CA (certification authority), 291
CachedFileManager class, 270
caching, data, 247–262
 app data, 248–258
 app data APIs, 249–256
 ESE (Extensible Storage Engine), 257–258
 IndexedDB technology, 257
 roaming profiles, 259–261
 understanding app and user data, 247–248
 user data, 260–262
Calendar class, 235
calendars, localizing apps, 235–236
callbacks
 asynchronous programming, 182
Called functions bar, 364
call method, 225
call stack, External Code, 41
Call Tree view, 363
CameraCaptureUI API, capturing pictures and video, 58–68
CameraCaptureUI class, 42
 leveraging camera settings, 64
CameraCaptureUIMaxPhotoResolution enumeration, 65
CameraCaptureUIMode parameter, 59
CameraCaptureUIPhotoFormat enumeration, 65
CameraCaptureUIVideoFormat property, 66
camera, media capture, 57–79
 CameraCaptureUI API, 58–68
 MediaCaptureUI API, 67–77
CameraOptionsUI class, 74
Cancelled value (Completion property), 131
cancel event, 217
cancellation requests, tasks, 16–19
canceling
 promises, 187–190
canceling tasks, 16–19
cancel method, 187
_cancelRequested variable, 17
capabilities, devices, 105–118
 DeviceWatcher class, 112–116
 enumerating devices, 106–112
 PnP (Plug and Play), 116–118
capCreateCaptureWindows function, 38
CaptureFileAsync method, 41, 44, 59, 341
Capture Photo button, 60
CapturePhotoToStorageFileAsync method, 71
capturing media
 camera, 57–79
 CameraCaptureUI API, 58–68
 MediaCaptureUI API, 67–77
 errors, 340–342
CertificateEnrollmentManager class, 294
Certificate Enrollment Requests store (Microsoft certificate store), 295
CertificateId attribute, 328
CertificateRequestProperties objects, 292
certificates, app security, 290–296
certification authority (CA), 291
change event, 217
changes
 print tasks, 140–142
CharacterGroupings class, 233
characterGroupings.lookup method, 233
charms
 Devices, 125, 142
 lay To-certified devices, 145
 Settings
 modifying Privacy settings, 339
CheckResult method, 13
CivicAddress property, 100
ClassCleanup attribute, 353
classes
 AccelerationReading, 83
 Accelerometer, 80
 ApplicationData, 249
 Assert, 348
 AsymmetricKeyProvider, 289
 BackgroundDownloader, 23
 BackgroundExecutionManager, 32
 BackgroundUploader, 23
 CachedFileManager, 270
 Calendar, 235
 CameraCaptureUI, 42
 leveraging camera settings, 64
 CameraOptionsUI, 74
 CertificateEnrollmentManager, 294, 296
 CharacterGroupings, 233
 Compass, 88
 CompassReading, 88
 Compressor, 276
 ControlChannelTrigger, 29–30
 keep-alive intervals, 35
 CryptographicBuffer, 281, 282

CryptographicEngine, 286
CryptographicHash, 282
CurrentApp, 310
CurrentAppSimulator, 311
DataProtectionProvider, 296–299
DataWriter, 273
DateTimeFormatter, 234
Debug, 20
Decompressor, 276
DeflateStream, 276
DeviceInformation, 108, 113
DeviceWatcher, 112–116
DOMEventMixin, 217
DownloadOperation, 25
EventListener, 376
EventSource, 374–377
FileIO, 251
FileOpenPicker, 264
Geolocator, 98
Gyrometer, 85
GZipStream, 276
HashAlgorithmNames, 281
HashAlgorithmProvider, 280
LicenseInformation, 310
LightSensorReading, 95
ListingInformation, 314
localStorage, 258
MacAlgorithmNames, 286
MacAlgorithmProvider, 285, 286
MaintenanceTrigger, 3, 10
MediaCaptureInitializationSettings, 73
MediaEncodingProfile, 74
MessageDialog, 185
OAuthToken, 167
OrientationSensor, 89, 91
PasswordVault, 299
PlayToConnection, 153
PlayToManager, 147
PlayToReceiver, 156
 events, 158
 initializing, 156
 Notify* methods, 160
PnpObject, 117
PrintManager, 127, 142
PrintTaskOptionDetails, 141
PrintTaskOptions, 136–138
ProximityDevice, 110
ResourceLoader, 233
sessionStorage, 258
SimpleOrientationSensor, 89
StandardPrintTaskOptions, 139
StorageFile, 260
StorageStreamTransaction, 273
StreamSocketControl, 35
SystemEventTrigger, 10
SystemTrigger, 3
TileUpdateManager, 14
TileUpdater, 165
VideoEffects, 71
VideosLibrary, 151
XMLHttpRequest, 185
ZipArchive, 276
ClassInitialize attribute, 353
ClearEffectsAsync method, 71
Clear method, 165
Clock, Language, and Region applet, 229–230
close method, 192
cloud storage, data caching, 261–262
CLR Windows 8 apps, consuming Windows Runtime, 41–42
code
 activating a background transfer, 24–25
 registering for Print contract, 127–128
coded UI testing, 346
Collation option (PrintTaskOptions class), 136
ColorMode option (PrintTaskOptions class), 136
combination approach (functional testing), 346
communication channels, keeping open, 27–36
Compare method, 282
Compass class, 88
CompassReading class, 88
compass sensor, 87–88
Completed event (PrintTask class), 131
Complete method, 8, 130
complete parameter, 189
Completion property, 131
components, WinMD, 38–50
 consuming a native WinMD library, 40–46
 creating a WinMD library, 47–50
composite settings, 249
compressing files, 276–277
Compressor class, 276
concurrency profiling, 358
conditions
 consuming background tasks, 10–12
 SystemConditionType enum, 11

confidentiality, 279
configuring
 animations, 204–206
 transitions, 197–201
`console.takeHeapSnapshot` method, 369
constraints (tasks), 15–16
consuming
 background tasks, 10–36
 cancelling tasks, 16–19
 debugging tasks, 20–21
 keeping communication channels open, 27–36
 progressing through and completing tasks, 12–15
 task constraints, 15–16
 task usage, 22
 transferring data in the background, 22–27
 triggers and conditions, 10–12
 updating tasks, 19–20
 WinMD library, 40–46
Containers enum, 250
`contextchanged` event, 232
continuation method, 190
contracts, `PlayTo`, 144–147
contracts, file pickers, 267
contrast mode, localizing images, 234
`ControlChannelReset` trigger, 11
`ControlChannelTrigger`, 11, 15
`ControlChannelTrigger` class, 29–30
 keep-alive intervals, 35
controls
 custom, 213–225
 creating, 218–222
 extending controls, 222–225
 understanding how existing controls work, 214–218
initializing, 184
rating
 constructor, 215
 CSS for, 214–215
 deriving from, 224–225
 extending, 223
 generated HTML, 214–215
`ConversionError` errors, 332
`ConvertStringToBinary` method, 281
cookies, 262
Coordinate property, 100
CostPolicy property, 25
CPU limits, task constraints, 15
CPU utilization graph, 370
crashes (failure rates), 379
`CreateContainer` method, 250
`_createControl` method, 223
`CreateDocumentFragment` method, 136
`createDownload` method, 25
Created value (`DeviceWatcherStatus` enum), 115
`createEventProperties` method, 217
`createFileAsync` method, 25
`CreateFileAsync` method, 251
`CreateFolderQuery` method, 271
`CreateHash` method, 282
`CreateKey` method, 286
`CreateMp4` method, 74
`CreatePrintTask` method, 128
`CreatePushNotificationChannelForApplicationAsync` method, 163–164
`CreateTileUpdaterForApplication` method, 14
`CreateUploadAsync` method, 27
`createUploadAsync(Uri, Iterable<BackgroundTransferContentPart>)` overload, 27
`createUploadAsync(Uri, Iterable<BackgroundTransferContentPart>, String)` overload, 27
`createUploadAsync(Uri, Iterable<BackgroundTransferContentPart>, String, String)` overload, 27
`CreateUploadFromStreamAsync` method, 27
`CreateWatcher` static method, 113
creating
 animations, 203–206
 background tasks, 1–8
 declaring background task usage, 5–7
 enumeration of registered tasks, 7–8
 using deferrals with tasks, 8
 custom controls, 218–222
 binding to custom controls, 220–221
 documentation, 221
 custom print templates, 133–136
 notification channels (WNS), 163–165
 promises, 188–190
 WinMD library, 47–50
`CroppedAspectRatio` property, 66
`CroppedSizeInPixels` property, 66
`CryptographicBuffer` class, 281–282
`CryptographicEngine` class, 286
`CryptographicEngine.Sign` method, 290
`CryptographicHash` class, 282
cryptography. *See* security
`CSS3` transitions, 196–203

activating transitions with JavaScript, 200–202
 adding/configuring transitions, 197–201
 CSS attributes, HTML, 133–136
 cubic Bézier curves, 200
 cubic-bezier() (transition timing function), 199
 currencies, localizing apps, 235
 CurrencyFormatter, 235
 CurrentApp class, 310
 CurrentAppSimulator class, 311
 CurrentState property, 153
 CurrentTimeChangeRequested event (PlayToReceiver class), 158
 custom controls, 213–225
 creating, 218–222
 binding to custom controls, 220–221
 documentation, 221
 extending controls, 222–225
 understanding how existing controls work, 214–218
 customizing
 animations, 203–206
 custom license information (apps), 316–317
 custom print templates, creating, 133–136
 C++ Windows 8 app
 consuming Windows Runtime, 42–47

D

data
 analytical, 377
 HTML5 Application Cache API storage, 261
 HTML5 File API storage, 261
 HTML5 Web Storage, 258
 integrity, 279
 ISAM files, 258
 libraries, 260
 local storage, 249–252
 management
 data caching, 247–262
 saving/retrieving files, 263–277
 securing app data, 278–299
 retrieval
 sensors, 79–103
 roaming storage, 252–255
 SkyDrive storage, 261–262
 telemetry, 377
 temporary storage, 255–257
 transferring, background tasks, 22–27

WinJS.Application.local storage, 258
 WinJS.Application.roaming storage, 258
 WinJS.Application.sessionState storage, 258
 data caching, 247–262
 caching app data, 248–258
 app data APIs, 249–256
 ESE (Extensible Storage Engine), 257–258
 IndexedDB technology, 257
 roaming profiles, 259–261
 understanding app and user data, 247–248
 user data, 260–262
 DataChanged event, 255
 DataProtectionProvider class, 296–299
 DataRow attribute, 354
 DataTestMethod attribute, 354
 data types, 40
 data-win-bind property, 221
 data-win-res property, 232
 DataWriter class, 273
 dates, localizing apps, 234–235
 DateTimeFormatter class, 234
 Deadline property, 152
 Debug class, 20
 debugging tasks, 20–21
 Debug Location toolbar (Visual Studio), 20–21
 Debug menu, Start Performance Analysis, 361
 DecimalFormatter, 235
 declarations, manifest, 151
 declaring, background task usage, 5–7
 Decompressor class, 276
 default constructor (DataProtectionProvider class), 297
 default contents, WinMD folders, 43
 Default.js file, 49
 DefaultTile element, 30
 Default value (PrintTaskOptions class options), 137
 default values, print options, 138
 deferrals, 130
 deferrals, using with tasks, 8
 define method, 216
 DeflateStream class, 276
 DeleteContainer method, 251
 #demoDiv:hover selector, 197
 derive method, 224–225
 deriving from existing controls (extending controls), 224–225
 design
 data caching, 247–262
 caching app data, 248–258
 roaming profiles, 259–261

DesiredAccuracy property

understanding app and user data, 247–248
user data, 260–262
diagnostics and monitoring strategies, 357–380
 JavaScript analysis tools, 365–371
 logging events, 371–377
 profiling Windows Store apps, 357–365
 reports, 377–380
error handling, 330–342
 app design, 331–335
 promise errors, 335–342
testing strategies, 344–355
 functional versus unit testing, 345–347
 test project, 348–355
DesiredAccuracy property, 100
development, Windows Store apps
 background tasks, 1–8
 consuming background tasks, 10–36
 integrating WinMD components, 38–50
DeviceClass enum, 108
device containers, 116
DeviceContainer value (PnpObjectType enum), 117
DeviceInformation class, 108, 113
device interface classes, 116
DeviceInterfaceClass value (PnpObjectType enum), 117
device interfaces, 116
DeviceInterface value (PnpObjectType enum), 117
devices
 capabilities, 105–118
 DeviceWatcher class, 112–116
 enumerating devices, 106–112
 PnP (Plug and Play), 116–118
 enumerating, 106–112
 media capture, camera and microphone, 57–79
 CameraCaptureUI API, 58–68
 MediaCaptureUI API, 67–77
 sensors, 79–103
 accessing, 80–96
 location data, 79
 user location, 96–102
Devices charm, 125, 142
 Play To-certified devices, 145
devices, PC Settings, 144–145
DeviceWatcher class, 112–116
DeviceWatcherStatus enum, 115
diagnostics strategies, 357–380
 JavaScript analysis tools, 365–371
 logging events, 371–377
 profiling Windows Store apps, 357–365
 reports, 377–380
dialog boxes
 Add Reference, 47
 Set Location, 97
 Windows Store, 318–319
digital signatures, app security, 288–290
Direct2D printing, 141
Disabled value (LocationStatus property), 103
disabling, default behavior of PlayTo feature, 146
dispatchEvent event, 217
dispatchEvent method, 217
DisplayedOptions property, 139
Dispose method, 375
DividedByZeroException exception, 352
_doBinding method, 221
documentation
 custom controls, 221
documents (user data), 248
DOMEventMixin class, 217
done method, 185, 335
 error function, 337
doSomething method, 222
DoSomeWork method, 332
DownloadOperation class, 25
Downloads reports, 378
doWork function, 2–3
dump files, 380
Duplex option (PrintTaskOptions class), 137

E

ease-in-out (transition timing function), 199
ease-in (transition timing function), 199
ease-out (transition timing function), 199
ease (transition timing function), 199
E_Fail error code, 322
Enable Multilingual App Toolkit option (Tools menu), 238
encryption
 MAC algorithms, 284–287
enhancements, UI (user interface)
 animations and transitions, 195–212
 animation library, 206–211
 creating/customizing animations, 203–206
 CSS3 transitions, 196–203
 HTML5 canvas element, 211–212
 custom controls, 213–225
 creating, 218–222

- extending controls, 222–225
- understanding how existing controls work, 214–218
- globalization and localization, 228–239
- responsiveness, 181–194
 - asynchronous strategy, 182–183
 - canceling promises, 187–190
 - handling errors, 185–186
 - promises, 183–186
 - web workers, 190–194
- enrollment, certificates, 290–296
- enterprise authentication capability, 297
- Enterprise Trust store (Microsoft certificate store), 295
- entities (user data), 248
- enumerating
 - registered tasks, 7–8
- enumerating devices, 106–112
- EnumerationCompleted event (DeviceWatcher class), 112
- EnumerationCompleted value (DeviceWatcherStatus enum), 116
- enumerations. *See* enums
- enums (enumerations)
 - ApplicationDataCreateDisposition, 250
 - BackgroundAccessStatus, 33
 - CameraCaptureUIMaxPhotoResolution, 65
 - CameraCaptureUIPhotoFormat, 65
 - Containers, 250
 - DeviceClass, 108
 - DeviceWatcherStatus, 115
 - KeyProtectionLevel, 292
 - PnpObjectType, 117
 - SimpleOrientation, 89
 - SystemConditionType
 - conditions, 11
 - SystemTriggerType, 4–5
 - Windows.Foundation.AsyncStatus, 336
 - WinRT PushNotificationType, 172
- Error event, 152
- error function, done method, 337
- error handling, 330–342
 - app design, 331–335
 - promise errors, 335–342
 - trial functionality, 320–321
 - UI responsiveness, 185–186
 - web workers, 193
- error parameter, 189
- errors
 - ConversionError, 332
 - RangeError, 332
 - ReferenceError, 332
 - TypeError, 332
 - URIError, 332
- ESE (Extensible Storage Engine), 257–258
- ETW (Event Tracing for Windows) mechanism, 371
- EventAttribute attribute, 375
- event-driven reading pattern, ReadChanging event, 82
- event handlers
 - app.onloaded, 283
 - onCanceled, 16
 - PrintTaskRequested, 128
 - progress, 14
 - receiving push notifications, 171
 - SourceRequested, 151
 - unregistering, 130
- EventListener class, 376
- events
 - Added, 114
 - addEventListener, 217
 - cancel, 217
 - change, 217
 - contextchanged, 232
 - DataChanged, 255
 - dispatchEvent, 217
 - EnumerationCompleted, 114
 - Error, 152
 - LicenseChanged, 317
 - LockScreenApplicationAdded, 12
 - LockScreenApplicationRemoved, 12
 - logging, 371–377
 - onCompleted, 12–13
 - onProgress, 14
 - OptionChanged, 141
 - PlayToReceiver class, 158
 - PositionChanged, 101
 - preview_change, 217
 - PrintTaskRequested, 127
 - ReadingChanged, 80, 82
 - RecordLimitationExceeded, 69
 - removeEventListener, 217
 - setOptions, 217
 - Shaken, 84
 - SourceRequested, 147, 151
 - SourceSelected, 151
 - StateChanged, 152

_events function

StatusChanged, 102
Transferred, 153
transitionEnd, 200–202
UseCamera_Click, 42
UserCamera_Click, 41
window.onerror JavaScript, 335
WinJS.Promise.error, 338
`_events` function, 217
EventSource class, 374–377
Event Tracing for Windows (ETW) mechanism, 371
exceptions, `DividedByZeroException`, 352
ExpectedExceptionAttribute attribute, 348
ExpirationDate property, 311
Exportable property, 293
Export method, 290
ExportPublicKey method, 290
export restrictions, cryptography, 279
extending controls, 222–225
Extensible Storage Engine (ESE), 257–258
External Code, call stack, 41
external scripts, web workers, 193–194
external services, data storage, 261–262
temporary files, 256
resource, 231–232
saving/retrieving, 263–277
accessing programmatically, 270–271
compressing files, 276–277
file extensions and associations, 274–276
file pickers, 264–270
files, folders, and streams, 272
Windows.Media.winmd, 43–44
WindowsStoreProxy.xml, 313
XLF, 238
FileSavePicker instance, 269
FileTypeFilter collection, 266
finally blocks, 331
FindAllAsync static method, 106
flow, file pickers, 267
fluid interface, 206
FlushTransport method, 36
FolderPicker, 268
folders, saving/retrieving files, 272
formats, video, 151
format templates, dates and times, 234
formatters, numbers and currencies, 235
fractionDigits property, 235
freshnessTime parameter, 11
FriendlyName property, 154
fulfilled state, promises, 183
functionality, trials, 307–329
 business model selection, 308–310
 custom license information, 316–317
 handling errors, 320–321
 in-app purchases, 322–327
 licensing state, 310–315
 purchasing apps, 318–320
 retrieving/validating receipts, 327–329
functional testing, 345–347
 coded UI testing, 346
 integration testing, 346–347
Function Code View pane, 364
Function Details view, 363–364
functions
 AddCondition, 5
 capCreateCaptureWindows, 38
 doWork, 2–3
 `_events`, 217
 takePicture_click, 59
 window.print (JavaScript), 126

F

Facedown value (`SimpleOrientation` enum), 89
Faceup value (`SimpleOrientation` enum), 89
failed state, promises, 183
Failed value (`Completion` property), 131
feature-based trials, 308, 320
feature lifetime, 309
file extensions, 274–276
file formats, SkyDrive, 261
FileIO class, 251
filename field (error event), 193
FileOpenPicker class, 264
file pickers, saving/retrieving files, 264–270
files
 Default.js, 49
 dump, 380
 ISAM (Indexed Sequential Access Method), 258
 Package.appxmanifest, 236
 Location capability, 97
 Private Networks capability enabled, 155
 Package.appxmanifest XML, 61–62
 reading values from
 local profiles, 252
 roaming profiles, 255

G

garbage collector, 367
 GC event category (UI Responsiveness Profiler tool), 370
 GenerateRandom method, 283
 GenerateRandomNumber method, 283
 generatin random numbers, app security, 283–284
 geographic data, determining user location, 98–101
 Geolocator class, 98
 get accessor, 217
 GetAppReceiptAsync method, 327
 getCurrentDownloadAsync method, 26
 GetCurrentReading method, 81
 GetDefault method, 80
 GetDeferral method, 8, 130, 152
 GetDeviceSelector method, 110
 GetFileAsync method, 271
 GetFilesAsync method, 151
 GetFoldersAsync method, 271
 GetForCurrentView method, 127, 151
 GetFromPrintTaskOptions static method, 142
 GetGeopositionAsync method, 100–101
 GetGlyphThumbnailAsync method, 106
 Getlids method, 45
 GetOAuthToken method, 168
 GetProductReceiptAsync method, 327
 GetRuntimeClassName method, 45
 getString method, 232
 GetThumbnailAsync method, 106
 GetTrustLevel method, 45
 GetValueAndReset method, 282
 globalization, 228–231
 globally unique identifiers (GUIDs), 110
 GPS sensor, 96
 guid property, BackgroundDownloader class, 25
 GUIDs (globally unique identifiers), 110
 Gyrometer class, 85
 gyrometer sensor, 85–86
 GZipStream class, 276

H

handling errors, 330–342
 app design, 331–335
 promise errors, 335–342
 trial functionality, 320–321

UI responsiveness, 185–186
 web workers, 193
 hardware slot, 29
 HashAlgorithmNames static class, 281
 HashAlgorithmProvider class, 280
 hash algorithms, app security, 279–282
 hash-based message authentication code (HMAC), 284
 HashData method, 281
 hash values, 279
 HasKey method, 250
 HeadingMagneticNorth property, 88
 HeadingTrueNorth property, 88
 HMAC (hash-based message authentication code), 284
 HolePunch option (PrintTastOptions class), 137
 HomeGroup content, 270
 Hot Paths, 363
 HSTRING data type, 40
 HTML5 Application Cache API storage, 261
 HTML5 canvas element, animating, 211–212
 HTML5 File API storage, 261
 HTML5 Web Storage, 258
 HTML (Hypertext Markup Language), CSS attributes, 133–136
 Hypertext Markup Language (HTML), CSS attributes, 133–136

I

IAsyncOperationWithProgress<DownloadOperation, DownloadOperation> interface, 26
 Id attribute, 328
 _<identifier>.comment key, 232
 IInspectable interface, 45
 ILDASM (Intermediate Language Disassembler) tool, 43
 IlluminanceLux property, 95
 Image decoding event category (UI Responsiveness Profiler tool), 370
 image (img) tags, 60
 images
 globalization, 228
 localization, 233–234
 img (image) tags, 60
 implementation
 data caching, 247–262
 caching app data, 248–258
 roaming profiles, 259–261
 understanding app and user data, 247–248
 user data, 260–262

importScripts method

PlayTo feature, 144–161
PlayTo contract, 144–147
PlayTo source applications, 149–155
registering apps as PlayTo receiver, 155–161
testing sample code, 147–149
printing, 125–142
choosing options to display in preview window, 139–140
creating user interface, 132–133
custom print templates, 133–136
in-app printing, 142
PrintTask events, 131–132
PrintTaskOptions class, 136–138
reacting to print option changes, 140–142
registering apps for Print contract, 126–130
testing strategies, 344–355
functional versus unit testing, 345–347
test project, 348–355
importScripts method, 193
in-app printing, 142
in-app purchases, 309, 322–327
inclimeter sensor, 92–94
IndexedDB technology, caching app data, 257–258
Indexed Sequential Access Method (ISAM) files, 258
InitializeAsync method, 72
InitializeSensor method, 84
initializing
 controls, 184
 PlayToReceiver class, 156
Initializing value (`LocationStatus` property), 102
`innertText` property, 220
`InstallCertificateAsync` method, 294–295
`instanceOf` statements, 332
instrumentation profiling, 358
`INT32` numeric type, 40
integrating WinMD components, 38–50
integration testing, 346–347
interfaces
 `IAsyncOperationWithProgress<DownloadOperation, DownloadOperation>`, 26
 `IInspectable`, 45
 `IPropertySet`, 70
 `IRandomAccessStream`, 60
Intermediate Certification Authorities store (Microsoft certificate store), 295
Intermediate Language Disassembler (ILDASM) tool, 43
InternetAvailable condition, 11
InternetNotAvailable condition, 11

`IPropertySet` interface, 70
`IRandomAccessStream` interface, 60
`IsActive` property, 311, 319–320
ISAM (Indexed Sequential Access Method) files, 258
`isGrouped` property, 235
`IsMailAddress` method, 48
isolation, unit testing, 347–348
`IsTrial` property, 311, 319–320

J

JavaScript
activating transitions, 200–202
analysis tools, 365–371
 Memory Analysis, 366–369
 UI Responsiveness Profiler, 369–371
single-threaded language, 182
`join` method, cancelling promises, 187–188

K

keep-alive connections, 28–29
keep-alive intervals, network apps, 35
keep-alive network triggers, 30
`KeepAlive` property, 35
`KeyAlgorithmName` property, 294
keyed hashing algorithms, 284–287
key frames, animations, 204
keypoints, 204
`KeyProtectionLevel` enum, 292
`KeySize` property, 293
key storage providers (KSPs), 293
`KeyUsages` property, 293
KSPs (key storage providers), 293

L

language settings, globalization, 229–231
Language window, 229–230
Launch Installed App Package option (JavaScript analysis tools), 366
Launch Startup Project option (JavaScript analysis tools), 366
layoutdir-RTL qualifier, 234

leveraging camera settings, CameraCaptureUI class, 64
 libraries, caching user data, 260
 LicenseChanged events, 317
 LicenseInformation class, 310
 LicenseInformation property, 310
 LicenseType attribute, 328
 licensing state (apps), 310–315
 light sensor, 95–96
 LightSensorReading class, 95
 linear (transition timing function), 199
 lineno field (error event), 193
 Link property, 314
 ListingInformation class, 314
 Live Connect, REST (Representational State Transfer) APIs, 261
 LoadCustomSimulator method, 324, 338
 Loading event category (UI Responsiveness Profiler tool), 370
 loading external scripts, web workers, 193–194
 LoadListingInformationAsync_GetResult method, 322
 LoadListingInformationAsync method, 314, 323
 load testing, 345
 local data storage, 249–252
 localeCompare method, 233
 LocalFolder property, 249
 localization, 231–239
 calendars, 235–236
 dates and times, 234–235
 images, 233–234
 manifest, 236–238
 numbers and currencies, 235
 string data, 231–233
 LocalSettings property, 249
 localStorage class, 258
 Location capability, App Manifest Designer, 97
 location data, 79
 LocationStatus property, 102
 lock screen
 applications, 28
 permission, 29
 registering an application, 30–33
 requesting use of, 32–33
 triggers, 11
 LockScreenApplicationAdded event, 12
 LockScreenApplicationRemoved event, 12
 logging events, 371–377
 logging to files
 local user storage, 251
 roaming user storage, 254

long-running server processes, 27
 longTaskAsyncPromise, 187
 LookUp method, 250
 lumen, 95
 lux (ambient lighting), 95

M

MacAlgorithmNames static class, 286
 MacAlgorithmProvider class, 285–286
 MAC (Message Authentication Code) algorithms, app security, 284–287
 MaintenanceTrigger class, 3, 10
 maintenance triggers, 10–12
 management, data and security
 data caching, 247–262
 saving/retrieving files, 263–277
 securing app data, 278–299
 manifest
 declarations, 151
 localization, 236–238
 maps, globalization, 229
 MaxCopies option (PrintTastOptions class), 137
 MaxResolution property, 66
 measuring angular velocity, gyrometer sensor, 85–86
 media capture, 57–79
 camera, 57–79
 CameraCaptureUI API, 58–68
 MediaCaptureUI API, 67–77
 errors, 340–342
 MediaCaptureInitializationSettings objects, 72
 MediaCaptureUI API, capturing media, 67–77
 MediaEncodingProfile class, 74
 media files (user data), 248
 MediaSize option (PrintTastOptions class), 137
 MediaType option (PrintTastOptions class), 137
 Memory Analysis tool, 365, 366–369
 Message Authentication Code (MAC) algorithms, 284–287
 MessageDialog class, 185
 message field (error event), 193
 MethodNam attribute, 322
 methods
 AddEffectAsync, 70
 anonymous, 189
 Append, 139, 282
 apply, 225

methods

AreEqual, 351
Assert.AreEqual, 352
Assert.IsTrue, 352
attachAsync, 26
call, 225
cancel, 187
CaptureFileAsync, 41, 44, 59, 341
CapturePhotoToStorageFileAsync, 71
characterGroupings.lookup, 233
CheckResult, 13
Clear, 165
ClearEffectsAsync, 71
close, 192
Compare, 282
Complete, 8, 130
console.takeHeapSnapshot, 369
continuation, 190
ConvertStringToBinary, 281
CreateContainer, 250
_createControl, 223
CreateDocumentFragment, 136
createDownload, 25
createEventProperties, 217
createFileAsync, 25
CreateFileAsync, 251
CreateFolderQuery, 271
CreateHash, 282
CreateKey, 286
CreateMp4, 74
CreatePrintTask, 128
CreatePushNotificationChannelForApplication-
 Async, 163–164
CreateTileUpdaterForApplication, 14
CreateUploadAsync, 27
CreateUploadFromStreamAsync, 27
CryptographicEngine.Sign, 290
define, 216
DeleteContainer, 251
derive, 224–225
dispatchEvent, 217
Dispose, 375
_doBinding, 221
done, 185, 335
 error function, 337
doSomething, 222
DoSomeWork, 332
Export, 290
ExportPublicKey, 290
FindAllAsync, 106
FlushTransport, 36
GenerateRandom, 283
GenerateRandomNumber, 283
GetAppReceiptAsync, 327
getCurrentDownloadAsync, 26
GetCurrentReading, 81
GetDefault, 80
GetDeferral, 8, 130, 152
GetDeviceSelector, 110
GetFileAsync, 271
GetFilesAsync, 151
GetFoldersAsync, 271
GetForCurrentView, 127, 151
GetFromPrintTaskOptions, 142
GetGeopositionAsync, 100–101
GetGlyphThumbnailAsync, 106
GetIids, 45
GetOAuthToken, 168
GetProductReceiptAsync, 327
GetRuntimeClassName, 45
getString, 232
GetThumbnailAsync, 106
GetTrustLevel, 45
GetValueAndReset, 282
HashData, 281
HasKey, 250
ImportPfxDataAsync, 296
importScripts, 193
InitializeAsync, 72
InitializeSensor, 84
InstallCertificateAsync, 294–295
IsEmailAddress, 48
join, 187
LoadCustomSimulator, 324, 338
LoadListingInformationAsync, 314, 323
LoadListingInformationAsync_GetResult, 322
localeCompare, 233
LookUp, 250
mix, 217
MSApp.GetHtmlPrintDocumentSource, 130, 136
Notify*, 160
Object.Equals, 351
OnEventWritten, 376
OnFileActivated, 276
OpenAlgorithm, 281, 286
OpenTransactedWriteAsync, 273
Pause, 25

PickSingleFileAsync, 267
 Play, 69
 PlayToManager.GetForCurrentView, 147
 postMessage, 190
 processAll, 184, 232
 ProtectAsync, 297
 ProtectStreamAsync, 299
 ReadBufferAsync, 273
 reateWatcher, 113
 ReloadSimulatorAsync, 316
 Remove, 251
 RequestAccessAsync, 11, 32
 RequestAppPurchaseAsync, 318
 RequestAppPurchaseAsync_GetResult, 322
 RequestProductPurchaseAsync, 325, 327
 RequestProductPurchaseAsync_GetResult, 322
 Resume, 25
 setOptions(this, options), 216
 setRequestHeader, 27
 SetSource, 130, 147
 Show, 74
 ShowPlayToUI, 155
 ShowPrintUIAsync, 142
 _showTentativeRating, 223
 Sign, 286
 SimulateRemoteServiceCall, 363
 Start, 116
 startAsync, 25
 StartAsync, 69, 159
 startDevice_click, 69
 StartReceivingButton_Click, 156
 StartRecordToCustomSinkAsync, 74
 StartRecordToStorageFileAsync, 74
 StartRecordToStreamAsync, 74
 Stop, 116
 StopAsync, 159
 StopRecordAsync, 75
 StopRecordingAsync, 69
 SubmitCertificateRequestAndGetResponse-
 Async, 294
 terminate, 192
 then, 184, 335
 ThrowsException<TException>, 348
 timeout, 188
 UnprotectAsync, 298
 UnprotectStreamAsync, 299
 _updateControl, 223
 VerifySignature, 286
 WaitForPushEnabled, 35
 WriteEvent, 377
 WriteTextAsync, 251, 272
 xhr, 185
 microphone, capturing audio, 76
 Microsoft.VisualStudio.TestTools.UnitTesting
 namespace, 348
 MinCopies option (PrintTastOptions class), 137
 MinimumReportInterval property, 82
 mix method, 217
 Modules view, 365
 monitoring strategies, 357–380
 JavaScript analysis tools, 365–371
 logging events, 371–377
 profiling Windows Store apps, 357–365
 reports, 377–380
 MovementThreshold property, 102
 MSApp.GetHtmlPrintDocumentSource method, 130,
 136
 Multilingual App Toolkit, 238
 MuteChangeRequested event (PlayToReceiver
 class), 158

N

namespaces
 ApplicationModel.Background, 32
 Microsoft.VisualStudio.TestTools.UnitTesting
 namespace, 348
 System.IO.Compression.FileSystem, 276
 System.Net.Sockets, 29
 System.Text.RegularExpressions, 47
 Windows.Devices.Enumeration, 105
 Windows.Devices.Enumeration.PnP, 116
 Windows.Devices.Geolocation, 79, 98
 Windows.Devices.Sensors, 79
 Windows.Globalization.Collation, 233
 Windows.Graphics.Printing, 127
 Windows.Networking.PushNotification, 163
 Windows.Security.Cryptography, 279
 Certificates, 292
 DataProtection, 296
 naming conventions, 351
 near-field communications (NFC), 110
 nested promises, 338
 .NET memory profiling, 358
 network access, task constraints, 15

NETWORK_ERROR

NETWORK_ERROR, 193
Network keep-alive interval, 35
network triggers, 29
NFC (near-field communications), 110
NoData value (LocationStatus property), 102
nonfunctional testing, 345
non-repudiation, 279
normal option (animation-direction property), 205
NotAvailable value (LocationStatus property), 103
NotAvailable value (PrintTaskOptions class options), 137
notification channels (WNS), requesting/creating, 163–165
notifications, progress, 186
notifications (WNS), sending to clients, 165–171
NotificationType property, 172
Notify* methods, 160
NotInitialized value (LocationStatus property), 103
NotRotated value (SimpleOrientation enum), 89
NumberOfCopies option (PrintTaskOptions class), 137
numbers, localizing apps, 235

O

OAuthToken class, 167
Object.Equals method, 351
objects
 BackgroundCompletedEventArgs, 13
 BackgroundTaskBuilder, 3
 BackgroundTaskCompletedEventArgs, 13
 BackgroundTaskRegistration, 7
 CertificateRequestProperties, 292
 MediaCaptureInitializationSettings, 72
 PnpDeviceWatcher, 117
 PrintManager, 126
 PrintTask, 126
 PrintTaskRequestedEventArgs, 128
 WebUIBackgroundTaskInstance, 2–3
onCanceled event handler, 16
onCompleted event, 12–13
oneShot parameter, 3, 11
OnEventWritten abstract method, 376
OnFileActivated method, 276
onProgress event, 14
OpenAlgorithm method, 281, 286
OpenTransactedWriteAsync method, 273
OptionChanged event, 141
OptionId property, 142

Orientation option (PrintTaskOptions class), 137
orientation sensor, 89–92
OrientationSensor class, 89
Other event category (UI Responsiveness Profiler tool), 370
Other People store (Microsoft certificate store), 295
overloaded constructor (DataProtectionProvider class), 297

P

Package.appxmanifest file, 236
Private Networks capability enabled, 155
Package.appxmanifest files
 Location capability, 97
Package.appxmanifest XML file, 61–62
PageSize setting, 249
parameters
 authenticating a cloud service to Windows Live service, 170
 CameraCaptureUIMode, 59
 freshnessTime, 11
 oneShot, 3, 11
 promise constructor, 189
 PasswordVault class, 299
 Pause method, 25
 PauseRequested event (PlayToReceiver class), 158
 PC Settings, panel listing of available devices, 144–145
 PercentFormatter, 235
 PermilleFormatter, 235
 permissions, lock screen, 29
 Personal store (Microsoft certificate store), 295
 PFX (Personal Information Exchange) messages, 296
 PhotoCaptureSource property, 72
 photoMessage div tag, 59
 PickSingleFileAsync method, 267
 pictures, CameraCaptureUI API, 58–68
 PitchDegrees property, 93
 pitch rotation, 92
 PKI (public key infrastructure), 291
 PlatformKeyStorageProvider KSP, 293
 PlaybackRateChangeRequested event (PlayToReceiver class), 158
 Play method, 69
 PlayRequested event (PlayToReceiver class), 158
 PlayToConnection class, 153
 PlayTo contract, 144–147

PlayTo feature, 144–161
 PlayTo contract, 144–147
 PlayTo source applications, 149–155
 registering apps as PlayTo receiver, 155–161
 testing sample code, 147–149

PlayToManager class, 147

PlayToReceiver class, 156
 events, 158
 initializing, 156
 Notify* methods, 160

PlayTo source applications, 149–155

Plug and Play (PnP) devices, 116–118

PnpDeviceWatcher objects, 117

PnpObject class, 117

PnpObjectType enum, 117

PnP (Plug and Play) devices, 116–118

polling sensor devices, 83

PortableStorageDevice value (DeviceClass enum), 108

PositionChanged event, 101

postMessage method, 190

preferences, globalization, 230

preview_change event, 217

Previewing event (PrintTask class), 131

PreviousState property, 153

PrinterCustom value (PrintTaskOptions class options), 137

printing implementation, 125–142
 choosing options to display in preview window, 139–140
 creating user interface, 132–133
 custom print templates, 133–136
 in-app printing, 142
 PrintTask events, 131–132
 PrintTaskOptions class, 136–138
 reacting to print option changes, 140–142
 registering apps for Print contract, 126–130

PrintManager class, 127, 142

PrintManager objects, 126

PrintQuality option (PrintTaskOptions class), 137

PrintSettings composite setting, 249

PrintTask events, 131–132

PrintTask objects, 126

PrintTaskOptionDetails class, 141

PrintTaskOptions class, 136–138

PrintTaskRequested event, 127

PrintTaskRequestedEventArgs objects, 128

PrintTaskRequested event handler, 128

print templates, creating, 133–136

Privacy settings (Permissions flyout), 339

Private Networks capability (Package.appxmanifest file), 155–156

processAll method, 184, 232

ProductId attribute, 328

ProductLicenses property, 311

ProductReceipt element, 328

ProductType attribute, 328

profile analysis report, 362

profiling Windows Store apps, 357–365

programmatically accessing files, 270–271

program user interaction
 implementing printing, 125–142
 choosing options to display in preview window, 139–140
 creating user interface, 132–133
 custom print templates, 133–136
 in-app printing, 142
 PrintTask events, 131–132
 PrintTaskOptions class, 136–138
 reacting to print option changes, 140–142
 registering apps for Print contract, 126–130

PlayTo feature, 144–161
 PlayTo contract, 144–147
 PlayTo source applications, 149–155
 registering apps as PlayTo receiver, 155–161
 testing sample code, 147–149

WNS (Windows Push Notification Service), 163–172
 requesting/creating notification channels, 163–165
 sending notifications to clients, 165–171

progress assignment, 14–15

progress event handlers, 14

Progressing event (PrintTask class), 131

progress notifications, 186

progress parameter, 189

promise constructor, parameters, 189

promise errors, 335–342

promises, 183–186
 cancelling, 187–190
 creating, 188–190

properties
 AccessToken, 167
 animation-delay, 206
 animation-direction, 205
 animation-duration, 205
 animation-fill-mode, 206
 animation-iteration-count, 205

ProtectAsync method

animation-name, 205
animation-play-state, 206
animation-timing-function, 204
AppId, 314
aria, 214
AudioDeviceId, 73
CameraCaptureUIVideoFormat, 66
CivicAddress, 100
Completion, 131
Coordinate, 100
CostPolicy, 25
CroppedAspectRatio, 66
CroppedSizeInPixels, 66
CurrentState, 153
data-win-bind, 221
data-win-res, 232
Deadline, 152
DesiredAccuracy, 100
DisplayedOptions, 139
ExpirationDate, 311
Exportable, 293
fractionDigits, 235
FriendlyName, 154
guid, BackgroundDownloader class, 25
HeadingMagneticNorth, 88
HeadingTrueNorth, 88
IlluminanceLux, 95
innertText, 220
IsActive, 311, 319–320
isGrouped, 235
IsTrial, 311, 319–320
KeepAlive, 35
KeyAlgorithmName, 294
KeySize, 293
KeyUsages, 293
LicenseInformation, 310
Link, 314
LocalFolder, 249
LocalSettings 4, 249
LocationStatus, 102
MaxResolution, 66
MinimumReportInterval, 82
MovementThreshold, 102
NotificationType, 172
OptionId, 142
PhotoCaptureSource, 72
PitchDegrees, 93
PreviousState, 153
ProductLicenses, 311
Quaternion, 92
Reading, 83
read-only requestedUri, BackgroundDownloader class, 25
ReportInterval, 81, 88, 92
Request, 128
resultFile, BackgroundDownloader class, 25
role, 214
RollDegrees, 93
RotationMatrix, 92
SourceRequest, 151
StreamingCaptureMode, 72
SuggestedStartLocation, 270
TemporaryFolder, 256
TriggerDetails, 36
userRating, 216
VideoDeviceId, 73
VideoSettings, 66
VideoStabilization, 71
YawDegrees, 93
ProtectAsync method, 297
ProtectStreamAsync method, 299
prototype functionality, controls, 222–223
prototypical inheritance, 224
ProximityDevice class, 110
public key infrastructure (PKI), 291
public/private key pairs, asymmetric encryption, 289
PurchaseDate attribute, 328
purchasing apps, 318–320
push notification, 28
push notification network triggers, 30
PushNotificationTrigger, 11, 15
PushNotificationTrigger trigger, 172

Q

quality panel, 379
Quality reports, 378
Quaternion property, 92

R

random number generation, app security, 283–284
RangeError errors, 332

rating control
 constructor, 215
 CSS for, 214–215
 deriving from, 224–225
 extending, 223
 generated HTML, 214–215
ReadBufferAsync method, 273
ReadingChanged event, 80, 82
Reading property, 83
 reading values from files
 local profiles, 252
 roaming profiles, 255
 temporary files, 256
read-only requestedUri property, **BackgroundDownloader** class, 25
Ready value (**LocationStatus** property), 102
ReceiptDate attribute, 328
ReceiptDeviceId attribute, 328
Receipt element, 328
 receipts, app purchases, 327–329
 receivers (**PlayTo** feature), registering apps as, 155–161
 recording video, 66
RecordLimitationExceeded event, 69
 reference content (app data), 248
ReferenceError errors, 332
 registered tasks
 enumerating, 7–8
 registering
 apps as **PlayTo** receivers, 155–161
ReloadSimulatorAsync static method, 316
 remote debugging, 347
Removed event (**DeviceWatcher** class), 112
removeEventListener event, 217
Remove method, 251
 Rendering event category (UI Responsiveness Profiler tool), 370
ReportInterval property, 81, 88, 92
 reports, 377–380
 Adoption, 378
 Downloads, 378
 profile analysis, 362
 Quality, 378
 Representational State Transfer (REST) APIs, 261
RequestAccessAsync method, 11, 32
RequestAppPurchaseAsync_GetResult method, 322
RequestAppPurchaseAsync static method, 318
 requesting
 certificates, 290–296
 notification channels (WNS), 163–165
RequestProductPurchaseAsync_GetResult method, 322
RequestProductPurchaseAsync method, 325, 327
Request property, 128
 resource files, 231–232
ResourceLoader class, 233
 response parameters, authentication to a Windows Live service, 170
 responsiveness, UI (user interface), 181–194
 asynchronous strategy, 182–183
 cancelling promises, 187–190
 handling errors, 185–186
 promises, 183–186
 web workers, 190–194
 REST (Representational State Transfer) APIs, 261
resultFile property, **BackgroundDownloader** class, 25
Resume method, 25
 retrieving
 compass sensor, 87–88
 data
 sensors, 79–103
 files, 263–277
 accessing programmatically, 270–271
 compressing files, 276–277
 file extensions and associations, 274–276
 file pickers, 264–270
 files, folders, and streams, 272
 receipts, app purchases, 327–329
 simulated license state, 311–312
 reverse option (animation-direction property), 205
 roaming data storage, 252–255
 roaming profiles, 259–261
 roaming settings, 252
 role property, 214
RollDegrees property, 93
 roll rotation, 92
Rotated90DegreesCounterclockwise value (**SimpleOrientation** enum), 89
Rotated180DegreesCounterclockwise value (**SimpleOrientation** enum), 89
Rotated270DegreesCounterclockwise value (**SimpleOrientation** enum), 89
RotationMatrix property, 92
Runtime Broker, 40
 runtime state (app data), 248

S

saving, files, 263–277
 accessing programmatically, 270–271
 compressing files, 276–277
 file extensions and associations, 274–276
 file pickers, 264–270
 files, folders, and streams, 272
scale factor, localizing images, 234
Script event category (UI Responsiveness Profiler tool), 370
secret, 167
security
 app data, 278–299
 certificate enrollment and requests, 290–296
 DataProtectionProvider class, 296–299
 digital signatures, 288–290
 hash algorithms, 279–282
 MAC algorithms, 284–287
 random number generation, 283–284
 Windows.Security.Cryptography namespaces, 279
security identifier (SID), 167
security testing, 345
Selling details section (Windows Store Dashboard), 308
sending
 notifications to clients (WNS), 165–171
sensitive devices, 340–341
Sensor platform, sensor change sensitivity, 86
sensors, 79–103
 accessing, 80–96
 accelerometer, 80–84
 compass, 87–88
 gyrometer, 85–86
 inclinometer, 92–94
 light, 95–96
 orientation, 89–92
determining user location
 geographic data, 98–101
 tracking position, 101–103
location data, 79
 user location, 96–102
server applications
 data caching, 261–262
Server keep-alive interval, 35
ServicingComplete system trigger type, 20
ServicingComplete task, 20
ServicingComplete trigger, 19

SessionConnected condition, 11
SessionConnected trigger, 11
SessionDisconnected condition, 11
sessionStorage class, 258
set accessor, 217
Set Location dialog box, 97
setOptions event, 217
setOptions(this, options) method, 216
setRequestHeader method, 27
SetSource method, 130, 147
settings
 composite, 249
 XML, 274–275
Settings charm, modifying Privacy settings, 339
Shaken event, 84
Sharing option, 144
Show method, 74
ShowPlayToUI static method, 155
ShowPrintUIAsync method, 142
_showTentativeRating method, 223
SID (security identifier), 167
Signature attribute, 329
Sign method, 286
SimpleOrientation enum, 89
SimpleOrientationSensor class, 89
simulated license state, retrieving, 311–312
SimulateRemoteServiceCall method, 363
SimulationMode attribute, 321
single-threaded language, JavaScript, 182
SkyDrive, data storage, 261–262
SmartcardKeyStorageProvider KSP, 293
Snapshot Detail view, 368
SoftwareKeyStorageProvider KSP, 293
software slot, 29
solution deployment
 diagnostics and monitoring strategies, 357–380
 JavaScript analysis tools, 365–371
 logging events, 371–377
 profiling Windows Store apps, 357–365
 reports, 377–380
error handling, 330–342
 app design, 331–335
 promise errors, 335–342
testing strategies, 344–355
 functional versus unit testing, 345–347
 test project, 348–355
trial functionality, 307–329
 business model selection, 308–310

custom license information, 316–317
 handling errors, 320–321
 in-app purchases, 322–327
 licensing state, 310–315
 purchasing apps, 318–320
 retrieving/validating receipts, 327–329
 sorting text
 globalization, 229
 source applications
 PlayTo, 149–155
 SourceChangeRequested event (PlayToReceiver class), 158
 SourceRequested event, 147, 151
 SourceRequested event handler, 151
 SourceRequest property, 151
 SourceSelected event, 151
 standard .NET 4.5 profile, 46
 StandardPrintTaskOptions class,, 139
 Standard UI (user interface)
 media capture, 57–79
 CameraCaptureUI API, 58–68
 MediaCaptureUI API, 67–77
 Staple option (PrintTaskOptions class), 137
 startAsync method, 25
 StartAsync method, 69, 159
 startDevice_click method, 69
 Started value (DeviceWatcherStatus enum), 115
 Start method, 116
 Start Performance Analysis (Debug menu), 361
 StartReceivingButton_Click handler method, 156
 StartRecordToCustomSinkAsync method, 74
 StartRecordToStorageFileAsync method, 74
 StartRecordToStreamAsync method, 74
 StateChanged event, 152
 state data, 259
 states, promises, 183–184
 StatusChanged event, 102
 step-end (transition timing function), 199
 stepping function, 200
 step-start (transition timing function), 199
 steps() (transition timing function), 199
 StopAsync method, 159
 Stop method, 116
 Stopped event (DeviceWatcher class), 112
 Stopped value (DeviceWatcherStatus enum), 116
 stopping, web workers, 192–193
 Stopping value (DeviceWatcherStatus enum), 116
 StopRecordAsync method, 75
 StopRecordingAsync method, 69
 StopRequested event (PlayToReceiver class), 158
 storage, data
 Extensible Storage Engine (ESE), 257–258
 HTML5 Application Cache API storage, 261
 HTML5 File API storage, 261
 HTML5 Web Storage, 258
 ISAM files, 258
 libraries, 260
 local, 249–252
 roaming, 252–255
 SkyDrive storage, 261–262
 temporary, 255–257
 WinJS.Application.local, 258
 WinJS.Application.roaming, 258
 WinJS.Application.sessionState, 258
 StorageFile class, 260
 StorageStreamTransaction class, 273
 StreamingCaptureMode property, 72
 streaming video
 PlayTo-certified devices, 149–155
 streams
 saving/retrieving files, 272
 StreamSocketControl class, 35
 stress testing, 345
 string data, localizing apps, 231–233
 strings directory, 231
 Styling event category (UI Responsiveness Profiler tool), 370
 SubmitCertificateRequestAndGetResponseAsync method, 294
 Submitted value (Completion property), 131
 Submitting event (PrintTask class), 131
 subscribing to the Completed event, 131
 SuggestedStartLocation property, 270
 Summary view, 366
 suspension
 checking tasks for, 15–16
 symmetric encryption, 284
 symmetric key algorithms, 285
 SystemConditionType enum
 conditions, 11
 SystemEventTrigger class, 10
 System.IO.Compression.FileSystem namespace, 276
 System.Net.Sockets namespace, 29
 System.Text.RegularExpressions namespace, 47
 SystemTrigger class, 3
 system triggers, 4–6, 10–12
 SystemTriggerType enum, 4–5

T

Take Heap Snapshot button, 366
takePicture_click function, 59
tasks, background
 checking for suspension, 15–16
 consuming, 10–36
 canceling tasks, 16–19
 debugging tasks, 20–21
 keeping communication channels open, 27–36
 progressing through and completing tasks, 12–15
 task constraints, 15–16
 task usage, 22
 transferring data in the background, 22–27
 triggers and conditions, 10–12
 updating tasks, 19–20
creating, 1–8
 declaring background task usage, 5–7
 enumeration of registered tasks, 7–8
 using deferrals with tasks, 8
usage, 22
TCP keep-alive interval, 35
telemetry (data), 377
templates
 creating custom print templates, 133–136
 format, dates and times, 234
temporary data storage, 255–257
TemporaryFolder property, 256
terminate method, 192
TestClass attribute, 351
TestCleanup attribute, 353
testing sample code, PlayTo feature, 147–149
testing strategies, 344–355
 functional versus unit testing, 345–347
 test project, 348–355
TestInitialize attribute, 353
TestMethod attribute, 353
text, globalization, 229
then method, 184, 335
third-party databases, data caching, 261–262
Third-Party Root Certification Authorities store
 (Microsoft certificate store), 295
ThrowsException<TException> method, 348
tier interaction profiling (TIP), 359
TileUpdateManager class, 14
TileUpdater class, 165
tile updates, 166
timed trials, 308, 320
timeout method, 188
timeouts, cancelling asynchronous operations, 188
times, localizing apps, 234–235
TimeTrigger, 11
time triggered tasks, 12
TimeUpdateRequested event (PlayToReceiver class), 158
timing functions, transitions, 199
TIP (tier interaction profiling), 359
toasts, 166
tools
 JavaScript analysis
 Memory Analysis, 365–369
 UI Responsiveness Profiler, 365, 369–371
 WinDbg.exe, 380
top-down approach (functional testing), 346
tracking, user position, 101–103
Transferred event, 153
transferring data, background tasks, 22–27
transform:scaleX(-1) style, 234
transition-delay property, transitions, 200
transition-duration property, transitions, 198
transitionEnd event, 200–202
transition-property property, transitions, 198
transitions, 195–212
 CSS3 transitions, 196–203
 activating transitions with JavaScript, 200–202
 adding/configuring transitions, 197–201
 UI enhancements
 animation library, 206–211
 creating/customizing animations, 203–206
 HTML5 canvas element, 211–212
 transition-timing-function property, transitions, 198
trial functionality, 307–329
 business model selection, 308–310
 custom license information, 316–317
 handling errors, 320–321
 in-app purchases, 322–327
 licensing state, 310–315
 purchasing apps, 318–320
 retrieving/validating receipts, 327–329
TriggerDetails property, 36
triggers, 4–6
 consuming background tasks, 10–12
 keep-alive network triggers, 30
 lock screen, 11
 push notification network triggers, 30
 ServicingComplete, 19

triggers (tasks)
 PushNotificationTrigger, 172
 Trusted People store (Microsoft certificate store), 295
 Trusted Publishers store (Microsoft certificate store), 295
 Trusted Root Certification Authorities store (Microsoft certificate store), 295
 Try button, 309
 try/catch blocks, 331
 TypeError error, 332
 types, requirements, 47

U

UINT64 numeric type, 40
 UI Responsiveness Profiler tool, 365, 369–371
 UITestMethodAttribute attribute, 348
 UI thread
 avoiding blocking of thread, 182
 UI (user interface)
 enhancements
 animations and transitions, 195–212
 custom controls, 213–225
 globalization and localization, 228–239
 responsiveness, 181–194
 printing implementation, 132–133
 unfulfilled state, promises, 183
 unhandled JavaScript exceptions, 379
 uniform resource identifiers (URIs), 163
 unit testing versus functional testing, 345–347
 Unit Test Library, 349–350
 Unknown value (PnpObjectType enum), 117
 UnprotectAsync method, 298
 UnprotectStreamAsync method, 299
 unregistering event handlers, 130
 unresponsiveness rate (failures), 379
 Untrusted Certificates store (Microsoft certificate store), 295
 _updateControl method, 223
 Updated event (DeviceWatcher class), 112
 updating tasks, 19–20
 URIError errors, 332
 URIs (uniform resource identifiers), 163
 usability testing, 345
 UseCamera_Click event, 42
 UserAway trigger, 12
 UserCamera_Click event, 41

user data
 caching, 260–262
 defined, 248
 understanding, 247–248
 user interaction
 implementing printing, 125–142
 choosing options to display in preview window, 139–140
 creating user interface, 132–133
 custom print templates, 133–136
 in-app printing, 142
 PrintTask events, 131–132
 PrintTaskOptions class, 136–138
 reacting to print option changes, 140–142
 registering apps for Print contract, 126–130
 PlayTo feature, 144–161
 PlayTo contract, 144–147
 PlayTo source applications, 149–155
 registering apps as PlayTo receiver, 155–161
 testing sample code, 147–149
 WNS (Windows Push Notification Service), 163–172
 requesting/creating notification channels, 163–165
 sending notifications to clients, 165–171
 user interface (UI)
 enhancements
 animations and transitions, 195–212
 custom controls, 213–225
 globalization and localization, 228–239
 responsiveness, 181–194
 UserNotPresent condition, 11
 user preferences (app data), 248
 UserPresent condition, 11
 UserPresent trigger, 12
 userRating property, 216
 users
 determining location with sensors, 96–102
 geographic data, 98–101
 tracking position, 101–103

V

validating
 receipts, app purchases, 327–329
 variables
 _cancelRequested, 17
 verification process, task usage, 22

VerifySignature method

VerifySignature method, 286
versioning compliance, 46
video
 CameraCaptureUI API, 58–68
 formats, 151
 recording, 66
 streaming to a PlayTo-certified device, 149–155
VideoCapture value (DeviceClass enum), 108
VideoDeviceId property, 73
VideoEffects class, 71
VideoSettings property, 66
Videos library
 recording video, 73
VideosLibrary class, 151
VideoStabilization property, 71
Visual Studio
 App Manifest Designer, 7–8
 Application UI tab, 237
 background taskApp settings, 32
 Badge and wide logo definition, 30–31
 enabling transfer operations in background, 23–24
 Location capability enabled, 97
 webcam capability, 61–62
 Debug Location toolbar, 20–21
VolumeChangeRequested event (PlayToReceiver class), 158

Windows.Devices.Enumeration.PnP namespace, 116
Windows.Devices.Geolocation namespace, 79, 98
Windows.Devices.Sensors namespace, 79
Windows.Foundation.AsyncStatus enum, 336
Windows.Globalization.Collation namespace, 233
Windows.Graphics.Printing namespace, 127
Windows Library for JavaScript (WinJS)
 catching exceptions, 333
Windows Live service, 166–167
 parameters for authenticating cloud service to, 170
Windows Location Provider, 96
Windows Media Audio (WMA) profile, 74
Windows Media Player instance, 148
Windows Media Video (WMV) profile, 74
Windows.Media.winmd file, 43–44
Windows Metadata (WinMD)
 components, 38–50
 consuming a native WinMD library, 40–46
 creating a WinMD library, 47–50
 default folder contents, 43
 Windows.Networking.PushNotification namespace, 163
 Windows Notification Service (WNS), 28
 Windows Performance Toolkit (WPT), 359
 Windows Push Notification Service. *See WNS*
Windows Runtime
 architecture, 40–41
 consuming from a CLR Windows 8 app, 41–42
 consuming from a C++ Windows 8 app, 42–47
 WinMD components, 38–50
Windows Runtime Component-provided template, 47
Windows.Security.CryptographyCertificates namespace, 292
Windows.Security.Cryptography.DataProtection namespace, 296
Windows.Security.Cryptography namespaces, 279
Windows Sensor and Location platform, 79
Windows Store apps
 accessing sensors, 80–96
 accelerometer, 80–84
 compass, 87–88
 gyrometer, 85–86
 inclinometer, 92–94
 light, 95–96
 orientation, 89–92
 development
 background tasks, 1–8
 consuming background tasks, 10–36
 integrating WinMD components, 38–50

W

WaitForPushEnabled method, 35
WCF (Windows Communication Foundation) APIs, 46
webcam capability, App Manifest Designer, 61–62
WebUIMBackgroundTaskInstance object, 2–3
web workers, UI responsiveness, 190–194
 available features, 190–192
 handling errors, 193
 loading external scripts, 193–194
 stopping, 192–193
WideLogo definition, 30
window.onerror JavaScript event, 335
window.print function (JavaScript), 126
windows
 Language, 229–230
Windows 8 Simulator, 97
Windows Communication Foundation (WCF) APIs, 46
Windows.Devices.Enumeration namespace, 105

- enhancements
 - animations and transitions, 195–212
 - custom controls, 213–225
 - globalization and localization, 228–239
- implementing printing, 125–142
 - choosing options to display in preview
 - window, 139–140
 - creating user interface, 132–133
 - custom print templates, 133–136
 - in-app printing, 142
 - PrintTask events, 131–132
 - PrintTaskOptions class, 136–138
 - reacting to print option changes, 140–142
 - registering apps for Print contract, 126–130
- PlayTo feature, 144–161
 - PlayTo contract, 144–147
 - PlayTo source applications, 149–155
 - registering apps as PlayTo receiver, 155–161
 - testing sample code, 147–149
- security, 278–299
 - certificate enrollment and requests, 290–296
 - DataProtectionProvider class, 296–299
 - digital signatures, 288–290
 - hash algorithms, 279–282
 - MAC algorithms, 284–287
 - random number generation, 283–284
 - Windows.Security.Cryptography
 - namespaces, 279
- solution deployment
 - diagnostics and monitoring strategies, 357–380
 - error handling, 330–342
 - testing strategies, 344–355
 - trial functionality, 307–329
- UI enhancements
 - responsiveness, 181–194
- WNS (Windows Push Notification Service), 163–172
 - requesting/creating notification channels, 163–165
 - sending notifications to clients, 165–171
- Windows Store Dashboard, 308
- Windows Store dialog box, 318–319
- WindowsStoreProxy.xml files, 313
- Windows.System.UserProfile.GlobalizationPreferences.
Listing, 230
- WinJS.Application.local, data storage, 258
- WinJS.Application.roaming, data storage, 258
- WinJS.Application.sessionState, data storage, 258
- WinJS controls, functionality, 214–218
- WinJS.Promise.error event, 338
- WinJS.UI.Animation API, 195
- WinJS (Windows Library for JavaScript)
 - catching exceptions, 333
 - event logging, 372–373
- WinMD components, event logging, 373–377
- WinMD library
 - consuming, 40–46
 - creating, 47–50
- WinMD (Windows Metadata)
 - components, 38–50
 - consuming a native WinMD library, 40–46
 - creating a WinMD library, 47–50
 - default folder contents, 43
- WinRT
 - media capture, 57–79
 - CameraCaptureUI API, 58–68
 - MediaCaptureUI API, 67–77
 - WinRT PushNotificationType enum, 172
 - WMA (Windows Media Audio) profile, 74
 - WMV (Windows Media Video) profile, 74
 - WNS (Windows Notification Service), 28
 - WNS (Windows Push Notification Service), 163–172
 - requesting/creating notification channels, 163–165
 - sending notifications to clients, 165–171
- WPT (Windows Performance Toolkit), 359
- WriteEvent method, 377
- WriteTextAsync method, 251, 272

X

- X.509 public key infrastructure (PKI), 291
- xhr method, 185
- XLF files, 238
- XMLHttpRequest class, 185
- XML settings, 274–275

Y

- YawDegrees property, 93
- yaw rotation, 92

Z

- ZipArchive class, 276

About the Authors

ROBERTO BRUNETTI is a consultant, trainer, and author with experience in enterprise applications since 1997. Together with Paolo Pialorsi, Marco Russo, and Luca Regnicoli, Roberto is a founder of DevLeap, a company focused on providing high-value content and consulting services to professional developers. He is the author of a few books about ASP.NET and Windows Azure, plus two books on Microsoft Windows 8, all for Microsoft Press. Since 1996, Roberto has been a regular speaker at major conferences.

VANNI BONCINELLI is a consultant and author on .NET technologies. Since 2010, he has been working with the DevLeap team, developing several enterprise applications based on Microsoft technologies. Vanni has authored many articles for Italian editors on XNA and game development, Windows Phone, and, since the first beta version in 2011, Windows 8. He also worked on *Build Windows 8 Apps with Microsoft Visual C# and Visual Basic Step by Step* (Microsoft Press, 2013).

