WatchKit App Development

Essentials

WatchKit App Development Essentials

WatchKit App Development Essentials – First Edition

ISBN-13: 978-1512302578

Rev 1.0

Table of Contents

Chapter 1

1. Start Here

Announced in September 2014, the Apple Watch family of devices is Apple's first foray into the market of wearable technology. The introduction of this new device category was accompanied by the release of the WatchKit framework designed specifically to allow developers to build Apple Watch app extensions to iPhone-based iOS apps.

WatchKit App Development Essentials is intended for readers with some existing experience of iOS development using Xcode and the Swift programming language. Beginning with the basics, this book provides an introduction to WatchKit apps and the WatchKit app development architecture before covering topics such as tables, navigation, user input handling, working with images, maps and menus.

More advanced topics are also covered throughout the book, including communication and data sharing between a WatchKit app and the parent iOS app, working with custom fonts and the design and implementation of custom notifications.

As with all the books in the "Development Essentials" series, WatchKit App Development Essentials takes a modular approach to the subject of WatchKit app development for the Apple Watch, with each chapter covering a self-contained topic area consisting of detailed explanations, examples and step-by-step tutorials. This makes the book both an easy to follow learning aid and an excellent reference resource.

1.1 Source Code Download

The source code and Xcode project files for the examples contained in this book are available for download at:

http://www.ebookfrenzy.com/direct/watchkit/

1.2 Feedback

We want you to be satisfied with your purchase of this book. If you find any errors in the book, or have any comments, questions or concerns please contact us at *feedback@ebookfrenzy.com*.

1.3 Errata

Whilst we make every effort to ensure the accuracy of the content of this book, it is inevitable that a book covering a subject area of this size and complexity may include some errors and oversights. Any known issues with the book will be outlined together with solutions at the following URL:

http://www.ebookfrenzy.com/errata/watchkit.html

In the event that you find an error not listed in the errata, please let us know by emailing our technical support team at *feedback@ebookfrenzy.com*.

Chapter 2

2. WatchKit Apps – An Overview

Before embarking on the creation of a WatchKit app it is important to gain a basic understanding of what a WatchKit app consists of and, more importantly, how it fits into the existing iOS application ecosystem. Within this chapter, a high level overview of WatchKit apps will be provided, together with an outline of how these apps are structured and delivered to the customer.

2.1 What is a WatchKit App?

Prior to the introduction of the Apple Watch family of devices, it was only possible to develop mobile applications for iPhone, iPad and iPod Touch devices running the iOS operating system. With the introduction of the Apple Watch, however, it is now possible for iOS developers to also create WatchKit apps.

In simplistic terms, WatchKit apps are launched on an Apple Watch device either as the result of an action by the user or in response to some form of local or remote notification. Once launched, the WatchKit app presents a user interface on the watch screen displaying information and controls with which the user can interact to perform tasks.

2.2 WatchKit and iOS Apps

It is important to understand that WatchKit apps are not standalone entities. A WatchKit app can only be created as an *extension* to an existing iOS app. It is not, therefore, possible to create a WatchKit app that is not bundled as part of a new or existing iOS application.

Consider, for example, an iPhone iOS application designed to provide the user with detailed weather information. Prior to the introduction of the Apple Watch, the only way for the user to access the information provided by the app would have been to pick up the iPhone, unlock the device, launch the iOS app and view the information on the iPhone display. Now that information can be made available via the user's Apple Watch device.

In order to make the information provided by the iOS app available via the user's Apple Watch, the developer of the weather app would add a WatchKit app extension to the iOS app, design a suitable user interface to display the information on the watch display and implement the logic to display the appropriate weather information and respond to any user interaction. Instead of having to launch the iOS app from the iPhone device to check the weather, the user can now launch the WatchKit app from the Apple Watch and view and interact with the information.

Clearly, the display size of an Apple Watch is considerably smaller than that of even the smallest of iPhone models. As such, a WatchKit app will typically display only a subset of the content available on the larger iPhone screen. For more detailed information, the user would still need to make use of the iOS application.

2.3 The WatchKit Framework

WatchKit apps are made possible by the WatchKit framework which is included as part of the iOS SDK and embedded into both Watch OS (the operating system installed on the Apple Watch) and iOS. The WatchKit framework contains a set of classes that provide the underlying functionality of both the WatchKit extension and WatchKit app. WatchKit, for example, provides the classes that can be used to construct a WatchKit app user interface (such as Button, Label and Slider classes).

WatchKit is also responsible for handling the communication between the iPhone device on which the WatchKit extension is running and the corresponding WatchKit app installed on the Apple Watch.

2.4 Understanding iOS Extensions

Extensions are a feature introduced as part of the iOS 8 SDK release and were originally intended solely to allow certain capabilities of an application to be made available for use within other applications running on the same device. The developer of a photo editing application might, for example, have devised some unique image filtering capabilities and decide that those features would be particularly useful to users of the iOS Photos app. To achieve this, the developer would implement these features in a Photo Editing extension which would then appear as an option to users when editing an image within the Photos app. Other extension types are also available for performing document storage, creating custom keyboards and embedding information from an application into the iOS notification panel.

With the introduction of the Apple Watch and WatchKit, however, the concept of extensions has now been extended to make the functionality of an iOS app available in the form of a WatchKit app.

Extensions are separate executable binaries that run independently of the corresponding iOS application. Although extensions take the form of an individual binary, they must be supplied and installed as part of an iOS application bundle. The application with which an extension is bundled is referred to as the *containing app* or *parent app*. The containing app must provide useful functionality and must not be an empty application provided solely for the purpose of delivering an extension to the user.

Once an iOS application containing a WatchKit extension has been installed on an iPhone device, it will be accessible from the Apple Watch device paired with that iPhone. When the user launches a WatchKit app on a watch device, the WatchKit framework will launch the corresponding WatchKit extension on the paired iPhone device, establish communication between the two entities and then begin the app initialization process.

2.5 Basic WatchKit App/Extension Structure

Simply by the nature of its physical size, an Apple Watch is always going to be resource constrained in terms of storage and processing power when compared to an iPhone. As previously outlined, the implementation of a WatchKit app is divided between the WatchKit app installed on the watch and the WatchKit extension bundled with the iOS app on the iPhone device. This raises the question of how the responsibilities of providing the functionality of the WatchKit app are divided between the app and the extension.

By necessity, the WatchKit app needs to be small and efficient. In fact, the WatchKit app bundle installed on the Apple Watch device consists solely of the storyboard file containing the user interface and corresponding resources (such as image and configuration files).

All of the code for providing the functionality of the WatchKit app and responding to user interaction is contained within the WatchKit extension and executed on the iPhone, with the WatchKit framework handling the communication between the app and the extension. This communication primarily takes the form of responding to user interactions (for example notifying the extension that a button was pressed in the user interface), making changes to the user interface on the watch (such as changing the text displayed on a label) and transferring data.

This separation ensures that WatchKit apps remain small and that the resource intensive code execution is performed almost entirely on the iPhone device.

2.6 WatchKit App Entry Points

There are number of different ways in which the user may enter a WatchKit app, each of which will be detailed in later chapters and can be summarized as follows:

- **Home Screen** – Once installed, the WatchKit app will be represented by an icon on the home screen of the Apple Watch display. When this icon is selected by the user the app will load and display the main user interface scene.
- **Glance** – When developing a WatchKit app, the option is available to add a *Glance* interface to the app. This is a single, non-scrollable, read-only scene that can be used to display a quick-look summary of the information normally presented by the full version of the app. Glances are accessed when the user performs an upward swiping motion on the watch display and, when tapped by the user, launch the corresponding WatchKit app.
- **Notifications** – When a notification for a WatchKit app appears on the Apple Watch device, the app will be launched when the notification is tapped.

2.7 Summary

A WatchKit app is an application designed to run on the Apple Watch family of devices. A WatchKit app cannot be a standalone application and must instead be created as an extension of an existing iOS application. WatchKit apps and extensions are built in Xcode using the WatchKit framework which, in turn, provides the classes necessary to build WatchKit apps. The WatchKit app is installed on the Apple Watch device and consists of a storyboard file containing the user interface of the app together with a set of resource files. The WatchKit extension, on the other hand, is bundled with the corresponding iOS app and resides and executes on the iPhone device. The extension is a separate binary from the main iOS app and contains all of the code logic to implement the behavior of the WatchKit app. When the app is launched on the Apple Watch, the WatchKit framework launches the corresponding extension on the iPhone device and handles the communication between the app and extension.

Chapter 3

3. Building an Example WatchKit App

Having outlined the basic architecture for a WatchKit app in the previous chapter, it is now time to start putting some of this knowledge to practical use through the creation of a simple example app.

The project created in this chapter will work through the creation of a basic WatchKit app that does nothing more than display a message and an image on an Apple Watch display.

3.1 Creating the WatchKit App Project

Start Xcode and, on the Welcome screen, select the *Create a new Xcode project* option. On the template screen choose the *Application* option located under *iOS* in the left hand panel and select *Single View Application.* Click *Next,* set the product name to *WatchKitSample,* enter your organization identifier and make sure that the *Devices* menu is set to *Universal* so that the user interface will be suitable for deployment on all iPhone and iPad screen sizes. Before clicking *Next*, change the *Language* menu to *Swift*. On the final screen, choose a file system location in which to store the project files and click on the *Create* button to proceed to the main Xcode project window.

3.2 Designing the iOS App User Interface

The next step in the project is to design the user interface for the iOS app. This layout is contained within the *Main.storyboard* file and is listed in the Project Navigator panel on the left hand side of the main Xcode window. Locate and click on this file to load it into the Interface Builder environment. Once loaded, locate the Label view object in the Object Library panel and drag and drop it so that it is centered in the layout canvas. Once positioned, double-click on the label and change the text so that it reads “Welcome to WatchKit” as illustrated in Figure 3-1:

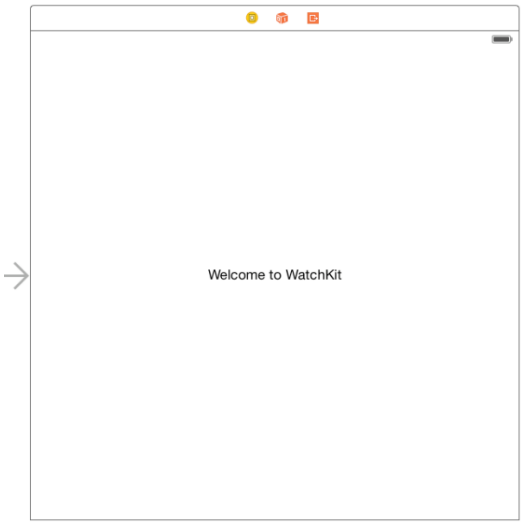

Figure 3-1

Select the new label in the layout canvas and display the *Resolve Auto Layout Issues* menu by clicking on the button in the lower right hand corner of the Interface Builder panel as indicated in Figure 3-2:

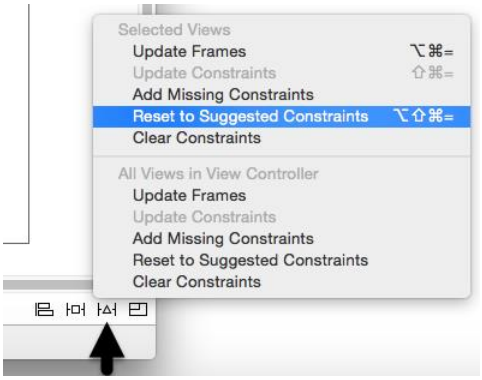

Figure 3-2

From the resulting menu, select the *Reset to Suggested Constraints* option. This will set up recommended layout constraints so that the label remains centered both horizontally and vertically within the screen regardless of whether the application is running on an iPhone or iPad display.

The user interface for the iOS application is now complete. Verify this by running the application on an iPhone device or iOS Simulator session before continuing.

3.3 Adding the WatchKit Extension and App

With a rudimentary iOS application created the next step is to add the WatchKit extension and WatchKit app to the project. This is achieved by selecting the Xcode *New -> File -> Target…* menu option. From within the resulting panel, select the *Apple Watch* option listed under *iOS* in the left hand panel as outlined in Figure 3-3:

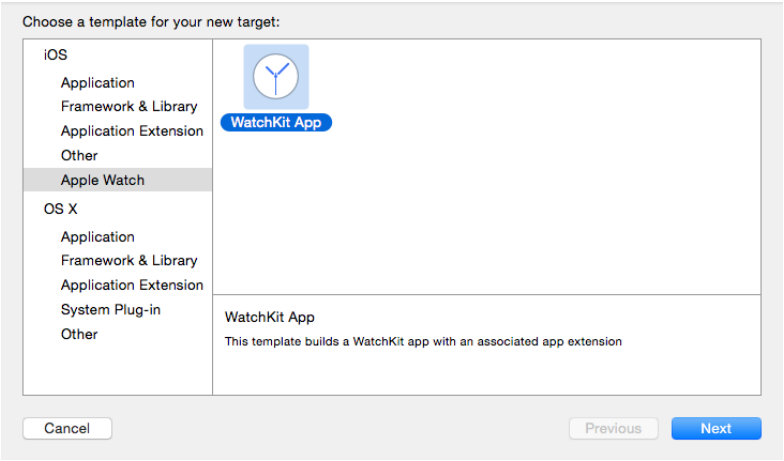

Figure 3-3

With the WatchKit App option selected, click on the *Next* button to proceed to the options screen shown in Figure 3-4. The product name will be preset based on the name of the containing app and cannot be changed. Before clicking on the *Finish* button, make sure that the Include Notification Scene and Include Glance Scene options are switched off (Glances and Notifications will be introduced in later chapters) and that the language menu is set to *Swift.*

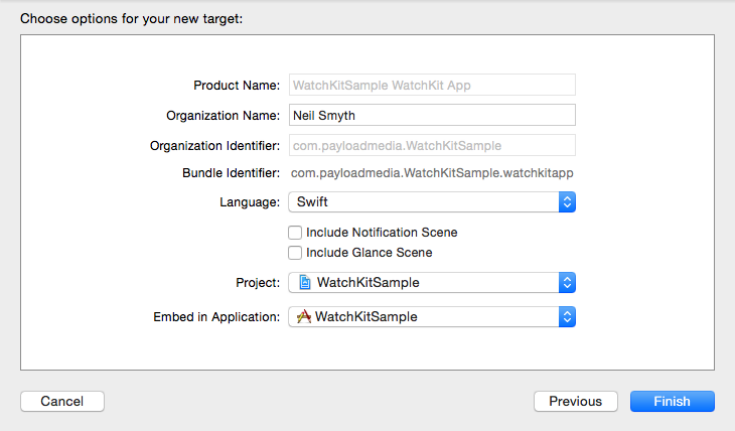

Figure 3-4

As soon as the extension target has been created, a new panel will appear requesting permission to activate the new scheme for the extension target. Every target within an Xcode project has associated with it a scheme which defines how that target is to be built. When an extension target is added to a project, Xcode automatically creates a corresponding scheme so that the extension can be built and run. Activate this scheme now by clicking on the *Activate* button in the request panel:

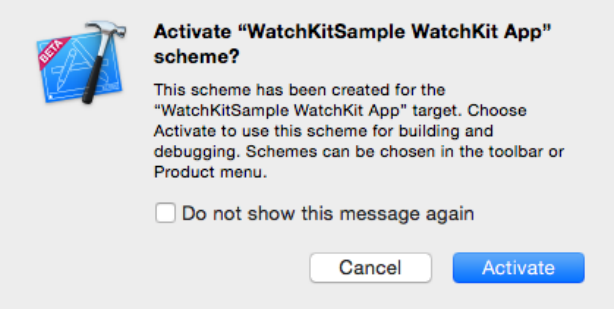

Figure 3-5

A review of the project files within the Project Navigator panel will reveal that new folders have been added for the WatchKit Extension and the WatchKit App (Figure 3-6), each of which contains the files that will need to be modified to implement the appearance and behavior of the WatchKit app:

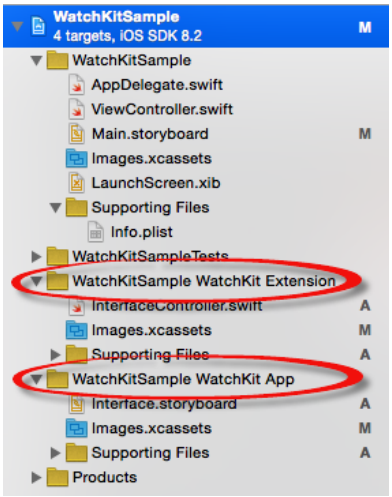

Figure 3-6

3.4 Designing the WatchKit App Storyboard

The next step in the project is to design the user interface for the WatchKit app. This is contained within the *Interface.storyboard* file located under the *WatchKitSample WatchKit App* folder within the Project Navigator. Locate and select this file to load it into the Interface Builder tool where the scene will appear as illustrated in Figure 3-7:

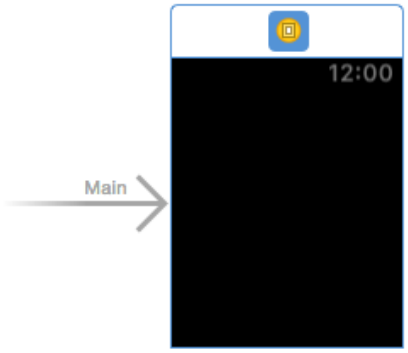

Figure 3-7

Designing the user interface for a WatchKit app involves dragging objects from the Object Library panel onto the layout canvas. When user interface objects are added to the layout canvas they are stacked vertically. These elements are then positioned at runtime by WatchKit based on the available display space combined with any sizing and positioning attributes declared during the storyboard design phase.

For the purposes of this example, the user interface will be required to display an image and a label. Locate the Image object in the Object Library panel and drag and drop it onto the scene layout. Repeat this step to position a Label object immediately beneath the Image object. Double click on the newly added Label object and change the text so that it reads “Hello WatchKit” such that the layout matches that of Figure 3-8:

Figure 3-8

Before testing the app, some additional attributes need to be set on the objects in the user interface. The first step is to configure the Image object to display an image. Before this can be configured, however, the image file needs to be added to the project. The image file is named *watch_image@2x.png* and can be found in the *sample_images* folder of the sample code archive which can be downloaded from the following URL:

http://www.ebookfrenzy.com/direct/watchkit/index.php

Within the Project Navigator panel, select the *Images.xcassets* entry listed under *WatchKitSample WatchKit App* so that the image asset catalog loads into the main panel. Locate the *watch_image@2x.png* image file in a Finder window and drag and drop it onto the left hand panel in the asset catalog as illustrated in Figure 3-9:

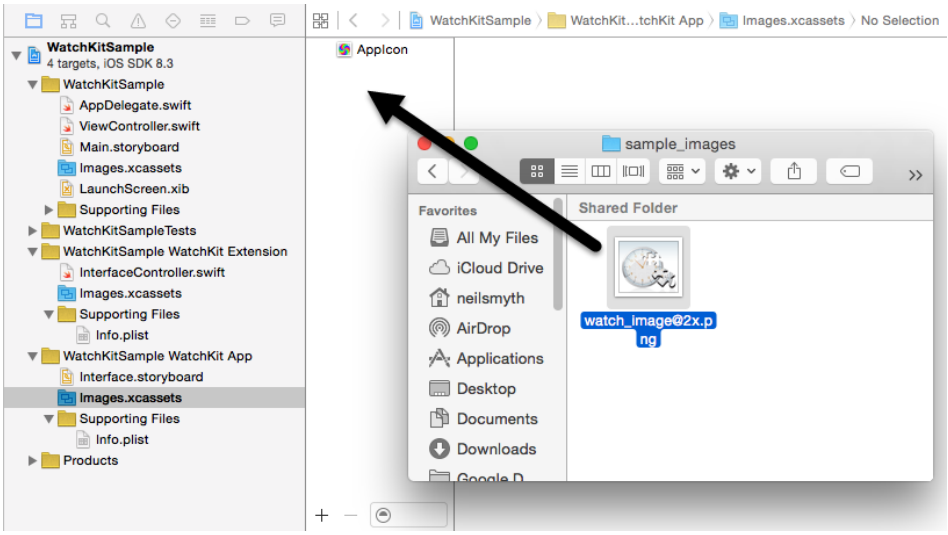

Figure 3-9

With the image file added to the project, the Image object needs to be configured to display the image when the app runs. Select the Image object in the storyboard scene and display the Attributes Inspector in the utilities panel (*View -> Utilities -> Show Attributes Inspector*). Within the inspector panel, use the drop down menu for the Image attribute to select the *watch_image* option:

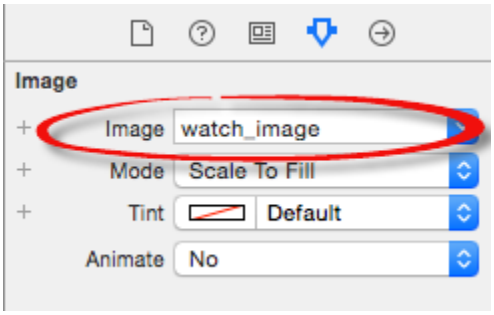

Figure 3-10

Finally, select the Label object in the scene and use the Attribute Inspector panel to change the *Alignment* attribute so that the text is centered within the label. Having set this attribute, a review of the scene will show that the text is still positioned on the left of the layout. The reason for this is that the text has been centered within the label but the Label object itself is still positioned on the left side of the display. To correct this, locate the *Position* section in the Attributes Inspector panel and change the *Horizontal* attribute from *Left* to *Center*. Figure 3-11 shows the Attributes Inspector panel with these attributes set:

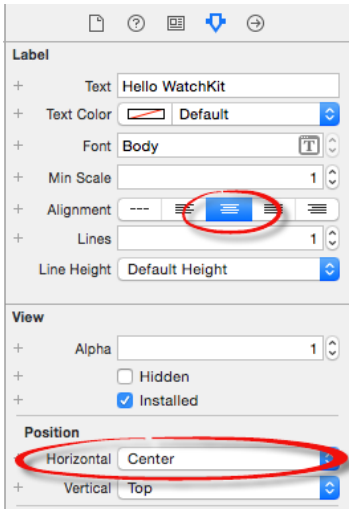

Figure 3-11

3.5 Running the WatchKit App

All that remains is to run the WatchKit app and make sure that it appears as expected. For the purposes of this example this will be performed using the simulator environment. In order to test the WatchKit app, the run target may need to be changed in the Xcode toolbar. Select the current scheme in the toolbar and use the drop down menu (Figure 3-12) to select the *WatchKitSample WatchKit App -> iPhone 6* option:

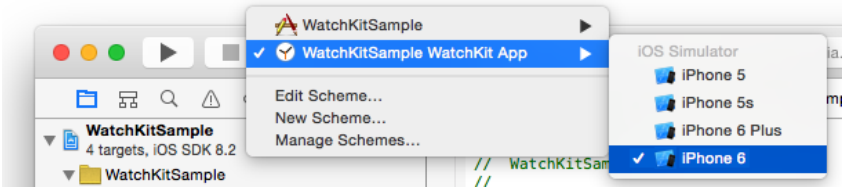

Figure 3-12

With the WatchKit app selected, click on the run button. Once the simulator has loaded, two windows should appear, one representing the iPhone 6 device and the other the Apple Watch device. After a short delay, the WatchKit app should appear on the watch simulator display as illustrated in Figure 3-13:

Figure 3-13

If the WatchKit simulator window fails to appear, use the iOS Simulator *Hardware -> External Displays* menu option to select one of the Apple Watch options.

By default Xcode will launch the 42mm Apple Watch simulator. To also test the layout on the 38mm Apple Watch model, select the iOS Simulator *Hardware -> External Displays -> Apple Watch – 38 mm* menu option and then stop and restart the WatchKit app from within Xcode.

3.6 Running the App on a Physical Apple Watch Device

In order to test the app on a physical Apple Watch device, connect the iPhone with which the Apple Watch is paired to the development system and select it as the target device within the Xcode toolbar panel (Figure 3-14).

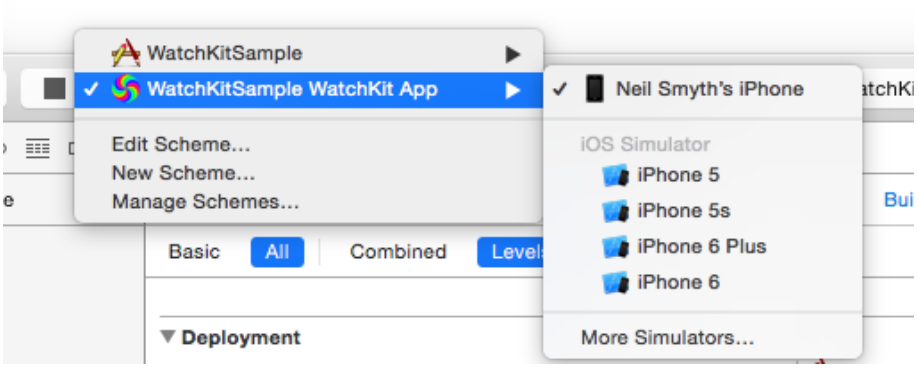

Figure 3-14

With *WatchKitSample WatchKit App* still selected as the run target, click on the run button and wait for the app icon to appear on the Apple Watch home screen. Once the icon appears, tap on it to launch the app.

Depending on the version of Xcode that you are using, the following error may occur when attempting to run the app on a watch device:

```
Embedded Binary Validation Utility error: WatchKit apps must have a deployment target
equal to iOS 8.2 (was 8.3).
```

If this error appears in Xcode, select the WatchKitSample target entry at the top of the Project Navigator panel to display the target settings. Within the settings panel, select the *Build Settings* screen as highlighted in Figure 3-15:

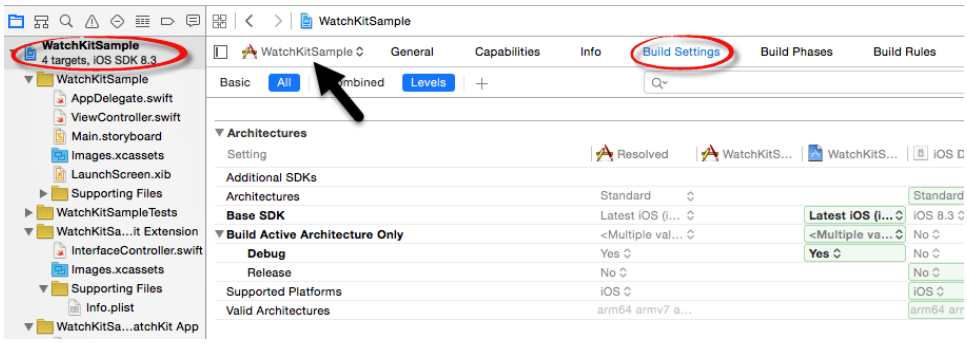

Figure 3-15

By default, the build settings for the iOS app will be displayed. Using the menu in the upper left hand corner of the settings panel (indicated by the arrow in Figure 3-15 above), select the *WatchKitSample WatchKit App* option as highlighted in Figure 3-16:

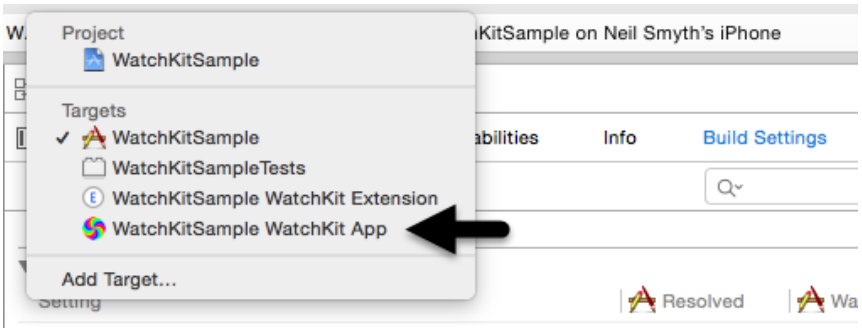

Figure 3-16

With the WatchKit app build settings displayed, enter *iOS Deployment Target* into the search box and change the deployment target settings to *iOS 8.2*:

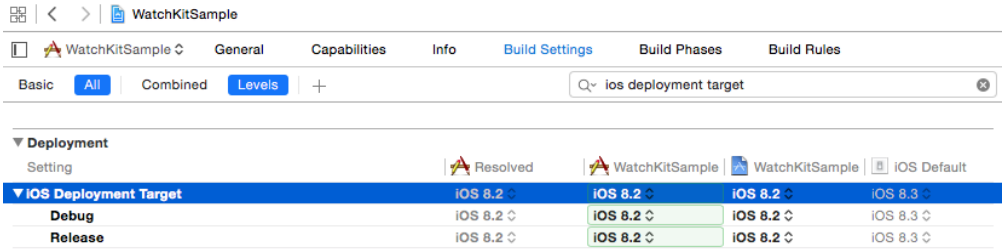

Figure 3-17

Having changed the deployment target the app should compile and run on the Apple Watch device.

3.7 Setting the Scene Title and Key Color

The area at the top of the Apple Watch display containing the current time is the *status bar* and the area to the left of the time is available to display a title string. To set this property, click on the scene within the storyboard so that it highlights in blue, display the Attributes Inspector panel and enter a title for the scene into the *Title* field:

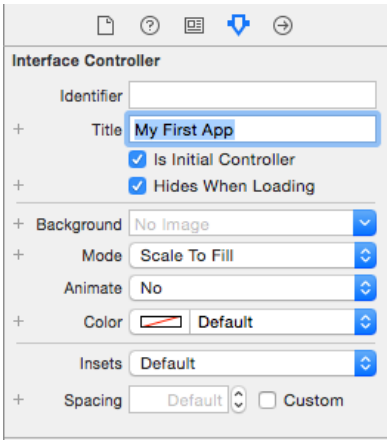

Figure 3-18

The foreground color of all of the scene titles in a WatchKit app may be configured by setting the *global tint* attribute for the storyboard file. To set this property, select the *Interface.storyboard* file in the Project Navigator panel and display the File Inspector panel (*View -> Utilities -> Show File Inspector*). Within the File Inspector panel change the color setting for the *Global Tint* attribute (Figure 3-19) to a different color.

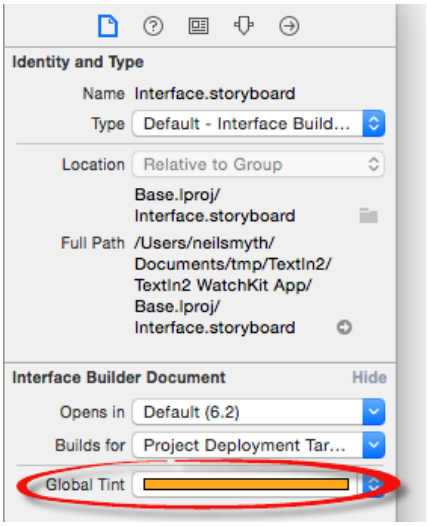

Figure 3-19

Next time the app runs, all of the titles in the scenes that make up the storyboard will be rendered using the selected foreground color.

The global tint color is also adopted by the app name when it is displayed in the short look notification panel, a topic area that will be covered in detail in the chapter entitled *An Overview of Notifications in WatchKit*.

3.8 Adding App Icons to the Project

Every WatchKit app must have associated with it an icon. This icon represents the app on the Apple Watch Home screen and identifies the app in notifications and within the iPhone-based Apple Watch app. A variety of icon sizes may need to be created depending on where the icon is displayed and the size of Apple Watch on which the app is running. The various icon size requirements are as outlined in Table 3-1:

Icon	38mm Watch	42mm Watch
Home Screen	80 x 80 pixels	80 x 80 pixels
Long Look Notification	80 x 80 pixels	88 x 88 pixels
Short Look Notification	172 x 172 pixels	196 x 196 pixels
Notification Center	48 x 48 pixels	55 x 55 pixels

Table 3-1

In addition to the icons in Table 3-1, icons are also required for the Apple Watch app on the paired iPhone device (a topic outlined in the chapter entitled *Configuring Preferences with the WatchKit Settings Bundle*). Two versions of the icon are required for this purpose so that the icon can be represented on both iPhone (@2x) and iPhone Plus (@3x) size models:

Icon	iPhone @2x	iPhone Plus @3x
Apple Watch App	58 x 58 pixels	87 x 87 pixels

Table 3-2

Since the app created in this chapter does not make use of notifications, only Home Screen and Apple Watch app icons need to be added to the project. The topic of notification icons will be addressed in greater detail in the chapter entitled *A WatchKit Notification Tutorial*.

The home screen icon needs to be circular and 80x80 pixels in size with a 24-bit color depth. The image must be in PNG format with a file name ending with "@2x", for example *homeicon@2x.png*.

Icons are stored in the image asset catalog of the WatchKit app target. Access the image set in the asset catalog by selecting the *Images.xcassets* file listed under the *WatchKitSample WatchKit App* folder in the project navigator panel. Within the asset catalog panel (Figure 3-20), select the *AppIcon* image set:

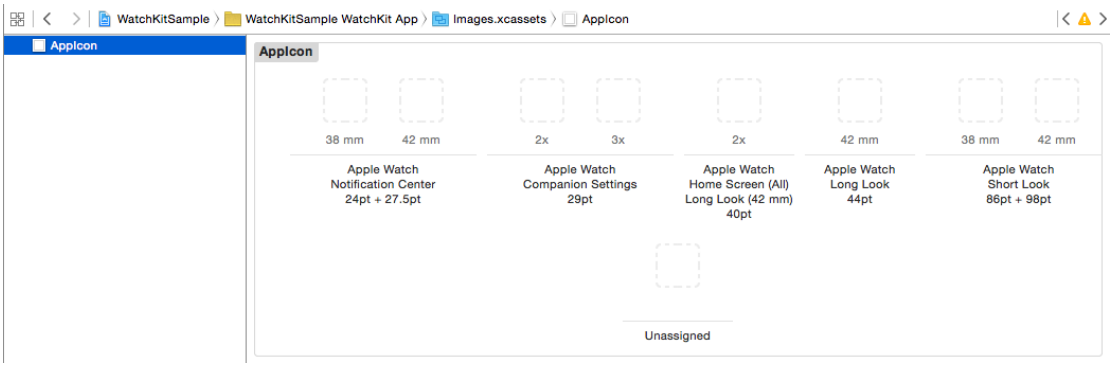

Figure 3-20

To add icons, locate them in a Finder window and drag and drop them onto the corresponding location within the image set. For the purposes of this example, app icons can be found in the *app_icons* folder of the sample code download.

Once the icons have been located, drag and drop the icon file named *HomeIcon@2x.png* onto the *Apple Watch Home Screen (All)* image location within the image asset catalog as shown in Figure 3-21:

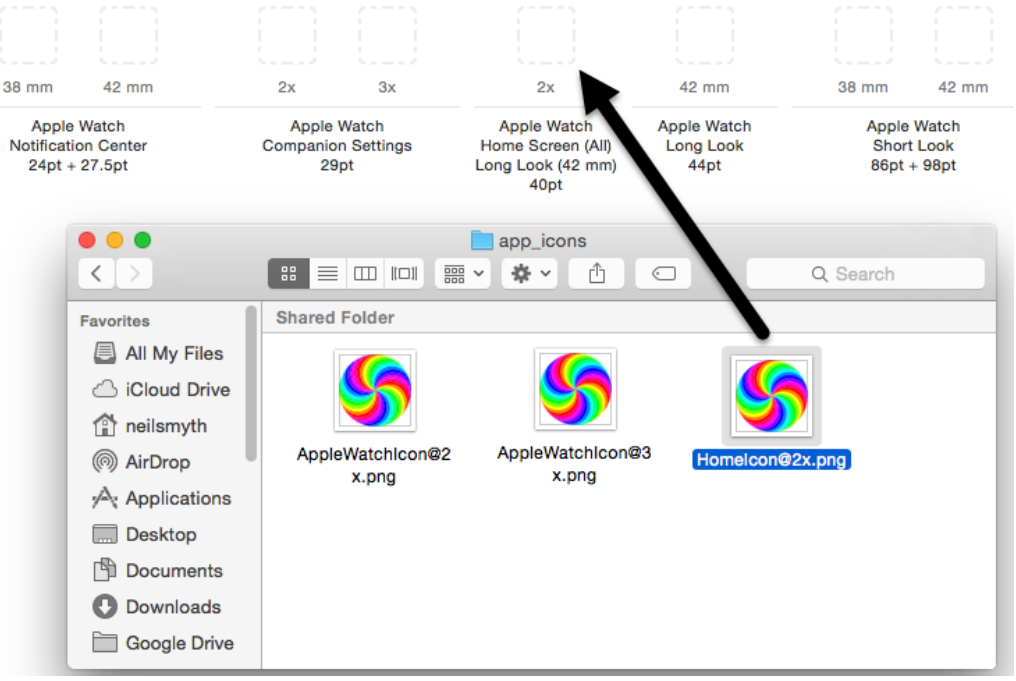

Figure 3-21

The two Apple Watch app icons are named *AppleWatchIcon@2x.png* and *AppleWatchIcon@3x.png* and should be placed in the *Apple Watch Companion Settings* 2x and 3x image locations respectively. Once these icons have been added the three icon categories in the AppIcon image set should resemble Figure 3-22:

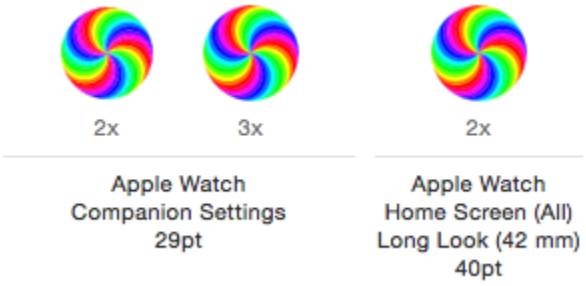

Figure 3-22

When the sample WatchKit app is now compiled and run on a physical Apple Watch device the app will be represented on the device Home Screen by the provided icon.

3.9 Summary

This chapter has worked through the steps involved in creating a simple WatchKit app and running it within the simulator environment. A WatchKit app is added as a target to an existing iOS app project. When a WatchKit target is added, Xcode creates an initial storyboard for the WatchKit app user interface and the basic code for the WatchKit Extension template. The user interface for the WatchKit app is designed in the storyboard file by selecting and positioning UI objects in the Interface Builder environment and setting attributes where necessary to configure the appearance and position of the visual elements. In order to test run a WatchKit app, the appropriate run target must first be selected from the Xcode toolbar.

Before a WatchKit app can be published, app icons must be added to the image asset catalog of the WatchKit App target. These icons must meet strict requirements in terms of size and format, details of which have also been covered in this chapter.

Chapter 4

4. An Overview of the WatchKit App Architecture

The previous chapters have explained that a WatchKit app consists of two main components, the WatchKit app itself residing on the Apple Watch and the WatchKit extension installed on the iPhone device. We have also established that the wireless communication between the WatchKit app and the WatchKit extension is handled transparently by the WatchKit framework.

It has also been established that the WatchKit app is primarily responsible for displaying the user interface while the programming logic of the app resides within the WatchKit extension.

It is less clear at this point, however, how the user interface elements in the app are connected to the code in the extension. In other words, how the project is structured such that tapping on a button in a scene causes a specific method in the extension to be called. Similarly, we need to understand how the code within the extension can manipulate the properties of a visual element in a storyboard scene, for example changing the text displayed on a Label interface object. These topics will be covered in this chapter and then put into practice in the next chapter entitled *An Example Interactive WatchKit App*.

This chapter will also provide an overview of the lifecycle of a WatchKit app and outline the ways in which this can be used to perform certain initialization tasks when a WatchKit app is launched.

4.1 Basic WatchKit App Architecture

As discussed in previous chapters, the WatchKit app itself consists only of the storyboard file and a set of resource files. The storyboard contains one or more scenes, each of which represents a different screen within the app and may, optionally, provide mechanisms for the user to transition from one scene to another. Clearly this does not provide any functionality beyond presenting user interfaces to the user. The responsibility of providing the behavior behind a user interface scene so that the app actually does something useful belongs to the *interface controller*.

4.2 WatchKit Interface Controllers

Each scene within a storyboard must have associated with it an interface controller instance. Interface controllers are subclassed from the WatchKit framework WKInterfaceController class and contain the code that allows the WatchKit app to perform tasks beyond simply presenting a user interface to the user. This provides a separation between the user interfaces (the storyboard) and the logic code (the interface controllers). In fact, interface controllers are similar to view controllers in iOS applications.

The interface controllers for a WatchKit app reside within the WatchKit extension associated with the app and are installed and executed on the iPhone device with which the Apple Watch is paired. It is the responsibility of the interface controller to respond to user interactions in the corresponding user interface scene and to make changes to the visual elements that make up the user interface. When a user taps a button in a WatchKit scene, for example, a method in the interface controller will be called by the WatchKit framework in order to perform some form of action. In the event that a change needs to be made to a user interface element, for example to change the text displayed on a label, the interface controller will make the necessary changes and the WatchKit framework will transmit those changes to the WatchKit app where the update will be performed.

This sounds good in theory but does not explain how the connections between the elements in the user interface on the watch device and the interface controller on the iPhone are established. This requires an understanding of the concepts of *outlets* and *action methods*.

4.3 WatchKit Action Methods

Creation of a WatchKit app typically involves designing the user interface scenes using the Interface Builder tool and writing the code that provides the logic for the app in the source code files of the interface controller classes. In this section we will begin to look at how the user interface scene elements and the interface controller code interact with each other.

When a user interacts with objects in a scene of a WatchKit app, for example touching a button control, an *event* is triggered. In order for that event to achieve anything, it needs to trigger a method call on the interface controller class. Use of a technique known as *target-action* provides a way to specify what happens when such events are triggered. In other words, this is how you connect the objects in the user interface you have designed in the Interface Builder tool to the back end Swift code you have implemented in the corresponding interface controller class. Specifically, this allows you to define which method of the interface controller gets called when a user interacts in a certain way with a user interface object.

The process of wiring up a user interface object to call a specific method on an interface controller is achieved using something called an *Action.* Similarly, the target method is referred to as an *action method*. Action methods are declared within the interface controller class using the IBAction keyword, for example:

```
@IBAction func buttonPress() {
    println("Button Pressed")
    // Perform tasks in response to a button press
}
```

4.4 WatchKit Outlets

The opposite of an *Action* is an *Outlet*. As previously described, an Action allows a method in an interface controller instance to be called in response to a user interaction with a user interface element. An Outlet, on the other hand, allows an interface controller to make changes to the properties of a user interface element. An interface controller might, for example, need to set the text on a Label object. In order to do so an Outlet must first have been defined using the *IBOutlet* keyword. In programming terms, an *IBOutlet* is simply an instance variable that references the user interface object to which access is required. The following line demonstrates an outlet declaration for a label:

```
@IBOutlet weak var myLabel: WKInterfaceLabel!
```

Once outlets and actions have been implemented and connected, all of the communication required to make these connections work is handled transparently by WatchKit. Figure 4-1 provides a visual representation of actions and outlets:

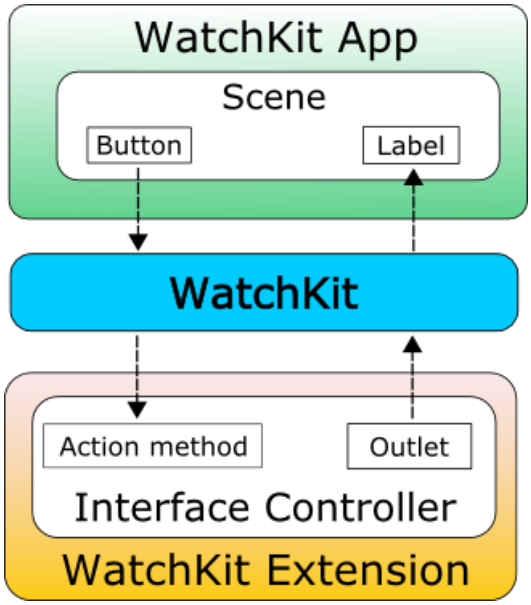

Figure 4-1

Outlets and actions can be created visually with just a few mouse clicks from within Xcode using the Interface Builder tool in conjunction with the Assistant Editor panel, a topic which will be covered in detail in the chapter entitled *An Example Interactive WatchKit App*.

4.5 The Lifecycle of a WatchKit App

The lifecycle of a WatchKit app and the corresponding WatchKit extension is actually very simple. When a WatchKit app is launched on the device, a scene will be loaded from within the storyboard file. When the scene has loaded, the WatchKit framework will request that the extension corresponding to the app be launched on the iPhone device (or woken up if it is currently suspended). The extension is then instructed to create the interface controller associated with the scene that was loaded.

As long as the user is interacting with the WatchKit app the extension will continue to run on the iPhone. When the system detects that the user is no longer interacting with the watch, or the user exits the app, the interface controller is deactivated and the extension suspended.

At various points during this initialization and de-initialization cycle, calls will be made to specific lifecycle methods declared within the interface controller class where code can be added to perform initialization and clean up tasks. These methods are as follows:

- **init()** - The first method called on the interface controller when the scene is to be displayed. This method can be used to perform initialization tasks in preparation for the scene being displayed to the user. It is also possible to make changes to user interface objects via outlets from within this method.
- **awakeWithContext() -** This method is called after the call to the init() method and may optionally be passed additional context data. This is typically used when navigating from one scene to another in a multi-scene app and allows data to be passed from the interface controller of the currently displayed scene to the interface controller of the destination scene. Access to user interface objects is available from within this method.
- **willActivate() –** Called immediately before the scene is presented to the user on the Apple Watch device. This is the recommended method for making final minor changes to the elements in the user interface. Access to user interface objects is available from within this method.
- **didDeactivate() –** The last method called when the user exits the current scene or the system detects that, although the app is still running, the user is no longer interacting with the Apple Watch device. This method should be used to perform

any cleanup tasks necessary to ensure a smooth return to the scene at a later time. Access to user interface objects is not available from within this method. Calls to this method may be triggered for testing purposes from within the WatchKit simulator environment by locking the simulator via the *Hardware -> Lock* menu option.

The diagram in Figure 4-2 illustrates the WatchKit app lifecycle as it corresponds to the WatchKit extension and interface controller:

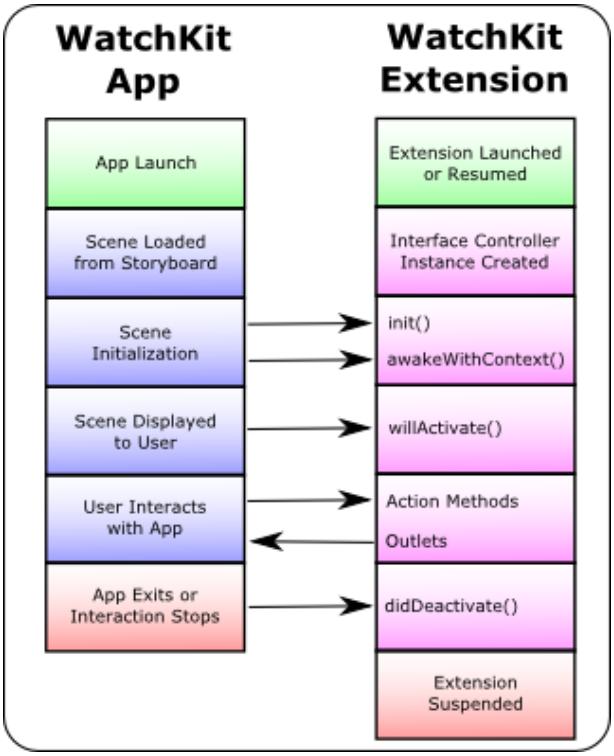

Figure 4-2

It should be noted that, with the exception of the app launch and extension suspension phases in the above diagram, the same lifecycle sequence is performed each time a scene is loaded within a running WatchKit app.

4.6 WatchKit Extension Guidelines

A key point to note from the lifecycle description is that the WatchKit extension is suspended when the user either exits or stops interacting with the WatchKit app. It is important, therefore, to avoid performing any long term tasks within the extension. Any tasks that need to continue executing after the extension has been suspended should be passed to the containing iOS app to be handled. Details of how this can be achieved are outlined in the *WatchKit App and Parent iOS App Communication* chapter of this book.

4.7 Summary

Each scene within a WatchKit app has associated with it an interface controller within the WatchKit extension. The interface controller contains the code that provides the underlying logic and behavior of the WatchKit app.

The interactions between the current scene of a WatchKit app running on an Apple Watch and the corresponding interface controller within the WatchKit extension are implemented using actions and outlets. Actions define the methods that get called in response to specific actions performed by the user within the scene. Outlets, on the other hand, provide a mechanism for the interface controller code to access and modify the properties of user interface objects.

When a WatchKit app is launched, and each time a new scene is loaded, the app has a lifecycle consisting of initialization, user interaction and termination. At multiple points within this lifecycle, calls are made to specific methods within the interface controller of the current scene thereby providing points within the code where initialization and de-initialization tasks can be performed.

Chapter 5

5. An Example Interactive WatchKit App

Having covered actions and outlets in the previous chapter, it is now time to make practical use of these concepts. With this goal in mind, this chapter will work through the creation of a WatchKit app intended to demonstrate the way in which the Interface Builder and Assistant Editor features of Xcode work together to simplify the creation of actions and outlets to implement interactive behavior within a WatchKit app.

5.1 About the Example App

The purpose of the WatchKit app created in this chapter is to calculate a recommended gratuity amount when dining at a restaurant. A mechanism will be provided for the user to select the amount of the bill and then tap a button to display the recommended tip amount (assuming a percentage of 20%).

5.2 Creating the TipCalcApp Project

Start Xcode and, on the Welcome screen, select the *Create a new Xcode project* option. On the template screen choose the *Application* option located under *iOS* in the left hand panel followed by *Single View Application* in the main panel. Click *Next,* set the product name to *TipCalcApp,* enter your organization identifier and make sure that the *Devices* menu is set to *Universal*. Before clicking *Next*, change the *Language* menu to Swift if necessary. On the final screen, choose a location in which to store the project files and click on the *Create* button to proceed to the main Xcode project window.

5.3 Adding the WatchKit App Target

For the purposes of this example we will assume that the iOS app has already been implemented. The next step, therefore, is to add the WatchKit app target to the project. Within Xcode, select the *File -> New -> Target…* menu option. In the target template dialog, select the *Apple Watch* option listed beneath the *iOS* heading. In the main panel, select the *WatchKit App* icon and click on *Next.* On the subsequent screen (Figure 5-1) turn off the *Include Glance Scene* and *Include Notification Scene* options before clicking on the *Finish* button:

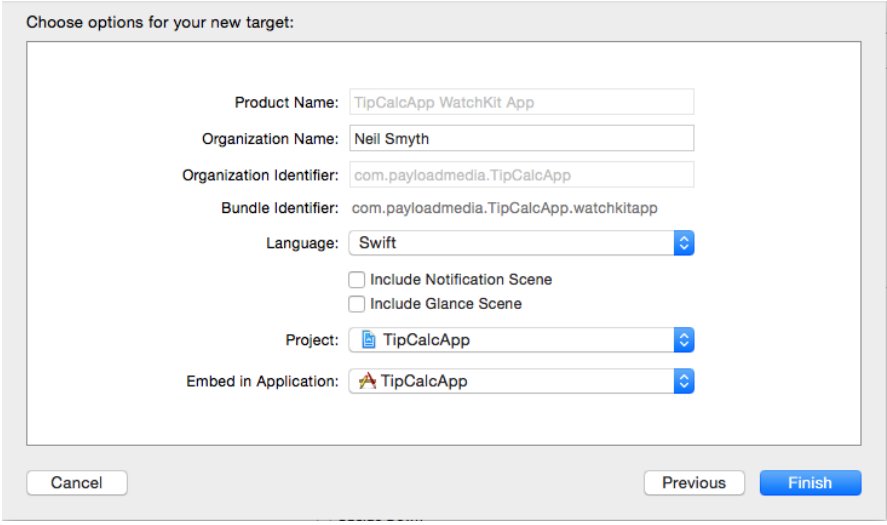

Figure 5-1

As soon as the extension target has been created, a new panel will appear requesting permission to activate the new scheme for the extension target. Activate this scheme now by clicking on the *Activate* button in the request panel.

5.4 Designing the WatchKit App User Interface

Within the Xcode Project Navigator panel unfold the *TipCalcApp WatchKit App* folder entry and select the *Interface.storyboard* file to load it into the Interface Builder tool.

The user interface for the app is going to consist of two Label objects, a Slider and a Button. Begin the design by locating the Label object in the Object Library panel and dragging and dropping it onto the scene so that it appears at the top of the scene layout. Select the newly added label and display the Attributes Inspector in the utilities panel (*View -> Utilities -> Show Attributes Inspector*). Within the inspector panel, change the text so that it reads $0.00 and change the Alignment setting so that the text is positioned in the center of the label.

Remaining within the Attributes Inspector panel, click on the 'T' icon located in the far right of the *Font* attribute text field to display the font setting panel. Within this panel, change the *Font* setting to *System*, the *Style* to *Bold* and the *Size* value to 28 as shown in Figure 5-2. Once the font settings are complete, click on the *Done* button to commit the changes.

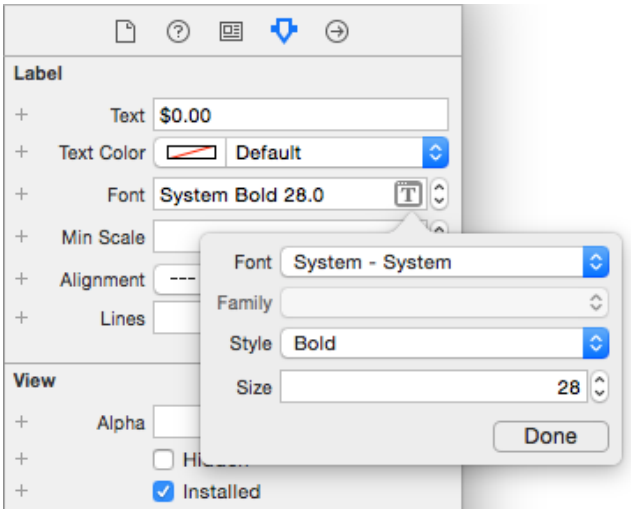

Figure 5-2

With the Label object still selected within the scene, locate the *Position* section within the Attribute Inspector panel and change the *Horizontal* property setting to *Center*.

Next, drag a Slider object from the library and drop it onto the scene so that it appears beneath the Label object. Select the Slider object in the scene and, within the Attributes Inspector, configure the *Minimum* and *Maximum* attributes to 0 and 100 respectively and enable the *Continuous* checkbox. Since we want the slider to adjust in $1 units the *Steps* value needs to be changed to 100.

Drag and drop a second Label object so that it is positioned beneath the slider. Select the new label and set the same alignment, font and positioning properties as those used for the first label. This time, however, change the *Text Color* attribute so that the text is displayed in green.

Finally, position a Button object beneath the second label. Double click on the button and change the text so it reads "Calculate Tip". With the button still selected, use the Attribute Inspector and change the *Vertical* property located in the *Position* section of the panel to *Bottom*.

At this point the scene layout should resemble that shown in Figure 5-3:

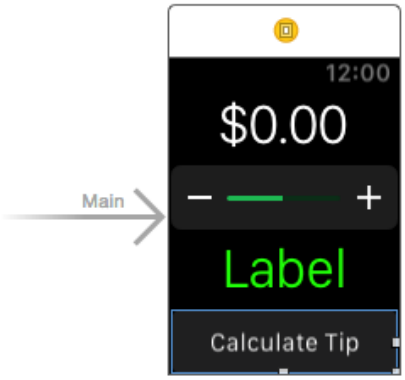

Figure 5-3

The user interface design is now complete. The next step is to configure outlets on the two Label objects so that the values displayed can be controlled from within the code of the interface controller class in the WatchKit extension. Before doing so, however, it is worth taking a look at the interface controller class file.

5.5 Reviewing the Interface Controller Class

As previously discussed, each scene within the storyboard of a WatchKit app has associated with it an interface controller class located within the WatchKit extension. By default, the Swift source code file for this class will be named *InterfaceController.swift* and will be located within the Project Navigator panel under the *<AppName> WatchKit Extension* folder where *<AppName>* is replaced by the name of the containing iOS app. Figure 5-4, for example, highlights the interface controller source file for the main scene of the TipCalcApp extension:

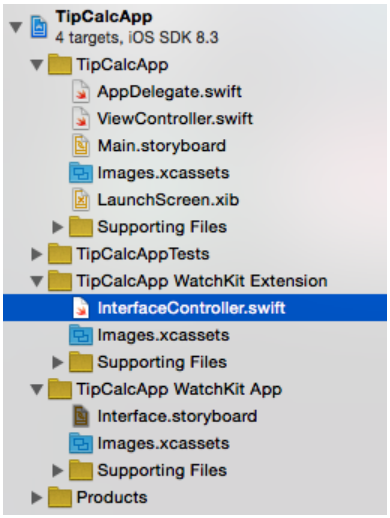

Figure 5-4

Locate and select this file so that it loads into the editing panel. Once loaded, the code should read as outlined in the following listing:

```
import WatchKit
import Foundation

class InterfaceController: WKInterfaceController {

    override func awakeWithContext(context: AnyObject?) {
        super.awakeWithContext(context)
```

```
        // Configure interface objects here.
    }

    override func willActivate() {
        // This method is called when watch view controller is about to be visible to
user
        super.willActivate()
    }

    override func didDeactivate() {
        // This method is called when watch view controller is no longer visible
        super.didDeactivate()
    }

}
```

Xcode has created an interface controller class implementation that overrides a subset of the lifecycle methods outlined in the chapter entitled *An Overview of WatchKit App Architecture*. Later in this chapter some initialization code will be added to the *willActivate()* method. At this point, some outlets need to be configured so that changes can be made to the Label objects in the main WatchKit app scene.

5.6 Establishing Outlet Connections

Outlets provide the interface controller class with access to the interface objects within the corresponding storyboard scene. Outlets can be created visually within Xcode by using Interface Builder and the Assistant Editor panel.

To establish outlets, begin by loading the *Interface.storyboard* file into the Interface Builder tool. Within Interface Builder, click on the scene so that it is highlighted before displaying the Assistant Editor by selecting the *View -> Assistant Editor -> Show Assistant Editor* menu option. Alternatively, it may also be displayed by selecting the center button (the one containing an image of interlocking circles) of the row of Editor toolbar buttons in the top right hand corner of the main Xcode window as illustrated in the following figure:

Figure 5-5

In the event that multiple Assistant Editor panels are required, additional tiles may be added using the *View -> Assistant Editor -> Add Assistant Editor* menu option.

By default, the editor panel will appear to the right of the main editing panel in the Xcode window. For example, in Figure 5-6 the panel to the immediate right of the Interface Builder panel is the Assistant Editor (marked A):

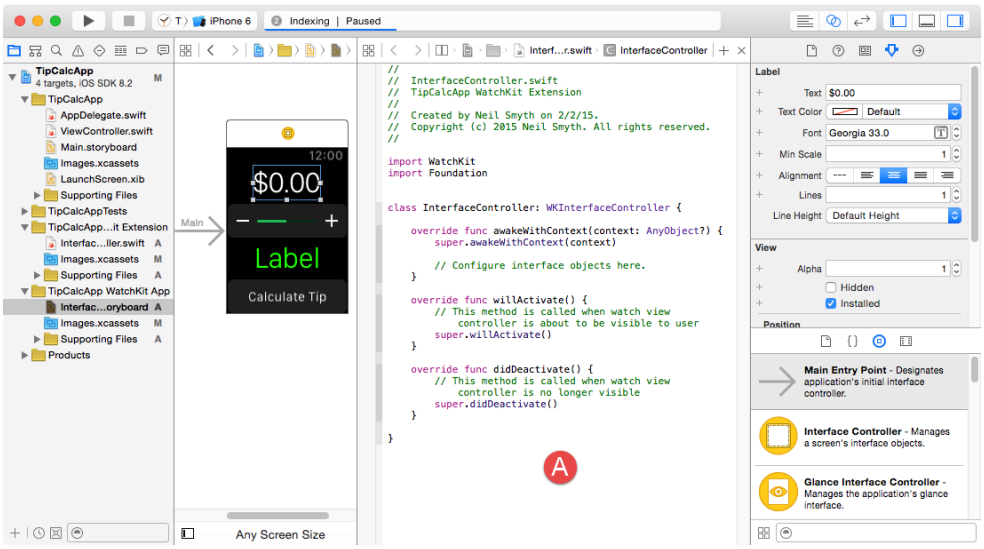

Figure 5-6

By default, the Assistant Editor will be in *Automatic* mode, whereby it automatically attempts to display the correct source file based on the currently selected item in Interface Builder. If the correct file is not displayed, use the toolbar along the top of the editor panel to select the correct file. The small instance of the Assistant Editor icon in this toolbar can be used to switch to *Manual* mode allowing the file to be selected from a pull-right menu containing all the source files in the project:

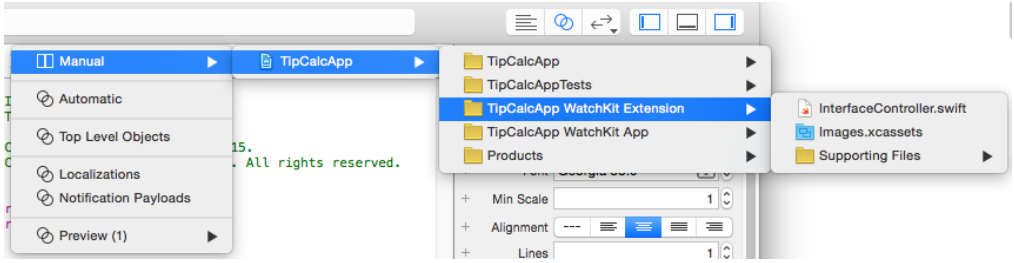

Figure 5-7

Make sure that the *InterfaceController.swift* file is displayed in the Assistant Editor and establish an outlet for the top-most label by Ctrl-clicking on the Label object in the scene and dragging the resulting line to the area immediately beneath the *class InterfaceController* declaration line in the Assistant Editor panel as shown in Figure 5-8:

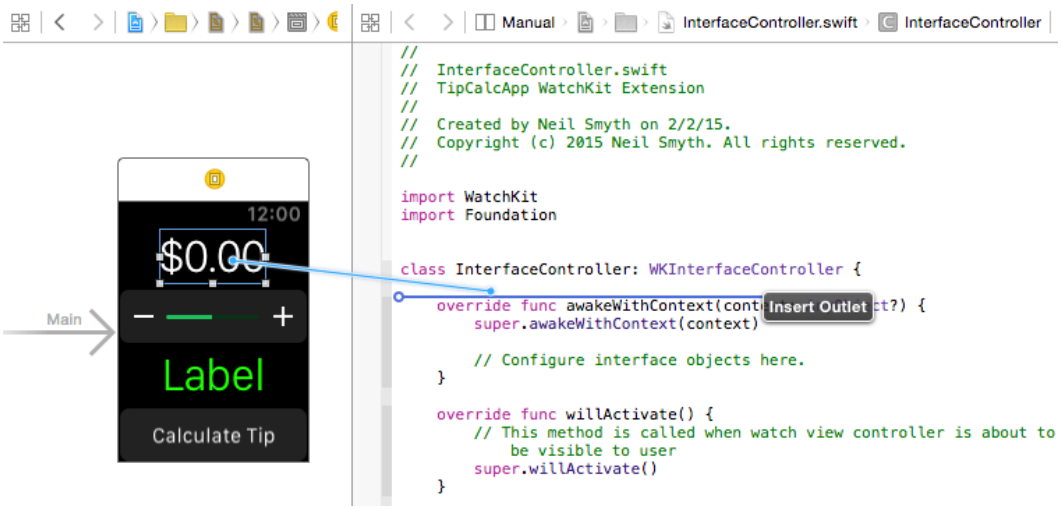

Figure 5-8

Upon releasing the line, the configuration panel illustrated in Figure 5-9 will appear requesting details about the outlet to be defined.

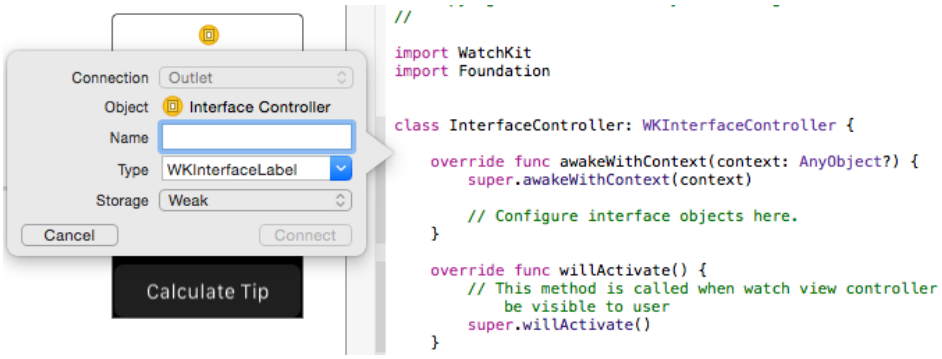

Figure 5-9

Since this is an outlet, the *Connection* menu should be set to *Outlet*. The type and storage values are also correct for this type of outlet. The only task that remains is to enter a name for the outlet, so in the *Name* field enter *amountLabel* before clicking on the *Connect* button.

Repeat the above steps to establish an outlet for the second Label object named *tipLabel.*

Once the connections have been established, review the *InterfaceController.swift* file and note that the outlet properties have been declared for us by the Assistant Editor:

```
import WatchKit
import Foundation

class InterfaceController: WKInterfaceController {

    @IBOutlet weak var amountLabel: WKInterfaceLabel!
    @IBOutlet weak var tipLabel: WKInterfaceLabel!
.
.
.
}
```

When we reference these outlet variables within the interface controller code we are essentially accessing the objects in the user interface.

5.7 Establishing Action Connections

Now that the outlets have been created, the next step is to connect the Slider and Button objects to action methods within the interface controller class. When the user increments or decrements the slider value the interface controller will need to change the value displayed on the *amountLabel* object to reflect the new value. This means that an action method will need to be implemented within the interface controller class and connected via an action to the Slider object in the storyboard scene.

With the scene displayed in Interface Builder and the *InterfaceController.swift* file loaded into the Assistant Editor panel, Ctrl-click on the Slider object in the storyboard scene and drag the resulting line to a position immediately beneath the *tipLabel* outlet as illustrated in Figure 5-10:

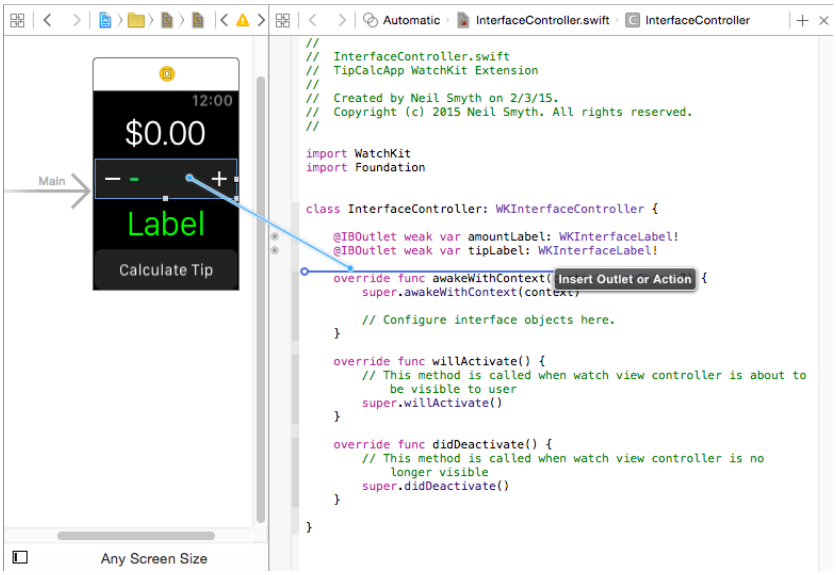

Figure 5-10

Once the line has been released, the connection configuration dialog will appear (Figure 5-11). Within the dialog, select *Action* from the *Connection* menu and enter *sliderChange* as the name of the action method to be called when the value of the slider is changed by the user:

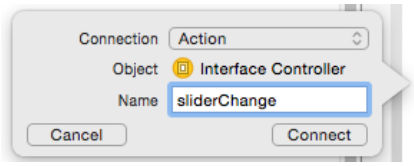

Figure 5-11

Click on the *Connect* button to establish the action and note that Xcode has added a stub action method at the designated location within the *InterfaceController.swift* file:

```
import WatchKit
import Foundation

class InterfaceController: WKInterfaceController {

    @IBOutlet weak var amountLabel: WKInterfaceLabel!
    @IBOutlet weak var tipLabel: WKInterfaceLabel!

    @IBAction func sliderChange(value: Float) {
    }
.
.
}
```

An action method will also need to be called when the user taps the Button object in the user interface. Ctrl-click on the Button object in the scene and drag the resulting line to a position beneath the *sliderChange* method. On releasing the line, the connection dialog will appear once again. Change the connection menu to *Action* and enter *calculateTip* as the method name before clicking on the *Connect* button to create the connection.

5.8 Implementing the sliderChange Action Method

The Slider object added to the scene layout is actually an instance of the WatchKit framework WKInterfaceSlider class. When the user adjusts the slider value, that value is passed to the action method assigned to the object, in this instance the *sliderChange* method created in the previous section.

It is the responsibility of this action method to display the current value on the amount label using the *amountLabel* outlet variable and to store the current amount locally within the interface controller object so that it can be accessed when the user requests that the tip amount be calculated. Select the *InterfaceController.swift* file and modify it to add a floating point variable in which to store the current slider value and to implement the code in the *sliderChange* method:

```
class InterfaceController: WKInterfaceController {

    @IBOutlet weak var amountLabel: WKInterfaceLabel!
    @IBOutlet weak var tipLabel: WKInterfaceLabel!

    var currentAmount: Float = 0.00

    @IBAction func sliderChange(value: Float) {
        let amountString = String(format: "%0.2f", value)
        amountLabel.setText("$\(amountString)")
        currentAmount = value
    }
.
.
}
```

The code added to the action method performs a number of tasks. First a new String object is created based on the current floating point value passed to the action method from the Slider object. This is formatted to two decimal places to reflect dollars and cents. The *setText* method of the *amountLabel* outlet is then called to set the text displayed on the Label object in the user interface, prefixing the *amountString* with a dollar sign. Finally, the current value is assigned to the *currentAmount* variable where it can be accessed later from within the *calculateTip* action method.

Make sure that the run target in the Xcode toolbar is set to *TipCalcApp Watchkit App* and click on the run button to launch the app. Once it has loaded into the simulator, click on the – and + slider buttons to change the current value. Note that the amount label is updated each time the value changes:

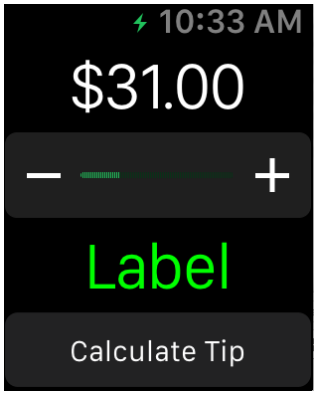

Figure 5-12

5.9 Implementing the calculateTip Action Method

The *calculateTip* action method will calculate 20% of the current amount and display the result to the user via the *tipAmount* outlet, once again using string formatting to display the result to two decimal places prefixed with a dollar sign:

```
@IBAction func calculateTip() {
    let tipAmount = currentAmount * 0.20
    let tipString = String(format: "%0.2f", tipAmount)
    tipLabel.setText("$\(tipString)")
}
```

Run the application once again, adjust the slider and click on the Calculate Tip button, verifying that the tip amount is displayed on the tip Label object as shown in Figure 5-13:

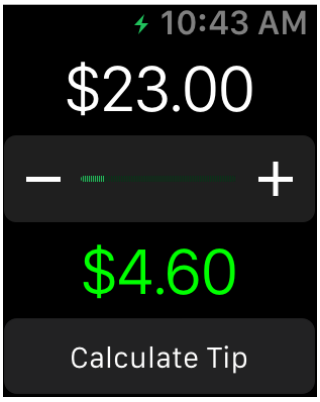

Figure 5-13

5.10 Hiding the Tip Label

Until the user taps the calculate button, the tip label is largely redundant. The final task for the project, therefore, is to hide the tip label until the app is ready to display the recommended tip amount. The Label object can be hidden by adding some code to the *willActivate* lifecycle method within the *InterfaceController.swift* class file as follows:

```
override func willActivate() {
    // This method is called when watch view controller is about to be visible to user
    super.willActivate()
    tipLabel.setHidden(true)
}
```

This method uses the *tipLabel* outlet to call the *setHidden* method on the Label object so that it is hidden from the user. When an object is hidden it is invisible to the user and the user interface layout behaves as though the object no longer exists. As such, the layout will re-arrange to occupy the vacated space. To hide an object whilst retaining the occupied space, call the *setAlpha* method passing through a value of 0 to make the object transparent.

Having hidden the label during the initialization phase, a line of code needs to be added to the *calculateTip* method to reveal the Label object after the tip has been calculated:

```
@IBAction func calculateTip() {
    let tipAmount = currentAmount * 0.20
    let tipString = String(format: "%0.2f", tipAmount)
    tipLabel.setText("$\(tipString)")
    tipLabel.setHidden(false)
}
```

Run the app one last time and verify that the tip label remains hidden until the calculate button is pressed.

5.11 Removing the WatchKit App

To avoid cluttering the Apple Watch Home screen with all of the sample WatchKit apps created in this book, it is recommended that the apps be removed from the Apple Watch device at the end of each chapter.

One option is to delete the containing iOS app from the iPhone device by pressing and holding on the app icon on the iPhone screen until the “x” marker appears. Tapping the “x” to delete the iOS app will also remove the WatchKit app from the paired Apple Watch.

Another option is to hide the app using the Apple Watch app. This app is installed by default on iPhone devices and is the app that you used when pairing your iPhone with the Apple Watch. Within this app, select the *My Watch* option in the bottom tab bar and scroll down and select the *TipCalcApp* entry. On the resulting preferences screen (Figure 5-14) turn off the *Show App on Apple Watch* option:

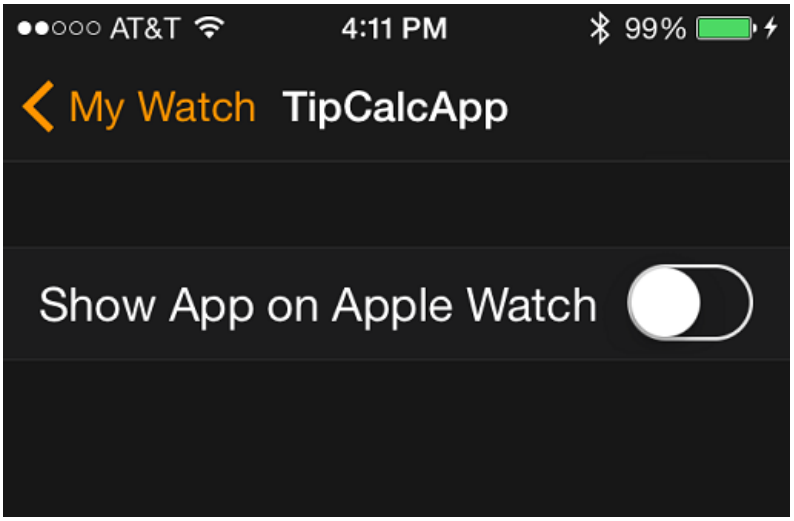

Figure 5-14

5.12 Summary

The Interface Builder tool and the Assistant Editor panel can be used together to quickly establish outlet and action connections between the user interface objects in a storyboard scene and the underlying interface controller scene in the WatchKit extension. This chapter has worked through the creation of a sample application project designed to demonstrate this technique. This chapter also made use of the *willActivate* lifecycle method to perform an initialization task and briefly covered the hiding and showing of objects in a WatchKit scene.