Microsoft

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

90000

9 780735 676800

U.S.A. $39.99
Canada $41.99
[Recommended]

Certification/Windows Store Apps

Microsoft

Microsoft

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

Roberto Brunetti
Vanni Boncinelli

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*.

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

Contents

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.

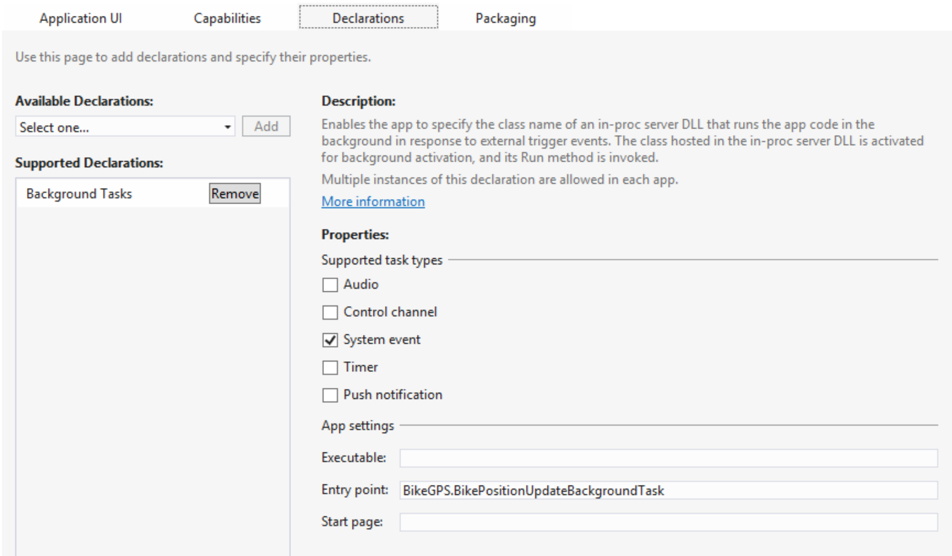

FIGURE 1-1 Background task declaration in Visual Studio App Manifest Designer

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.moveNext();
}
```

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.

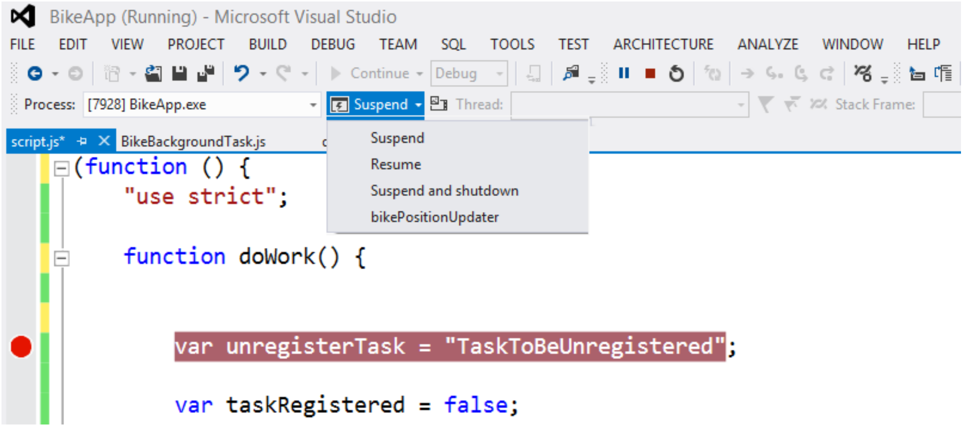

FIGURE 1-2 Visual Studio "hidden" Debug Location toolbar to start tasks

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

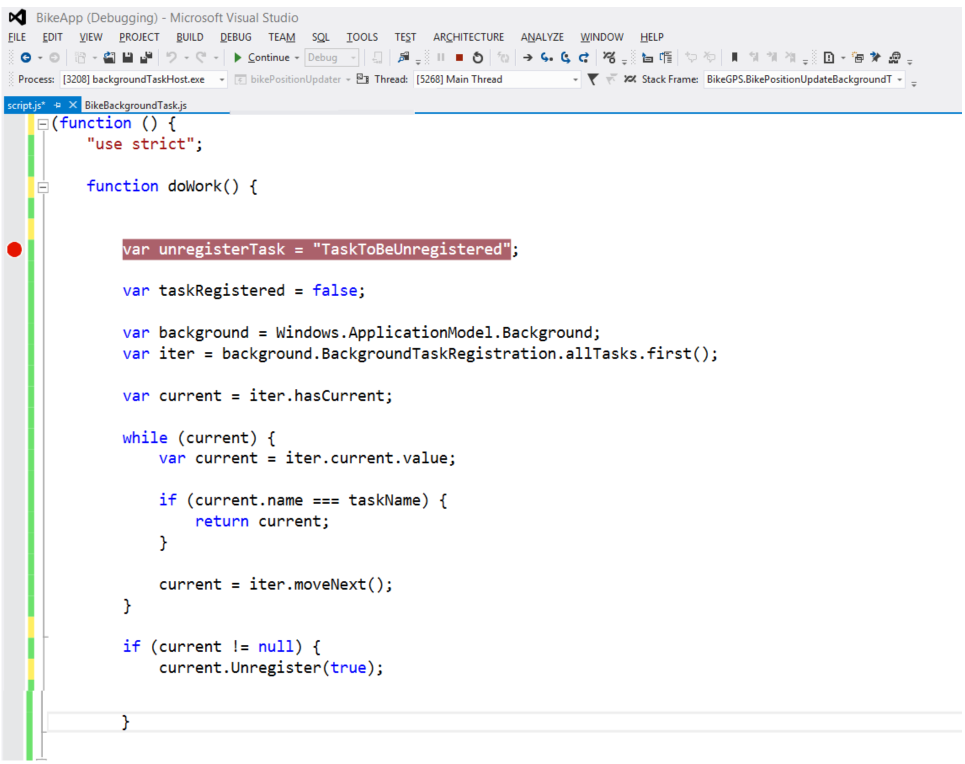

FIGURE 1-3 Debugging tasks activated directly from Visual Studio

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 *BackgroundTranferCostPolicy*. 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.

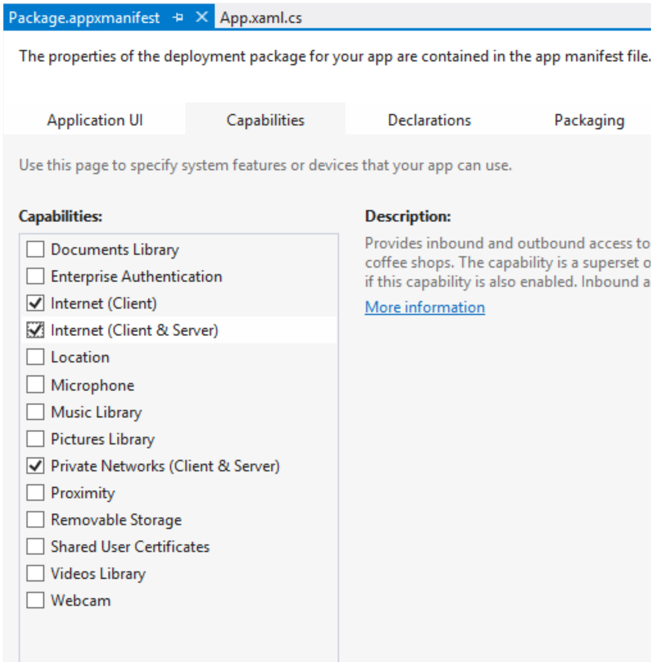

FIGURE 1-4 Capabilities for background transferring

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 g*etCurrentDownloadAsync* 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 timeiouts 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 s*erverCredential* 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 is 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, HttpClientHander,* or related classes in the *System.Net.Http* namespace in .NET Framework 4.5. The *IXMLHttpRequest2*, an extension of the classic *XMLHttpRequest*, can also be used to activate a *ControlChannelTrigger.*

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

EXAM TIP

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

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

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

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

There are two trigger types:

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

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

> ***NOTE*** **LOCK SCREEN REMOVAL**
>
> **You have also to handle the situation where the user removes the application from the lock screen.**

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

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

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

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

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

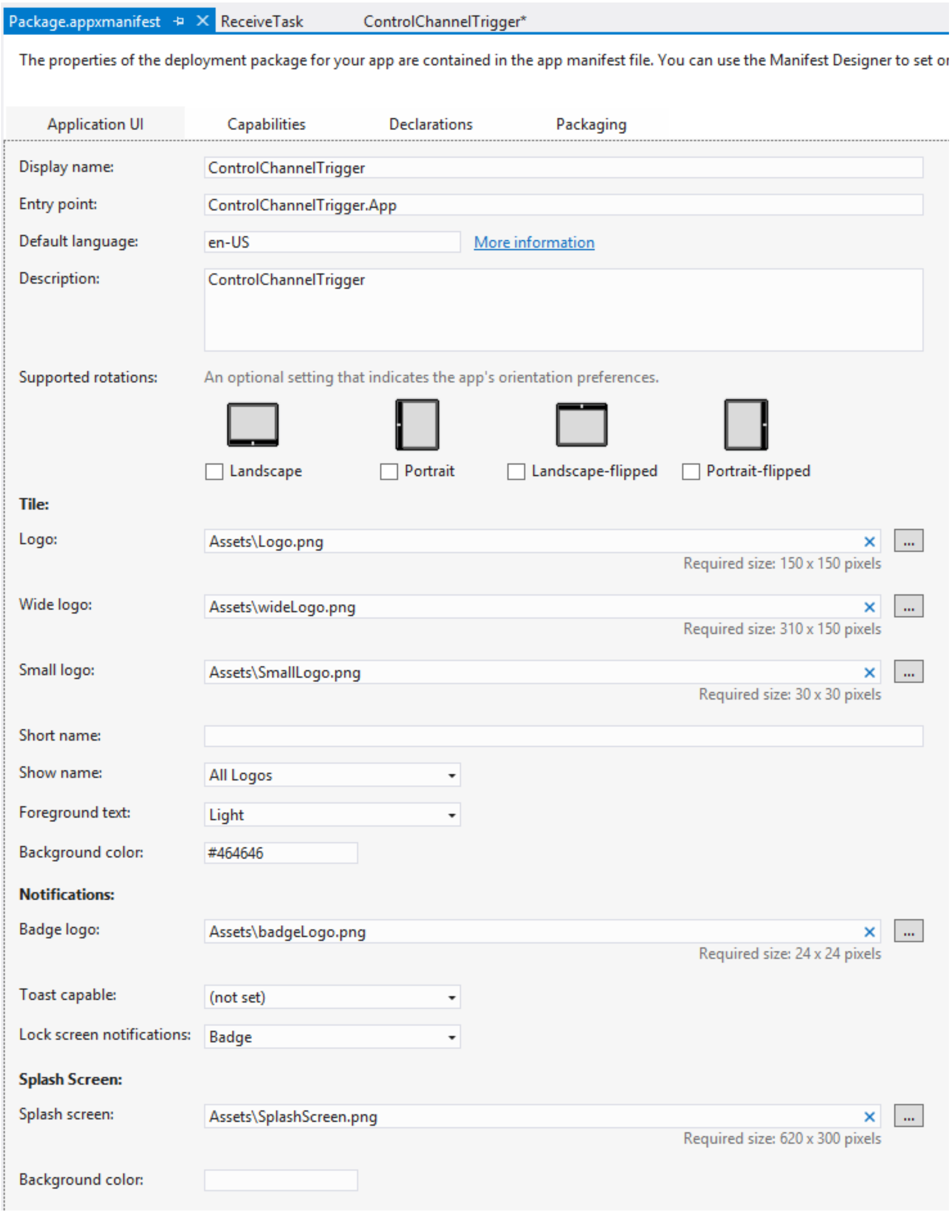

FIGURE 1-5 Badge and wide logo definition

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

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

Sample of XML code

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

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

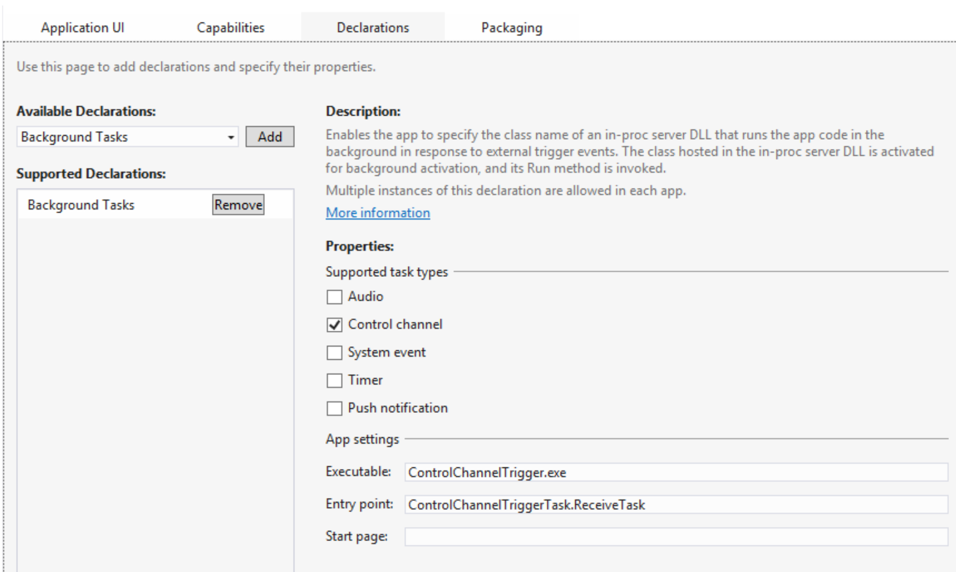

FIGURE 1-6 Background task app settings

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

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

```
var lockScreenEnabled = false;

function ClientInit() {

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

EXAM TIP

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

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

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

When the application is ready for the lock screen, you have to implement the WinMD library to perform the network operations. Remember you cannot implement a WinMD library in a Windows Store app using JavaScript. You have to create a C#, VB, or C++ WinMD component and call it from the main application.

The component code has to:

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

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

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

LISTING 1-14 Control channel creation and connection opening

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

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

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

     //
     // Create the trigger.
     //
     BackgroundTaskBuilder controlChannelBuilder = new BackgroundTaskBuilder();
         controlChannelBuilder.Name = "ReceivePacketTaskChannelOne";
     controlChannelBuilder.TaskEntryPoint =
         "ControlChannellTriggerTask.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 =
            (IControlChannelTriggerEventDetails)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 Regnicoli (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.

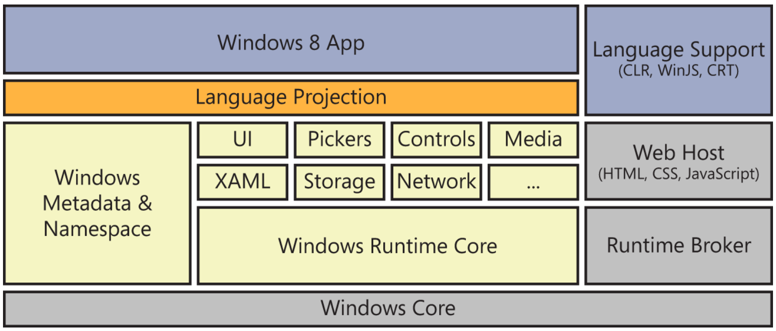

FIGURE 1-7 The Windows Runtime architecture

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.

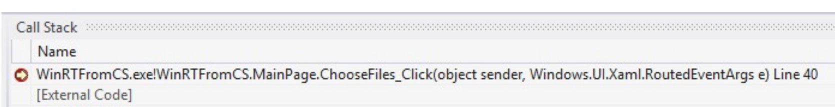

FIGURE 1-8 Call stack showing external code

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."
```

```
                    }
                    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.

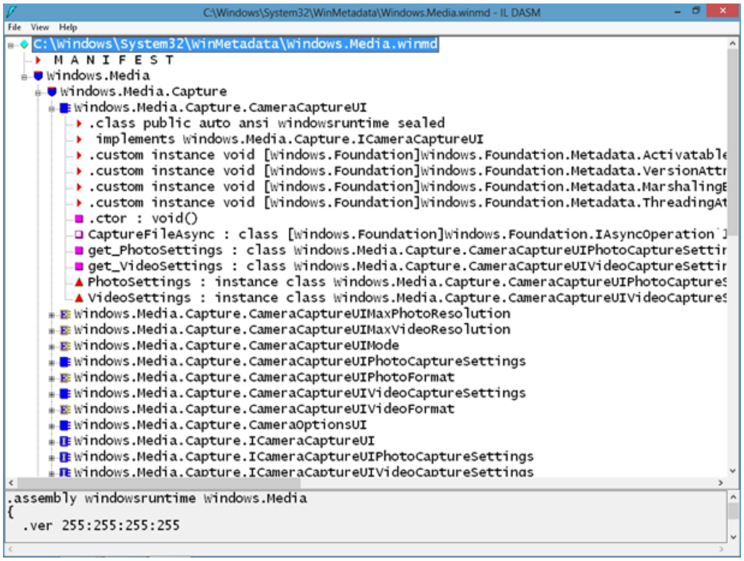

FIGURE 1-9 ILDASM displaying the outline of the Windows.Media.winmd file

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:

- ***GetIids*** 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.

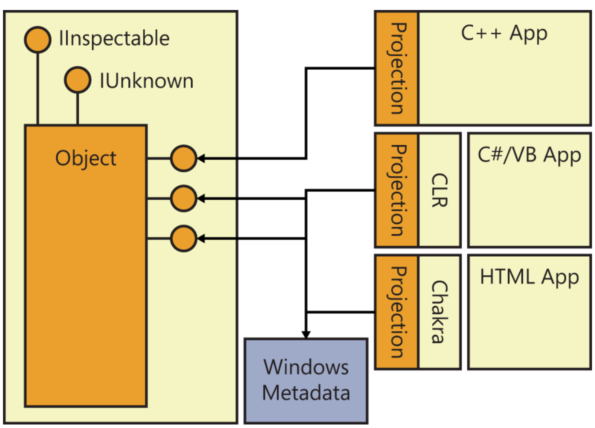

FIGURE 1-10 Projection schema

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 IsMailAddress(String email)
{
    Regex regexMail = new Regex(@"\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b");
    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->IsMailAddress("paolo@devleap.com");
}
```

Build the solution and place a breakpoint in the *IsMailAddress* 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.

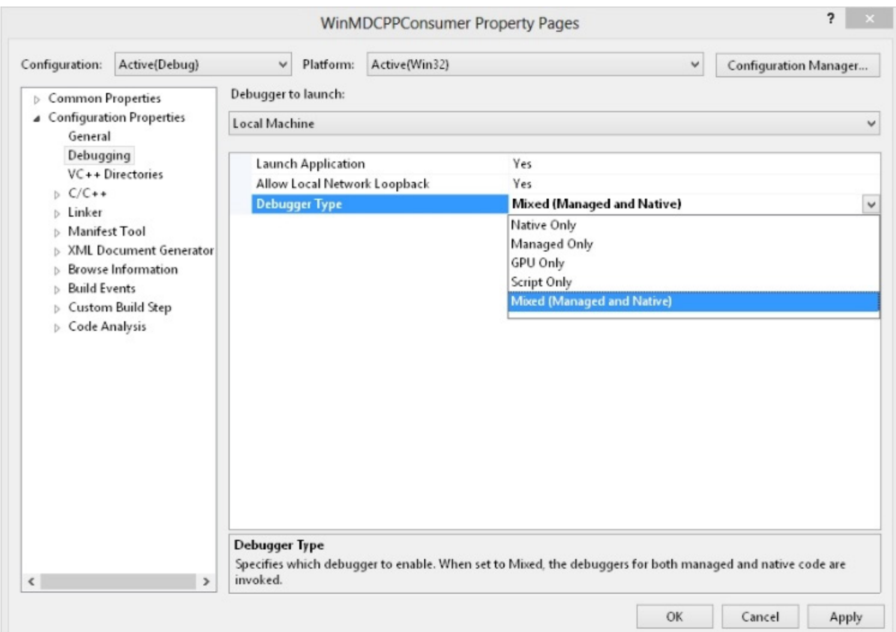

FIGURE 1-11 Debugger settings to debug mixed code

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.isMailAddress("paolo@devleap.com");
}
```

Notice that the case of the *IsMailAddress* method, defined in C#, has been translated into *isMailAddress* 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.isMailAddress("paolo@devleap.com");
    }

    app.start();
})();
```

Place a breakpoint in the *IsMailAddress* 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

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

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).