



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.

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.

microsoft.com/mspress

ISBN: 978-0-7356-7680-0



U.S.A. \$39.99

Canada \$41.99

[Recommended]

Certification/Windows Store Apps





Exam Ref 70-482:

Advanced Windows Store

App Development Using

HTML5 and JavaScript

**Roberto Brunetti
Vanni Boncinelli**

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, 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.

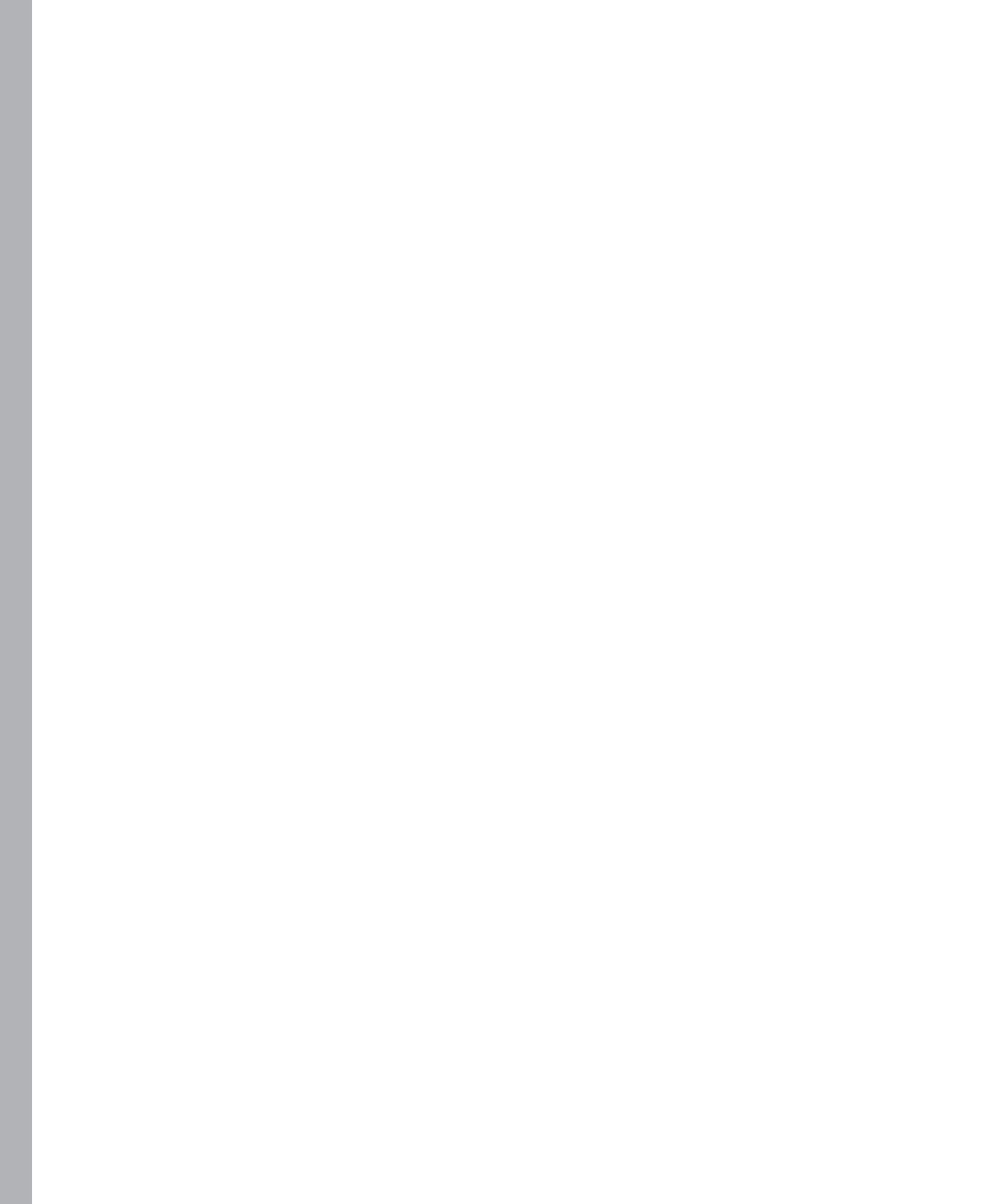
— ROBERTO BRUNETTI

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:

<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.

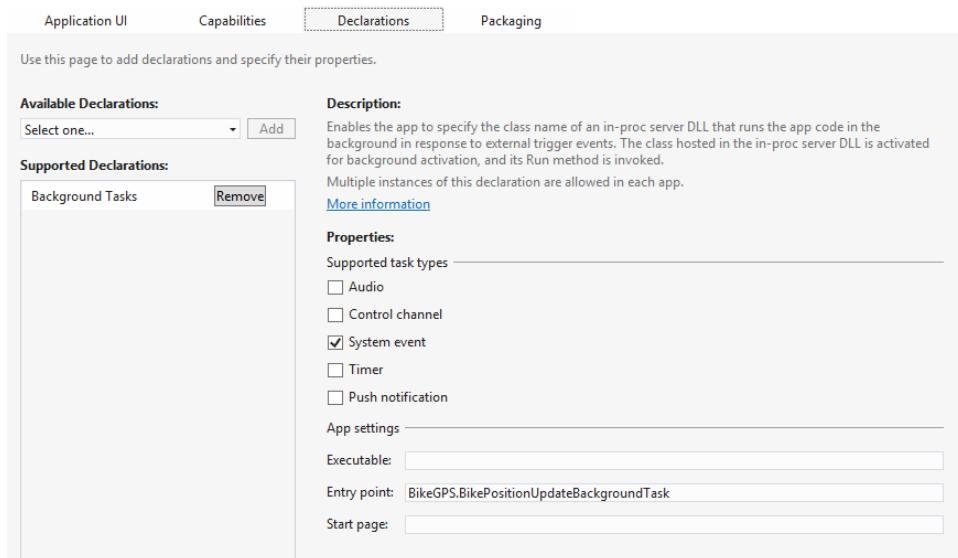


FIGURE 1-1 Background task declaration in Visual Studio App Manifest Designer

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.

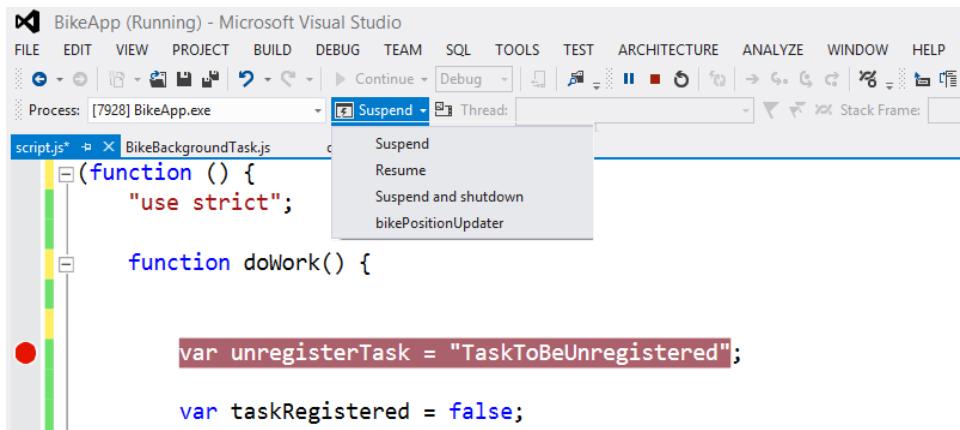


FIGURE 1-2 Visual Studio “hidden” Debug Location toolbar to start tasks

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

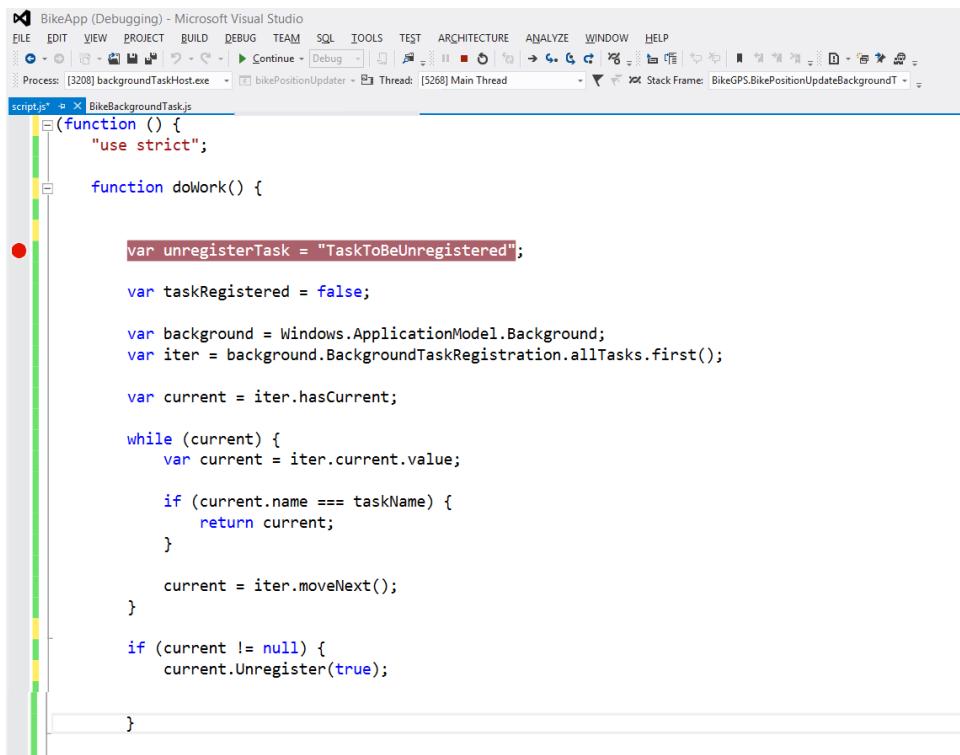


FIGURE 1-3 Debugging tasks activated directly from Visual Studio

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.

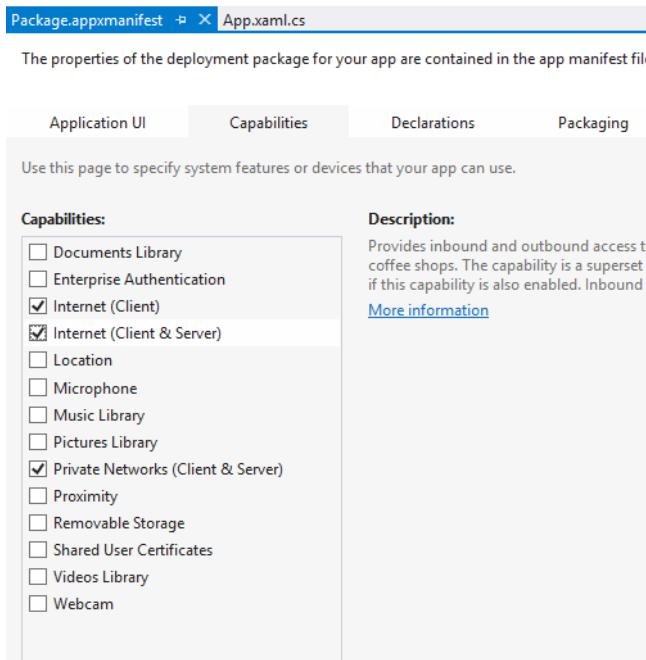


FIGURE 1-4 Capabilities for background transferring

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 URI 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.

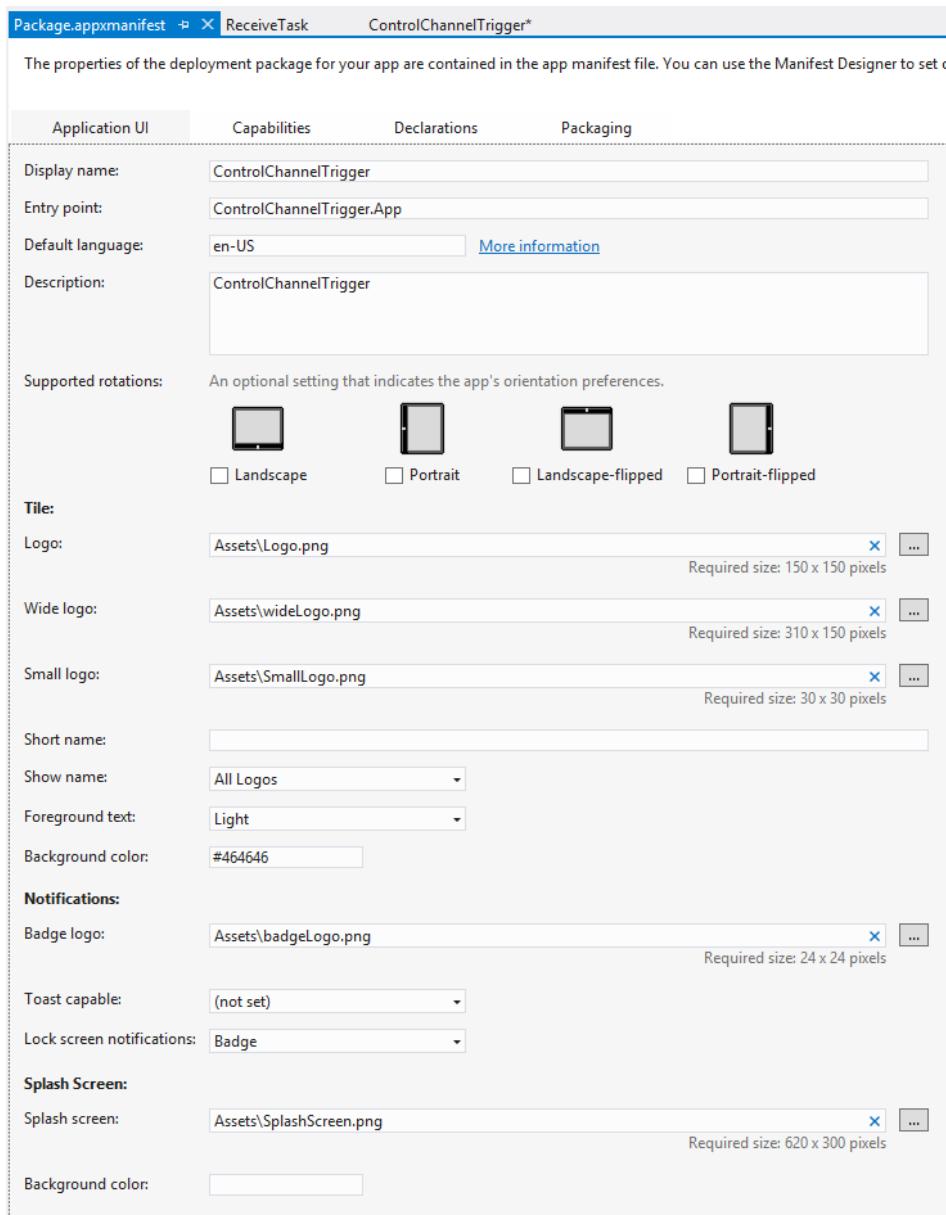


FIGURE 1-5 Badge and wide logo definition

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.

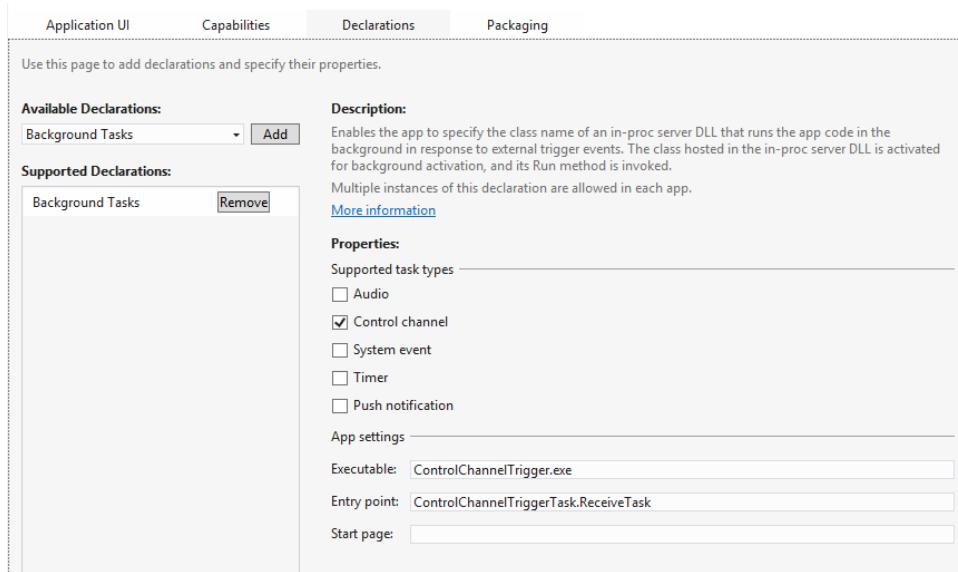


FIGURE 1-6 Background task app settings

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 `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.

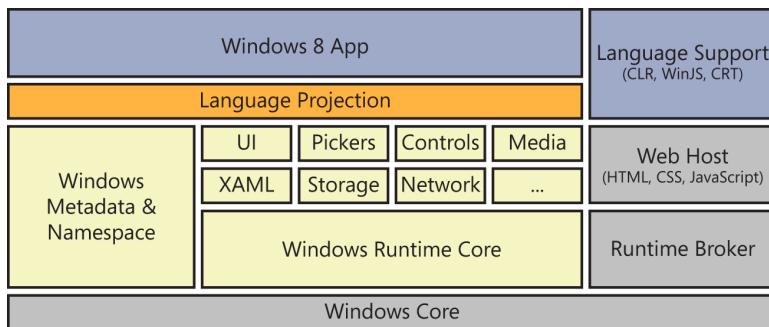


FIGURE 1-7 The Windows Runtime architecture

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.

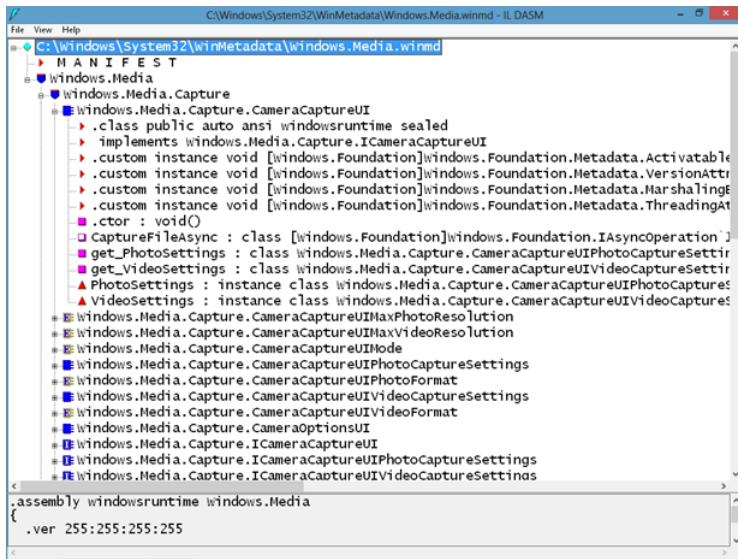


FIGURE 1-9 ILDASM displaying the outline of the Windows.Media.winmd file

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.

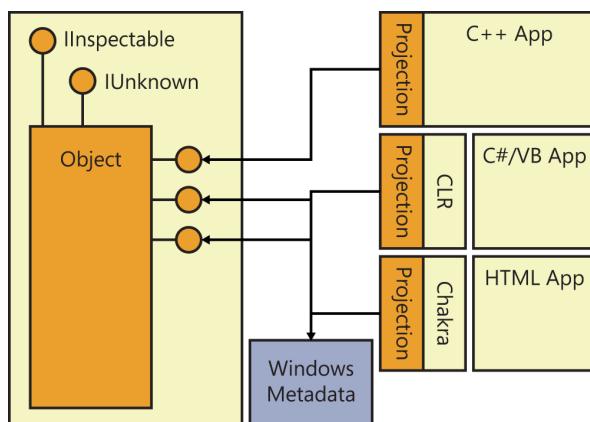


FIGURE 1-10 Projection schema

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.

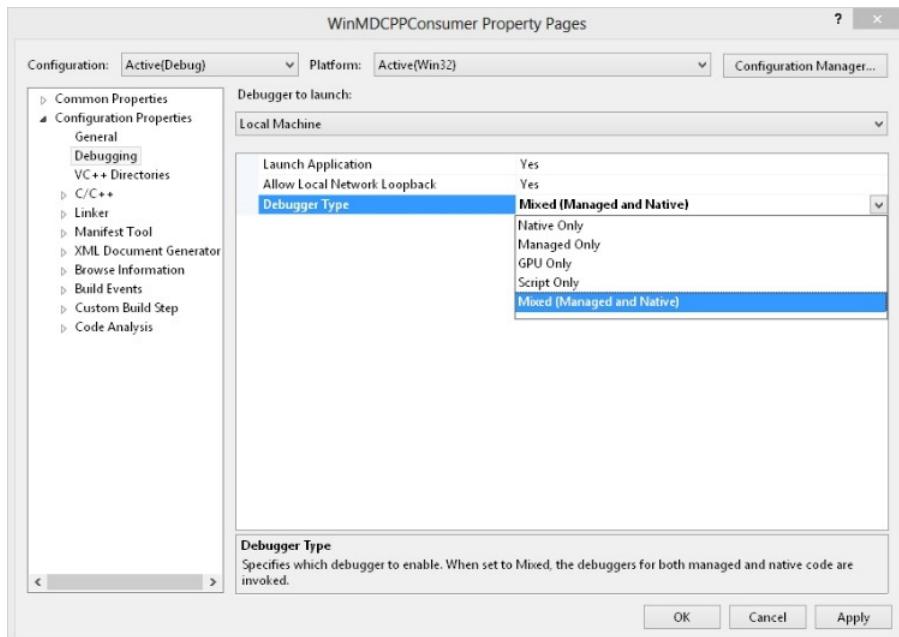


FIGURE 1-11 Debugger settings to debug mixed code

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.

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
 MethodName, 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
 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
 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 (PrintTaskOptions 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
 WebUIContentBackgroundTaskInstance, 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).

