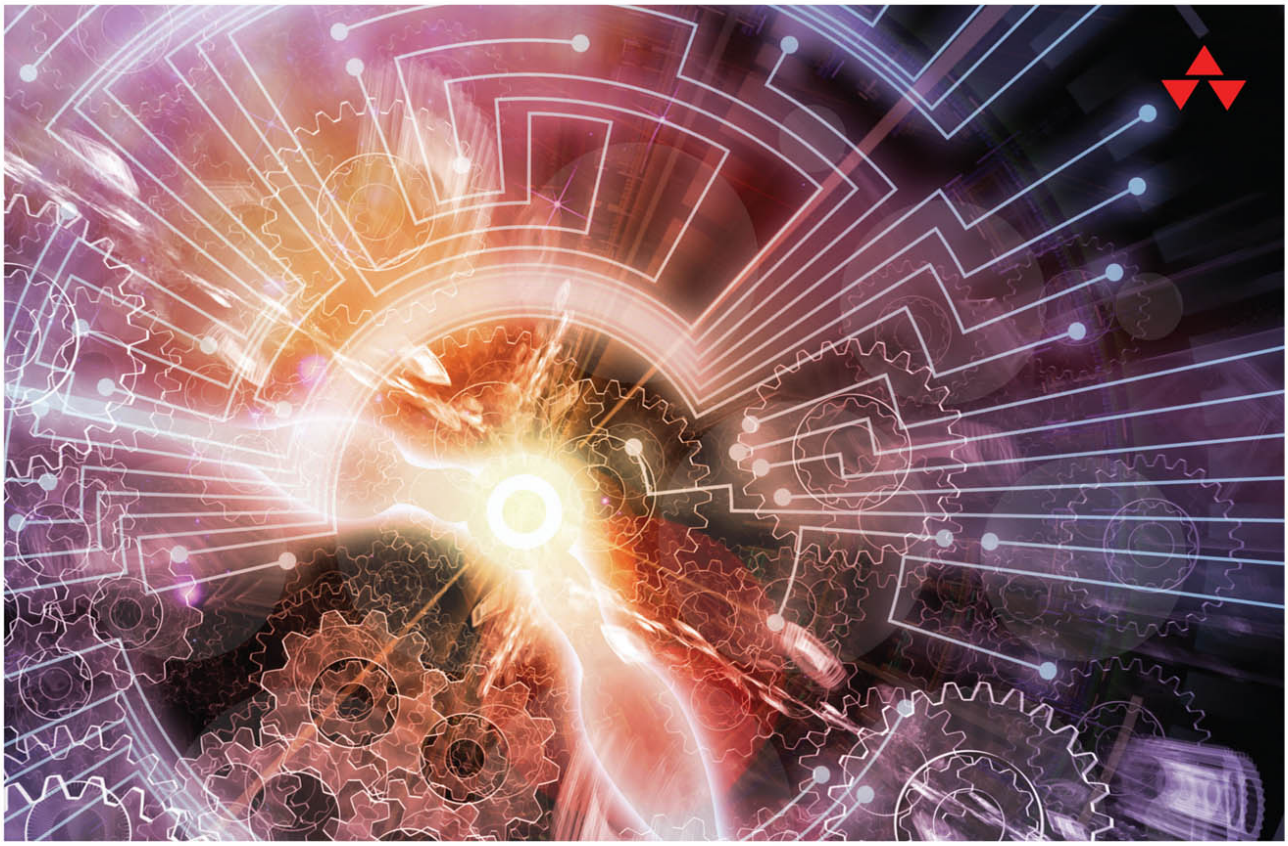

LEARNING WatchKit PROGRAMMING

A Hands-On Guide to Creating watchOS 2 Applications

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Learning WatchKit Programming

Second Edition

Addison-Wesley Learning Series

Visit **informit.com/learningseries** for a complete list of available publications.

The **Addison-Wesley Learning Series** is a collection of hands-on programming guides that help you quickly learn a new technology or language so you can apply what you've learned right away.

Each title comes with sample code for the application or applications built in the text. This code is fully annotated and can be reused in your own projects with no strings attached. Many chapters end with a series of exercises to encourage you to reexamine what you have just learned, and to tweak or adjust the code as a way of learning.

Titles in this series take a simple approach: they get you going right away and leave you with the ability to walk off and build your own application and apply the language or technology to whatever you are working on.

Learning WatchKit Programming

A Hands-On Guide to Creating watchOS 2 Applications

Second Edition

Wei-Meng Lee

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
Sao Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: 2015952426

ISBN-13: 978-0-13-439898-3
ISBN-10: 0-13-439898-X

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, December 2015

Editor-in-Chief
Mark L. Taub

Senior Acquisitions Editor
Trina MacDonald

Development Editor
Sheri Replin

Managing Editor
John Fuller

Full-Service Production Manager
Julie B. Nahil

Copy Editor
Barbara Wood

Indexer
Jack Lewis

Proofreader
Anna Popick

Technical Reviewers
Mark H. Granoff
Chaim Krause
Niklas Saers

Editorial Assistant
Olivia Basegio

Cover Designer
Chuti Prasertsith

Compositor
The CIP Group

❖

I dedicate this book with love to my family, and to my dearest wife, who has had to endure my irregular work schedule and take care of things while I was trying to meet writing deadlines!

❖

This page intentionally left blank

Contents at a Glance

This page intentionally left blank

Contents

Preface

Welcome to *Learning WatchKit Programming, Second Edition*!

This is an exciting time to be a programmer, as we are witnessing a new era of wearables. Although the Apple Watch is not the first wearable device in the market, its launch signified the intention of Apple to enter the wearable market in a big way. After successfully changing various industries—music, computer, phone, and mobile computing—Apple looks set to change the wearable industry. And nobody is taking this lightly.

As with the iPhone, much of the usefulness and functionality of the Apple Watch device actually come from the creativity of the third-party developers. In the early days of the iPhone, Apple restricted all third-party apps to web applications, as it wanted to retain the monopoly on developing natively for the device. However, due to the overwhelming protests of developers, Apple finally relented by releasing an SDK to support third-party apps. It was this decision that changed the fate of the iPhone; the iPhone would never have been so successful without the ability to support third-party apps.

When the Apple Watch was announced, Apple was quick to learn its lesson and realized that the success of the Apple Watch largely depends on the availability of apps that support it. Hence, before the release of the Apple Watch, the SDK was made available to developers to have a hand in developing Apple Watch apps.

Barely two months after the Apple Watch was made available for sale, Apple announced the second version of the Apple Watch OS, aptly named watchOS 2. Unsurprisingly, watchOS 2 now supports native apps and comes with a slew of new features.

The book you are holding in your hands right now (or reading on your phone or tablet) is a collection of tutorials that help you navigate the jungle of Apple Watch programming. This book contains all the fundamental topics that you need to get started in Apple Watch programming. In particular, this second edition has been fully updated to cover watchOS 2 programming.

Because this is a book on Apple Watch programming, I make a couple of assumptions about you, the reader:

- You should already be familiar with the basics of developing an iOS application. In particular, concepts like outlets and actions should not be new to you.
- You should be comfortable with the Swift programming language, but see the next section on how to get started with Swift if you are new to it.

What You'll Need

To get the most out of this book, note the following:

- You need a Mac, together with **Xcode**.
- Your Mac should be running at least **Mac OS X Yosemite (v10.10)** or later.
- You can download the latest version of Xcode from the Mac App Store. All of the code samples for this book have been tested against Xcode 7.
- If you plan to test your apps on a real device, you need to register to become a paying Apple developer (https://developer.apple.com/programs/). The program costs $99 per year for individuals. Once registered, you can register your Apple Watch's UDID with Apple (necessary for testing on Apple Watch). The Apple Watch works only with iPhone 5, iPhone 5c, iPhone 5s, iPhone 6, iPhone 6 Plus, iPhone 6s, and iPhone 6s Plus (or newer versions of the iPhones).
- Most of the code samples in this book can be tested and run on the iPhone Simulator without the need for a real device or Apple Watch. However, for some code examples, you need access to a real Apple Watch (for example, to access the hardware features like accelerometer, microphone, etc.).
- A number of examples in this book require an Internet connection in order to work, so ensure that you have an Internet connection when trying out the examples.
- All of the examples in this book are written in Swift 2.0. If you are not familiar with Swift, you can refer to Apple's web page on Swift at https://developer.apple.com/swift/resources/.

How This Book Is Organized

This book is styled as a tutorial. You try out the examples as I explain the concepts. This is a proven way to learn a new technology, and I strongly encourage you to type in the code as you work on the examples.

- **Chapter 1, "Getting Started with WatchKit Programming"**: In this chapter, you learn about the architecture of Apple Watch applications and how it ties in with your iOS apps. Most importantly, you get your chance to write a simple Apple Watch app and deploy it onto the Apple Watch Simulator.
- **Chapter 2, "Apple Watch Interface Navigation"**: In this chapter, you dive deeper into how your Apple Watch application navigates between multiple screens. You get to see how data is passed between screens and how to customize the look and feel of each screen.

- **Chapter 3, "Responding to User Actions"**: Designing the user interface (UI) for your Apple Watch application is similar to designing for iPhone apps. However, space is at a premium on the Apple Watch, and every millimeter on the screen must be put to good use in order to convey the exact intention of your app. In this chapter, you learn how to use the various UI controls in the Apple Watch to build your application. You will start off with the controls with which the user interacts.
- **Chapter 4, "Displaying and Gathering Information"**: While Chapter 3 covers the various controls with which the user interacts through the tap gesture, this chapter continues to explore the various controls available in the WatchKit framework, focusing on controls that display information, as well as controls that gather information.
- **Chapter 5, "Accessing the Apple Watch Hardware"**: In watchOS 1, Apple did not provide third-party developers access to the various hardware features of the Apple Watch, such as accelerometer, microphone, and Taptic Engine. However, in watchOS 2, Apple has exposed some of these features to developers so that they can create more exciting watch apps. In this chapter, you learn how to access some of these hardware features and see how they can be useful to the apps you are building.
- **Chapter 6, "Programming Complications"**: A complication is a function on a timepiece that does more than just tell the time. Complications on a timepiece include alarms, tachymeters, chronographs, calendars, and so on. In watchOS 2, third-party apps can now also display data in watch face complications. In this chapter, you learn the process of creating an application that displays complication data.
- **Chapter 7, "Interfacing with iOS Apps"**: This chapter discusses the Watch Connectivity Framework, a set of APIs that allow the containing iOS app to communicate with the watch app (and vice versa). In addition to discussing how apps intercommunicate, this chapter also discusses how to use location services in your watch app, as well as how to consume web services. Last, but not least, this chapter ends with a discussion on persisting data on your watch.
- **Chapter 8, "Displaying Notifications"**: In this chapter, you learn how to display notifications on your Apple Watch. Notifications received by the iPhone are sent to the Apple Watch, and you have the chance to customize the notifications so that you can display their essence quickly to the user.
- **Chapter 9, "Displaying Glances"**: Glances on the Apple Watch provide the user a quick way to gather information from apps. For example, Instagram's glance on the Apple Watch may show the most recently shared photo, and Twitter may show the latest trending tweets. In this chapter, you learn how to implement glances for your own apps.

About the Sample Code

The code samples in this book are written to provide the simplest way to understand core concepts without getting bogged down with details like beautifying the UI or detailed error checking. The philosophy is to convey key ideas in the simplest manner possible. In real-life apps, you are expected to perform detailed error handling and to create a user-friendly UI for your apps. Although I do provide several scenarios in which a certain concept is useful, it is ultimately up to you, the reader, to exercise your creativity to put the concepts to work, and perhaps create the next killer app.

Getting the Sample Code

To download the sample code used in this book, visit the book's web page on informIT.com at http://informit.com/title/9780134398983, click the **Extras** tab, and register your book.

Contacting the Author

If you have any comments or questions about this book, drop me an email at weimenglee@learn2develop.net, or stop by my web site at http://learn2develop.net.

Acknowledgments

Writing a book on emerging technology is always an exciting and perilous journey. On one end, you are dealing with the latest developments, going where not many have ventured, and on the other end you are dealing with many unknowns. To endure this journey, you need a lot of help and family support. I want to take this opportunity to thank the people who make all this happen.

I am indebted to Trina MacDonald, senior acquisitions editor at Pearson, for giving me the chance to work on this book. She has always been supportive of my proposals for new titles, and I am really glad that we have the chance to work together on this project. Thank you very much for the opportunity and guidance, Trina! I hope I did not disappoint you.

I want to thank the many heroes working behind the scenes—copy editor Barbara Wood; production editor Julie Nahil; and technical reviewers Mark H. Granoff, Chaim Krause, and Niklas Saers—for turning the manuscript into a book that I am proud of!

Last, but not least, I want to thank my family for all the support that they have always given me. Without their encouragement, this book would never have been possible.

This page intentionally left blank

About the Author

Wei-Meng Lee is a technologist and founder of Developer Learning Solutions (http://learn2develop.net), a technology company specializing in hands-on training on the latest web and mobile technologies. Wei-Meng speaks regularly at international conferences and has authored and coauthored numerous books on .NET, XML, Android, and iOS technologies. He writes extensively for informIT.com and mobiForge.com.

This page intentionally left blank

3

Responding to User Actions

If you haven't found it yet, keep looking. Don't settle. As with all matters of the heart, you'll know when you find it. And like any great relationship, it just gets better and better as the years roll on.

Steve Jobs

Designing the user interface (UI) for your Apple Watch application is similar to designing for the iPhone. However, space is at a premium on the Apple Watch, and every millimeter on the screen must be put to good use in order to convey the exact intention of your app.

The UI of an Apple Watch application is represented by various controls (commonly known as *views* in iOS programming), and they are divided into two main categories:

- **Responding to user actions**: Users directly interact with these controls to perform some actions. Examples of such controls are Button, Switch, Slider, Picker, and Table.
- **Displaying information**: These controls mainly display information to the user. Examples of such controls are Label, Image, and Table.

In this and the next chapter, you learn how to use these various controls to build the UI of your application.

Using the Tap Gesture to Interact with Controls

One key way to interact with the Apple Watch is to use the tap gesture. You can tap the following controls:

- Button
- Switch
- Slider
- Table

Let's take a more detailed look at these objects!

> **Note**
> I cover the Table control in the next chapter where we discuss controls that display information.

Button

The Button control is the most direct way of interacting with an Apple Watch application. A button can display text as well as a background image. Tapping a button triggers an action on the Interface Controller where you can write the code to perform the appropriate action.

Adding a Button to an Interface Controller

In this section, you create a project that uses a Button control. Subsequent sections show you how to customize the button by creating an action for it and then displaying its title using custom fonts.

1. Using Xcode, create a new iOS App with WatchKit App project and name it **Buttons**. Uncheck the option Include Notification Scene so that we can keep the WatchKit project to a bare minimum.
2. Select the Interface.storyboard file to edit it in the Storyboard Editor.
3. Drag and drop a Button control onto the storyboard, as shown in Figure 3.1.

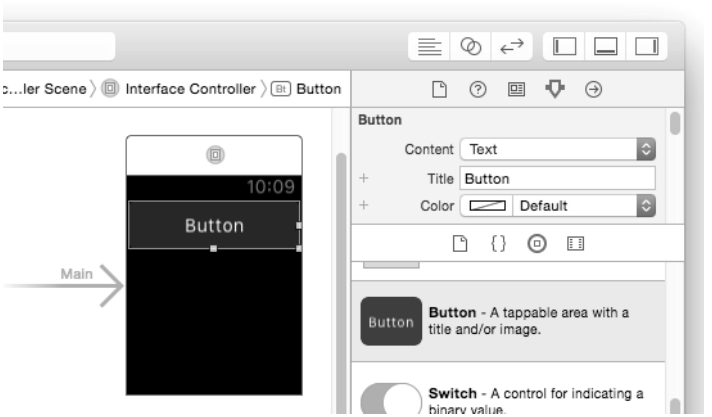

Figure 3.1 Adding a Button control to the Interface Controller

4. In the Attributes Inspector window, set the Title attribute to **Play** (see Figure 3.2).

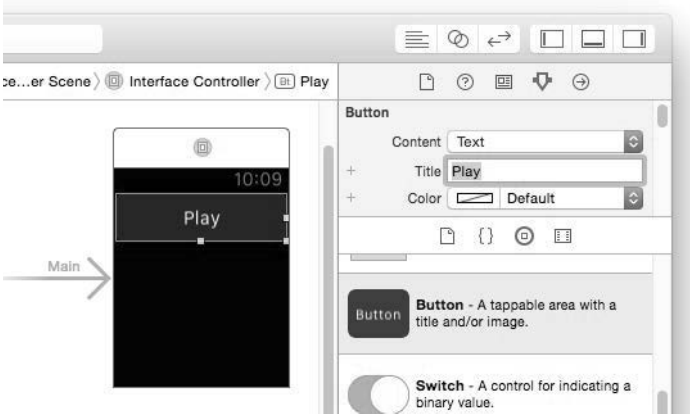

Figure 3.2 Changing the title of the button

5. Select the WatchKit App scheme and run the project on the Apple Watch Simulator. You should see the button on the Apple Watch Simulator (see Figure 3.3). You can click it (or tap it on a real Apple Watch).

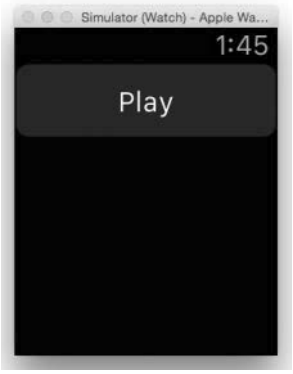

Figure 3.3 Testing the button on the Apple Watch Simulator

Creating an Action for a Button

For the Button control to do anything useful, you need to create an action for it so that when the user taps it, your application performs some actions. To create this action, follow these steps:

1. In the Storyboard Editor, select the **View** | **Assistant Editor** | **Show Assistant Editor** menu item to show the InterfaceController.swift file.
2. Control-click the Button control in the Interface Controller and drag it over the `InterfaceController` class (see Figure 3.4).

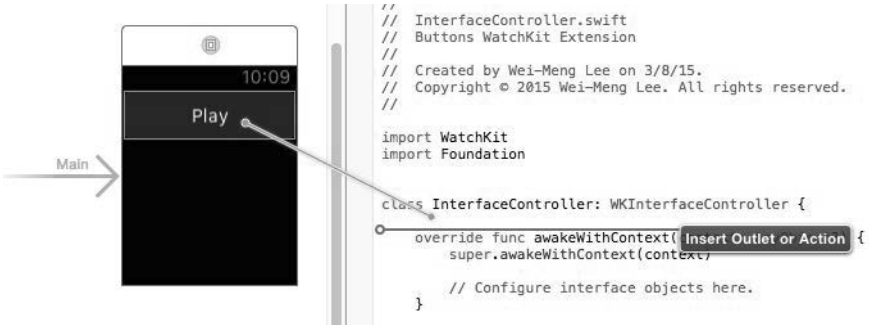

Figure 3.4 Creating an action for the button

3. Create an action for the button and name it **btnPlay** (see Figure 3.5). Click **Connect**.

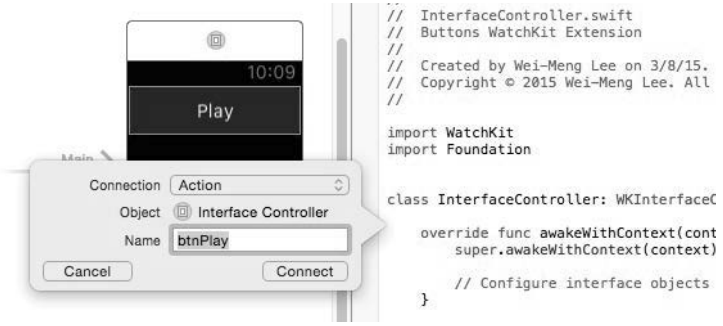

Figure 3.5 Naming the action

4. You now see the action created in the InterfaceController.swift file:

```
import WatchKit
import Foundation

class InterfaceController: WKInterfaceController {

    @IBAction func btnPlay() {
    }
```

5. Add the following statement in bold to the InterfaceController.swift file:

```
@IBAction func btnPlay() {
    print("The button was tapped!")
}
```

6. Select the WatchKit App scheme and run the project on the Apple Watch Simulator. Click the **Play** button and observe the statement printed in the Output window (see Figure 3.6).

Figure 3.6 Clicking the button fires the action

Creating an Outlet for a Button

You can also programmatically change the title of the Button control during runtime. To do so, you need to create an outlet for the button:

1. With the Assistant Editor shown, control-click the button and drag it over the InterfaceController.swift file. Name the outlet **button1** (see Figure 3.7) and click **Connect**.

Figure 3.7 Creating an outlet for the button

2. This creates an outlet in the InterfaceController.swift file:

```
import WatchKit
import Foundation

class InterfaceController: WKInterfaceController {

    @IBOutlet var button1: WKInterfaceButton!

    @IBAction func btnPlay() {
        print("The button was tapped!")
    }
```

3. Add the following statements in bold to the InterfaceController.swift file:

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

    // Configure interface objects here.
    button1.setTitle("Play Video")
}
```

> **Note**
>
> Observe that, while you can change the title of a button, you cannot get the title of the button programmatically.

4. Select the WatchKit App scheme and run the project on the Apple Watch Simulator. You should now see the title of the button changed to "Play Video" (see Figure 3.8).

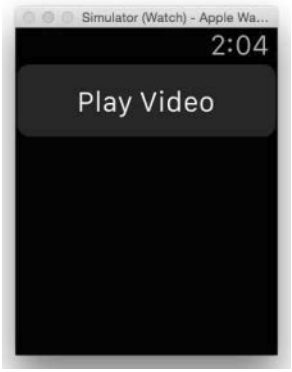

Figure 3.8 Changing the title of the button dynamically

Displaying Attributed Strings

The Button control supports *attributed strings*. Attributed strings allow you to specify different attributes (such as color, font, size, etc.) for different parts of a string. In the following steps, you display the title of the button using different colors:

1. Add the following statements in bold to the InterfaceController.swift file:

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

    // Configure interface objects here.

    // button1.setTitle("Play Video")
    let str = NSMutableAttributedString(
        string: "Hello, Apple Watch!")
```

```
        //------display the Hello in yellow---
        str.addAttribute(NSForegroundColorAttributeName,
            value: UIColor.yellowColor(),
            range: NSMakeRange(0, 5))

        //---display the , in red---
        str.addAttribute(NSForegroundColorAttributeName,
            value: UIColor.redColor(),
            range: NSMakeRange(5, 1))

        //---display Apple Watch! in green---
        str.addAttribute(NSForegroundColorAttributeName,
            value: UIColor.greenColor(),
            range: NSMakeRange(7, 12))
        button1.setAttributedTitle(str)
    }
```

2. Select the WatchKit App scheme and run the project on the Apple Watch Simulator. You should see the title of the button displayed in multiple colors, as shown in Figure 3.9 (readers of the print book will not see the colors in the figure).

Figure 3.9 Displaying the button title with mixed colors

Using Custom Fonts

Using attributed strings, you can also use different fonts for parts of a string. To illustrate this, let's modify the example in the previous section to display part of the button's title using a custom font.

For this example, use the Impact font that is installed on your Mac. The Impact font is represented using the Impact.ttf file located in the /Library/Fonts/ folder.

1. Drag and drop a copy of the Impact.ttf file onto the Extension project in Xcode.
2. You are asked to choose a few options. Select the options shown in Figure 3.10. This adds the Impact.ttf file onto the Extension and WatchKit App projects.

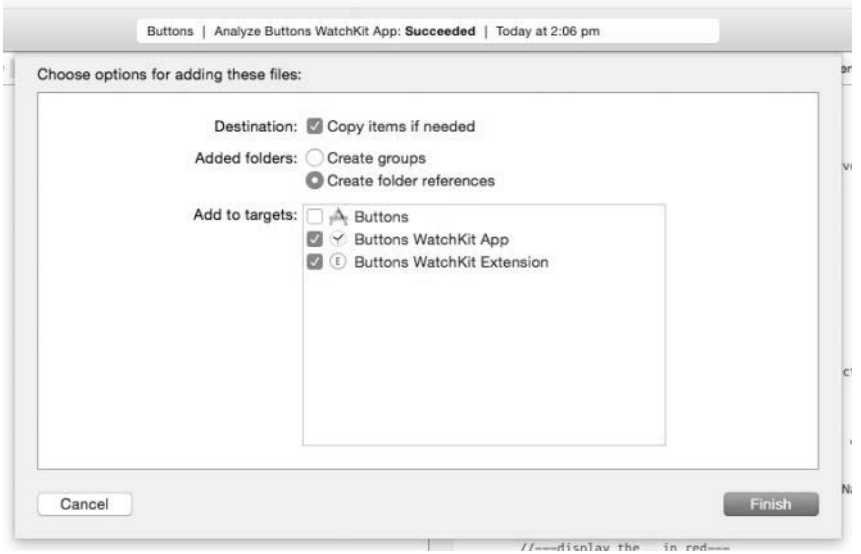

Figure 3.10 Adding the font file to the Extension and the WatchKit App

> **Note**
>
> Remember to add the font file to both the WatchKit Extension and WatchKit App. Also, be aware that adding custom fonts to the project adds considerable size and memory usage to your watch app. So, try to use the system font unless you have a very good reason not to.

3. Figure 3.11 shows the Impact.ttf file in the project.

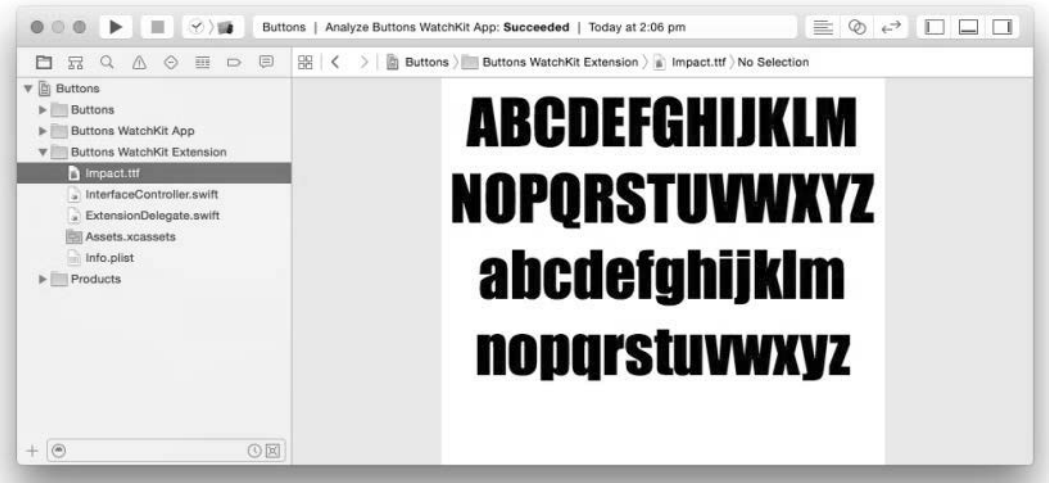

Figure 3.11 The font file in the project

4. Add a new key named **UIAppFonts** to the Info.plist file located in the Extension and set its Item 0 to **Impact.ttf** (see Figure 3.12).

> **Note**
>
> If your Info.plist file does not show the items as shown in Figure 3.12, simply right-click any of the items in it and select **Show Raw Keys/Values**.

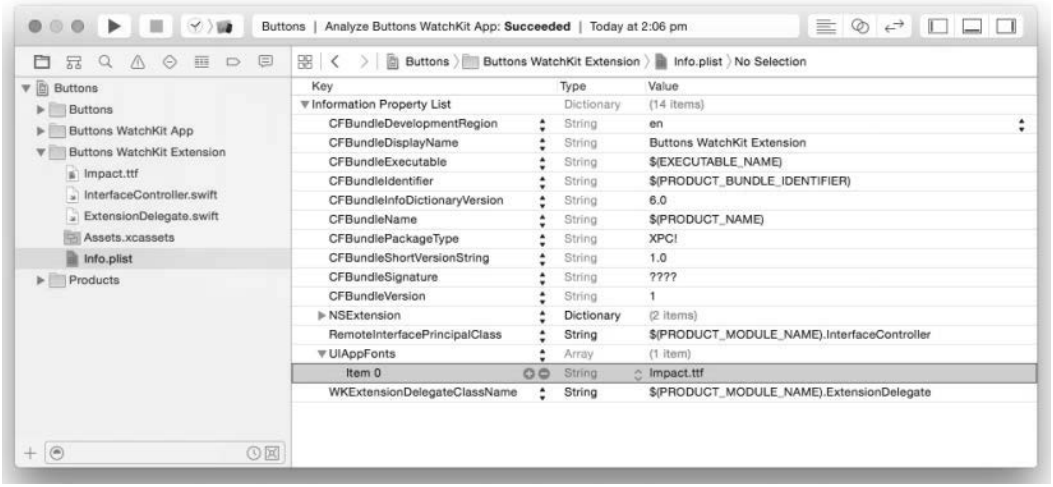

Figure 3.12 Specifying the font filename in the Extension project

5. Likewise, add a new key named **UIAppFonts** to the Info.plist file located in the WatchKit App and set its Item 0 to **Impact.ttf** (see Figure 3.13).

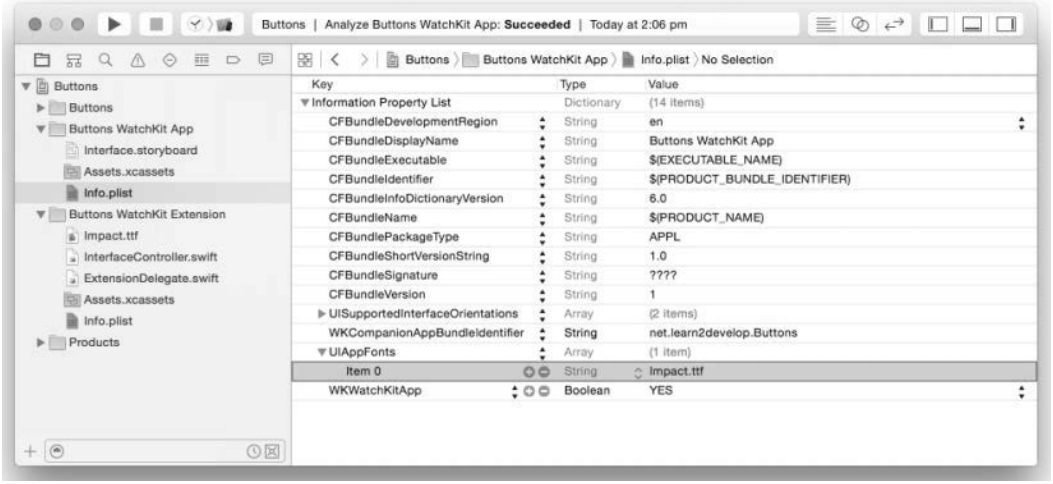

Figure 3.13 Specifying the font filename in the WatchKit app project

6. Add the following statements in bold to the InterfaceController.swift file:

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

    // Configure interface objects here.
    // button1.setTitle("Play Video")
    let str = NSMutableAttributedString(
        string: "Hello, Apple Watch!")

    //---display the Hello in yellow---
    str.addAttribute(NSForegroundColorAttributeName,
        value: UIColor.yellowColor(),
        range: NSMakeRange(0, 5))

    //---display Hello using the Impact font, size 22---
    str.addAttribute(NSFontAttributeName,
        value: UIFont(name: "Impact", size: 22.0)!,
        range: NSMakeRange(0, 5))

    //---display the , in red---
    str.addAttribute(NSForegroundColorAttributeName,
        value: UIColor.redColor(),
        range: NSMakeRange(5, 1))

    //---display Apple Watch! in green---
    str.addAttribute(NSForegroundColorAttributeName,
        value: UIColor.greenColor(),
        range: NSMakeRange(7, 12))
    button1.setAttributedTitle(str)
}
```

7. Select the WatchKit App scheme and run the project on the Apple Watch Simulator. You should now see "Hello" displayed using the Impact font (see Figure 3.14).

Figure 3.14 Displaying "Hello" using a custom font

Note

Once you have added a custom font to your project, you can use the font directly in Interface Builder by setting the Font attribute of a control to **Custom** and then selecting the font that you want to use in the Family attribute.

Getting the Font Name

One common problem in dealing with fonts is that the filename of the custom font that you are using is not always the same as the font name. The following code snippet allows you to print out the name of each font family and its corresponding font name:

```
for family in UIFont.familyNames() {
   print(family)
   for name in UIFont.fontNamesForFamilyName(family as String) {
   print("--\(name)")
   }
}
```

This code snippet prints the output as shown in Figure 3.15. For example, if you want to use the Helvetica Neue font, you have to specify in your code one of the font names printed: HelveticaNeue-Italic, HelveticaNeue-Bold, etc.

Figure 3.15 Printing out the font families and their associated font names

Changing the Background Image of Button

Besides displaying text, the Button control can also display a background image. The following exercise shows you how to add an image to the project and use it as the background of a button:

1. Drag and drop the image named play.png onto the Assets.xcassets item in the WatchKit App (see Figure 3.16).

> **Note**
>
> You can find a copy of this image in the source code download for this book.

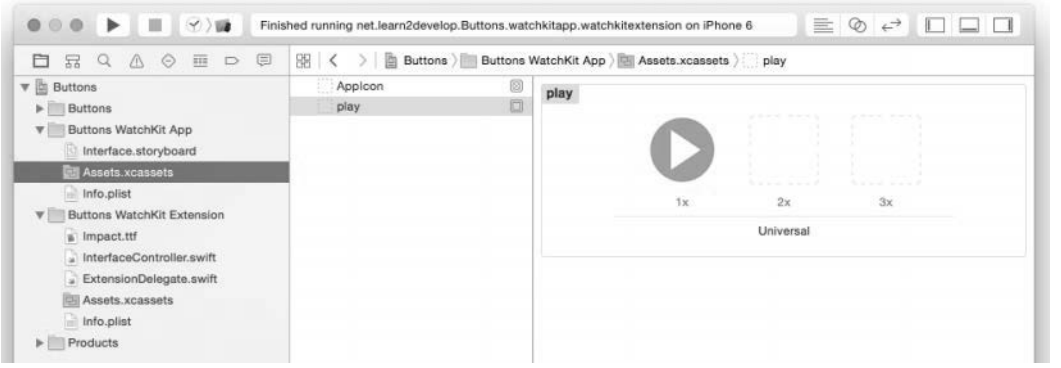

Figure 3.16 Adding an image to the project

2. In the Attributes Inspector window for the play.png image, check the watchOS checkbox (see Figure 3.17, right). Then, move the play.png into the box labeled

Figure 3.17 Specifying device-specific images to use

2× (see Figure 3.17, middle). This signifies that this image will be displayed for all sizes of Apple Watch. If you want to use different images for the 38mm Apple Watch and the 42mm Apple Watch, you can drag and drop different images onto the boxes labeled "38 mm 2×" and "42 mm 2×." For this example, you will use the same image for the two different watch sizes.

3. In the InterfaceController.swift file, add the following statements in bold:

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

    // Configure interface objects here.
    // button1.setTitle("Play Video")

    /*
    let str = NSMutableAttributedString(
        string: "Hello, Apple Watch!")

    //---display the Hello in yellow---
    str.addAttribute(NSForegroundColorAttributeName,
        value: UIColor.yellowColor(),
        range: NSMakeRange(0, 5))

    //---display Hello using the Impact font, size 22---
    str.addAttribute(NSFontAttributeName,
        value: UIFont(name: "Impact", size: 22.0)!,
        range: NSMakeRange(0, 5))

    //---display the , in red---
    str.addAttribute(NSForegroundColorAttributeName,
        value: UIColor.redColor(),
        range: NSMakeRange(5, 1))

    //---display Apple Watch! in green---
    str.addAttribute(NSForegroundColorAttributeName,
        value: UIColor.greenColor(),
        range: NSMakeRange(7, 12))
    button1.setAttributedTitle(str)
    */

    button1.setBackgroundImageNamed("play")
}
```

4. Select the WatchKit App scheme and run the project on the Apple Watch Simulator. You should now see the image on the button (see Figure 3.18).

Figure 3.18 Displaying an image on the button

Do not use the `setBackgroundImage:` method by passing it a `UIImage` instance, like this:

```
button1.setBackgroundImage(UIImage(named: "play"))
```

This is because the `UIImage` class looks for the specified image `("play")` in the main bundle (the Extension). And because the play.png file is in the WatchKit App, the image cannot be found and, therefore, the image will not be set successfully.

5. You can also set the background image of the button in the storyboard via the Background attribute in the Attributes Inspector window.

Switch

The Switch control allows the user to toggle between the ON and OFF states. It is commonly used in cases where you allow users to enable or disable a particular setting. In the following example, you will create a project and see how the Switch control works:

1. Using Xcode, create a new iOS App with WatchKit App project and name it **Switches**. Uncheck the option Include Notification Scene so that we can keep the WatchKit project to a bare minimum.
2. Select the Interface.storyboard file to edit it in the Storyboard Editor.
3. Drag and drop a Switch control onto the default Interface Controller (see Figure 3.19).
4. In the Attributes Inspector window, set the Title attribute of the Switch control to **Aircon** (see Figure 3.20).

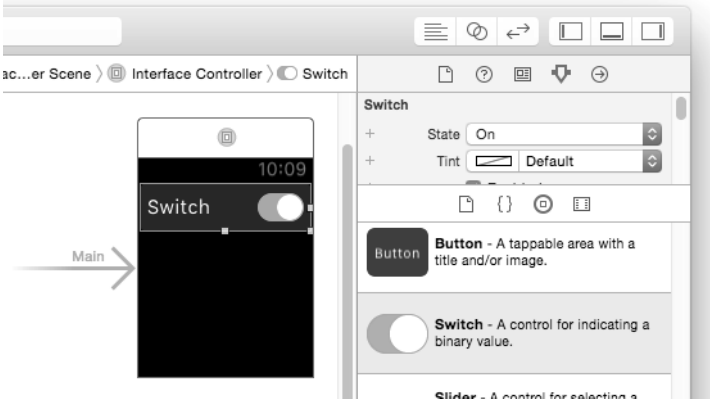

Figure 3.19 Adding a Switch control to the Interface Controller

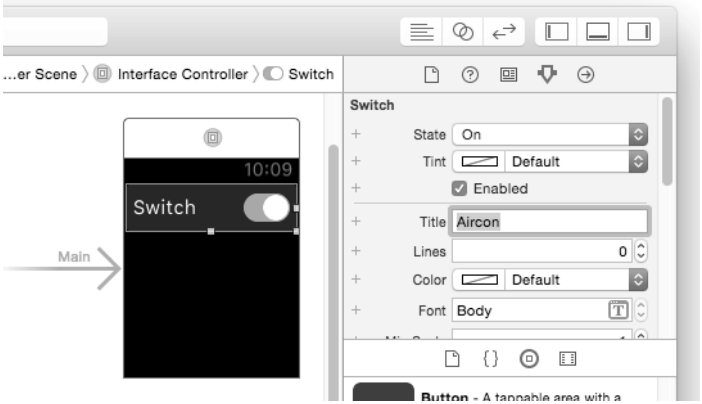

Figure 3.20 Changing the title of the Switch control

5. Add a Label control to the Interface Controller (see Figure 3.21).

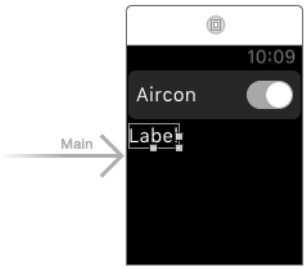

Figure 3.21 Adding a Label control to the Interface Controller

6. Create an outlet for the Switch control and name it **switch**. Likewise, create an outlet for the Label control and name it **label**. Then, create an action for the Switch control and name it **switchAction**. The InterfaceController.swift file should now look like this:

```
import WatchKit
import Foundation

class InterfaceController: WKInterfaceController {

    @IBOutlet var `switch`: WKInterfaceSwitch!
    @IBOutlet var label: WKInterfaceLabel!
    @IBAction func switchAction(value: Bool) {
    }
```

> **Note**
>
> Because `switch` is a reserved word in the Swift programming language, if you try to use it as the name of an outlet, you have to enclose it with a pair of back quotes (``).

8. Add the following statements in bold to the InterfaceController.swift file:

```
    @IBAction func switchAction(value: Bool) {
        value ? label.setText("Aircon is on") :
            label.setText("Aircon is off")
    }

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

        // Configure interface objects here.
        `switch`.setOn(false)
        label.setText("")
    }
```

> **Note**
>
> You can programmatically set the value of a Switch control, but you will not be able to get its value. To know its value, you need to implement the action of the Switch control and save its value whenever its state changes.

9. Select the WatchKit App scheme and run the project on the Apple Watch Simulator. On the Apple Watch Simulator, click the Switch control to turn it on and off and observe the message printed in the Label control (see Figure 3.22).

Figure 3.22 Testing the Switch control

Slider

The Slider control is a visual control with two buttons (– and +) that allow the user to decrement or increment a floating-point value. It is usually used in situations where you want the user to select from a range of values, such as the temperature settings in a thermostat or the volume of the iPhone.

1. Using Xcode, create a new iOS App with WatchKit App project and name it **Sliders**. Uncheck the option Include Notification Scene so that we can keep the WatchKit project to a bare minimum.
2. Select the Interface.storyboard file to edit it in the Storyboard Editor.
3. Drag and drop a Slider control onto the default Interface Controller (see Figure 3.23).

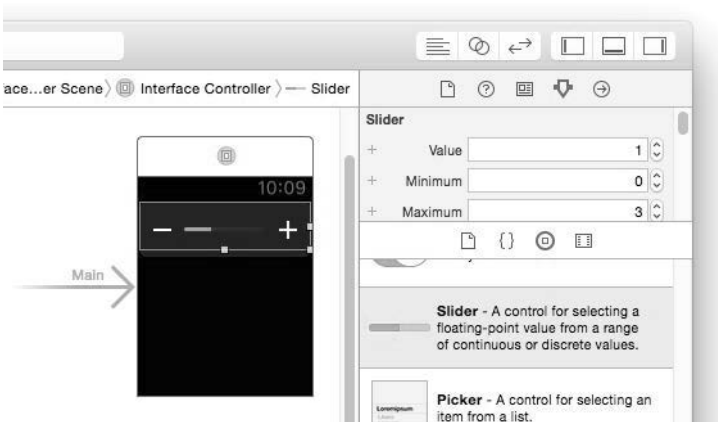

Figure 3.23 Adding a Slider control to the Interface Controller

4. Select the WatchKit App scheme and run the project on the Apple Watch Simulator. On the Apple Watch Simulator, click the + and – buttons (see Figure 3.24) and observe the slider.

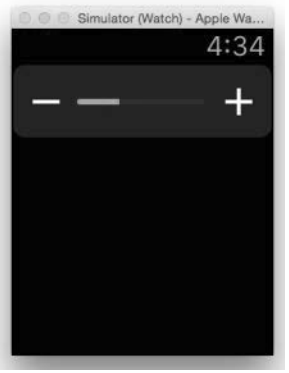

Figure 3.24 Testing the slider

5. Add a Label control to the Interface Controller (see Figure 3.25).

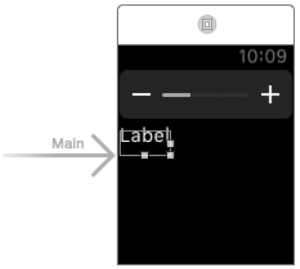

Figure 3.25 Adding a label to the Interface Controller

6. Create an outlet for the Slider control and name it **slider**. Likewise, create an outlet for the Label control and name it **label**. Then, create an action for the Slider control and name it **sliderAction**. The InterfaceController.swift file should now look like this:

```
import WatchKit
import Foundation
```

```
class InterfaceController: WKInterfaceController {

    @IBOutlet var slider: WKInterfaceSlider!
    @IBOutlet var label: WKInterfaceLabel!

    @IBAction func sliderAction(value: Float) {
    }
```

7. Set the attributes for the Slider control as follows (see Figure 3.26):
   Maximum: **10**
   Steps: **5**

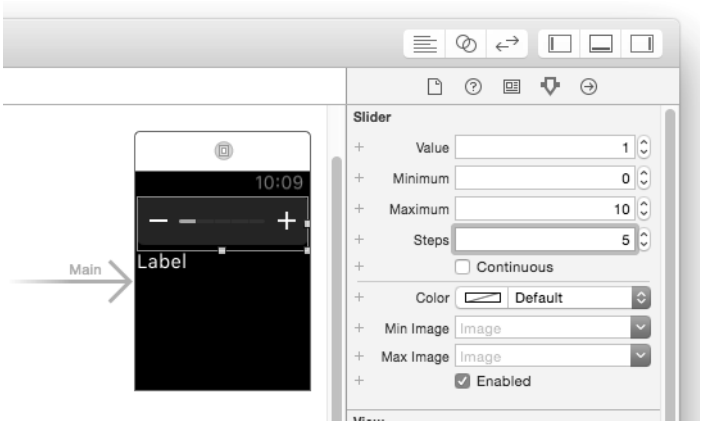

Figure 3.26 Setting the attributes for the Slider control

8. Add the following statements in bold to the InterfaceController.swift file:

```
@IBAction func sliderAction(value: Float) {
    label.setText("\(value)")
}

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

    // Configure interface objects here.
    slider.setValue(0.0)
    label.setText("0.0")
}
```

> **Note**
>
> You can programmatically set the value of a Slider control, but you will not be able to get its value. To know its value, you need to implement the action of the Slider control and save its value whenever the value changes.

9. Select the WatchKit App scheme and run the project on the Apple Watch Simulator. Click the – and + buttons and observe the value printed on the Label control (see Figure 3.27).

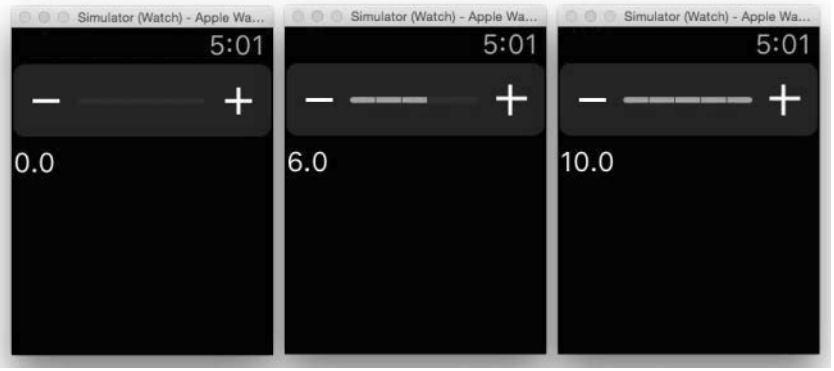

Figure 3.27 Testing the slider

The Steps attribute specifies how many times you can click the slider to reach its maximum value. The increment or decrement value of the slider at any point is dependent on the length of the slider (Maximum value minus Minimum value) divided by the value of Steps. In this example, the length of the slider is 10 (maximum of 10 minus minimum of 0) and the value of Steps is 5; hence, the slider increments or decrements by 2 whenever the + or – button is clicked.

Alerts and Action Sheets

In watchOS 2, Apple now allows developers to display alerts and actions just like they did in iPhone and iPad:

1. Using Xcode, create a new iOS App with WatchKit App project and name it **UsingAlerts**. Uncheck the option Include Notification Scene so that we can keep the WatchKit project to a bare minimum.
2. Select the Interface.storyboard file to edit it in the Storyboard Editor.
3. Drag and drop a Button control onto the default Interface Controller (see Figure 3.28) and set its title to **Show Alerts**.
4. Create an action for the Button control and name it **btnShowAlerts**. The InterfaceController.swift file should now look like this:

```
import WatchKit
import Foundation

class InterfaceController: WKInterfaceController {

    @IBAction func btnShowAlerts() {
    }
```

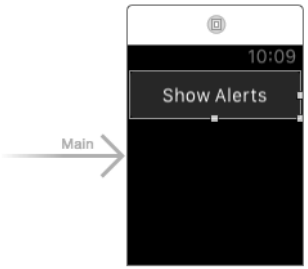

Figure 3.28 Adding a button to the Interface Controller

5. Add the following statements in bold to the InterfaceController.swift file:

```
import WatchKit
import Foundation

class InterfaceController: WKInterfaceController {

    func performAction(actionStyle: WKAlertActionStyle) {
        switch actionStyle {
        case .Default:
            print("OK")
        case .Cancel:
            print("Cancel")
        case .Destructive:
            print("Destructive")
        }
    }

    @IBAction func btnShowAlerts() {
        let okAction = WKAlertAction(title: "OK",
            style: WKAlertActionStyle.Default) { () -> Void in
                self.performAction(WKAlertActionStyle.Default)
        }

        let cancelAction = WKAlertAction(title: "Cancel",
            style: WKAlertActionStyle.Cancel) { () -> Void in
                self.performAction(WKAlertActionStyle.Cancel)
        }

        let abortAction = WKAlertAction(title: "Abort",
            style: WKAlertActionStyle.Destructive) { () -> Void in
                self.performAction(WKAlertActionStyle.Destructive)
        }
```

```
        presentAlertControllerWithTitle("Title",
            message: "Message",
            preferredStyle: WKAlertControllerStyle.Alert,
            actions: [okAction, cancelAction, abortAction])
    }
```

Here, you first defined a function named `performAction:` that prints out a message depending on the style that is passed in as the argument. Next, in the `btnShowAlerts` action, you created three `WKAlertAction` instances, each with a specific style (`Default`, `Cancel`, and `Destructive`). Within each instance, you have a closure that is fired when the user clicks on the action buttons. When each button is clicked, you simply call the `performAction:` function to print out a message so that you know which button was clicked. Finally, you called the `presentAlertControllerWithTitle:message:preferredStyle:actions:` method to display an alert, together with the three action buttons.

6. Select the WatchKit App scheme and run the project on the Apple Watch Simulator. Clicking the button displays an alert (see Figure 3.29).

Figure 3.29 Displaying an alert in the Apple Watch

> **Note**
>
> Note that the **Abort** button in the alert is displayed in red as its style is set to `Destructive`.

7. Modify the `presentAlertControllerWithTitle:message:preferred-Style:actions:` method, as follows:

```
//---SideBySideButtonsAlert supports exactly two actions---
presentAlertControllerWithTitle("Title",
    message: "Message",
    preferredStyle:
        WKAlertControllerStyle.SideBySideButtonsAlert,
    actions: [okAction, cancelAction])
```

8. Select the WatchKit App scheme and run the project on the Apple Watch Simulator. Clicking the button displays an alert with the two buttons displayed side by side (see Figure 3.30).

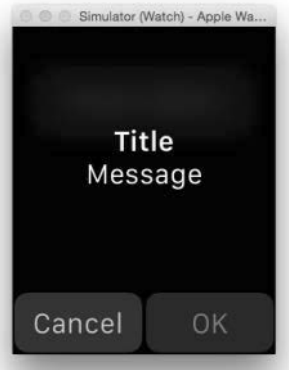

Figure 3.30 Displaying an alert with two buttons side by side in the Apple Watch

> **Note**
>
> For the `SideBySideButtonsAlert` style, you need to specify exactly two action buttons.

9. Modify the `presentAlertControllerWithTitle:message:preferredStyle:actions:` method as follows:

```
presentAlertControllerWithTitle("Title",
    message: "Message",
    preferredStyle: WKAlertControllerStyle.ActionSheet,
    actions: [okAction, cancelAction, abortAction])
```

10. Select the WatchKit App scheme and run the project on the Apple Watch Simulator. Clicking the button displays an alert, as shown in Figure 3.31.

> **Note**
>
> When using the `ActionSheet` style, the action button that is set to the `Cancel` style is displayed at the top-left corner of the screen. Even if you do not specify the cancel action button, a default **Cancel** button is still displayed to close the action sheet (though in this case you cannot handle the event that is fired when the user taps the **Cancel** button).

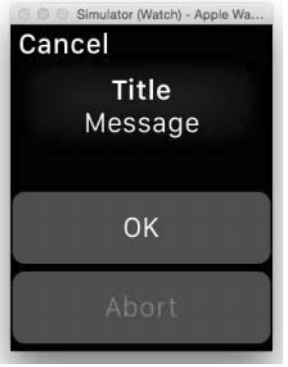

Figure 3.31 Displaying an action sheet in the Apple Watch

Summary

In this chapter, you looked at the various controls that you can use to build the UI of your Apple Watch application. In particular, you saw the various controls that you can interact with by using the tap gesture, such as the Button, Switch, and Slider controls. In addition, you learned about the new alerts and action sheets that you can use to display information in watchOS 2. In the next chapter, you learn more about the other controls that primarily display information to the user.

Index