Professional Expertise Distilled

Xcode 6 Essentials

Create exciting native apps for your Apple devices with Xcode

Jayant Varma

www.allitebooks.com

Xcode 6 Essentials

Create exciting native apps for your Apple devices with Xcode

Jayant Varma

BIRMINGHAM - MUMBAI

www.allitebooks.com

Xcode 6 Essentials

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: January 2015

Production reference: 1220115

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78439-443-1

www.packtpub.com

www.allitebooks.com

Credits

Author
Jayant Varma

Reviewers
Smart Cai
Ben Dodson

Acquisition Editor
Sonali Vernekar

Content Development Editor
Akashdeep Kundu

Technical Editor
Tanvi Bhatt

Copy Editors
Shivangi Chaturvedi
Gladson Monteiro
Jasmine Nadar
Merilyn Pereira

Project Coordinator
Neha Thakur

Proofreaders
Bridget Braund
Stephen Copestake
Maria Gould

Indexer
Tejal Soni

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat

www.allitebooks.com

About the Author

Jayant Varma is a technophile with a career spanning over 2 decades and was introduced to computing in the days of 8-bit computers and Z80 chips. While managing the IT and telecom department at the BMW dealerships in India and Oman, and Nissan in Qatar, he worked extensively on Windows, AS/400, and Unix. His love for travelling inspired him to work and travel to several countries. He is currently based in Australia.

His technological journey began as a Microsoft Technologies developer and has diversified; his focus is now on Apple and mobile technologies. He has a masters degree in Business Administration and IT from James Cook University, Australia. He also lectured at James Cook University and co-ordinated the on-shore and off-shore teaching for the Linux/Unix Administration subject. He worked closely with the Australian Computer Society (ACS) and Apple University Consortium (AUC) on workshops and projects.

He authored the book *Learn Lua for iOS Game Development*, *Apress*, and has also been a technical reviewer on several titles.

As a founder, consultant, and developer at OZApps (`www.oz-apps.com`), he helps organizations and individuals integrate technology into their business and strategies. He also conducts trainings and workshops, and writes blogs to share his knowledge with the community.

www.allitebooks.com

About the Reviewers

Smart Cai is a Chinese iOS developer with 3 years of experience in the field of IT. Currently, he works as a software engineer at Beijing Effective Biz Info Tech Co, LTD. He has also published an application on the App Store named, 今目标 (a business iOS App).

Ben Dodson is a freelance iOS developer in the UK. He has been working on application development since the introduction of the App Store and has built a number of high-ranking applications for clients, including Channel 4, Nectar, ExpenseMagic, and NBC. Ben has also published a number of his own applications including Highlights (featured as "iPad App of the Week" and "Best of 2011 Travel Apps" by Apple) and WallaBee (voted "Best European Gaming Startup 2012" at Nonick 012).

Ben spends the majority of his time working on WallaBee, the largest collecting game on the App Store, of which he is the founder and the lead developer. He is currently working on a number of exciting projects written in Swift and using the latest iOS 8 technologies. He is also working on a major game for the upcoming Apple Watch.

Find out more about him at `www.bendodson.com`.

www.allitebooks.com

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

www.allitebooks.com

Table of Contents

Preface 1
Chapter 1: Introduction to Xcode 7
Requirements and installing Xcode 7
Features of Xcode 10
What's new in Xcode 6? 10
Interactive coding using Playgrounds 11
Mac OS X storyboards 11
Live design and responsive UI 11
Visual debugging 11
Improved debugger 12
Summary 12
Chapter 2: Tour of Xcode 13
Starting Xcode 13
Creating a new project 14
Selecting the project type 14
Setting the project options 15
Setting the project properties 15
Xcode overview 15
The project section 17
The editor and assistant editor sections 18
Source code editors 18
The assistant editor 20
The utility/inspector section 20
The Apple development languages 23
Swift 23
Summary 24

www.allitebooks.com

Chapter 3: Playgrounds 25
Introduction 25
Swift Playgrounds 26
The UI 26
Learning Swift 29
Console output printing 29
Working with variables 29
Variable types 30
Declaring variables 30
What are Int, Double, and Float variables 31
Adding different types 31
Booleans 32
Arrays and dictionaries 32
Tuples 37
Strings 37
Any and AnyObject 38
Control flows and code execution 39
The if statement 39
The if..else statement 39
The if..elseif..else statement 39
The ternary operator 40
For loops 40
The While loop 42
The switch condition 42
Functions 47
Return values 47
Parameters 48
Using closures 50
Objects 51
Classes 51
Enumerations 54
Extensions – extending the classes 55
Operator overloading 56
Summary 57
Chapter 4: Interface Builder 59
Introducing Interface Builder 60
Creating a basic interface 61
Elements on the view 62
Adaptive UI 64
Adding scenes 68
Navigating between View Controllers 69
Building a simple application 71
Adding items to Table View 71
Different elements for different orientations and dimensions 76
Adding the code 77

www.allitebooks.com

Autolayout 79
Subclassing 80
Quick help 82
Managing connections 83
Adding gesture recognizers 84
Segues and connections 86
Summary 87
Chapter 5: Custom Controls 89
An introduction to custom controls 89
The basics 89
Creating a class 90
Properties 91
Changing properties from the inspector 93
Enhancing custom controls 93
Adding some text 95
Frameworks 96
Creating a framework 96
Debugging custom controls 98
Summary 99
Chapter 6: Debugging 101
Breakpoints 102
Listing all the breakpoints 103
Navigating 103
The console 104
Understanding the debug information 105
Visualizing variables with the Quick Look functionality 107
Using images with quick view 109
Debugging the view hierarchy 110
Introducing debug gauges 112
Summary 113
Chapter 7: Building and Running 115
Simulator 115
Choosing the device 116
Preparing for distribution 116
Configuring the project settings 117
The Identity section 117
The Deployment Info section 118
Application icons and launch images 118
Linked frameworks and libraries 120

www.allitebooks.com

The Capabilities tab 120
The Info tab 120
The Build tabs 121
The Targets tab 121
Setting project-wide properties 122
Code signing 123
Organizer 124
Summary 125
Appendix: Conditional Execution and Interface Designing 127
More?.Swift! 127
Differences between iOS 7.x and 8.x SDK 132
Location services 132
Conditional execution 134
Displaying alerts 135
Showing an alert with ActionController 136
Showing ActionSheet with AlertController 136
Designing in Interface Builder 138
Adaptive UI 138
Summary 140
Index 141

Preface

Welcome to *Xcode 6 Essentials*. Development has moved on from the realm of computer science students to practically everyone. A couple of decades ago, computers were expensive and hard to use and now everyone has one in their pockets. Similarly, developmental tools have also changed and made life easier and allowed people from many other aspects to get with developing. With the popularity of iOS devices and Macs, the lure of being the App Store millionaire appeals to one and all. With a growing number of developers using Xcode, Apple has been adding a lot of features to it. In 2014, Apple released Xcode 6 and Swift. This book will cover using Xcode, and the language of choice is Swift. This book will take the reader through the various aspects of Xcode that might be important to use. The book employs a combination of a step-by-step approach and a theory approach to help the reader develop.

What this book covers

Chapter 1, *Introduction to Xcode,* introduces what Xcode is all about. It helps you install Xcode, if you do not have it on your system, and set it up for use.

Chapter 2, *Tour of Xcode*, offers a quick tour of the Xcode UI, the various panels, windows, and settings. This is what you will interact with on a regular basis when developing. It is helpful to know the shortcuts, what each displays, and how to activate or deactivate the windows.

Chapter 3, *Playgrounds*, takes you through a quick journey of Swift, allowing you to learn Swift interactively using the new feature called Playgrounds. This allows you to see the results of your commands and variables; your code is run in the background with no need to compile and run it.

Chapter 4, *Interface Builder*, introduces Interface Builder and Storyboards. It further helps non-developers or beginners to create applications with little to no code. This can also be used to create functional mock-ups as required.

Chapter 5, *Custom Controls*, introduces another new feature added to Xcode: the inclusion of custom controls and live previews. This chapter helps you understand the stub of a custom control and then add code. It further explains how to create this as a framework.

Chapter 6, *Debugging*, helps you as you progress and write your code, given the possibility of errors creeping in. Powerful debugging tools are introduced; they will help you understand where to look for basic help and will set up breakpoints, monitor variables, and quick look values.

Chapter 7, *Building and Running*, helps you understand the next step after creating your application: packaging it for distribution to the App Store. After going through this chapter, you will have run a full life cycle introduction to creating an application for iOS devices using Xcode.

Appendix, *Conditional Execution and Interface Designing*, has some interesting gotchas and tips for using Xcode more efficiently and also learning how the current versions differ from previous versions of Xcode and SDK.

What you need for this book

To be able to run the code and work through the chapter step-by-step, you will need the following:

- An Apple computer running OS X 10.9 or higher
- Xcode 6.x installed on this Apple computer

Who this book is for

This book is aimed at intermediate developers who have little to no experience with iOS SDK. It will also be useful to those who want to get up-to-speed with the new features of Xcode briefly. A basic understanding of programming logic is necessary and any experience with iOS is beneficial. It also introduces and uses Swift, a new language, and will be helpful to those who want a quick introduction to Swift.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "New `UIAlertController` is introduced to replace the older `UIAlertView`."

A block of code is set as follows:

```
var name: String = "XCode Esentials"
var version: Int = 1.0
var title = "\(name) - ver (\version)"
println("The book is \(title)")
```

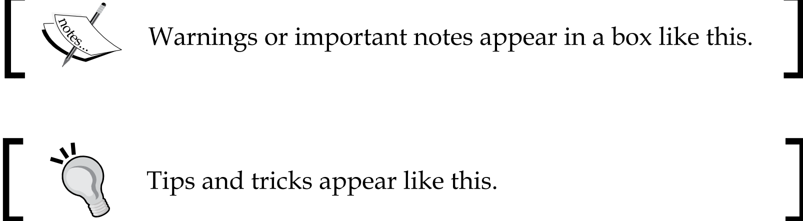
When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
var width: Int = 200
var height: Int = 200
@IBInspectable theColor: UIColor = UIColor.redColor()
var title: String = "Untitled"
```

Any command-line input or output is written as follows:

```
# po UIScreen().bounds
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Clicking on the **Next** button moves you to the next screen."



Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Introduction to Xcode

A carpenter needs his tools to build furniture; similarly, as a developer, you need to have a set of tools that allow you to work in a better and more efficient manner. However, in the analogy of the carpenter, the tools could be used to make different types of furniture. With software, it is a little different. There are plenty of tools and they are dependent on many factors, such as the operating system (Windows, Mac OS X, or Linux) and the language of choice (for example, C, C++, Python, Ruby, JavaScript, and Objective-C). This book is about Xcode, the IDE that Apple makes available to developers who want to develop for Apple devices (iPhone, iPad, Mac, and so on). In this book, we will explore working with Xcode 6.x and all the features that it offers to write code; build UI; and debug, build, and distribute your apps.

Xcode helps you to build native applications that run on and use the features of the current iOS and Mac OS. The current one as of Xcode 6.x is iOS 8 for mobile devices and Mac OS X 10.10 Yosemite for desktops. Xcode 6.x has backward compatibility for iOS 7.0 and OS X 10.9. While there is backward compatibility, new features are available mostly with the newer version or later.

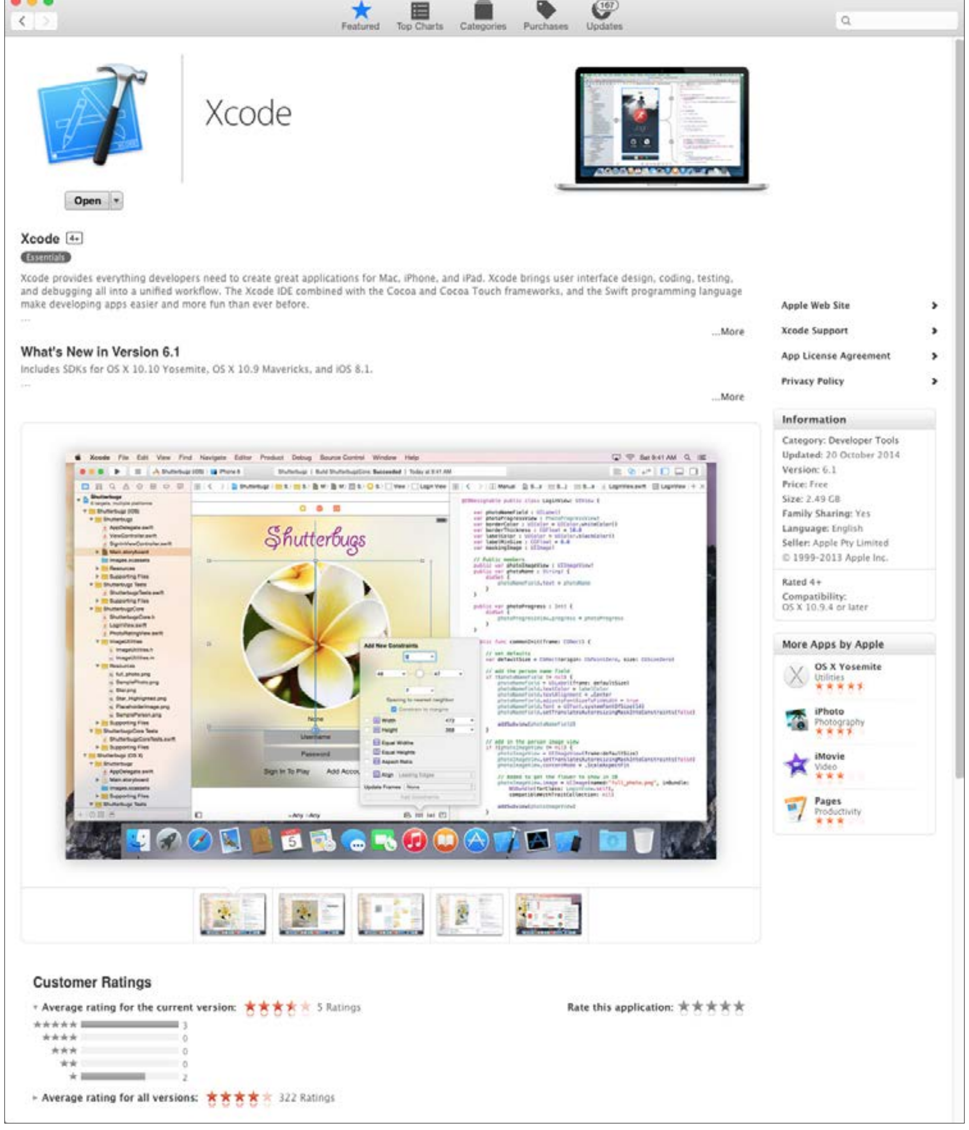
Requirements and installing Xcode

At first, Xcode was available with all Mac OS X installation disks, but then Apple made it available as a purchase from the App Store. Now it is available on the App Store for free. The minimum requirement for Xcode 6 is that you need to have Mac OS X 10.9.x installed with at least 10 GB of disk space. To run the simulator with newer, larger resolutions, it is advisable that you have a large screen display. The retina MacBook Pro or the newer iMac 5K retina would be a very good suggestion to help you work and design at 1:1 resolution.

The method to install Xcode via the App Store eliminates all the setting up and creation of the correct directories, which are generally associated with the installation. This also ensures that all of the upgrades are applied appropriately. If there is an issue, performing a simple reinstallation is as simple as downloading it again from the App Store.

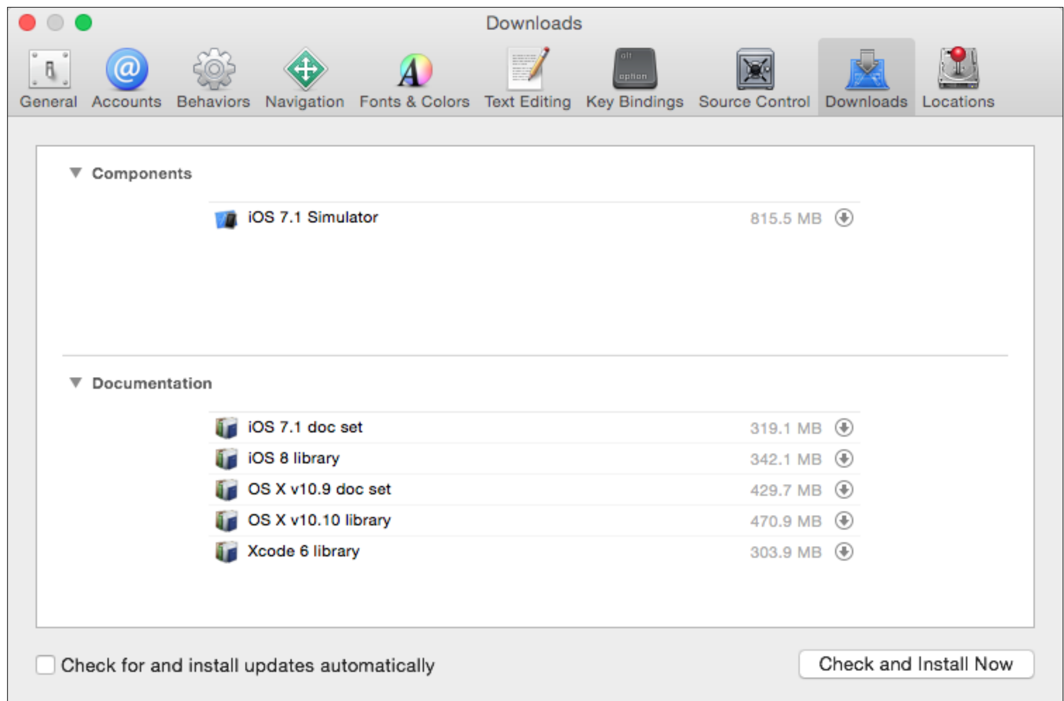
> If you like to work with ß software, you could also consider installing beta versions of Xcode from your Apple Developer Portal. Generally, since Apple does not allow you to create apps with ß versions, unless you are an advanced user, it is advisable to steer clear and wait for the release versions.

Here is a screenshot of the Xcode installation via App Store:



Xcode installation via App Store

Upon first run, Xcode downloads some of the required components and utilities. These can be found under the **Developer** tools menu. Other components, utilities, and documentation can be downloaded via preferences. After these are installed, the API documentation and iOS simulators need to be downloaded:



You can build and test your iOS applications on the simulator from within Xcode and run OS X applications. To test your iOS app on a device or distribute the iOS or OS X app, you will have to enroll in the Apple Developer program for either iOS or Mac OS X, as appropriate, or both (this is a paid option).

The advantage of joining a developer program is that it provides access to pre-release software. It also offers access to the Apple forums, including two incident supports with Apple (per year).

Features of Xcode

Xcode is an all-inclusive IDE, and it offers a variety of tools required by a developer. IDE stands for Integrated Development Environment, which basically means that most of the tools required for a developer are available in this environment. The developer will not have to go outside of this environment to perform other tasks. Some basic features of an IDE include the ability to edit, write, and browse code and be able to run and debug the code from the program and return to the code editor when done. The Xcode environment is made up of the following components:

- **Editor**: Xcode offers a code editor with syntax highlighting for some languages, such as C, C++, Objective-C, JavaScript, Ruby, AppleScript, and Java. With third-party add-ins, support has been added for other languages as well, such as Pascal and C#.
- **File View**: Xcode also features file viewers for code, images, media, and data models.
- **Interface Builder**: Xcode features an Interface Builder that allows you to create interfaces for your apps, mobiles, or desktops. This provides a drag-and-drop interface to lay out the components of your app.
- **Debugger**: Xcode also features a debugger; earlier versions of Xcode used GDB, which has been replaced by the LLDB debugger since 4.3.
- **Versioning**: Xcode has Git integration that allows for versioning and integrated source control.

What's new in Xcode 6?

Xcode has been around for a while, and it was at WWDC 2014 that Xcode 6 was introduced. There were a couple of new features that were introduced in Xcode, and we will explore some of them in this book along with the standard features.

The first important thing that Apple introduced is a new Language called **Swift** (detailed in *Chapter 3*, *Playgrounds*). Since the days of NeXT, when Steve Jobs returned to Apple, Macs has been using the Objective-C language to write applications for the Mac environment. With the introduction of iPhone, Apple introduced their PhoneSDK, which allowed developers to write apps for their phones using Objective-C. This was later named iOS, as the devices were not just phones. Like any language, Objective-C has its own set of followers and haters. With the uptake of iOS devices, many users wanted to try their hand at developing apps. However, Objective-C largely remained a hurdle so they considered other options. At WWDC 2014, Apple unveiled a new language they had been working on, called Swift. Swift is an easy-to-learn-and-use language that looked more like a scripting language than a structured C variant. The code looked smaller and more manageable.

The most important bit was that there was no more memory management in the form of `alloc` and `retain` statements. To add to this, the introduction to Swift by Apple mentioned, "Semi-colons are optional".

Interactive coding using Playgrounds

Apple also introduced a feature called **Playgrounds**. This allows you as the developer to quickly type in your commands in an editor, like you would write in a text editor or Microsoft Word. The Playground compiles, runs, and displays the results of each line of code that is typed. This topic is further detailed in *Chapter 3*, *Playgrounds*.

> There is an open source project that runs your Objective-C code like Playgrounds. So you can simply alter your code in the editor and the running application is altered on the fly. This is not part of this book but could be interesting for some advanced users.

Mac OS X storyboards

Before storyboards, there were XIB files that described the user interface or the form in XML format. Apple introduced the storyboard, which allowed users to define the relationship between the elements, mainly to direct the workflow. This was originally introduced only for iOS projects, but with Xcode 6, storyboards are now available for OS X projects too. This is detailed in *Chapter 4*, *Interface Builder*.

Live design and responsive UI

In earlier versions of Xcode, there was little control over debugging and testing controls you created. Now, Apple offers a way to interact with your controls while in design mode in Interface Builder.

Xcode now offers new functionality to allow you to create a single storyboard that would adapt to different screen sizes with the help of size classes and autolayout. They are detailed in *Chapter 4*, *Interface Builder*.

Visual debugging

When debugging your code, Xcode can now display variables and objects visually. This helps make debugging much simpler and easy to use. It also offers functionality to provide custom previews for your own classes/objects while debugging. They are detailed in *Chapter 5*, *Custom Controls*.

Improved debugger

Even the debugger has some new features added to it. Some of these allow you to test and optimize your code and eliminate errors, including visual debugging with a hierarchical view of the views and elements on the screen. This is detailed in *Chapter 6*, *Debugging*.

In addition to these, we will also cover some of the other features that were present in earlier versions, which are equally important in getting things done.

Summary

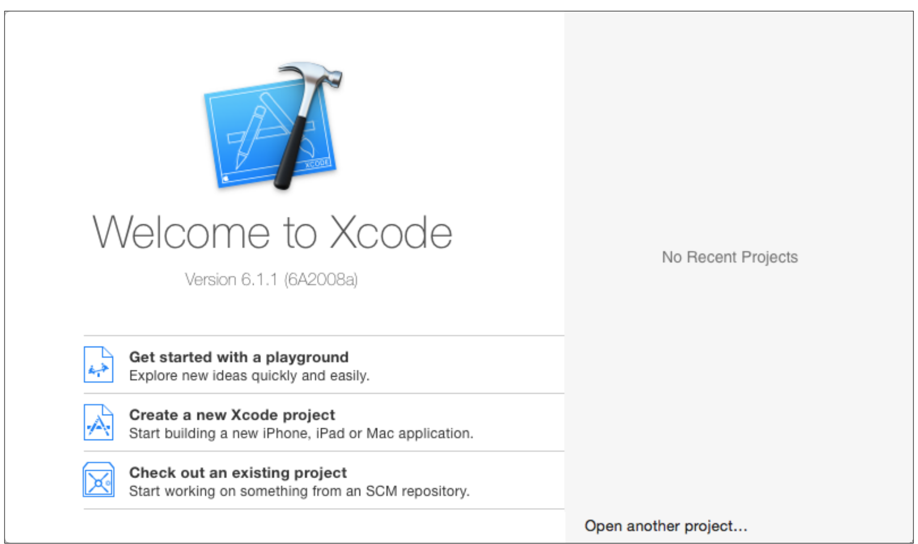
This short chapter talks about how Xcode fits in with your development requirements. Every year, Apple adds new features to Xcode (and hopefully continues). This time around, out of the many features added, one of the major new additions that has everyone talking is a language called Swift and Playgrounds. In our next chapter, we will have a detailed look at Xcode, its components, and UI.

2
Tour of Xcode

In the previous chapter, we looked briefly at Xcode. In this chapter, we shall look at Xcode closely as this is going to be the tool you would use quite a lot for all aspects of your app development for Apple devices. It is a good idea to know and be familiar with the interface, the sections, shortcut keys, and so on.

Starting Xcode

Xcode, like many other Mac applications, is found in the `Applications` folder or the Launchpad. On starting Xcode, you will be greeted with the launch screen that offers some entry points for working with Xcode. Mostly, you will select **Create a new Xcode project** or **Check out an existing project**, if you have an existing project to continue work on. We shall look at the option to get started with a playground slightly later in the book.



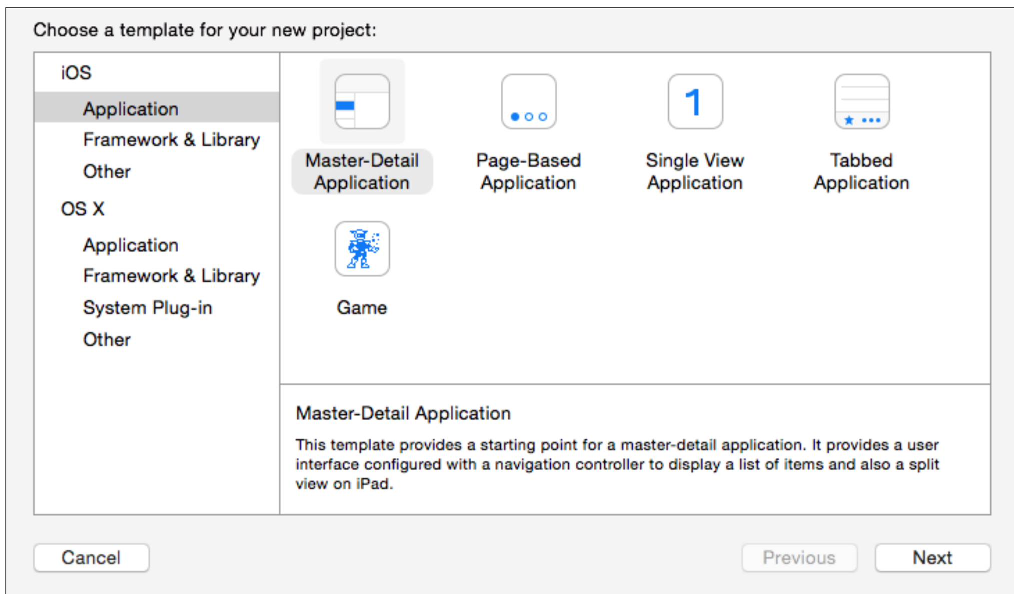
Xcode remembers what it was doing last, so if you had a project or file open, it will open up those windows again.

Creating a new project

After selecting the **Create a new project** option, we are guided via a wizard that helps us get started.

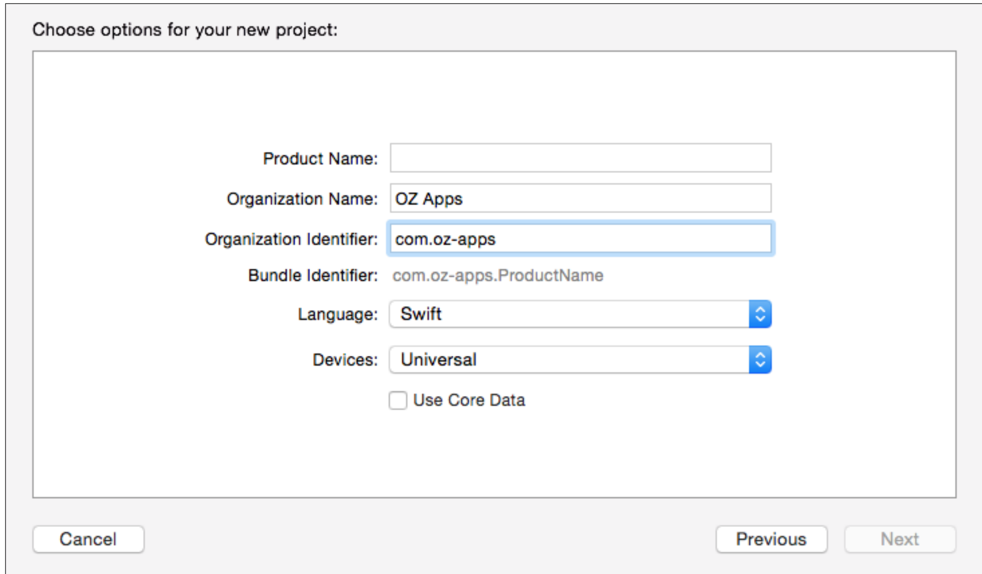
Selecting the project type

The first step is to select what type of project you want to create. At the moment, there are two distinct types of projects, mobile (iOS) or desktop (OS X) that you can create. Within each of those types, you can select the type of project you want. The screenshot displays a standard configuration for iOS application projects. The templates used when the selected type of project is created are self sufficient, that is, when the **Run** button is pressed, the app compiles and runs. It might do nothing, as this is a minimalistic template. On selecting the type of project, we can select the next step:



Setting the project options

This step allows selecting the options, namely setting the application name, the organization name, identifier, language, and devices to support. In the past, the language was always set to Objective-C, however with Xcode 6, there are two options: objective-C and Swift (more of which we shall see later in this book):



Setting the project properties

On creation, the main screen is displayed. Here it offers the option to change other details related to the application such as the version number and build. It also allows you to configure the team ID and certificates used for signing the application to test on a mobile device or for distribution to the App Store. It also allows you to set the compatibility for earlier versions. The orientation and app icons, splash screens, and so on are also set from this screen. If you want to set these up later on in the project, it is fine, this can be accessed at any time and does not stop you from development. It needs to be set prior to deploying it on a device or creating an App Store ready application. These are covered later in the book in *Chapter 7, Building and Running*.

Xcode overview

Let us have a look at the Xcode interface to familiarize ourselves with the same as it would help improve productivity when building your application.

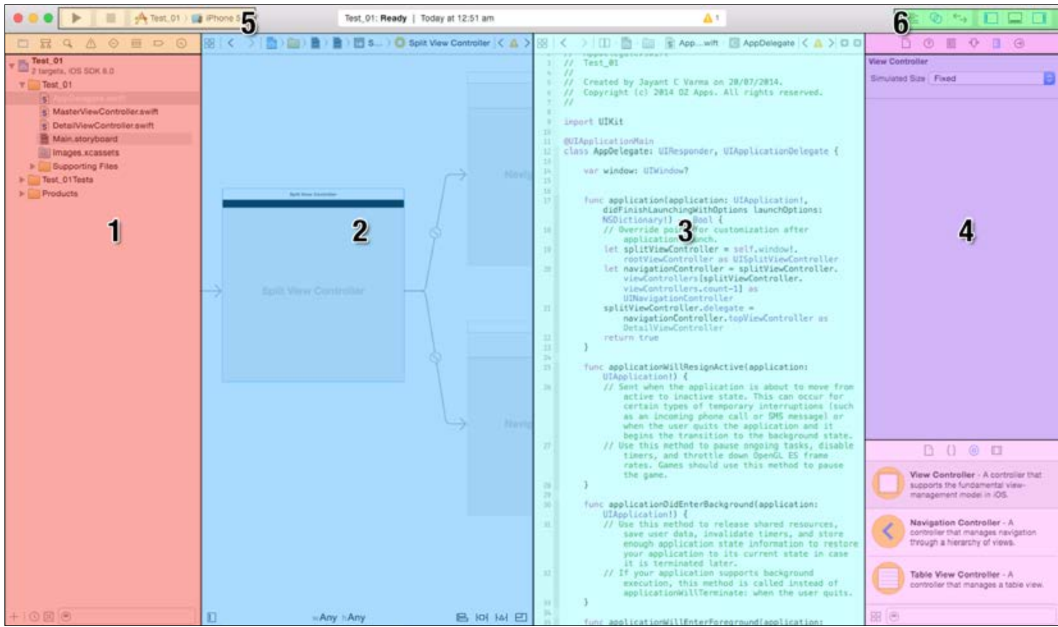
The top section immediately following the traffic light (window chrome) displays a **Play** and **Stop** button. This allows the project to run and stop. The breadcrumb toolbar displays the project-specific settings with respect to the product and the target. With an iOS project, it could be a particular simulator for iPhone, iPad, and so on, or a physical device (number **5** in the following screenshot).

Just under this are vertical areas that are the main content area with all the files, editors, UI, and so on. These can be displayed or hidden as required and can be stacked vertically or horizontally.

The distinct areas in Xcode are as follows:

- Project navigation (number **1**)
- Editor and assistant editor (number **2**) and (number **3**)
- Utility/inspector (number **4**)
- The toolbar (number **5**) and (number **6**)

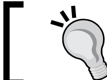
These sections can be switched on and off (shown or hidden) as required to make space for other sections or more screen space to work with:



Sections in Xcode

The project section

The project navigation section has three sub sections, the topmost being the project toolbar that has eight icons. These can be seen as in the following screenshot. The next sub section contains the project files and all the assets required for this project. The bottom most section consists of recently edited files and filters:



You can use the keyboard shortcuts to access these areas quickly with the *CMD + 1...8* keys.

The eight areas available under project navigation are key and for the beginner to Xcode, this could be a bit daunting. When you run the project, the current section might change and display another where you might wonder how to get back to the project (file) navigator. Getting familiar with these is always helpful and the easiest way to navigate between these is the *CMD + 1..8* keys.

- **Project navigator** (*CMD + 1*): This displays all of the files, folders, assets, frameworks, and so on that are part of this project. This is displayed as a hierarchical view and is the way that a majority of developers access their files, folders, and so on.
- **Symbol navigator** (*CMD +2*): This displays all of the classes, members, and methods that are available in them. This is the easiest way to navigate quickly to a method/function, attribute/property.
- **Search navigator** (*CMD + 3*): This allows you to search the project for a particular match. This is quite useful to find and replace text.
- **Issues navigator** (*CMD + 4*): This displays the warning and errors that occur while typing your code or on building and running it. This also displays the results of the static analyzer.
- **Tests navigator** (*CMD + 5*); This displays the tests that you have present in your code either added by yourself or the default ones created with the project.
- **Debug navigator** (*CMD + 6*): This displays the information about the application when you choose to run it. It has some amazing detailed information on CPU usage, memory usage, disk usage, threads, and so on.

- **Breakpoint navigator** (*CMD + 7*): This displays all the breakpoints in your project from all files. This also allows you to create exception and symbolic breakpoints (symbolic breakpoints is an advanced feature and not discussed in the scope of this book).
- **Log navigator** (*CMD + 8*): This displays a log of all actions carried out, namely compiling, building, and running. This is more useful when used to determine the results of automated builds (automated builds is an advanced topic and not covered in the scope of this book).

The editor and assistant editor sections

The second area contains the editor and assistant editor sections. These display the code, the XIB (as appropriate), storyboard files, device previews, and so on. Each of the sub sections have a jump bar on the top that relates to files and allow for navigating back and forth in the files and display the location of the file in the workspace. To the right from this is a mini issues navigator that displays all warnings and errors. In the case of the assistant editors, it also displays two buttons: one to add a new assistant editor area and another to close it.

Source code editors

While we are looking at the interface, it is worth noting that the Xcode code editor is a very advanced editor with a lot of features, which is now seen as standard with a lot of text editors. Some of the features that make working with Xcode easier are as follows:

- **Code folding**: This feature helps to hide code at points such as the function declaration, loops, matching brace brackets, and so on. When a function or portion of code is folded, it hides it from view, thereby allowing you to view other areas of the code that would not be visible unless you scrolled.
- **Syntax highlighting**: This is one of the most useful features as it helps you, the developer, to visually, at a glance, differentiate your source code from variables, constants, and strings. Xcode has syntax highlighting for a couple of languages as mentioned earlier.

- **Context help**: This is one of the best features whereby when you hover over a word in the source code with OPT pressed, it shows a dotted underline and the cursor changes to a question mark. When you click on a word with the dotted underline and the question mark cursor, it displays a popup with details about that word. It also highlights all instances of that word in the file. The popup details as much information as available. If it is a variable or a function that you have added to the code, then it will display the name of the file where it was declared. If it is a word that is contained in the Apple libraries, then it displays the description and other additional details.
- **Context jump**: This is another cool feature that allows jumping to the point of declaration of that word. This is achieved by clicking on a word while keeping the *CMD* button pressed. In many cases, this is mainly helpful to know how the function is declared and what parameters it expects. It can also be useful to get information on other enumerators and constants used with that function. The jump could be in the same file as where you are editing the code or it could be to the header files where they are declared.
- **Edit all in scope**: This is a cool feature where you can edit all of the instances of the word together rather than using search and replace. A case scenario is if you want to change the name of a variable and ensure that all instances you are using in the file are changed but not the ones that are text, then you can use this option to quickly change it.
- **Catching mistakes with fix-it**: This is another cool feature in Xcode that will save you a lot of time and hassle. As you type text, Xcode keeps analyzing the code and looking for errors. If you have declared a variable and not used it in your code, Xcode immediately draws attention to it suggesting that the variable is an unused variable. However, if it was supposed to be a pointer and you have declared it without `*`; Xcode immediately flags it as an error that the interface type cannot be statically allocated. It offers a fix-it solution of inserting `*` and the code has a greyed `*` character showing where it will be added. This helps the developer fix commonly overlooked issues such as missing semicolons, missing declarations, or misspelled variable names.
- **Code completion**: This is the bit that makes writing code so much easier, type in a few letters of the function name and Xcode pops up a list of functions, constants, methods, and so on that start with those letters and displays all of the required parameters (as applicable) including the return type. When selected, it adds the token placeholders that can be replaced with the actual parameter values. The results might vary from person to person depending on the settings and the speed of the system you run Xcode on.

www.allitebooks.com

The assistant editor

The assistant editor is mainly used to display the counterparts and related files to the file open in the primary editor (generally used when working with Objective-C where the `.h` or `.m` files are the related files). The assistant editors track the contents of the editor. Xcode is quite intelligent and knows the corresponding sections and counterparts.

When you click on a file, it opens up in the editor. However, pressing the *OPT* + *Shift* while clicking on the file, you would be provided with an interactive dialog to select where to open the file. The options include the primary editor or the assistant editor. You can also add assistant editors as required.

Another way to open a file quickly is to use the **Open Quickly** option, which has a shortcut key of *CMD* + *Shift* + *O*. This displays a textbox that allows accessing a file from the project.

The utility/inspector section

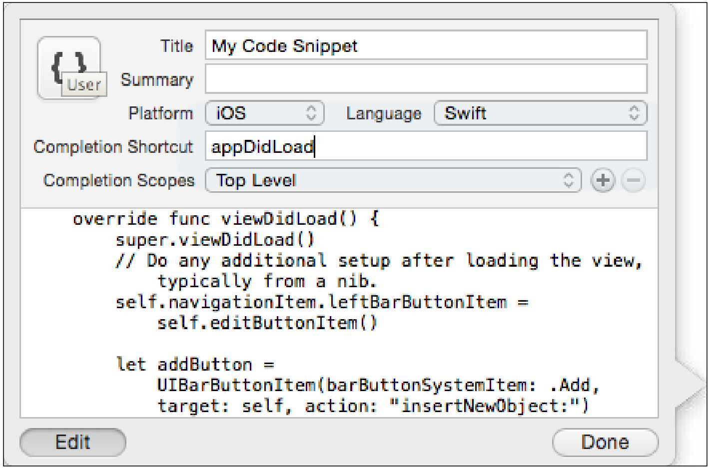
The last section contains the inspector and library. This section changes based on the type of file selected in the current editor. The inspector has 6 tabs/sections and they are as follows:

- **The file inspector** (*CMD* + *OPT* + *1*): This displays the physical file information for the file selected. For code files, it is the text encoding, the targets that it belongs to, and the physical file path. While for the storyboard, it is the physical file path and allows setting attributes such as auto layout and size classes (new in Xcode 6).
- **The quick help inspector** (*CMD*+*OPT* + *2*): This displays information about the class or object selected.

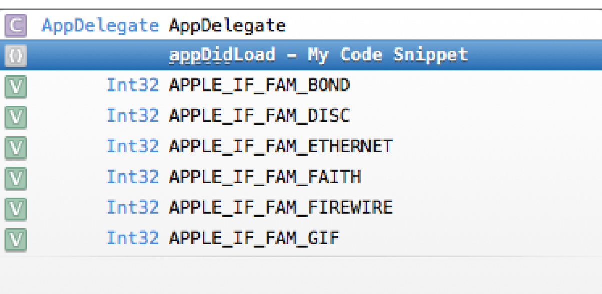
- **The identity inspector** (*CMD* + *OPT* + *3*): This displays the class name, ID, and others that identify the object selected.
- **The attributes inspector** (*CMD* + *OPT* + *4*): This displays the attributes for the object selected as if it is the initial root view controller, does it extend under the top bars or not, if it has a navigation bar or not, and others. This also displays the user-defined attributes (a new feature with Xcode 6).
- **The size inspector** (*CMD* + *OPT* + *5*): This displays the size of the control selected and the associated constraints that help position it on the container.
- **The connections inspector** (*CMD* + *OPT* + *6*): This displays the connections created in the Interface Builder between the UI and the code.

The lower half of this inspector contains four options that help you work efficiently, they are as follows:

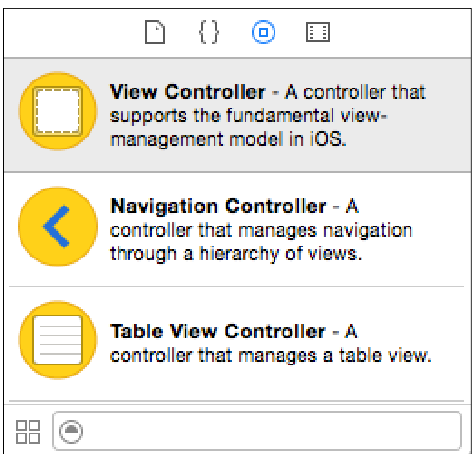
- **The file template library**: This contains the options to create a new class, protocol. The options that are available when selecting the **File** | **New** option from the menu.
- **The code snippets library**: This is a wonderful but not widely used option. This can hold code snippets that can help you avoid writing repetitive blocks of code in your app. You can drag and drop the snippet to your code in the editor. This also offers features such as shortcuts, scopes, platforms, and languages. So you can have a shortcut such as `appDidLoad` (for example) that inserts the code to create and populate a button. This is achieved simply by setting the platform as appropriate to iOS or OS X.



After creating a code snippet, as soon as you type the first few characters, the code snippet shows up in the list of autocomplete options;



- **The object library**: This is the toolbox that contains all of the controls that you need for creating your UI, be it a button, a label, a Table View, view, View Controller, or anything else.



Adding a code snippet is as easy as dragging the selected code from the editor onto the snippet area. It is a little tricky because the moment you start dragging, it could break your selection highlight. You need to select the text, click (hold) and then drag it.

- **The media library**: This contains the list of all images and other media types that are available to this project/workspace.

The Apple development languages

We need to choose a language to write our code in. Till now, developing for an Apple device was synonymous with Objective-C. It is something that has carried on from the days of NeXT. Now Apple has boldly offered a new modern language alternative, unveiled at WWDC 2014, called Swift. All developers that used Xcode till now used Objective-C for development. From now on, Apple is offering the choice to use either Swift or Objective-C as the language to develop with. In the future, there might be just one or there could be more languages, for now, we have these options. We shall focus on Swift as the language of choice for the remaining chapters. Do not worry if you have no idea about Swift, because in the next chapter we shall have a crash course on Swift. Enough swift to get you started up and running.

Swift

In simple terms, Swift is a language that closely reads like a scripting language in comparison to many of the C and C++ variants. Of the long list of advantages and features, some of the key features that would be of interest to someone starting off are as follows:

- Automatic memory management
- Multiple return values and tuples
- Typesafe Language
- Closures
- Function pointers
- Generics
- Structures and enums with support for methods, extensions, and protocols
- It is fast, comparable to Objective-C
- Official language for development supported by Apple

Though Xcode has syntax highlighting for several languages, the official languages offered by Apple are Objective-C and Swift. Objective-C has been the only choice to develop native apps for iOS or OS X. There are plenty of books on Objective-C and we shall look at a crash course introduction to Swift in the next chapter. If you want to know more about either Objective-C or Swift, the best sources are obviously from Apple. Apple has a blog dedicated to resources for Swift including articles, iBooks, and videos to learn Swift from the time they introduced it. This can be found at `https://developer.apple.com/swift/`.

Summary

In this chapter, you have seen a quick tour of Xcode, keep the shortcuts and tips handy as they really do help get things done faster. The code snippets are a wonderful feature that allow for quickly setting up commonly used code with shortcut keywords. Now that you are settled in with Xcode, we can start playing with code. The language of choice for all samples from here on in the book would be Swift.

Let's head on to the next chapter and get our hands into Swift so you can learn all about this new language that everyone is excited about.

3
Playgrounds

In this chapter, we will cover the following topics:

- Introduction to Playgrounds
- Introduction to the Swift language
- Handling loops
- Creating logic (conditionals)
- Functions and parameters
- Understanding classes and structures
- Extensions to classes
- Operator overloading

Introduction

Programming has always been considered difficult. It is not only because of the complexities of logic or the syntax of the language; it has been so also because of the tools available to work with. The second issue is that to test a single line, you have to write the code in an editor, then compile it, and run it to finally see the results; tedious for a single line of code. If you have to test a function, and there is a line that has a bug, it is not easy to debug that unless you have an integrated debugger that allows you to go line by line and add watches, and print out the values of the variables. The mere mention of the "issue" makes it sound complicated. Apple has released a new feature as part of Xcode 6 called Playgrounds. This is largely based on the principles of live coding where one can change the code without having to recompile it. Think of it as a scratchpad that executes your code with each keystroke you type into the editor, without having to even run it. It displays errors and warnings unobtrusively allowing you to continue writing your code and also indicating where the issues are.

Swift Playgrounds

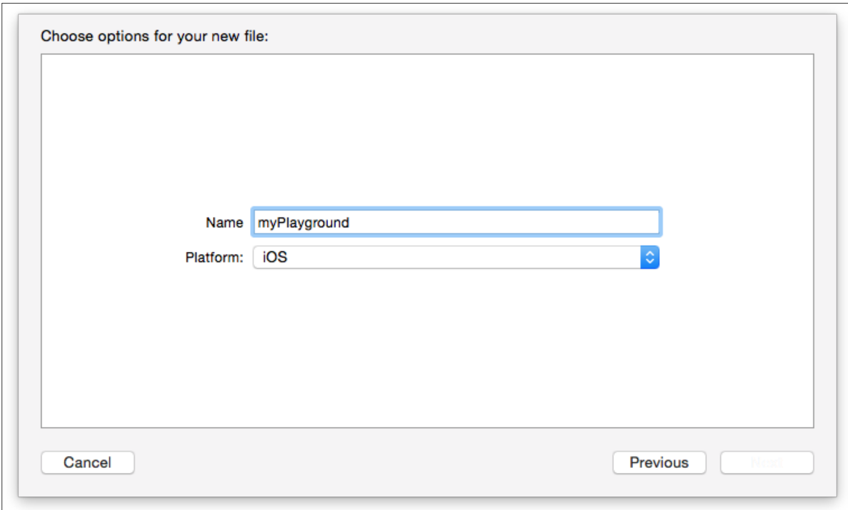
For this chapter, we shall look at this brilliant new feature called **Playground**. You can simply type the code you see in this chapter in a Playground, and it will run and show the results without having to compile or run the application. However, there are a few things to note before we start:

- The language supported by Playgrounds is a new language called Swift, introduced by Apple. You cannot have (as yet) a Playground with Objective-C as the code language.
- The entire file is a single Playground and called so too. This means that the compiler treats it like a single source file. This can give rise to issues where, for example, if you used a variable called `temp` as `string`, and further down you used it as an `int`, the compiler would throw an error complaining that the variable is being redefined.
- The code you can type in a Playground is specific to the platform of the Playground, that is, iOS or OS X. This affects the code in terms of the classes available for use, CocoaTouch or Cocoa. If your Playground is for iOS, you cannot have the classes that are available for Mac OS X, and vice-versa.
- The code is compiled and run at every change as you type/alter the code. The results could vary from machine to machine, depending on the specs of the Mac you are running it on.

You can open only one Playground at a time.

The UI

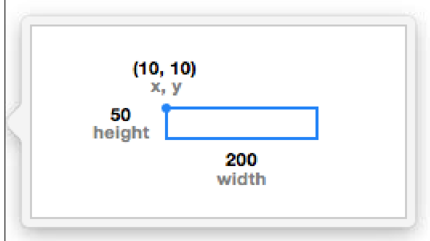
We can create a new Playground by navigating to **File** | **New** | **Playground** or by *OPT* + *CMD* + *N*:



The Playground is the Xcode editor without the toolbar, or the other elements. It has a sidebar that displays the results of the line of code. It is suggested that you turn on the assistant editor as it displays additional information that would be quite useful:

```
Chapter7 〉 Chapter7 〉 MyPlayground.playground 〉 No Selection
1  // Playground - noun: a place where people can
       play
2
3  import UIKit
4
5  var str = "Hello, playground"                    "Hello, playground"
6
7
```

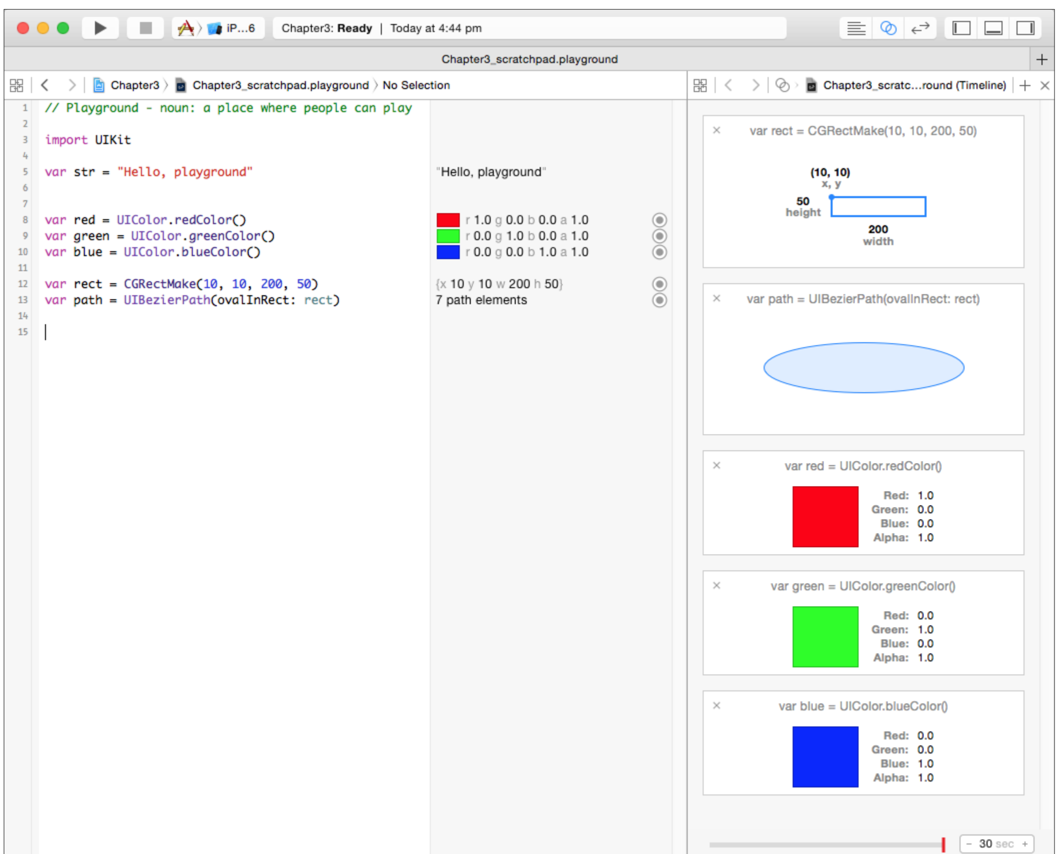
If you hover over the results in the main code editor, you see two icons, an eye and a circle. The eye allows you to view the value in a pop-up, if it is a primitive, such as `Int`, `String`, and `Double`, it will display the value. If it is a complex class such as `CGRect`, `CGPoint`, `UIBezierCurve`, and `UIColor`, it is displayed visually, as in the following screenshot:



Playgrounds can show a lot of information that is otherwise difficult to visualize. For the simplest amazing example to illustrate this point, type the following code in a Playground to see the results:

```
var red = UIColor.redColor()
var rect1 = CGRectMake(10, 10, 200, 50)
var path = UIBezierPath(ovalInRect:rect1)
```

The following screenshot shows the Playground:



The items in the sidebar can be added by simply hovering over the line and clicking on the **+** icon. If the item is displayed, the icon shows as a filled circle, as shown in the preceding screenshot.

Learning Swift

Apple released Swift officially as an alternative language for development, apart from Objective-C. It looks and reads like an easy-to-use scripting-like language. It is a modern language, and has the features of most modern languages such as Python, Ruby, Lua and Scala. Swift removes the need for semicolons and uses the standard double slash for comments, amongst other things, as shown here:

```
//This is a comment
```

Console output printing

The `println` function is used for printing data. Swift also has a more evolved `println` function that allows for formatted printing. The value of a variable can be inserted into a string by enclosing the variable name in brackets preceded by a slash: `\(varname)`. This format has to be enclosed in double quotes to make it work.

```
println("Hello World")
println("This is fine too");

var ver = 6
println("Hello from Xcode \(ver)")
```

Working with variables

We can add two numbers, say 1 and 3, but if our program just added 1 and 3, we would not have much use of the same. If we wanted to write a program that adds two numbers that we pass and it returns a result, then it would be of some use. There are a couple of things that need attention here; we need to first store the numbers in some space and then be able to access them. Variables are basically storage spaces in the system that hold a value of a particular type, and access is via named labels. It would be easier to remember a variable called `age` than one called `0xff22400d3`. Swift has two types of variables; mutable and immutable. Immutable variables are called constants. The values that they hold cannot be changed.

When dealing with large numbers, it is difficult to type them in or read them while trying to group them mentally. A number like 123456789 is not as easy to decipher as 123,456,789 is. While assigning the same number to variables, it is not easy to read and check the same, as you cannot set something like `var largeNum = 1,234,567.89`. However, Swift allows you to format your numbers with an underscore as `1_234.50`, or in the preceding example as `1_234_567.89`:

```
var thousands = 4_693.45
var billions = 3_142_857_143
var crazyNumber = 1_23_456_7_890
```

Variable types

In Swift, a variable can be of any type as available. However, there are some internal types that are provided by the system, they are as follows:

- `Int`, `Float`, `Double`
- `String`
- `Array`, `Dictionary`
- `Bool`

Declaring variables

All variables in Swift need to be declared prior to using them, as Swift is a typesafe language. The compiler checks for the types and highlights of potential issues with the use of non-matching variable types used. So, a variable that is declared as `Int` cannot hold any other type than `Int`. The two keywords used to declare variables are `let` and `var`; `let` declares constants (the value for these variables can be set once and not changed after that) and `var` declares mutable variables:

```
var publisher:String
let name:String = "Learning Xcode 6"
var year:Int = 2014
```

Variables can either just be declared or can be both, declared and assigned a value. Once declared, a mutable variable can be reassigned a new value; whereas, if we try to do so with a constant, the compiler would complain:

```
name = "Learning Visual Studio"    // Compiler complains
year = 2015                        // Works ok
```

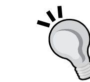
The Swift compiler is quite intelligent and it can implicitly define variables. It knows that when you type the following:

```
let author = "Jayant Varma"
```

The variable `author` is of type `String` as the value assigned to it is of type `String` and it does not have to be explicitly declared.

Multiple variables can be declared on the same line too by separating them with commas.

```
var a = 1, b = 2.3, c = "Four"
var x, y, x : Double
```



If you have to declare a variable that is a keyword, you can enclose it in back ticks `. You can access all the variables using back ticks, though it's not very convenient.

What are Int, Double, and Float variables

Implicit assignment works even with numbers, and all the numbers without the decimal point are considered of type `Int` and all with a decimal point are of type `Double`:

```
var nextYear = year + 1        // autocasts to type Int
```

In cases where you wanted this variable to be `Double` and not `Int`, you could explicitly assign it to type `Double` as follows:

```
var nextYearAndHalf = Double(year) + 1.5
```

When working with iOS SDK, you will realize that you will need another type of variable for numbers, called `CGFloat`; while it is a floating point number, it is a different structure than `Double` of `Float` or `Int`.

Adding different types

As mentioned earlier, the Swift compiler expects the types to be of the expected types, so while a number is a number and broadly we consider 1, 1.0, and "1.0" as the same, for computers, and more so for Swift, these are three different types, namely `Int`, `Double`, and `String`. When working with different types, we need to convert them to the same type in order for it to work:

```
var pi = 3.14           // autocast to Double
var e = 3               // autocast to Int
var pie = pi + e        // Compiler error - not similar types
```

To resolve such issues, we need to cast the values to the expected type (if they can be converted), we can fix this by using the following line of code:

```
var pie = pi + Double(e)    // works as expected
```

Or, as we discussed in the previous example, regarding `CGFloats`, you could use the following line of code:

```
var cgPie = CGFloat(pie)    // this is now of CGFloat type
```

The same is true for strings; we can easily do something like the following:

```
var hello = "Hello"
var world = "World"
var test = hello + world
```

However, we get an error when we try something like this:

```
var appname = "Xcode"
var appversion = 6
var bookname = appname + appversion         // we get an error
```

This is because we are trying to add type `String` with type `Int`. We can convert `Int` to `String` and it would work fine, we can do that easily by using the following line of code:

```
var bookname = appname + String(appversion)
```

Or, use the string formatting feature in Swift:

```
var sBookname = "\(appname) \(appversion)"
```

Booleans

Booleans are simple `true` and `false` values, and the variable type in Swift is called `Bool`. They form the basis for a lot of logic and conditions. It can be used either by setting them explicitly via declaration, or can be set via conditions and never assigned to variables, as shown here:

```
var worksOnWindows = false
if appversion > 6 {
    println("Cutting edge version")
 if worksOnWindows == true {
  println("and it works on Windows too")
 }
}
```

In the first line, we explicitly set the value of a variable to type `Bool`, and in the second line, we use the result of the condition `appversion > 6`, which evaluates to either `true` or `false`.

Arrays and dictionaries

Sometimes, we need multiple variables, say a student enrolled at university, and we want to store the name of the classes they are enrolled in. We can have something like this:

```
var class1 = "Math"
var class2 = "Creative English"
var class3 = "Compiler Theory"
```

This works but is not practical as the more classes the student is enrolled in, the more variables there will be. The student might enroll in a few classes this semester and might take additional classes in the next semester. How would we know how many classes the student was enrolled in? How would we access the variable names in our code? If we were to use the `classXX` format by name, we are hardcoding our logic and this would not be good. For this purpose we use an `Array`.

Array objects

An array stores a list of values and these can be accessed via a sequential index. In simple terms, we can access the list using a numeric index that is in sequence, starting from `0` up to the number of values held in the array. Arrays are declared in Swift as follows:

```
var arrClasses:[String]    //shorthand declaration
```

The alternative way is to declare an array as follows:

```
var arrClasses = Array<String>()  //declaration
```

This tells the compiler that `arrClasses` is a type `Array` variable that holds `String` values. With arrays, we also get additional functions that allow us to work with the array, such as, to get the number of elements in the array we can use the `count` function.

Initializing array objects

Arrays need to be initialized, we can either initialize them with a blank list, or we can populate the values while initializing them, as shown in the following code:

```
var arrClasses:[String]
```

This only declares the variable called `arrClasses` to be of type array of string, but it is not initialized and cannot be used prior to initialization:

```
arrClasses = []            //Initialized as blank
```

A blank array can be declared and initialized by using `var arrBlank = [Int]()`. It can also be initialized with a set of default repeating values like this:

```
var arrZeros = [Int](count:10, repeatedValue:0)
// Array of Ints with 10 items and all initialized to 0
```

Arrays can also be initialized with default values, as follows:

```
var arrSlots = ["Morning", "Afternoon", "Evening", "Night"]
```

This would have created an array with four string elements. If we will not have the need to modify this array, we can declare it with `let` instead.

In our example, we can now have the student enroll in several classes and we can know the classes the student is enrolled in and also the number by accessing the array:

```
var numberOfClasses = arrClasses.count
```

Apart from the `count` property, there is another property called `isEmpty` that returns `true` if there are no elements in the array.

Appending values

More elements can be added to the array using the `append` method, as shown in the following code:

```
arrClasses.append("Finance")
arrClasses.append("Learning MS Office")
```

Or, we can also use the `+=` operator as follows:

```
arrClasses += ["Algorithms"]
arrClasses += ["Basket Weaving", "Bread Making", "Cheese Tasting"]
numberOfClasses = arrClasses.count
```

The index can be used to access the elements or to modify the elements, as follows:

```
arrClasses[0] = "Basics of Finance"  // Modifies the 1st element
```

Removing values

Elements can be removed using the `remove` method. The `remove` function returns the element value it is removing:

```
arrClasses.removeAtIndex(1)          // Removes the 2nd element
```

We can also remove the last element or remove all the elements using the `removeLast` or `removeAll` functions.

Inserting values

We can also add an item specifically at an index position using the `insert` function:

```
arrClasses.insert("Creative Writing", atIndex: 2)
```

Iterating values

We can iterate through the array using a `for` loop:

```
for sClass in arrClasses {
 println("class name : \(sClass)")
}
```

Enumerating values

Another way is to enumerate the items in an array using the `enumerate` function:

```
for (index, value) in enumerate(arrClass){
  println("Class at index \(index+1) is \(value)")
}
```

We used `index+1` in the code because all the array indexes are `0` based.

Dictionary objects

Dictionaries are also called *key-value* pairs or associative index. This is because, instead of a numeric index value, we have an associative value to access the element or a key to retrieve the value. This is quite useful in cases where you might want to access an item (the value) by a key (usually a string) instead of using a numerical index. Declaring dictionaries is similar to declaring arrays. However, the only difference is that Swift expects that the variable types for both the key and the value be specified:

```
var dClassTimes : [String:String]
// Specifies both the key and the value would be of type string
```

An example for using an associative array / dictionary could be to store the subject code with the subject name:

```
var  myCourse : [String:String] = ["CP1001":"Computing Basics",
"FP1001":"Finance Basics", "CP2002":"Algorithms"]
```

Then, we can simply get the subject name using the following code:

```
var theSubject = myCourse["CP1001"]
println("The subject name for CP1001 is \(theSubject!)")

var _CODE_ = "CP1005"
println("The subject name for CP1001 is \(myCourse[_CODE_])")
```

```
if let invSubject = myCourse["CPxxxx"] {
    println("The invalid subject is \(invSubject)")
}
```

The functions to work with dictionary variables are similar to the ones available with arrays, including the `isEmpty` and the `count` functions.

Adding values

In a dictionary object, new values can be added by simply specifying the value for a key:

```
myCourse["OZ1001"] = "Custom Subject"
```

Replacing values

This would replace the value that has the key `CP1001` with this new text:

```
myCourse["CP1001"] = "Basics of Computing"
```

Another alternative for replacing a value in the dictionary object is using the `updateValue` function and pass the `forKey` parameter as the key:

```
myCourse.updateValue("Basics of Finance", forKey:"FP1001")
```

Removing Values

To remove a value from the dictionary, either set the value of the key to `nil` or use the function `removeValueForKey`:

```
myCourse["CP2002"] = nil
myCourse.removeValueForKey("CP1001")
myCourse.removeValueForKey("LP1001") // Non existent key-value
```

The function returns the value being removed or `nil` if none existed.

Iterating values

The elements in a dictionary object can be iterated using the `for` loop:

```
for (key, value) in myCourse {
  println("The key :\(key) has a value of : \(value)")
}
```

Each item in a dictionary object is returned as a tuple (key, value) and all of the keys, or all of the values can also be accessed as arrays:

```
var allKeys = myCourse.keys.array
var allValues = myCourse.values.array
```

Tuples

Tuples are a simple concept. It is a data structure that holds multiple parts. The values inside of a tuple can be of the same type or different types. Declaring a tuple is easy and can be defined as TupleName and the value parts inside brackets:

```
var webError = (Int, String)
webError = (404, "Page not found")
var myTuple = ("07:00", "Morning Run", true)
```

Accessing the members of a tuple

To access the members of a tuple, the values can be stored into normal variables. This reverse assignment is called decomposition:

```
var (errorCode:Int, errorDescription:String) = webError
```

Since tuples are not arrays or dictionary objects, they cannot be accessed using indices, sequences, or associates. In such a scenario, Swift creates sequential member properties that allow access to the tuple members:

```
println("The components of webError are code: \(webError.0) and
Description: \(webError.1)")
```

Named access to tuple members

It is not always very easy to use the sequential members to access tuples. How would you know what was `.0` and what was `.1`, this is where named members make the access easier. Creating named members just involves defining a name for the tuple members:

```
var namedError = (code:200, description:"OK")
println("The components of named error are code : \(namedError.code)
and description : \(namedError.description)")
```

These components can now be accessed using the named members, as follows:

```
var theCode = namedError.code        // returns the error code
```

Strings

Working with strings is also quite easy in Swift. Like arrays, strings can be iterated using a `for` loop:

```
var testing = "This is a test"
for char in testing {
  println("The char is \(char)")
}
```

Appending strings

The `+` and `+=` operators can be used to concatenate strings:

```
var strSmall = "Small "
strSmall = strSmall + "string "
strSmall += "with added text"
```

Formatting strings

Like the `println` function, strings can be formatted and stored in strings instead of using them for printing:

```
var theStr = "The test string"
var length = countElements(theStr)
var strRes = "The length of \(theStr) is \(length)"
println(strRes)
```

Any and AnyObject

`Any` needs to be mentioned, as it is a type that matches any of the types. This is useful to store a type that could change:

```
var someValue:Any = "Frogs"
someValue = 6
someValue = "Green Tree Frogs"
```

If instead of using the `Any` type in the preceding example, you used something like this:

```
var stringValue = "Green Frogs"
stringValue = 6
```

Then Playgrounds would complain, as the `stringValue` variable has been implicitly defined as `String`. Assigning an integer value to a string variable is not acceptable. This functionality allows Xcode to be *typesafe* and ensures that even before you run your code, you are accidentally assigning wrong value types to the variables. The `Any` variable type is similar to `id` in Objective-C:

```
var  castedString = someValue as String
// the last value of someValue was "Green Tree Frogs"
```

The `AnyObject` type is similar to the `Any` variable type and the values need to be cast to a specific type to work with. If a value is of type `Any` and you compare it to a `String`, it would not work because type `Any` is not the same as type `String` and there is no function that compares these two using the `==` operator. So, it must be cast to a variable of type `String` and then it could be easily compared.

Control flows and code execution

Like the C variants, Swift offers a series of control flow options that allow conditional execution or multiple executions of other statements. Swift has most of the C-type constructs providing the familiar `for` loops, `if..else`, `switch`, and a couple of newer ones.

The if statement

The classic `if` statements are available in Swift, and the commands that need to be executed if the condition evaluates to `true` or `false` is enclosed in curly brackets:

```
var berry = "black"
if berry == "red" {
 println ("This is not a blackberry")
}
```

We see that the condition does not evaluate to `true` as the value in the variable called `berry` is `black` and we are checking it for `red`. So we do not see any output.

The if..else statement

The `if` statement also has functionality to do something when the condition does not evaluate to `true`. So, it basically has two sets of statements; one that evaluates when the condition is `true` and the other when it does not:

```
var numApples = 3
if numApples > 5 {
  println("You have enough Apples")
} else {
  println("You might want to buy some more Apples")
}
```

The if..elseif..else statement

Sometimes, there are more than a single `if..else` type conditions. In cases where you need more than two conditions, we use the `if..elseif..else` statement:

```
var platform = "MAC"
if platform == "MAC" {
  println("This is running Apple OS X")
}else if platform == "WIN" {
  println("This is running Microsoft Windows")
```


www.allitebooks.com

```
}else{
  println("This could be running Linux")
}
```

We can add more `else if` statements to check for more conditions.

The ternary operator

The ternary operator `?:` is also referred to as the Elvis Operators since it looks like Elvis' hairdo in ASCII. This is the shortcut for an `if...else` condition and can be used as follows:

```
var eggs = 0
var bread = eggs > 0 ? 12 : 1
```

This checks for a condition, and if it is `true`, evaluates to the statement between `?` and `:`, and if the condition is not satisfied, it evaluates to the statement after the `:`.

For loops

When you want to perform some statements repeatedly, you can use loops. This makes it easy rather than write them several times over.

The for-increment loop

The classic `for` loop is `for-condition-increment` loop:

```
for var i=0; i<=10; i++ {
  println("The value of i is \(i)")
}
```

You can also have multiple assignments or increments in the preceding `for` loop, as shown:

```
for var i=0, j=0, k=0; i<=10; i++, j--, k+=5 {
  println("The value of i=\(i) j=\(j) k=\(k)")
}
```

The for-in loop

The new `for` loop added in Swift is the `for-in` loop that we have already seen in use earlier for enumerating arrays, and with strings. The `for-in` loop provides a range of values for the loop to iterate through.

Using for with strings

The following is an example of using the `for` loop with strings:

```
for i in "Hello" {
  println("The value of i = \(i)")
}
```

Using for with numeric ranges

The following is an example of using the `for` loop with numeric ranges:

```
for i in 1...10 {
  println("The value of i = \(i)")
}
```

Swift has the concept of half closed and full closed ranges. The preceding example is a full closed range that prints the value of `i` from 1 through to 10:

```
for i in 1..<10 {
  println("The value of i = \(i)")
}
```

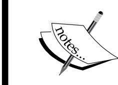
This would print the values of `i` from 1 to 9, not including the last value that we specified of 10.

Stepped ranges

While we can create ranges with the three dots operator (`...`), we can create these stepped ranges or strides using the `stride` functions:

```
for i in stride(from:1, through:20, by:3){
 println("The value of i = \(i)")
}
```

The other alternative `stride` function is `stride(from:1, to:20, by:3)`.



The `stride` function with the `through` parameter is `...` (full closed range) and the function with the `to` parameter is `..<` (half closed range).

The While loop

The other alternative to `for` loops is a `while` loop; this performs the statements repeatedly as long as the condition provided evaluates to `true`, as shown in the following code

```
var coins = 0
while coins < 10 {
  coins += 1
  println("We have mined \(coins) coins")
}
```

This will run and keep incrementing our coins, and when it reaches the value of 10, it stops, as our condition is that the `coins` should be less than 10, so the maximum `coins` we should have are 9. However, because when the value of `coins` is 9, the `while` condition is evaluated as `true` and run once again, incrementing our `coins` count to 10.

There is a variation on the `while` condition and that is the `do..while` loop. The difference between the `while` loop and the `do..while` loop is that the `while` loop may or may not execute, depending on the condition; whereas the `do..while` loop will execute at least once before evaluating the condition to determine running the code or not:

```
coins = 0
do {
  coins += 1
  println("We have mined \(coins) coins")
}while coins < 10
```

We find that the loop ran once again and now we have 10 coins. When the value of coins reaches 10, the loop is stopped and not executed again.

Seeing the two variations, be careful when you use them, they could give you unexpected results due to the way they work.

The switch condition

The `if` statement works with conditions and mostly we use a series of `if` statements to check for the value of a variable. `switch` is a better approach that allows checking for multiple values.

```
var age1 = 28
switch age1 {
  case 0:
   println("Baby")
```

```
  case 1:
   println("Toddler")
}
```

This code would cause the compiler to complain, especially since the Swift compiler attempts to reduce programming errors. We have the code that is not faulty, but we have not handled the scenario where none of the cases are matched as in the preceding example. Swift expects a `default` value to handle the other scenarios that do not match. The working code would look like this:

```
var age1 = 28
switch age1 {
  case 0:
   println("Baby")
  case 1:
   println("Toddler")
 default:
  println("Others")
}
```

If we were to classify the ages, it would be quite a hassle to type in every possible match, we might just want that 1 to 3 be a toddler and 4 to 12 be a child, as shown ahead:

```
var age1 = 28
switch age1 {
  case 0:
   println("Baby")
  case 1, 2, 3:
   println("Toddler")
 case 4, 5, 6, 7, 8, 9, 10, 11, 12:
 println("Child")
 default:
  println("Other")
}
```

This is a bit tedious, typing in all of the values. Swift helps us out here with ranges, we can use the range operator `...` (three dots) as shown in the following code:

```
var age1 = 28
switch age1 {
  case 0:
   println("Baby")
  case 1...3:
   println("Toddler")
  case 4...12:
```

```
     println("Child")
    case 13...19:
     println("Teen")
    default:
     println("Other")
  }
```

We can also use non-contiguous ranges, such as 0 to 17 and 60 to 99 in this example to segregate them as working and non-working population:

```
var age1 = 28
switch age1 {
  case 0...17, 60...99:
   println("Not Working")
default:
   println("Working")
}
```

The `switch` statement can handle pretty much any type of value, not just numeric. It can handle strings, tuples, and many more.

```
var berry1 = "red"
switch berry1 {
  case "red" :
    println("This is a red berry")
  case "blue" :
    println("This is a blue berry")
  case "black" :
    println("This is a black berry")
  default
    println("This is a not a berry")
}
```

Character ranges can also be used like `"a"..."i", "l"..."x"`.

Tuples can also be used for switch case matching:

```
var aScore = (100, "Temple Bun")
switch aScore {
  case (1000, "Temple Bun"):
  println ("You made the high score")
  default:
  println("A little more \(1000-aScore.0) and you can beat the high
score")
}
```

With tuples, you can choose to ignore a value, that is, say you wanted to check the scores of a particular game or just a particular score for any game, as in the following code:

```
switch aScore{
  case (_, "Temple Bun"):
    println("Wow, I could never score on this game")
  case ( 100, _):
    println("Yay! You are a player… I have not yet tried \(aScore.1)")
  default:
    println("All play makes Jack a dull boy")
}
```

There are a couple more tricks with Swift. In the previous example, since our score is 100, the first condition matches and it prints out the "`Wow, I could never …`" message. This is the default behavior, and the expected one too. In some cases, you might want it to continue matching the next case. In such a scenario, you can use the `fallthrough` keyword. By default, the code exits the switch block and does not check the next case, with `fallthrough`, the statements for the next case are also executed, irrespective of the match, as shown here:

```
switch aScore{
  case (_, "Temple Bun"):
    println("Wow, I could never score on this game")
    fallthrough
  case ( 100, _):
    println("Yay! You are a player… I have not yet tried \(aScore.1)")
  default:
    println("All play makes Jack a dull boy")
}
```

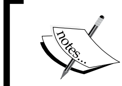
Now, we will see both the messages, "`Wow, I could…`" and "`Yay! You are a …`".

In another example, we can see this:

```
var theNumber = 1
switch (theNumber) {
  case 1:
    println("One")
    fallthrough
  case 2:
    println("Two")
  case 3:
```

```
    println("Three")
  default:
    println("Default")
}
```

You will see that in the output it prints `One` and `Two`. This is useful in cases where you might want to only execute `println("Two")` if `theNumber` is `2`, but display both `One` and `Two` if it is 1.



If you need to exit a loop or a switch, the `break` statement exits the current scope and continues execution.

In the examples, you will notice that we used `aScore.0` or `aScore.1` to access the first or the second member of the tuple. This is fine, but sometimes you want to access them with a proper variable name. We can do this via *value binding*, and use it as follows:

```
switch aScore{
  case (let score, "Temple Bun"):
    println("Wow, I could never score \(score) ")
  case ( 100, let game):
    println("Yay! You are a player… I have not yet tried \(game)")
  default:
    println("All play makes Jack a dull boy")
}
```

`switch` is even more powerful in Swift and it offers more condition checking options via the `where` clause:

```
var marks = (80, "Math")
switch marks {
  case (let score, let subject) where score>75 && subject == "Math" :
    println("You got a HD in Math")
  case (50..<80, let subject):
   println("You passed the subject \(subject)"
  case (let score, let subject):
   println("You scored \(score) in \(subject)")
  default:
   println("")
}
```

You can also check for compound cases, such as `switch score + bonus` in one go, or use functions to check conditions.

Functions

Functions in Swift are declared using the `func` keyword. A function encloses a set of statements inside a block. When a function is called, all of the statements inside that block that make up the function are executed.

Return values

Functions may or may not return any values. Although, functions in Swift can return multiple values via tuples.

The no-return value function

The following is an example of a no-return value function:

```
func noValues(){
 println("this returns no values")
}
```

The single-return value function

The following is an example of a single-return value function:

```
func returnOne() -> Int{
 return 1
}
var one = returnOne()
```

You might note that when a function returns a value, we need to specify the return type of the value and that the function is written with a `->` at the end followed by the return type.

The multiple-return value function

The following is an example of a multiple-return value function:

```
func returnMany() -> (Int, String){
  return (404, "Page not found")
}
var error = returnMany()
```

The multiple return values are in the form of a tuple and it is imperative to define the return values. In the previous example, we return an `Int` and a `String` value. While in this example, we are returning the tuple without named parameters; depending on your needs, you could return tuples with named parameters.

Parameters

Functions might accept parameters that allow you to pass values that may influence the functionality of the function. We have seen examples of functions that do not require any parameters in the previous examples. If we need to pass parameters, we need to specify the type of the parameter value we shall pass to the function.

The single parameter function

The following is an example of a single parameter function:

```
func strLength(theStr:String) -> Int {
  return countElements(theStr)
}
var len = strLength("Hello World")
```

The multiple parameter function

The following is an example of a multiple parameter function:

```
func multiplyBy(num1:Int, num2:Int) -> Int {
  return num1 * num2
}
var res = multiplyBy(7, 4)
```

Named parameters

You might have noticed that we pass parameters to the functions, but as a developer you might want to have named parameters that would be self-explanatory as to what the parameters are. It would not be very clear on the purpose of the parameters passed. To name the parameters, we need to give them a name. These are called *external parameter names*, the *internal names* are used by the function:

```
func appendString(theString first: String, withString second: String)
-> String {
    return "\(first) \(second)"
}

var strRes = appendString("Hello", "World!")
```

The preceding code will give a compiler error because `appendString` was declared to use named functions and it is missing the named arguments of first and second. To be able to call the `appendString` function, we need to use the named parameters:

```
var strRes = appendString(theString:"Hello", withString: "World!")
```

Shorthand external parameter names

Swift has a shorthand method that helps avoid this scenario by simply using a hash symbol (#). This ensures that both the parameters (internal and external) have the same name. The following code explains this:

```
func appendStrings(#theString: String, #withString: String) -> String
{
    return "\(theString) \(withString)"
}
var theBook = appendStrings(theString: "Swift", withString: "Rocks")
```

Optional parameters with default values

For some functions, you can also provide default values that allow you to omit passing that parameter:

```
func replicate(times:Int, theString:String = "-")->String{
  var result = ""
  for i in 1...times {
  result += theString
  }
  return result
}
var dashes = replicate(20)
```

This calls the `replicate` function and passes 20 for `times` and a value of `"-"` as `theString`. If we want to call this function with a different string to replicate, we cannot simply call it as follows:

```
var plusses = replicate(20, "+")
```

This would give an error and to fix it, we need to call it as follows:

```
var plusses = replicate(20, theString:"+")
```

Passing multiple parameters

While we can have functions with no parameters, we could also have functions with single or multiple parameters; but in some cases, there might be a scenario to pass an unknown number of parameters. With a known number of parameters, we can define the number of parameters. With unknown, we would never know if there will be any parameters and if there are; the number of parameters would be unknown. These are called parameter arrays or *variadic parameters*. This is denoted by three dots `...` after the parameter type, as indicated in the following code:

```
func add(numbers:Int...)->Int{
  var result = 0
```

```
    for i in numbers{
      result += i
    }
  return result
}
var total = add(3, 7, 2, 9, 12, 11)        // 44
```

You can have only one variadic parameter in a function and it must be at the last position in the list of parameters.

There are more advanced features associated with functions. This chapter is a quick intro to Swift and hence they are not covered here. Please refer to a Swift book for comprehensive coverage of advanced features such as in-out parameters, constant and variable parameters, function types, and using functions as return types. Apple has a free book called *The Swift Programming Language* that you can download and refer to, among other options.

Using closures

Closures might sound like a fanciful name but in simple terms, they are what Apple introduced as *Blocks* for Objective-C. This is functional code that can be called at a later time. These retain their own scopes for variables and contained functions and can be executed in a different scope than they were created in. These are so called because they form an enclosure over the variables and constants, as shown in the following code:

```
func startFrom(startValue:Int) -> (()->Int) {
  var _startVal = startValue
  func increment() -> Int{
    return ++_starVal
  }
  return increment
}
```

In the previous example, we return a function and if you noticed the return type, it is `(()-> Int)`, which specifies that the return type is a function. This function takes no parameters and returns an `Int` value. First, we save the value passed into a local variable called `_startVal`. Then we return the value while incrementing it using `++_startVal` as shown here:

```
var counter1 = startFrom(1)
var counter2 = startFrom(10)
counter1()                // return  2
counter2()                // return 11
```

The preceding example demonstrates that both of the counter functions contain their own set of variables and are independent of each other.

Objects

Swift has an object-oriented paradigm for you to declare and create classes, structures and enumerations. The primitives such as `Int`, `String`, `Float`, `Double`, to name a few, are structures or structs as they are now called.

Classes

To define a class, we simply use the keyword `class`. A class can contain variables, functions and methods that have initializers to set up the initial values; they can be extended and they can also conform to protocols to provide standard functionality. The following code explains this:

```
class Stooge{
 var name:String = ""
  func getName() -> String{
    return self.name
  }
}
var stooge1 = Stooge()
stooge1.name = "Moe"
println("The name of the first stooge is \(stooge1.getName())")
```

This is a bit cumbersome. What if we could initialize our object with a default name? How about using the following:

```
var stooge2 = Stooge("Larry")
```

We get a compiler error. To be able to use it this way, we need to create initializers using the `init` function. Our `Stooge` class would need modifications, and will now look something like the following code snippet:

```
class Stooge{
    var name: String = ""

    func lengthName() -> Int {
        return countElements(self.name)
    }

    init(forName: String){
        self.name = forName
    }
}
```

However, we still get a compiler error and this time the compiler tells us that it is looking for a named parameter name. So we cannot simply call this as `Stooge("Larry");` instead, we need to call it as `Stooge(forName:"Larry")`.

Swift provides us a way to ignore the external name by adding an underscore before the internal name. Then we can call this without the `forName:` parameter:

```
init(_ forName:String){
  self.name = forName
}
```

And now, we can initialize our new `stooge` variable as follows:

```
var stooge2 = Stooge("Larry")
```

Properties

We created a property called `name` in the previous example, and it can be read and updated directly. Say we add another property in the year of first appearance as:

```
class Stooge{
 var name:String
 var firstAppear:Int

init(_ forName:String){
  self.name = forName
  self.firstAppear = 1900
}
```

```
convenience init(){
  self.init("Unknown")
}

convenience init(_ forName:String, _ year:Int){
  self.init(forName)
  self.firstAppear = year
}

 func lengthName() -> Int {
        return countElements(self.name)
    }
  }
```

We have two `convenience` functions for `init`. The `convenience` functions are defined to allow different function signatures. In our case, we can call the `init` function without any parameters, with just the name or with both, the name and the year. If we did not use the `convenience` keyword, the compiler would complain thinking we are redefining the function with a new signature:

```
var stooge3 = Stooge("Moe", 1934)
```

The properties in this class, `name` and `firstAppear`, are both directly accessible. You could also define setters and getters that could be used to set the values as required. These are useful as calculated properties:

```
  var theYear:Int {
    get {
      return self.firstAppear - 1900
    }
    set {
      self.firstAppear = newValue + 1900
    }
  }
```

We could now use this like a property and it will interact and set/get the value from the `firstAppear` variable:

```
println("\(stooge3.firstAppear)")  // Prints 1934
stooge3.theYear = 67
println("\(stooge3.firstAppear)")  // Prints 1967
```

If you want to run some code *before* a value is set or *after* a value is set, Swift offers the `willSet` and the `didSet` functions like the `get` and `set`.

Structures are similar to classes in most regards whereby they can have properties, functions, initializers, and so on. One of the differences between structures and classes in Swift is that structures are copied and do not use reference counting.

Enumerations

Enumerations are data types that have named values and behave like constants. They too, like structures and classes, have properties and functions associated including initializer functions:

```
enum eStooges:Int{
  case Moe = 1
  case Larry
  case Curly
  case Shemp
  case Joe
}
var aStooge = eStooges.Moe
```

You could add a function to this like `toString`, which returns the name of `Stooge` as a string:

```
enum eStooges:Int{
  case Moe = 1
  case Larry
  case Curly
  case Shemp
  case Joe

  func toString()->String{
    switch self{
     case .Moe:
      return "Moe"
     case .Larry:
      return "Larry"
     case .Curly:
      return "Curly"
```

```
      case .Shemp:
       return "Shemp"
      case .Joe:
       return "Joe"
     }
   }
 }
 var aStooge = eStooges.Moe
 println(aStooge.toString())    // Moe
```

You can also use the enumerations without the enumeration name directly with the member name, like this:

```
var theStooge: eStooges = .Curly
println("\(theStooge.toString())")      // Curly
println("\(theStooge.rawValue)")        // 3
```

When using enumerations, it will always number them starting from 0, however, if you want to change that start index, you can define the first value and the rest will be assigned a sequential value. You can also assign values to each of the members or they are assigned the next number (that is, sequential).

Extensions – extending the classes

This is one of the most interesting language features in Swift. It allows adding of functions and features to an existing class. We can extend an `Int` variable and add an extension called "square" that returns the square of the integer:

```
extension Int{
  var square:Int {
    return self * self
  }
}
```

You can now type `4.square` in the Playground and the result `16` will show.

We can create another extension for strings called `reverse`:

```
extension String {
  var reverse:String{
    var result:String = ""
    for i in self {
      result = String(i) + result
    }
```

```
        return result
    }
}
"Hello".reverse        // shows olleH
```

Operator overloading

We can use most of the basic operators to perform basic operations, as you have seen in the previous examples. With Swift, we can also re-declare these operators to perform custom operations. Some of the basic operators can be overridden to provide additional functionality.

There are three positions for operators, `prefix`, `postfix`, and `infix`. They are defined depending on the position of where the operator lies.

The prefix operator

Prefix operators are those that have the operator before the expression, such as `++a` or `!b`. Here's an example of using the square root symbol operator (√) and making it function as expected:

```
prefix operator √ {}
prefix func √ (theNumber:Double) -> Double {
  return sqrt(theNumber)
}
var sqRoot1 = √144         // 12.0
var sqRoot2 = √16          // 4.0
```

The postfix operator

Postfix operators appear at the end of the expression, such as `a++`. Here is an example of using the `%` postfix operator that returns the percentage:

```
postfix operator %{}
postfix func %(theNumber: Double) -> Double {
    return theNumber / 100
}

var rads1 = 60%         // 0.6
var rads2 = 152%        // 1.52
```

The infix operator

The infix operator appears in between two expressions, such as the equality operator `==` or `+=`, or `>>`, `<<` and the bitwise operators `&&`, `||`,. Let us create a custom infix operator that replicates a string as many times as the number following it:

```
infix operator ** {}
func ** (theString:String, times:Int) -> String{
  var res = theString
  for i in 1..<times {
    res += theString
  }
  return res
}
var buffer = "Hello" ** 5    //HelloHelloHelloHelloHello
```

Summary

We have tried to cover as much of Swift to get started with coding as possible. A single chapter would not do justice to cover the breadth of this topic. We saw that we can use the `println` function to output our data, and while in the Playground, the values for all variables are seen in the editor space. In the next chapter, we will take a break from code and look at creating interfaces visually in Interface Builder. If you want to learn more about Swift, Apple has a blog and lots of resources including some free iBooks on Swift. One thing to note is that Swift is evolving, and in the time since Swift was announced at WWDC to the time it was released in September, the language and the syntax has undergone many changes. The possibility of it changing in the near future cannot be ruled out. So be aware of this fact when working with Swift.

4
Interface Builder

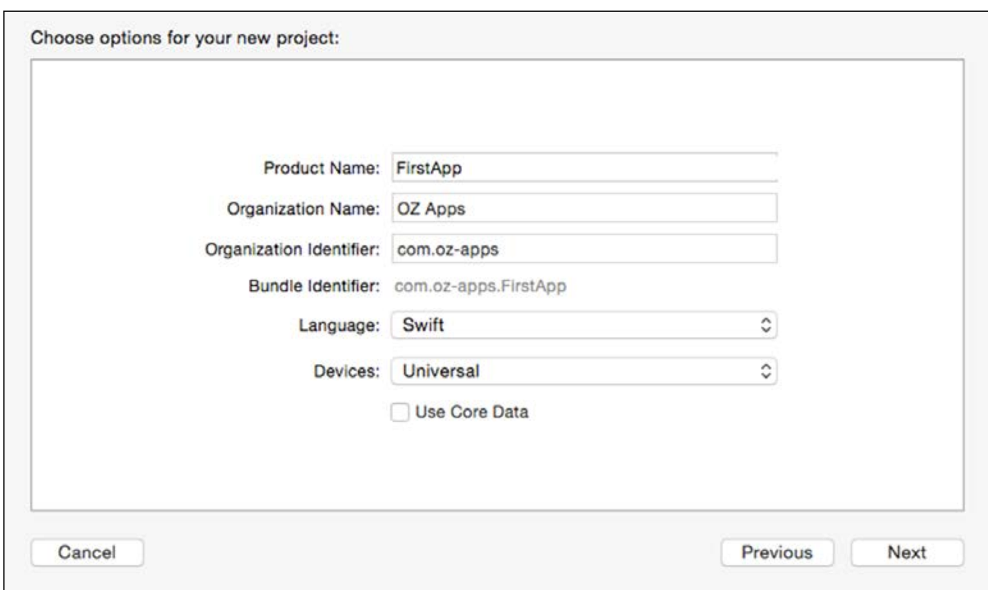
In this chapter, we will cover the following topics:

- Interface Builder – the storyboard
- Creating a mockup
- Adaptive layouts
- Connecting code

In the last chapter, you learned about Swift, the alternative programming language. There is an increasing pressure on developers to make a pretty-looking application; the designers deliver a design and it is mostly up to the developers to convert this design into an application. Earlier, an external component of Xcode, **Interface Builder** (**IB**) now is an integrated part of Xcode. Interface Builder allowed you to visually layout the UI of the application in forms/screens that were saved as XIBs. With iOS5 and Xcode 4.3, Apple introduced storyboards. Storyboards were generally used in the movie and animation industry, where sequences of images were used to previsualize the scene or flow of the movie. The concept remains and a storyboard in Interface Builder also allows you to previsualize the different scenes in your project including the interaction between them. The different scenes in a storyboard are called views and the transitions between the two scenes linking them is called a *segue*. Segues can be drawn between the UI elements in a scene to the destination scene.

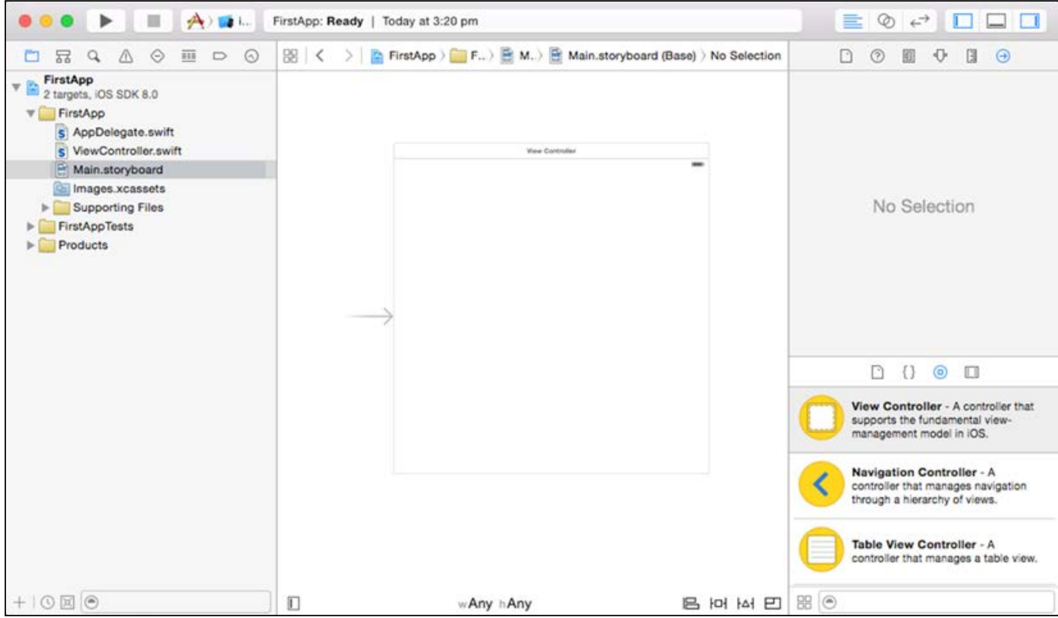
Introducing Interface Builder

Interface Builder has now been a part of Xcode for a while and can be used as you can use the code editor. Create a new project with a storyboard to create a simple UI. To start, create a new project (**File** | **New** | **Project**) and create an iOS project:



Creating a basic interface

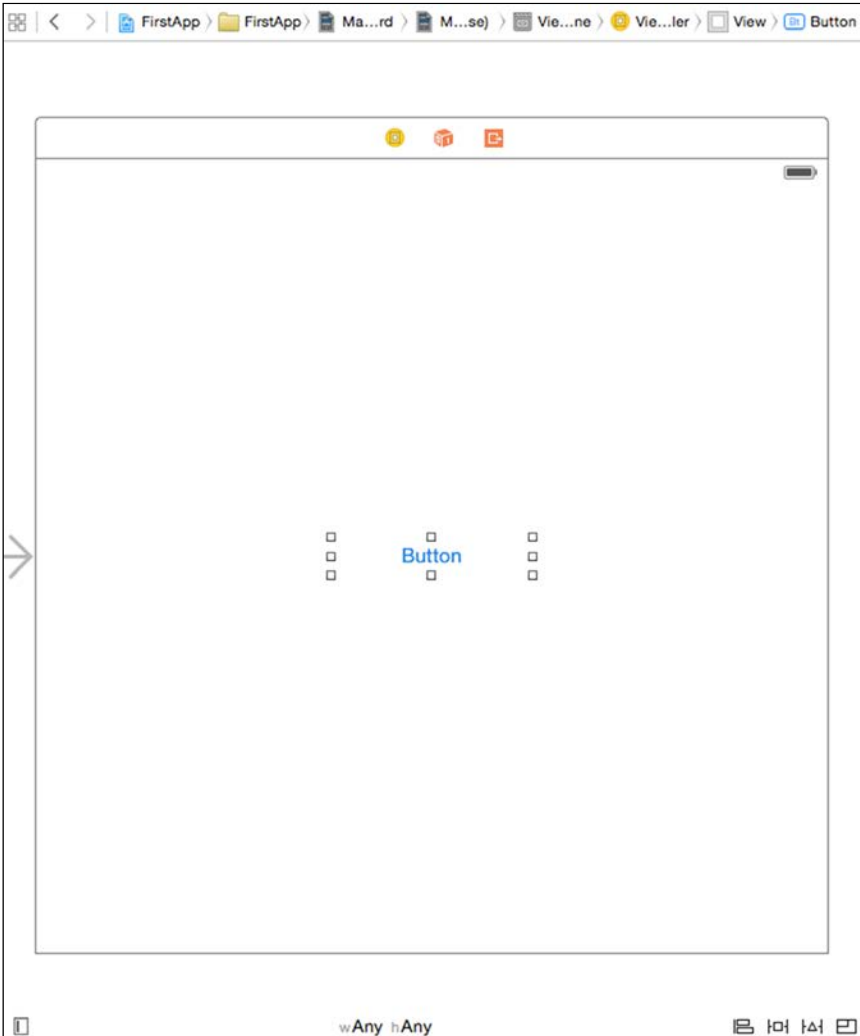
To work with the storyboard UI, first click on the `Main.storyboard` option to open the storyboard in the editor for you to work with. This area is called the canvas; this is where you drag-and-drop your elements:



Now you can place items onto the canvas. You can drag-and-drop elements from the library, scroll down to a button object, and drag and drop it onto the canvas. Use the blue guidelines to align it right at the center of the view.

Elements on the view

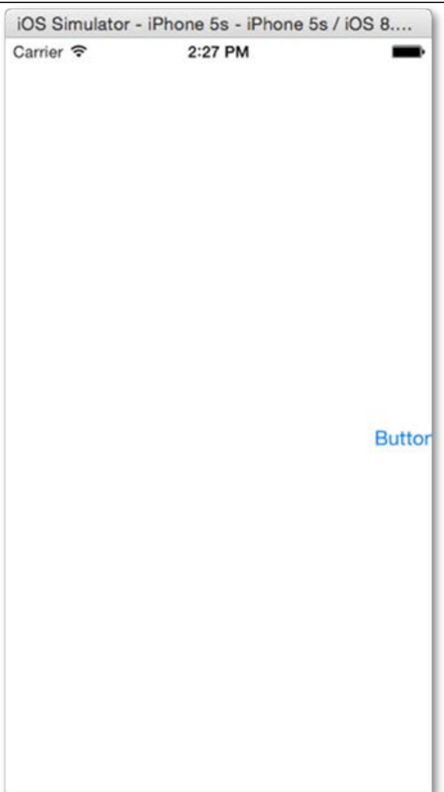
On top of the scene, there are three icons: the first yellow and the next two orange. These are the View Controller, First Responder, and Exit as can be seen in the following screenshot:



The following is the description of the three icons:

- The View Controller element is the object that handles the interface events and it is also the `UIViewController` class. You can change it to make it manage your custom class.
- The First Responder element is a placeholder that informs the application that it should be the first one to handle the events. This is mainly useful for `UITextViews` and other controls that require a first responder to be set.
- The Exit element is used to connect code that is called when exiting a segue scene.

You can simply press *CMD+R* or click on the Run button. You will see it running in the simulator, as shown in the following screenshot:



You might wonder why the button is being cut off even when you centered it and aligned it. The main reason for this is that Interface Builder sets the size of the view to 600 x 600 by default. When this is run and positioned on an iPhone-sized screen, the elements are culled or clipped and hence half the button is shown in the preceding screenshot. When you switch off the size class, the sizes are retained for either the iPhone or the iPad screen size and thereby they remain as you might have laid out your elements. You can switch off the size class from your storyboard by navigating to the first tab of **Utilities Navigator** (*OPT* + *CMD* + *1*) under **Interface Builder Document** and deselecting **Use Size Classes**. With size classes, you can create a single view that adapts based on the device and the orientation it is run on. You can also switch off **Autolayout** for the project.

Adaptive UI

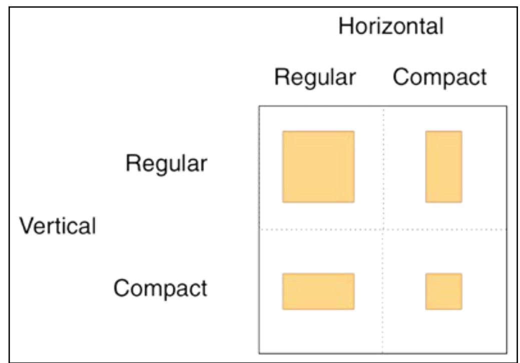
Adaptive UI is quite exciting, especially since it allows defining how the UI should adapt to different orientations and resolutions. The first thing that Apple has done to make this possible is to do away with `UIInterfaceOrientation` and `UIDeviceIdiom` and replace them with the size class. Earlier, your code could have been littered with a lot of conditional code determining the orientation and the device to run code. Obviously, you would not have done so with Swift as it is a new feature with Xcode 6, but this is a sample of what it would look like with Objective-C:

```
UIDevice *device = [UIDevice currentDevice];
UIDeviceOrientation currentOrientation = device.orientation;
BOOL isPhone = (device.userInterfaceIdiom ==
UIUserInterfaceIdiomPhone);
BOOL isPortrait = ([[UIScreen mainScreen] bounds].size.height ==
568.0);
```

This would then be followed up with a series of conditions such as:

```
if (isPhone == YES) {
    if (isPortrait == YES) {
        // Do something when in Portrait orientation
    } else {
        // Do something when in Landscape orientation
    }
else
    {
        //You could check for iPad and its orientations
    }
```

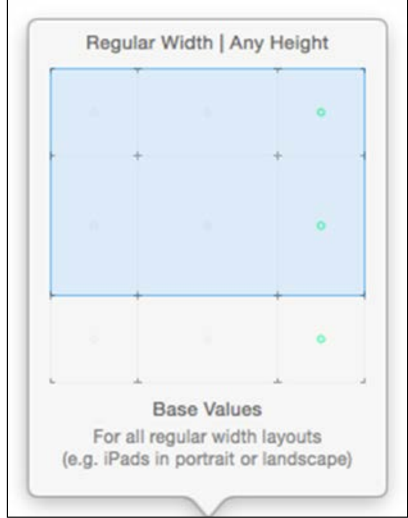
This is quite cumbersome. Instead, the size class offers three options on each of the dimensions: width and height. The size options are `Compact` and `Regular` and the third option is `Any`, which is basically a way of saying it could be either.



Xcode has nine values available for creating your UI; this includes the third option `Any` that we mentioned earlier. In the following figure, you can see that this allows you to adapt to more UI's easily, so you do not have to explicitly check for sizes any more; it is all about the space available. The sizes as displayed by 1, 3, 7, and 9 are final values, specifically for the desired orientations, whereas 2, 4, 5, 6, and 8 are base values. Final values are more like fixed values that conform to a particular device size, whereas the base values are those that can adapt to or be used for more than one device and/or orientation. Constraints created in the `Any` | `Any` layout are applied to all of the layouts, while constraints created for a specific layout are applied for that layout only.

	Compact	Any	Regular
Compact	1	2	3
Any	4	5	6
Regular	7	8	9

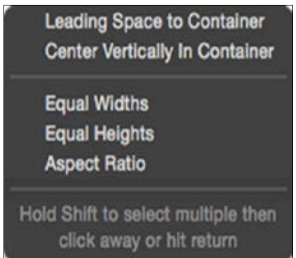
If you look at the bottom of the Interface Builder screen, you can see `wAny` and `hAny`, if you click on that, it will show a pop-up as shown in the following screenshot. This allows you to select the size you want to create your UI at:



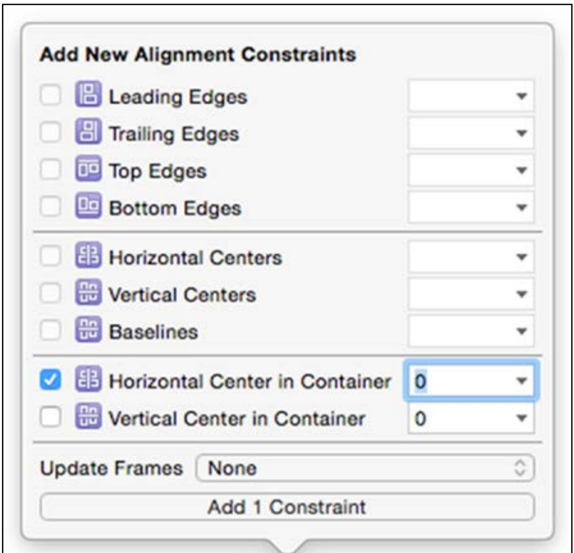
For now, we will work with the `Any | Any` layouts. This is the universal layout that would work on most devices.

The scene size in Interface Builder can be adjustable (in case you need to) by first setting **Simulated Size** to **Freeform** in **Size Inspector** (*OPT + CMD + 5*) after you select the view. Then, set the width and height to the desired size.

The first task is to ensure that the button remains in the center as we expected it to be. To do that, we need to add constraints. Constraints are not new; they were introduced with iOS 6 and Auto Layout. This allows you to specify how the elements would be displayed on the view, and how they will be displayed when viewed on different devices. To ensure that the button stays in the center of the screen, you can add two constraints, one to center it horizontally and another to center it vertically. To do that, press Control and then click and drag from the button towards the left and release; it displays a pop-up that allows you to center vertically in the container.



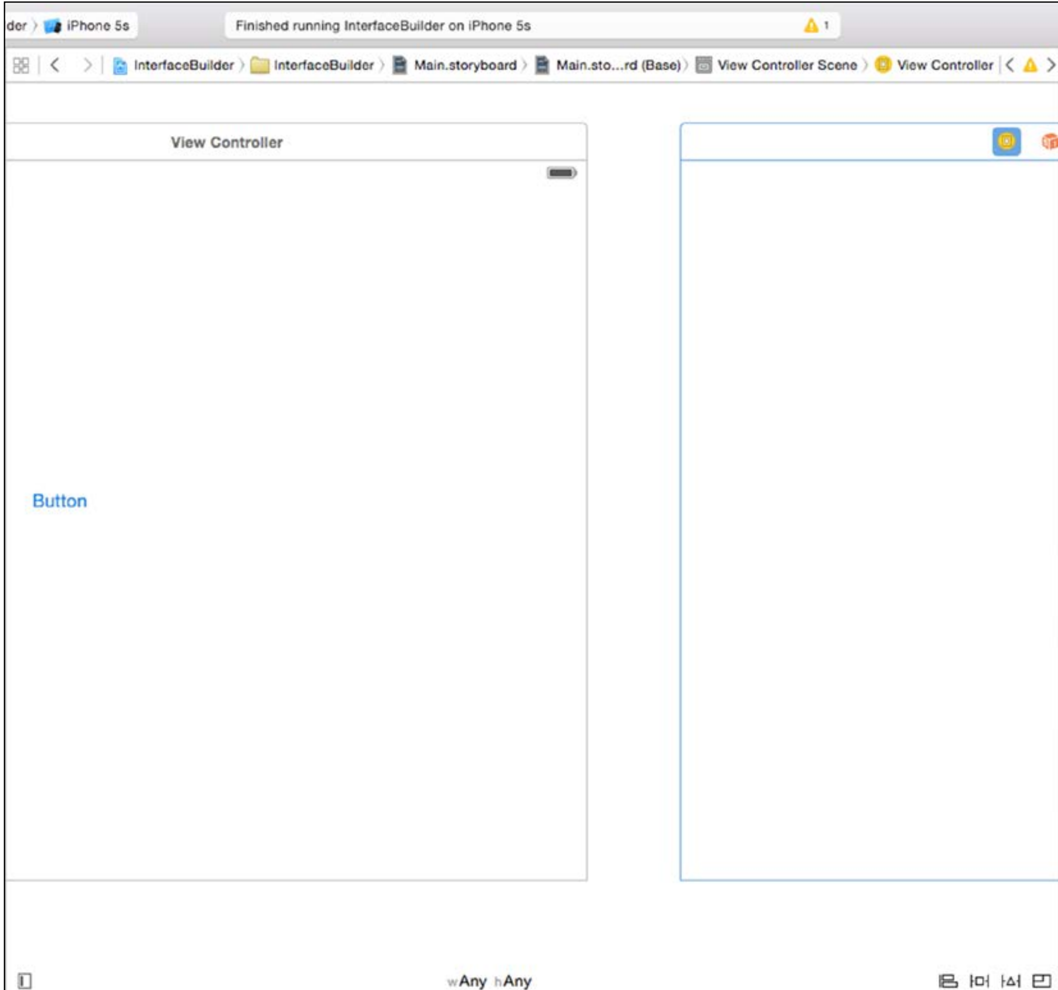
Alternatively, you can also click on the Align button and select the alignment options:



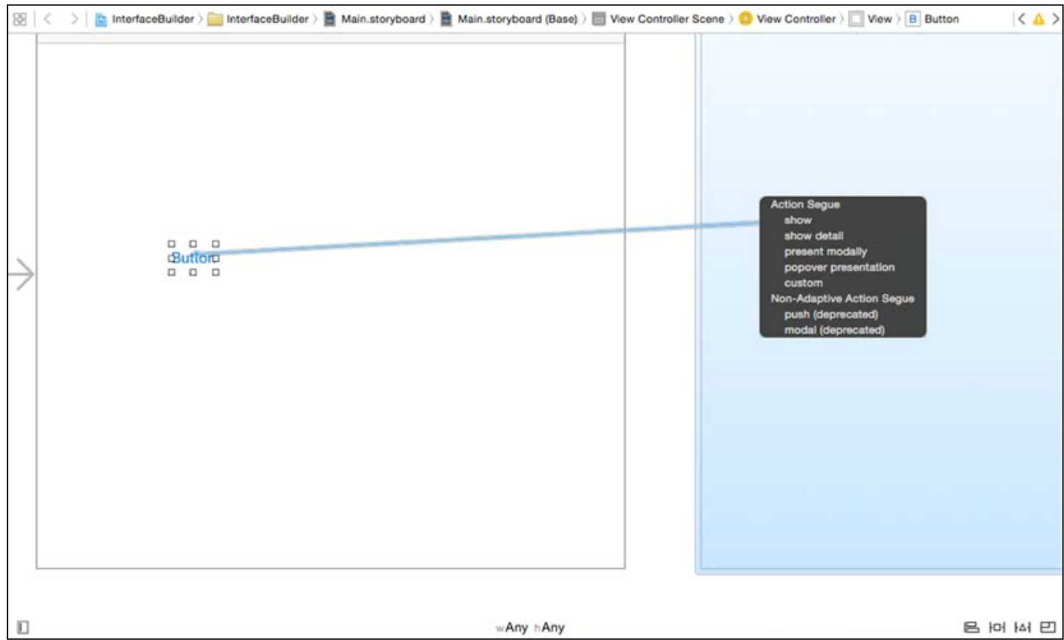
Now, when you run the project again, you will see the button display in the center as expected to start with.

Adding scenes

If you click on the button that we just created, nothing happens; that is OK because we have not specified anything for it to do. Let's look at creating a mock-up without writing a single line of code. The first thing you need is another view that will be displayed when you click on the button. Drag-and-drop **View Controller** from the library and position it next to the existing one.



Now press *control* and click and drag from the button to the new View Controller that you just added to the storyboard. This displays a pop-up option to select the method to use when displaying the new View Controller; select **show**:



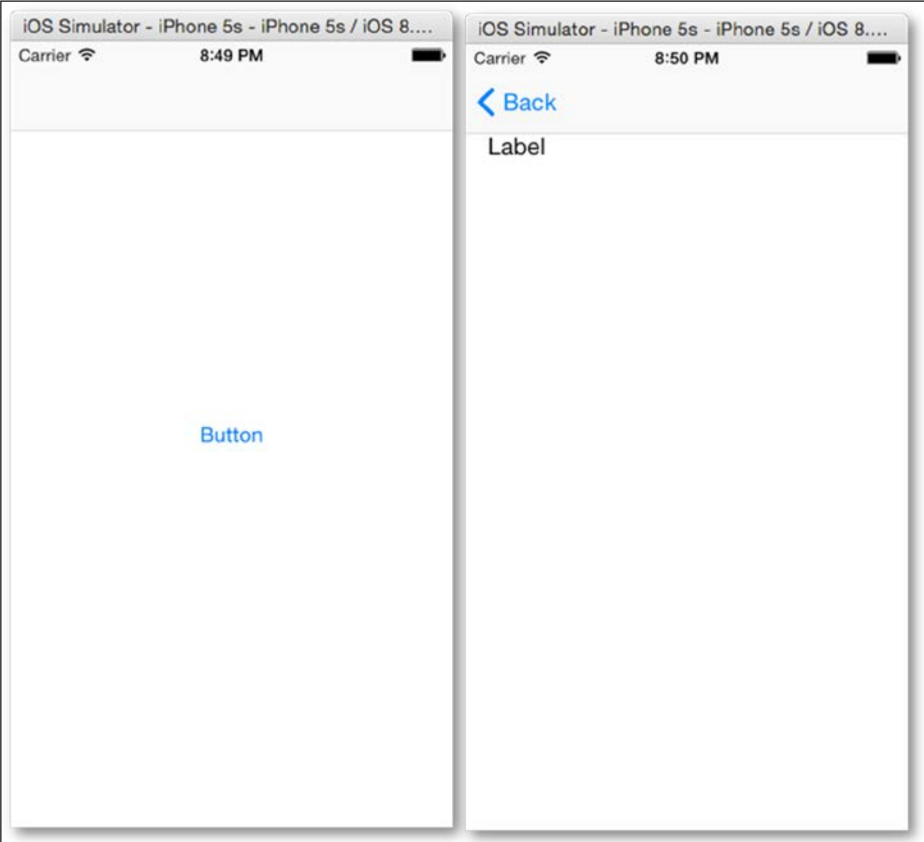
Let us quickly add a label to the new view and set the constraints to the top-left corner. Now, if you run the application and click on the button, it will display the second view. There is no way to go back to the previous view. We do not see the back button that you would have expected from many iPhone applications that navigate between views.

Navigating between View Controllers

To get the View Controller that you would have expected along with the back button, you need `NavigationViewController`. An easy way to add `NavigationViewController` is to select the root View Controller (the root View Controller can be identified as the one with the half arrow that is not connected from any other View Controller) and navigate to the menu option **Editor** | **Embed In** | **Navigation Controller**. You will notice that the root View Controller changes to new `navigationViewController`.

To change any View Controller to be the root View Controller, look for the settings in the Attributes Inspector (*OPT* + *CMD* + *4*); under the View Controller, select the **Initial View Controller** checkbox.

Now, when you run the project, you will notice that the view has space for a navigation bar in Interface Builder. Run the project again and you will notice the top navigation bar space in gray; and when you click on **Button**, the new presented view also has the **Back** button that works as expected without doing anything much.



You can add more UI elements and scenes that can be connected with segues.

Building a simple application

Start a new project of the **Single View Application** type, name it as `fixedDataTable`. Click on the `Main.storyboard` option to open it in the editor. Drag **Table View Controller** from **Utilities Library** and arrange it to the left of the view. Delete the main View Controller; now select **Table View Controller** and check the **Is Initial View Controller** checkbox. This will make **Table View Controller** the main controller. Run the project and you will see **Table View** blank for now.

Adding items to Table View

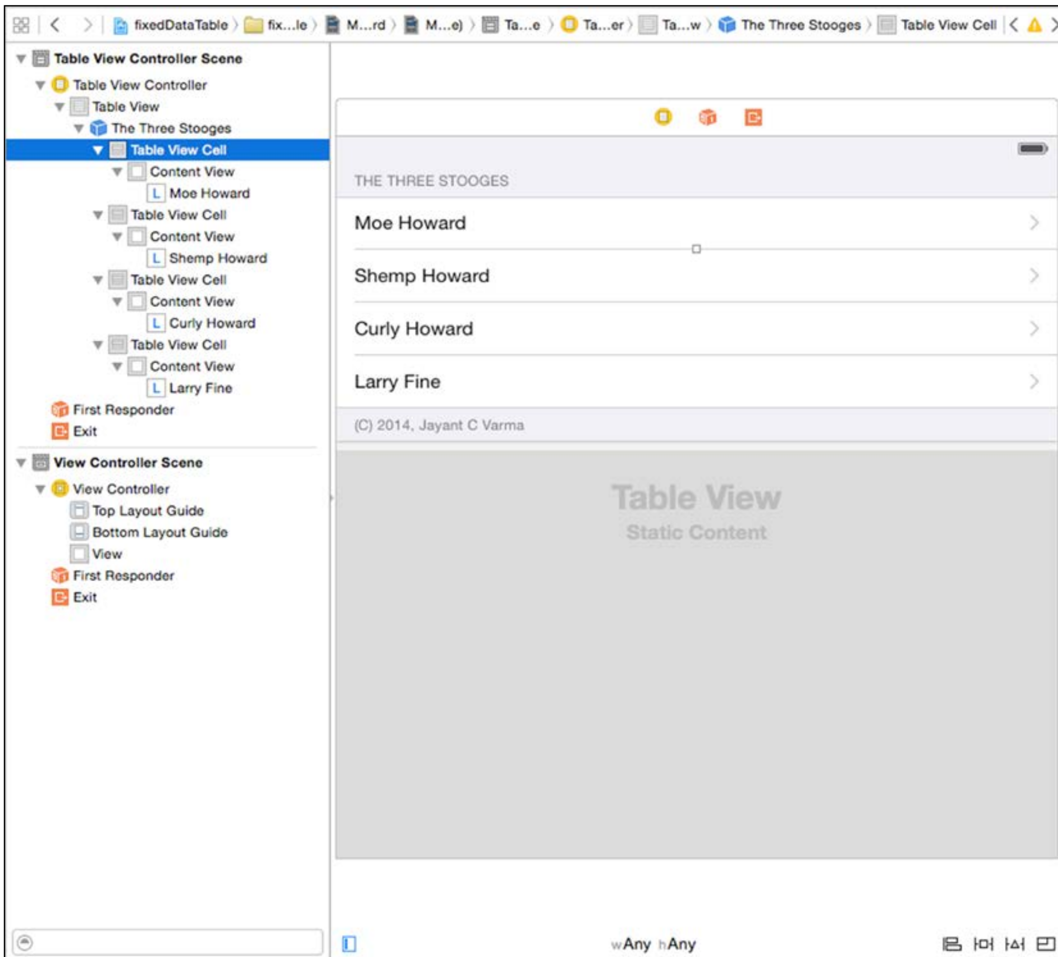
To be able to access the various parts of **Table View**, such as the cells and the content, you can activate the document outline that allows you to access each element more accurately than by simply clicking on them.

Click on **Table View** and, under the **Attributes Inspector** tab, change the value of **Prototype Cells** to `4`. This determines the number of items displayed in **Table View**. Change the **Content** option to **Static Cells** from **Dynamic** properties. Change the style to **Grouped** from **Plain**. Next, click on each of the **Table View Cell** options and change the type to **Basic** from **Custom**. Now, select the **Table View** section. ensure that the value of rows is `4,` and add **Header** and **Footer** text.

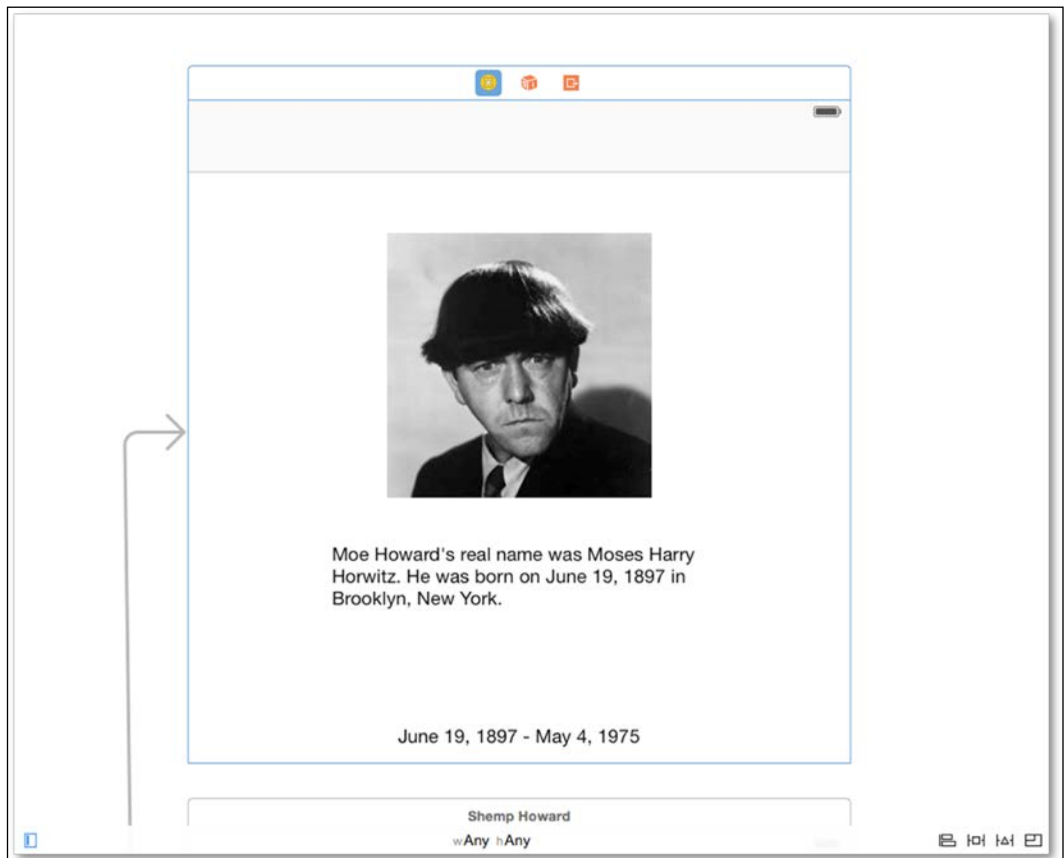
Click on each of the **Table View Cell** options and, under Accessory, change the value to **Disclosure Indicator** from **None**. This will display the `>` indicating that, when the tableCellView is clicked, it will display some other detail.

Now, double-click on each of the cells to edit and set its text. We are making a *Stooge-o-dex* so we have the names of the four stooges. The show was called the *Three stooges;* however, there were as many as six stooges but that's beside the point here. We will have information about four stooges in our sample application.

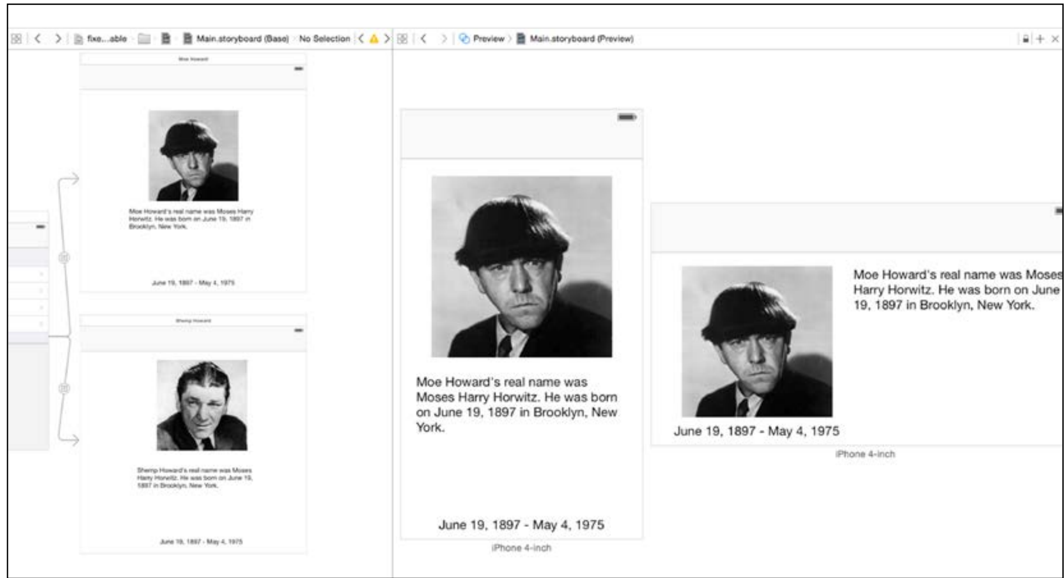
Name the cells as `Moe Howard`, `Shemp Howard`, `Larry Fine`, and `Curly Howard`:



Add four View Controllers, add an image view, and add two labels on each of them. The image will hold the picture of the stooge and the two labels will display some information about the stooges. Position the image view to be centered horizontally and towards the top of the view. Position the first label just below the image view and position the last label at the bottom of the view. Refer to the following screenshot. In the **Attributes Inspector** tab, name the View Controllers as per the names of the stooges. Add the four images (you can get these from the companion files download) to your project; then select the image view and set its image property from the **Attributes** navigator. Type relevant text into the two labels. Set your constraints to have them display as per the following screenshot:

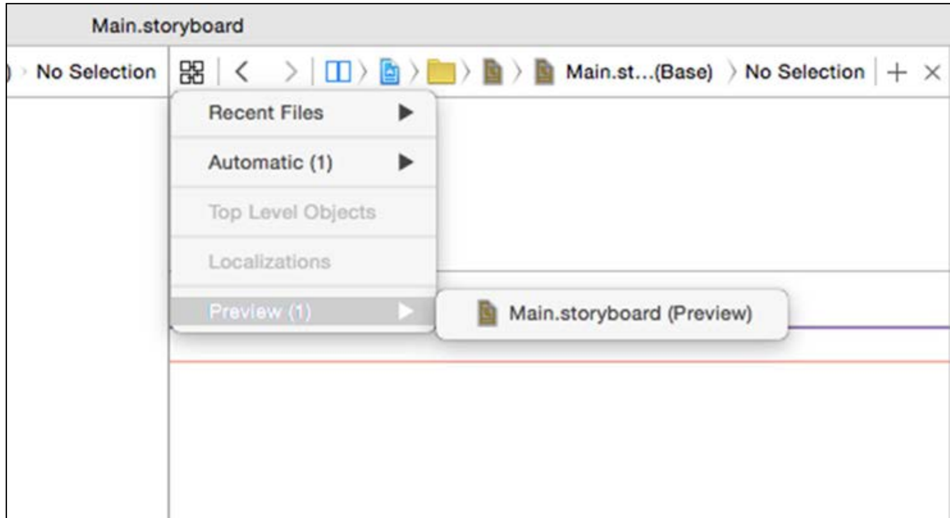


Next, press Control and click-and-drag from each of **Table View Cells** to the View Controller to connect them with a segue. Select *show* when asked how you would like to display the View Controller. After you are done connecting **Table View Cells** to the View Controllers, you should have a storyboard with multiple arrows connecting the scenes, as seen in the following screenshot:



A storyboard with multiple arrows connecting different scenes

Open the assistant editor (press *OPT* + *CMD* + *Enter*) and then select the storyboard. Next, click on the first icon on the left; it looks like a set of 4 squares. Scroll down to the **Preview** option and select it. This displays a preview of the view. If you do not get the Preview option, but see a code file in the assistant editor, select the storyboard from the list of recent files (in the assistant editor), and then select the **Preview** option again.



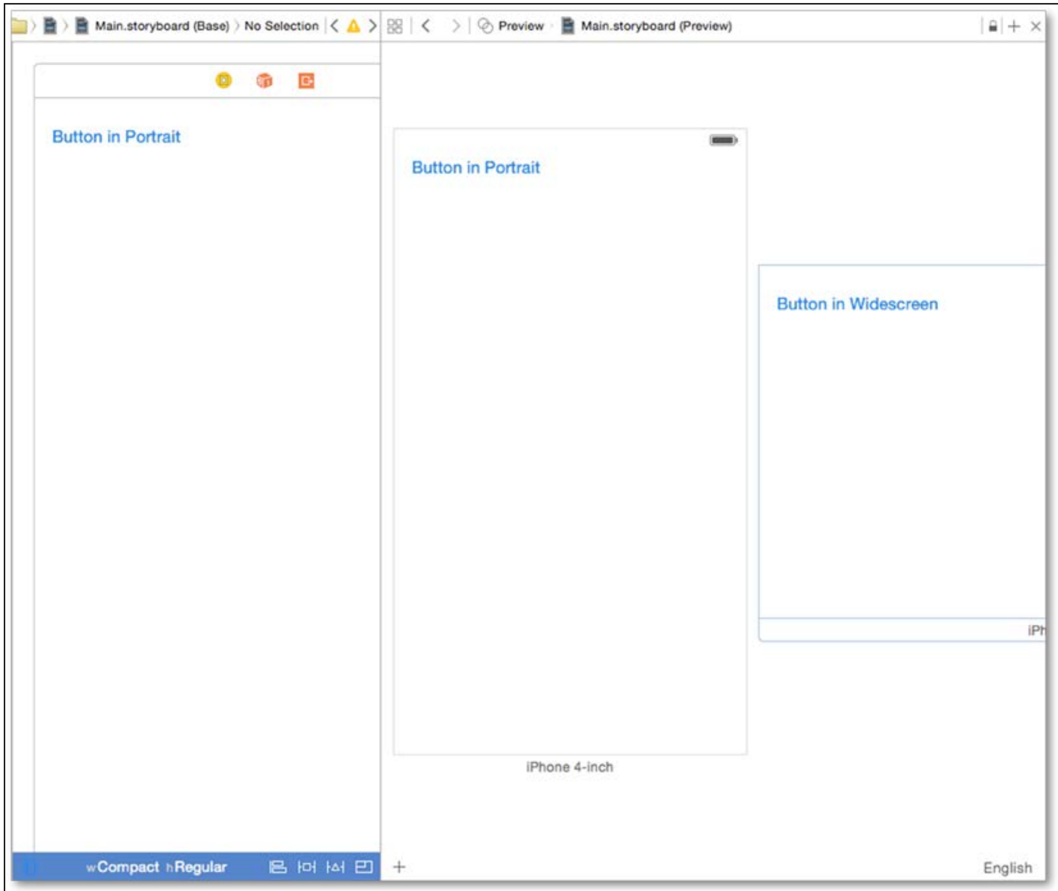
> You can quickly open the storyboard in the main editor by clicking on it. However, if you press the option key and click on the file, it will open it in the assistant editor.

Click on the plus button (**+**) at the bottom; it offers three choices: the iPhone 3.5, iPhone 4.0, and iPad. When you select a particular device, it is added to the preview and the view in the main editor is displayed as adapted for that particular device. It is very useful to preview your work live on various-sized devices. You can add two of the same and keep them in different orientations, as shown in the previous screenshot. You can see how the View Controller for Moe Howard in both portrait and landscape is adapted and displayed based on the view and constraints. This would speed up the process to create adaptive UIs. In addition to this, designers can create the UIs with Xcode and add basic functionality without writing a single line of code. This book does not go into details of auto layout and constraints as that is a topic that might require a chapter or more by itself. It could be used to create pixel-perfect and aligned UIs as your designer might expect while designing it in some image manipulation software.

Once you have populated the screens for all four of the View Controllers, you are all set. You have just created a static application without writing a single line of code.

Different elements for different orientations and dimensions

You can also select which controls are seen in a particular orientation, work with the `Any | Any` option for the common elements, then switch to the desired orientation or size; for example, navigate to `Compact | Any` for widescreen layouts on an iPhone device. Rearrange or add the component you want for that orientation.



You might notice in the preceding screenshot that the two previews have different buttons depending on the orientation they are displayed in.

Adding the code

While it is easy to create UIs in Interface Builder, there is only so much that you can do. If you want to add a button that displays another view, it is rather difficult to do that in Interface Builder.

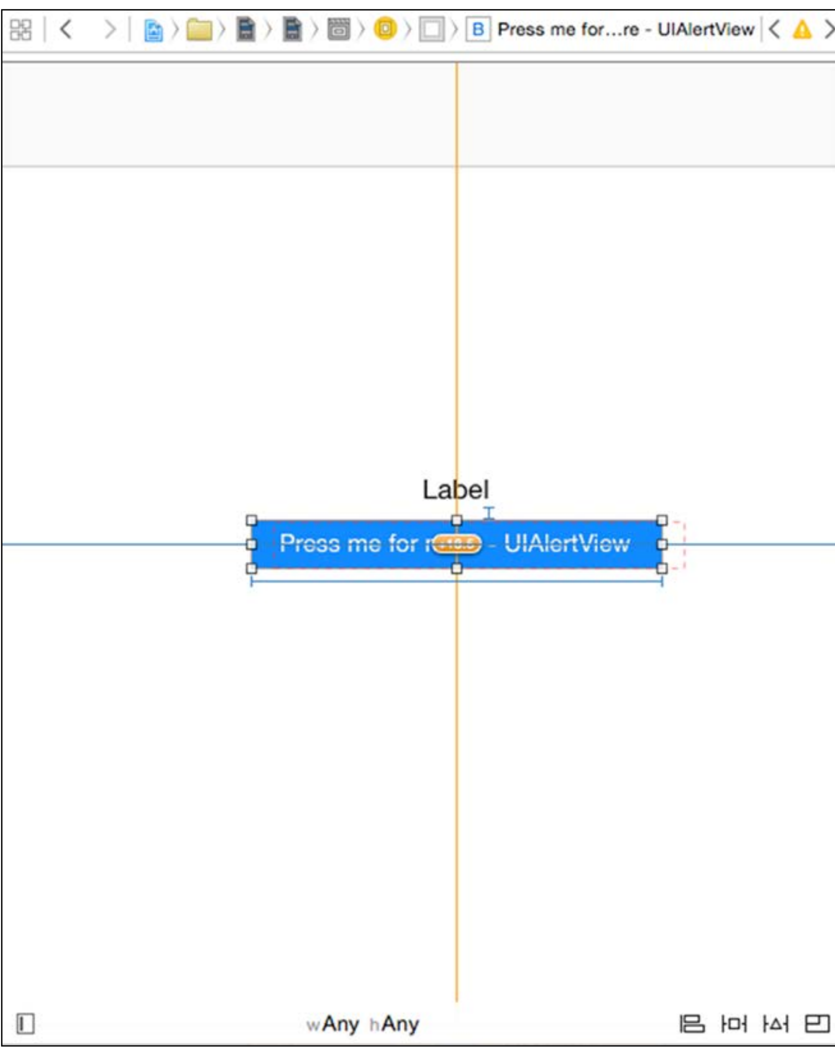
Before we start writing some code, there are a couple of things that we need to know about Interface Builder and Xcode:

- Interface Builder follows the **Model-View-Controller** (**MVC**) model. This, in simple terms, means that there is a model that holds the data, a view that you see on the screen that displays the data, and a controller that liaises between the view and the controller.
- You can create a custom class of the `UIViewController` type that allows you to work with the View Controller—that is, add code that can be executed.
- You need to make connections between the elements on a View Controller in Interface Builder and the code via outlets. These are similar to variables but are connections to your UI elements, thereby allowing you to refer to them from code.

If you are new to Xcode, you would want to take care of what you connect from. This is because there are differences between, say, connecting an element to the first responder and connecting the first responder to an element.

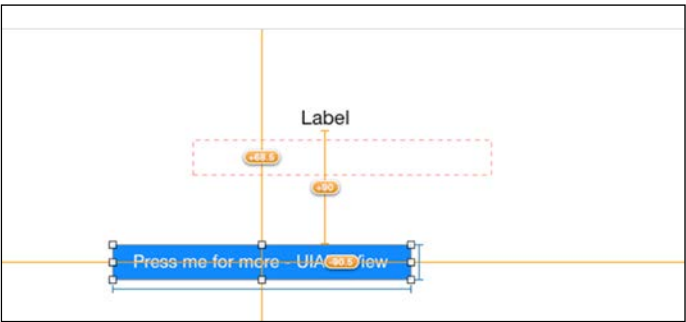
1. Simply add a new row to the existing **Table View** and name it `counter`.
2. Add a new View Controller to the storyboard and connect the table cell to the View Controller by pressing control and dragging from the table cell to the View Controller.
3. Next, add a button, position it in the center of the screen, and set the appropriate constraints, aligning the button centrally in the container, both horizontally and vertically.
4. Double-click to change the title of the button and type in `Press me for more – UIAlertView`.
5. You can also change the **TextColor** color to **White** and the **Background** color to **Blue**. This would give it contrast while on the screen.

6. Next, add a label above the button and set its constraints accordingly.

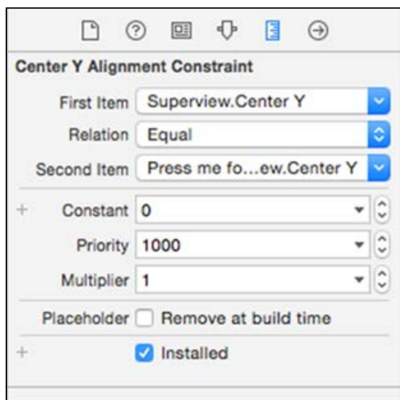


Autolayout

This book will not delve into autolayout, but a quick look at it would help you to understand a bit more about autolayouts. In the following screenshot, the button has its constraints set to position it in the center, both horizontally and vertically. There is another constraint that sets the vertical spacing between the button and the label. There are another two constraints: the width and the height. Interface Builder shows you where the element would be shown at runtime as seen via the orange dotted lines. In fact, it would even show you inconsistencies and missing constraints, including if any of them are ambiguous or conflict with others. It even tells you that the frame will be different at runtime.

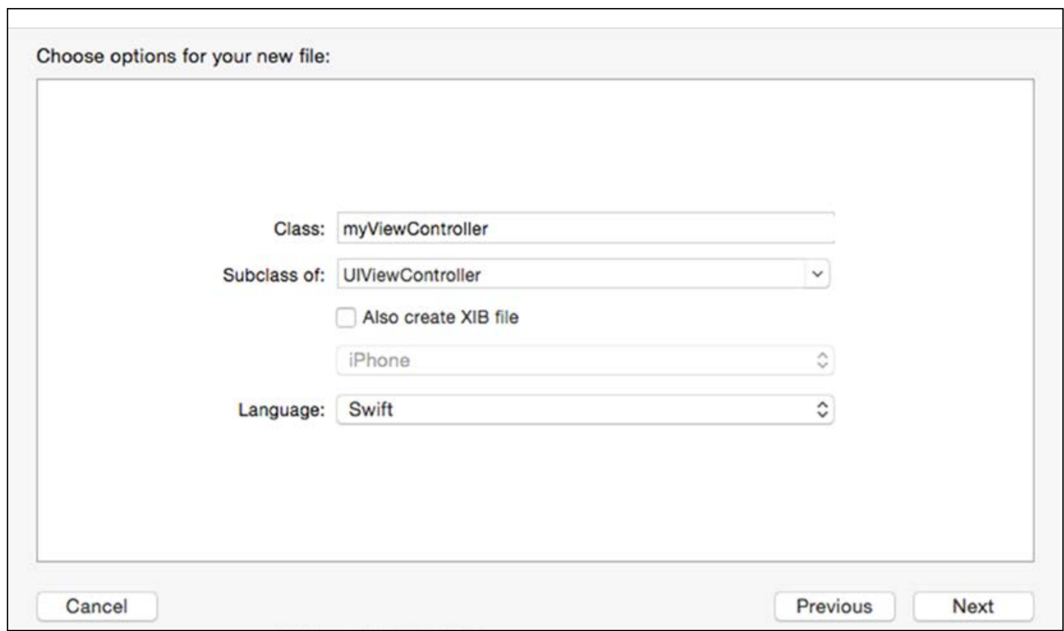


You can also individually click on each of the constraints like any other object in Interface Builder and it will show up in **Properties Inspector**, which you can tweak as required.



Subclassing

You have the simple UI that you need (also referred to as the view). To create the controller, we navigate to **File** | **New** | **File**. Under **iOS Source**, select **Cocoa Touch Classes**, name this `myViewController` and select it as **Subclass of** `UIViewController`. Do not select **Also create XIB file** as we are using storyboards, and choose the language as **Swift**.



To create the connections, open the storyboard in the editor and position the view, in **Identity Inspector**, change the class from `UIViewController` to `myViewController`. Now, if you open the assistant editor (by pressing *OPT* + *CMD* + *Enter*) it will open the `myViewController.swift` code in the assistant editor. Now press *control* and drag from the label to the code at the top just after the class name declaration, and select to create an outlet connection, name it `theText`, and ensure that storage is of type weak.

This would create the code:

```
@IBOutlet weak var theText: UILabel!
```

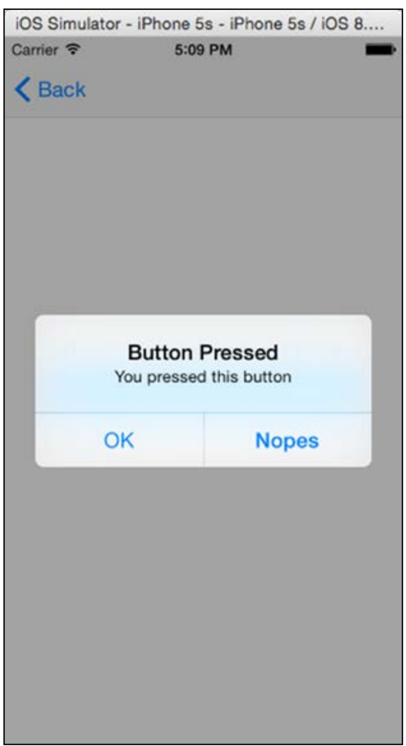
Next, press *control* and drag from the button to the code and create a connection of type action, name it `pressMe`. This would generate the following lines of code:

```
@IBAction func pressMe(sender: AnyObject){
}
```

You need to supply the code for that action. In this case, to create `UIAertViewController`, type the following code between the curly braces:

```
let alert = UIAlertController(title: "Button Pressed", message: "You
pressed this button", preferredStyle: .Alert)
let actionOK = UIAlertAction(title: "OK", style: .Default, handler:
nil)
let actionNo = UIAlertAction(title:"Nopes", style: .Cancel, handler:
nil)
alert.addAction(actionOK)
alert.addAction(actionNo)
self.presentViewController(alert, animated: true, completion: nil)
```

This would create an alert as expected when the button is pressed:



To be able to process the alert button, that is, handle the button pressed, you need to add a function block to the action's handler. We had specified it as nil previously. For now, we will simply display a message to the Xcode console window using `println`. Modify `UIAlertAction` as follows:

```
let actionOK = UIAlertAction(title:"OK", style:.Default, handler: {_in
println("We clicked OK"})
```

If you are a keyboard ninja and work with the keyboard and find that the mouse/trackpad slows you down, you can switch between the editor and the assistant editor or other elements by quickly pressing *CMD* + *J* which will pop up the elements for you to choose from.

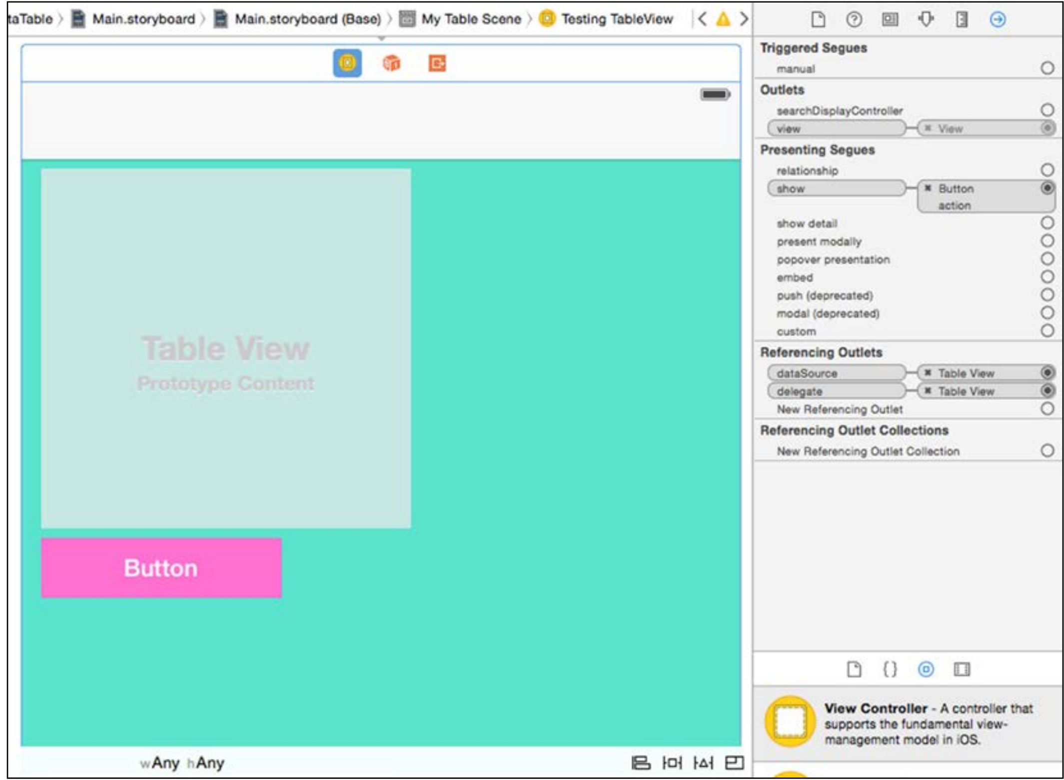
Quick help

When you are writing code, there are times when you are unsure of the methods or properties; Xcode offers a quick and easy method to lookup the words. If you keep the option key pressed and hover over a class, it underlines the word with a dotted line and the cursor changes to a question mark, this will display a pop up with the details of the word under the cursor. If it is a function or a method it displays its details:

If you have the *command* key pressed, the cursor changes to a hand cursor and the word under the cursor changes to a solid line. Then, if you click on the word, it would open the corresponding header declaration and even display the declares (from the system file headers) and so on as their Swift equivalents. If the class is your own custom class, then the source code file is displayed.

Managing connections

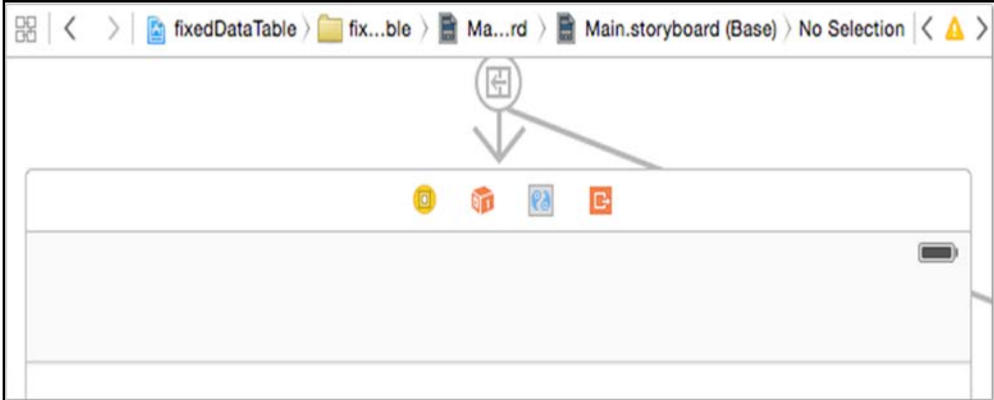
Connections created using Interface Builder can end up with multiple connections if you create, change elements and maybe recreate the connections. Most importantly, if these connections are invalid or changed, they may cause crashes. It is best to check the connections and removing them if they are no longer required. To bring up the connections, you can right-click on the object and it will display a list of items; it will also show the connections (if any). Another way to look at the connections is in **Connections Inspector** (*OPT* + *CMD* + *6*) under **Utilities Inspector**.



Connections Inspector

Adding gesture recognizers

Once you start to connect code and views in storyboards, you can do a lot more than display simple alert boxes. You can attach gesture recognizers to make your UI interact with gestures. Continuing with the example project, you can add a rotation gesture recognizer and then rotate the button, which looks really cool and still works like a button.

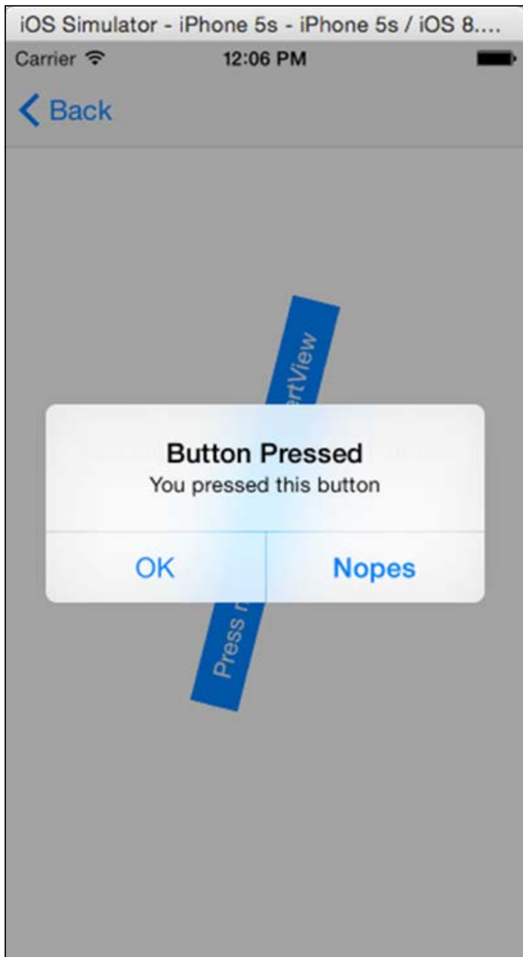


First, look up for **Rotation Gesture Recognizer** from **Object Library** and drag it to the view. Notice that this does not show up on the view; it sits on top of it. It looks like a yin-yang symbol. Then, set up the gesture recognizer by connecting it. Press *Control* and drag from the button to the gesture recognizer; from the pop-up menu outlet collections, select `gestureRecognizers`. Next, open up `myViewController.swift` in the assistant editor; press *control* and drag from the gesture recognizer to the editor and add a new `IBAction` called `rotateMe`. This creates a blank function with the name `rotateMe` in `myViewController.swift`. You can add the following code to this function:

```
@IBAction func rotateMe(sender:AnyObject){
    let gesture = sender as UIRotationGestureController
    let transform = gesture.view?.transform as CGAffineTransform!
gesture.view?.transform = CGAffineTransformRotate(transform, gesture.
rotation)
    gesture.rotation = 0
}
```

First of all, the parameter passed to this function is of the `AnyObject` type; we could have instead specified that we need this to be `UIRotationGestureController`. The code simply changes the rotation of the view by using `CGAffineTransformationRotate` and then sets the rotation as `0`, so it continues from that point. Otherwise, it goes a bit crazy with the rotation adding up. You can comment the `gesture.rotation = 0` line and have a look. Alternatively, Apple offers you something called the CGAffineTransformIdentity. This is basically the default position with no transformations applied, no rotations, no scaling, etc. So you could use this as the starting point to apply your transforms.

If you are in the simulator, you can press the option key to get the second touch and then rotate the button. The rotated button works just as it did when it was normal.



Segues and connections

The last thing you need to know with Interface Builder is **segues**. It is pronounced as seg'ways as in the two-wheeled self-balancing transportation vehicle. Segues are the lines with arrows that connect various scenes with the elements on another scene. If you are a designer, you can use this to quickly create mock-ups that work natively and move between scenes without writing a single line of code. The other thing about segues is that you can name them. In the attributes inspector, you can give them an identifying name. This name can be used via code to present or identify segues.

Add another button and position it at the bottom of the view; create **Center Horizontally in Container** and a **Bottom Space to Bottom Layout Guide** constraints. Next, add a new View Controller, press *control* and drag from the button to the View Controller, and select show. Click on the newly created segue and, in the attributes property, select `mySegue` as the identifier.

Now, in the assistant editor, open `myViewController.swift`. Add the following code to that file:

```
override func prepareForSegue(segue: UIStoryboardSegue!, sender:
AnyObject!) {
    println("The segue '\(segue.identifier)' was called")
}
```

The `prepareForSegue` function is called every time a new scene is called. This allows you to prepare and be able to pass data to the new scene or set up the new scene. It provides you with access to the new scene's View Controller . However, this function is invoked in the current View Controller before the new View Controller is displayed. Now if you run the project, navigate to this scene, and click on the button, you will see the message **The segue 'mySegue' was called** displayed in the console.

You can also manually invoke segues from code then rely on design-time connections. If you called the `presentSegueWithIdentifier` function at the end of the `viewDidLoad` function, it will navigate to the new view when this view is displayed:

```
self.performSegueWithIdentifier("mySegue", sender:self)
```

Summary

In this chapter, you saw how a storyboard contains several scenes and how they could be connected via segues. This helps create quick interactive mock-ups that can be provided to clients or your teams. You also learned how you can set up constraints and how to create adaptive UI's using the size classes. You also learned about dragging to the code to create outlets or action functions.

If you are feeling adventurous or when you feel comfortable, try to use segues and display all of the stooges data using a single view, having **Table View** populate dynamically from a data source.

In the next chapter, we will have a look at creating custom controls. Custom controls are the controls that are not available with the standard controls that come with Xcode; they are created by developers.

5
Custom Controls

In this chapter, we will cover the following topics:

- An introduction to custom controls
- The basics
- Interactive properties
- Frameworks
- Debugging

An introduction to custom controls

In the previous chapter, we saw that we could quickly build our UIs in Interface Builder. We added components or elements from the library onto the views. However, Apple cannot provide developers with every conceivable control as developers keep coming up with new and innovative ideas to add beautiful features to an app, other than just simple basic TextBoxes, Labels, TableViews, and so on. Creating a custom control is not exactly new. However, the new features that Apple has added in Xcode 6 are quite exciting and these will unfold in this chapter.

The basics

At the very base, all visual elements are derived from the `UIView` object described in `UIKit`. A custom control is basically a custom class that is derived from a `UIView` class and is responsible for displaying itself visually on the screen.

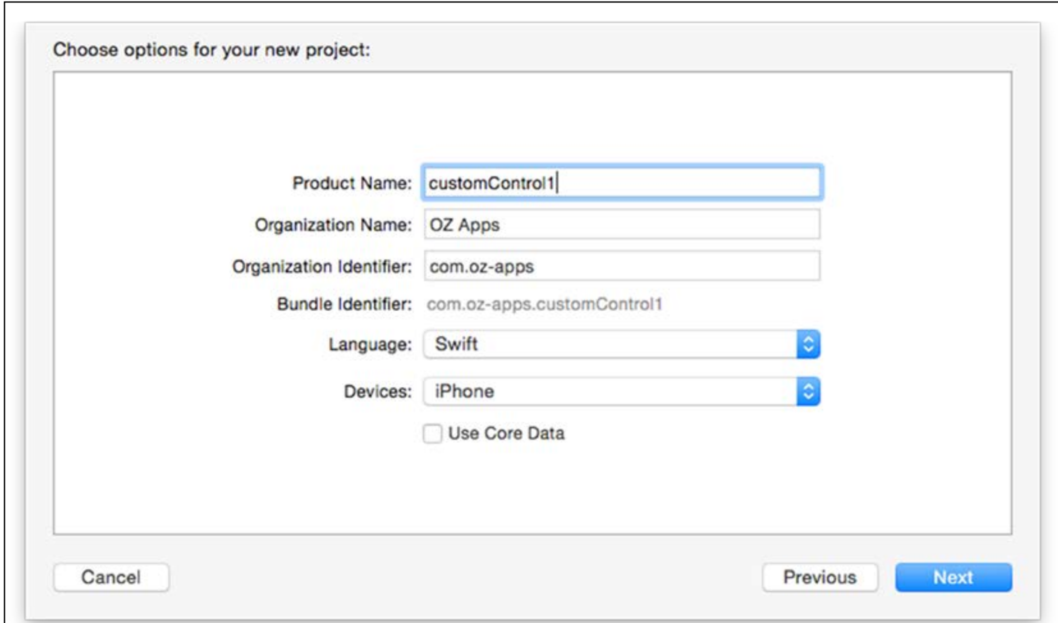
Some controls such as the stock ticker will display the data based on the stocks that you might want to track. In the case of a progress bar, however, you might want to set the progress and then also retrieve the progress status at some point.

www.allitebooks.com

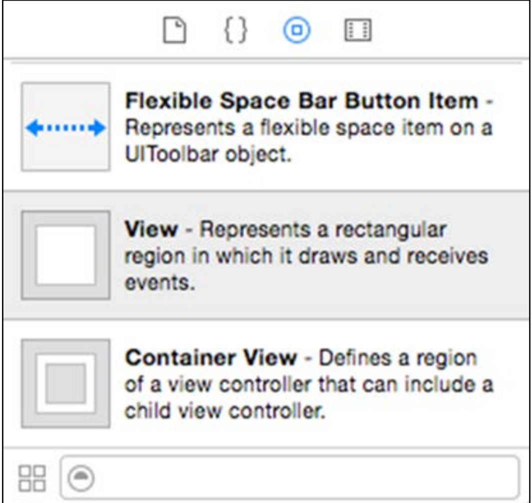
Creating a class

First, let us start with the basics; we will create a class and give it some properties. Since this is a visual class, we will need to derive it from `UIView`. Perform the following steps:

1. Start a new project and name it `customControl1`:



2. Click on `Main.storyboard` to open it in the editor and then drag a view control onto the View Controller scene. This will be the placeholder for our custom control. Since the view has a white background and so does our scene, it is a little difficult to see white on white. Change the background color to **Light Gray** and set the size of the control to `100 x 100`. Now it will be manageable and visible.



3. Next, create a new Cocoa Touch Class that is a subclass of `UIView`. Select File | New | File and under Source, select the Cocoa Touch Class. Name the class CustomControlView and make it a subclass of UIView.
4. Open the storyboard in the main editor and open the newly created `CustomControlView.swift` class in the assistant editor.
5. Select the `UIView` control and, in the Identity inspector (*CMD* + *3*), change the class to `CustomControlView`. Remove all the commented code provided by the Xcode template—that is, between the `/*` and `*/`. Just leave the top lines of import and the class declaration, as it is easier to work with a clean blank slate.

Properties

The custom control that we create will be a simple placeholder that displays the size of the control. We need a property, `backColor`, for this control. This can be easily created with Swift, as follows:

```
var backColor = UIColor.lightGrayColor()
```

We also need to override the `layoutSubViews` function, called when the view is created, resized, or scrolled:

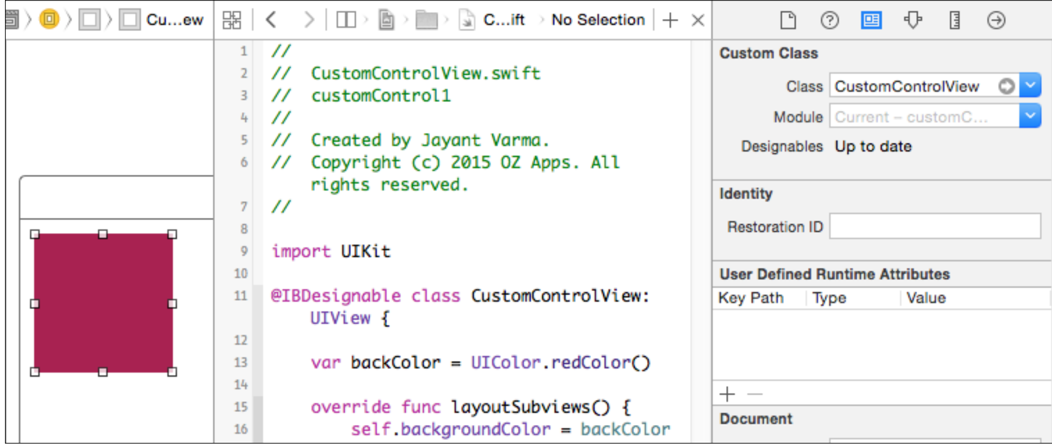
```
override func layoutSubViews() {
    //set the background color
    self.backgroundColor = backColor
}
```

Change `backColor` to `UIColor.redColor()`. You will see no change in the view. What we are after is a custom control that is interactive and changes when we make changes in our code.

View Controller › View | CustomCo...lView.swift › No Selection

```
//
//  CustomControlView.swift
//  customControl1
//
//  Created by Jayant Varma.
//  Copyright (c) 2015 OZ Apps. All rights reserved.
//

import UIKit

class CustomControlView: UIView {

    var backColor = UIColor.redColor()

    override func layoutSubviews() {
        self.backgroundColor = backColor
    }

}
```

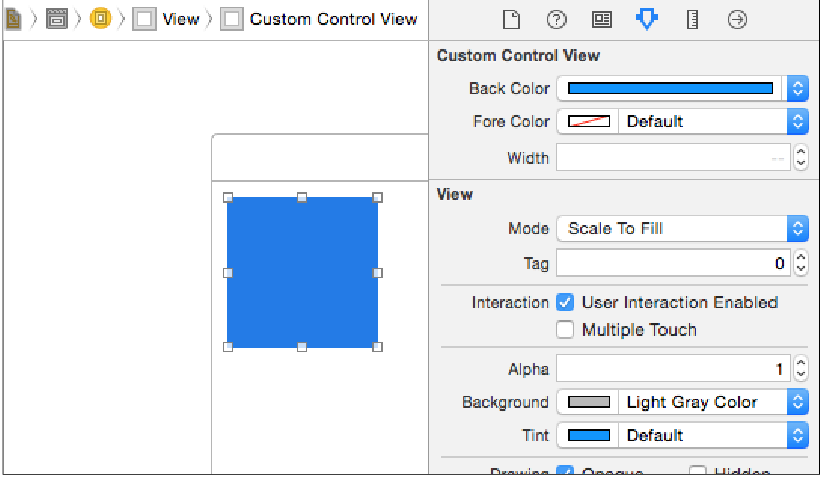
This is pretty simple and easy. All we need to do is add the `@IBDesignable` attribute before the class declaration. Xcode will compile the code in the background. Voilà, the color changes to red! You will also notice that there is a new attribute added under the class called `Designables` with the text, `Up to date`:



This should help you to understand and look at what Xcode has to offer in terms of live previews.

Changing properties from the inspector

It would be nice if, instead of changing the color in the code, we could change it from the inspector. For us to be able to do this, we need to set the `@IBInspectable` attribute before the variable that we want to expose in the **Attributes Inspector**. In our case, it would be `@IBInspectable var backColor = UIColor.redColor()`, and Xcode will show this attribute under the **Attributes Inspector** option. you can see the same show up as can be seen in the following figure.



Select a color from the drop-down list. If you navigate to **Identity Inspector**, you will be able to see the color under **User Defined Runtime Attributes**. You will see that there is absolutely nothing more you need to do. Now if you select a new color from Xcode, the view updates automatically.

Enhancing custom controls

This is the easiest example to demonstrate a very simple custom control and how it works. However, when we want something a bit more complex that requires drawing lines and images, we can use `CALayer`, which is more efficient in comparison. The `CALayer` class is available in the `QuartzCore` framework, so you will need to import it before you can start using the `CALayer` class reference or `CAShapeLayer`:

```
var theLayer:CAShapeLayer!
override func layoutSubviews() {
    if theLayer == nil {
        theLayer = CAShapeLayer()
        layer.addSublayer(theLayer)
    }
}
```

First, we need to declare `theLayer` as an optional variable. Then, when the views are laid out, we need to check whether the variable is `nil`—that is, not declared—and create it as required.

Now that we have created the layer, we can get some information about the bounds:

```
let rect = CGRectInset(bounds, width/2, width/2)
var path = UIBezierPath(rect: rect)
theLayer.path = path.CGPath

theLayer.fillColor = backColor.CGColor
```

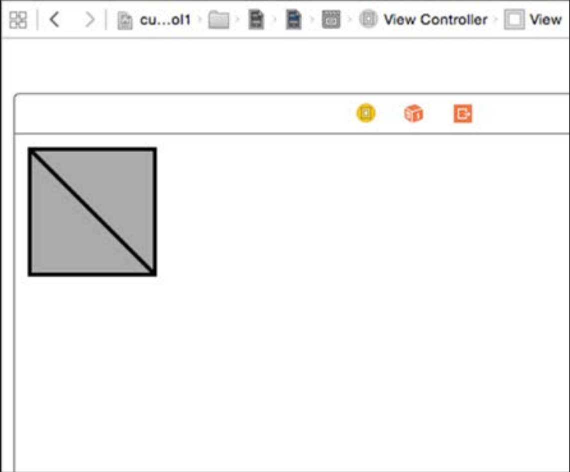
Now, instead of setting `self.backgroundColor`, we can set the color for the layer. Layers use color as `CGColor` and not as `UIColor`. We simply pass it in that format:

```
theLayer.fillColor  = backColor.CGColor
theLayer.strokeColor = foreColor.CGColor
theLayer.lineWidth = width
theLayer.frame = bounds
```

We need to add a couple of properties with the custom control that we will use in our project:

```
@IBInspectable var backColor:UIColor = UIColor.redColor()
@IBInspectable var foreColor:UIColor = UIColor.whiteColor()

@IBInspectable var width:CGFloat = 1
var theLayer: CAShapeLayer!
```

Our control now has an outline via the `foreColor` property and the background color via the `backColor` property. The `width` property determines the thickness of the line used in the drawing of the custom control, as shown in the following screenshot:



To draw a line diagonally across from the top-left to the bottom-right, we need to add a line to the path with, like a strike-out like a strike-out. Add this code before the `theLayer.path = path.CGPath` line:

```
path.addLineToPoint(CGPointMake(rect.width, rect.height))
```

Adding some text

To add some text to this control that shows the size of the control over it, we need to create new `CATextLayer`; this is similar to `CAShapeLayer` but is more efficient in comparison. After all, it is more suited for text.

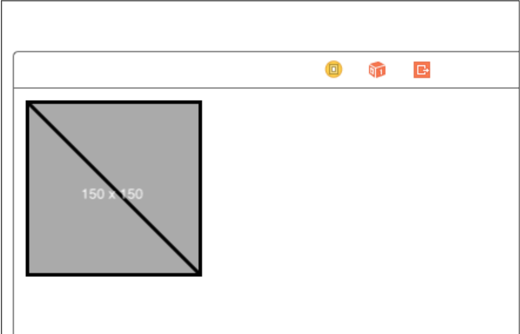
First, we need to add another property the variable declarations:

```
var txtLayer:CATextLayer!
```

Then, at the end of the function, after `theLayer.frame = frame`, we can add the following piece of code:

```
If txtLayer==nil {
    txtLayer = CATextLayer()
    layer.addSublayer(txtLayer)
}
let size = txtLayer.bounds
let szW = Int(size.width)
let szH = Int(size.height)
txtLayer.string = "\(szW) x \(szH)"
txtLayer.foregroundColor = foreColor.CGColor
txtLayer.backgroundColor = nil
txtLayer.fontSize = 12
txtLayer.position = CGPointMake(CGFloat(szW/2), CGFloat(szH) -
(txtLayer.fontSize/2))
txtLayer.alignmentMode = kCAAlignmentCenter
txtLayer.bounds = bounds
```

Now, we can see that the custom control displays its size at the center of the control. This can be used as a placeholder in your applications to quickly see the size of the control:



Frameworks

When you want to create a custom control, Apple recommends that you simply follow these four easy steps:

1. Create a framework.
2. Create a subclass.
3. Mark the class as IBDesignable.
4. Set the class of the view to the custom class.

To be able to debug the code of the custom control, it needs to be run; however, while you are working with Interface Builder, writing the code, it is not really running. When you create a framework, Xcode compiles and runs the code and also allows you to debug it.

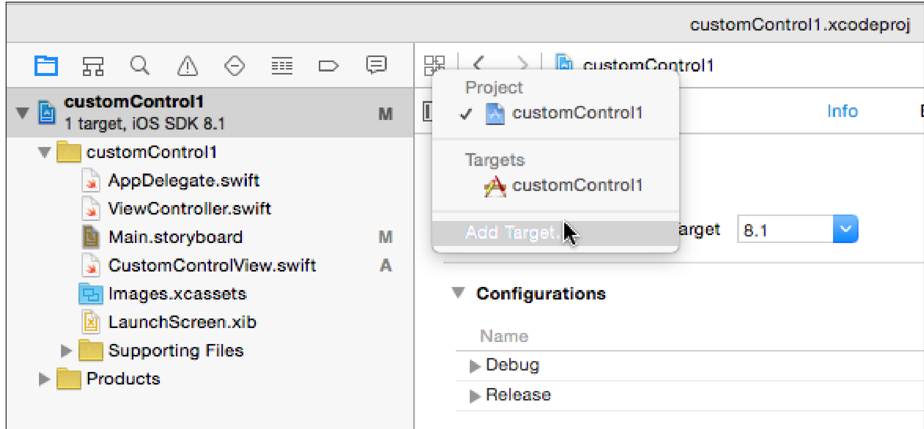
A simple rule about frameworks is that *if the code appears more than once, it probably belongs in a framework*.

Creating a framework

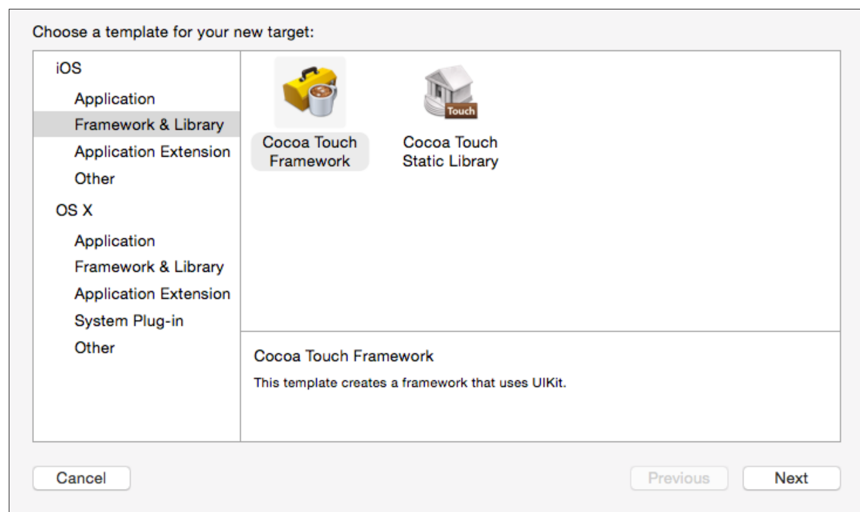
We saw from the recommendations made by Apple, that custom controls should be placed in their own framework. The simple reason is that, if you want to use this control in an Objective-C project or distribute it to your friends and customers, you might want to keep it on its own as a framework. Moving an embedded custom control to a framework is easy, as we will see through the following steps:

1. First, select the project in the project navigator. In the main editor, this will bring up the information and build settings.

2. Click on the **CustomControl1** drop down on the top-left of the editor window:



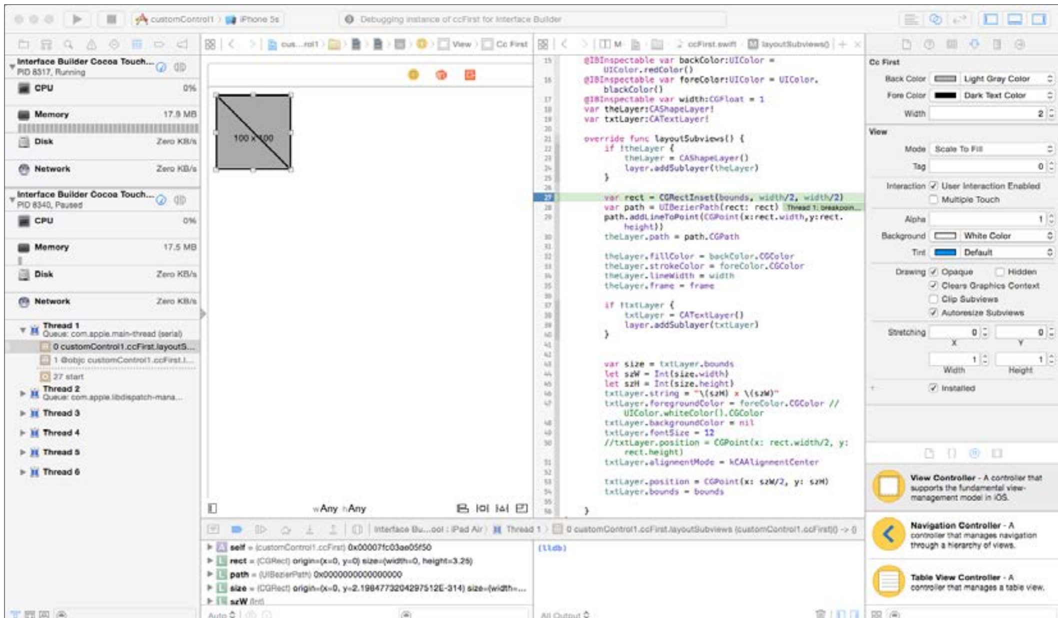
3. Then, click on the drop down and select **Add Target**.
4. Select the template type for the new project, which in this case will be **Cocoa Touch Framework**. Give it a name; I am using `MyCustomControl`. This is similar to creating a new project that is embedded in the main project.
5. Next, click on the `MyCustomControl` folder in the project navigator and navigate to **File** | **New** | **File**.
6. Select the **Cocoa Touch Class** template, make it a subclass of `UIView`, and call it `MyCustomControlView`. The process is mainly the same as the one we did earlier at the start of this chapter.



7. Lastly, as before, set the class to `@IBDesignable`, add the appropriate `@IBInspectable` to properties as required, and finally write the code for your custom control.
8. Now, place a view control on the storyboard scene from the object library and change the class name to `MyCustomControlView`. You will be able to see that this control is interactive during design time.

Debugging custom controls

Xcode makes it easy to debug a custom control; it is as simple as selecting the custom control and then navigating to the **Editor** | **Debug Selected Views** menu. However, the first thing to do is to set breakpoints in the source file and then select **Debug Selected View** from the menu. All the debugging tools and features will be available as soon as the breakpoint is hit.



Debugging tools and features

Summary

In this chapter, we had a look at the four-step recommendation made by Apple to create custom controls. You learned how to subclass `UIView` to use it as a custom control. You also learned about the `IBDesignable` and `IBInspectable` attributes, that allow Xcode to display the control and let the user interact with it. We also saw briefly that we can debug the control in design time rather than at runtime.

In the next chapter, we will look closely at debugging.

6
Debugging

In this chapter, we will cover the following topics:

- Breakpoints
- Console
- Debug view hierarchy
- Data tips and quick look
- Debug gauges

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you are as clever as you can be when you write it, how will you ever debug it?

– Brian Kerninghan

No one writes code with the intention of introducing bugs. However, there are situations where despite the best intentions our worst nightmare comes true—the code crashes. Even with the new advancements that check for code issues and type-safe code, to make a long story short, even then bugs can creep in and cause crashes. Crashes and bugs are not just code errors; they could be missing or unreferenced libraries. There could also be a case where you might be attempting to optimize your code. Debugging provides you with a way to go through your code, and Apple has a whole heap of tools and utilities to help you achieve this task.

Breakpoints

When you run your project, it opens up in the Simulator or the device (mainly in the simulator). You can stop it from running with the stop button. Clicking on the stop button returns you to your code. You can then update your code and rerun the project. Everything is fine but, if there was an error while the project was running, it goes back to Xcode and displays a source, either your own source code or some other code where the error occurred. However, instead of waiting for errors to break your code and provide you with an opportunity to debug it, you could set breakpoints. Breakpoints are, as the name suggests, points in your code that cause the running code to break (go to debugging mode) when the execution reaches that point.

One of the ways in which most developers try to resolve errors is by including statements that output some text and check what happened or is happening. This could work for some. However, these will have to be removed when you build your application for release. Then, there are times when your code does not do what you expect. For example, it loads the wrong image or does not display one, and so on. This will not crash your code and hence will not bring up the debugger, but you might still want to know the reasons why it did not work. The worst thing after an application that crashes is an application that does not do what it is expected to do.

Creating a breakpoint is quite simple; just click on the line number in the code editor and you will see a blue mark on that line. Click on it again and it will turn lighter indicating that the bookmark is inactive. Removing a bookmark is easy; simply drag the bookmark onto the code and it will turn into a dust ball and be removed once you release the drag.

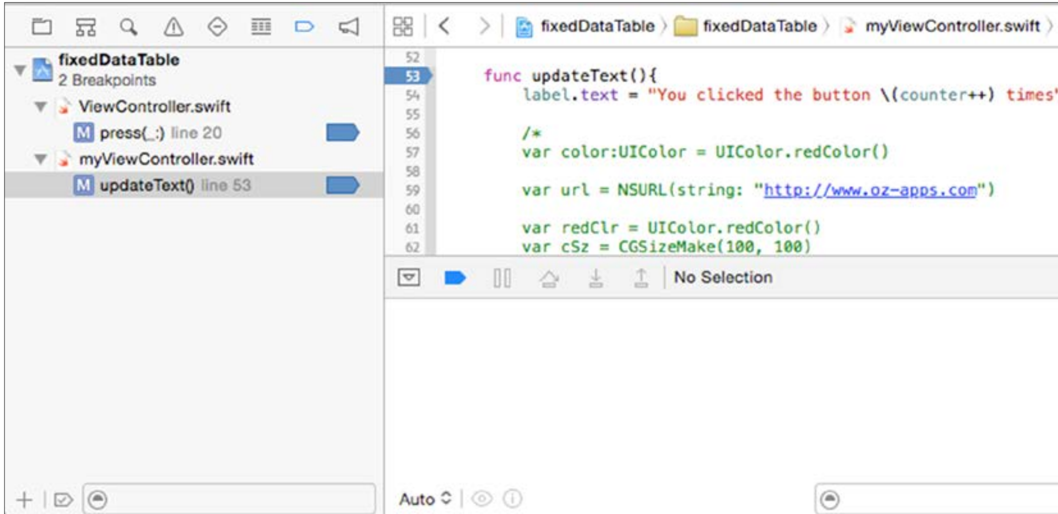
There are additional types of breakpoints, as follows:

- Exception breakpoints
- Symbolic breakpoints
- OpenGL ES error breakpoints
- Test failure breakpoints

On the breakpoints navigator screen, the **+** button at the bottom can be used to add these additional breakpoints.

Listing all the breakpoints

The breakpoint navigator (*CMD + 7*) displays all the breakpoints. The listing displays the source file and the line number where the breakpoint is created. Clicking on these breakpoints will quickly navigate to the location in the code where the breakpoint is set:



Navigating

When Xcode breaks execution at a breakpoint, it offers you a couple of options to work with the code. You can continue executing the code (unless there is an error, which is equivalent to a crash if it was not being debugged). Then, there are the options to step over, step into, and step out, hierarchy view and location:

- **Step over**: This executes the line or function and goes to the next line
- **Step into**: This executes the line or jumps to the function being called
- **Step out**: This exits the function and jumps to the line of code it was called from
- **Hierarchy view**: This allows you to see all the elements on the screen

- **Location**: This allows you to set the location for the simulator since it does not have GPS capabilities

Xcode also displays a list of variables in the current module and function. These variables can be expanded, viewed, inspected, and even printed onto the console. The variables also display their type, which can be useful to determine the type of the variable when debugging.

When an error occurs in your code, there are a questions that need to be answered to help get a solution for the crashing code:

- What happened and why has the code stopped execution? Xcode displays this on the right-hand side of the editor.
- Which code resulted in this outcome? This line of code is highlighted in green with the reason why the code stopped execution.
- Investigate the variable and its values. Xcode displays the list of variables and their values just under the debug toolbar.
- Lastly, fix the code as required and step through the code.

The console

The console is the area that displays all your system-generated messages and the messages that are printed using `println` or `NSLog`. The messages are displayed only when the application is running. This console provides interactivity when in debug mode. In interactive mode, you can view the values of variables and objects, run the debugger commands, inspect code, evaluate it, step through it, and even skip code.

For most developers that are new to debugging with Xcode, the GUI is self-sufficient and provides all the tools and icons to access these tools. However, if you want to use the command line in the console, you have the `lldb` commands at your disposal. Some of the commands that might be useful are as follows:

- `thread info`: This displays information about the thread including the reason why the break occurred.

- `thread backtrace`: This dumps the trace and the frames. This information is displayed graphically as the thread information, which you can see in the previous screenshot.
- `frame select n`: This is quite useful when you are trying to focus on what the piece of code being executed is. It displays a couple of lines of source code around the line of that frame.
- `br`: The `b` or `br` command allows you to work with breakpoints. Although you can set the breakpoints by simply clicking on the line in which you want to set them, you can also do this from the console. However, these do not set visual breakpoints:
    - `br l`: This lists all the breakpoints that are set.
    - `b function_name`: This sets a breakpoint for a function.
    - `b filename:line`: This sets a breakpoint in the file specified by the filename at the specified line number.
    - `br del num`: This deletes the breakpoints. Use the list command to see a list of the breakpoints that are set.
    - `br m -c "condition"`: This sets the condition for the breakpoint.

When you list the breakpoints, it displays detailed information about the breakpoint and also the number of times the breakpoint was reached.

A life saver in the console is the command po, which is the short for 'print out'. This simply prints out the value of the variable following the po command. Sometimes this is more useful as you can use this to cast the values. For example, **po theData as myClass**.

The ability to switch to the **REPL** mode (**Read-Evaluate-Print-Loop**) is new in Xcode 6. The REPL mode is specific for Swift. Getting into the REPL mode is as simple as typing `repl` in the console. Once you're in the REPL mode, you can type in the Swift commands or even write functions that can call or be called from your code. When you are done, simply type `:` to exit the REPL mode back to the `lldb` debugger.

Understanding the debug information

When you reach a breakpoint or an exception/error, you can look at the information that is displayed; but what does it all mean? Here's a quick look at the information displayed and how to use it to debug and resolve the issues.

First, when the execution is interrupted either by reaching a breakpoint or an exception or by physically stopping the execution, `lldb` displays the reason for the code interrupting execution and all the frames and threads, as relevant.

To quickly understand debugging, let us create a new single-view project and call it `myDebugging`. Now, open the `ViewController.swift` file in the editor and add the code immediately after the `ViewController` class declaration line:

```
var counter = 0
```

Next, add the following code to the `viewDidLoad` function:

```
NSTimer.scheduledTimerWithTimeIntervals(0.5,
           target: self,
         selector: "runme:",
         userInfo: nil,
          repeats: true)
```

Lastly, create a function called `runme`. This function is called every time the timer that we created in the previous code fires:

```
func runme(timer:NSTimer) {
    counter++
    if counter <= 10 {
        println("timer fired \(counter)")
    } else {
        timer.invalidate()
    }
}
```

Now, when we run the project, it displays `timer fired 1`, goes on till `10`, and then stops printing. This is because of the `timer.invalidate` function that stops the timer from firing more events.

To start debugging, click on the line that has `func runme`. So, now whenever the function is called, Xcode will go to the debug mode.

You will notice that the line is highlighted and, to the right, **Thread 1: breakpoint 1.1** is displayed. This means that the code has stopped because of a breakpoint.

We asked a couple of questions earlier, among which was: What caused the break? We can see the reason for the break in the code editor and also type thread information into the console. It will display something like the following code:

```
thread #1: tid = 0x1302c, 0x00000001006afab3 debugging`debugging.
ViewController.runme (timer=0x00007fb6c1b6d250,
self=0x00007fb6c0d23e70) (ObjectiveC.NSTimer) -> () + 19 at
ViewController.swift:34, queue = 'com.apple.main-thread', stop reason
= breakpoint 1.1
```

You can see that `debugging`debugging.ViewController.runme` is the representation of `<Project>`<Folder>.<ClassName>.<function>`. It simply means that the project is called `debugging` and the folder in this case is also called `debugging`. The class name for this is `ViewController` and the function is called `runme`. Further down, you can see `ViewController.swift:34` that indicates a human-readable form for the source filename and the line where the breakpoint is created.

Let's list out the breakpoints. We can list them by typing the `br l` command in the console; it will display something like this:

```
Current Breakpoints:

1: file = '/Users/Jayant/Desktop/Projects/debugging/debugging/
ViewController.swift', line=33, locations = 1, resolved = 1, hit count =
1


 1.1: where = debugging`debugging.ViewController.runme (debugging.
ViewController)(ObjectiveC.NSTimer) -> () + 19 at ViewController.
swift:34, address = 0x00000001006afab3, resolved, hit count = 1
```

We can inspect the variable counter by using the `po counter` command; if this were the first run, it would return the value of `0`. We can also modify the value by using the `expr counter = 5` command; this will set the value of the counter to `5`. You can click on the Continue button to continue running the code till the next breakpoint stops it (or use the key combination *CTRL* + *CMD* + Y).

The scope of this chapter is a simple introduction and this can be a bit difficult to take in at one go; so play around with it and try the commands listed in this chapter. If you get stuck, just stop the project and rerun it.

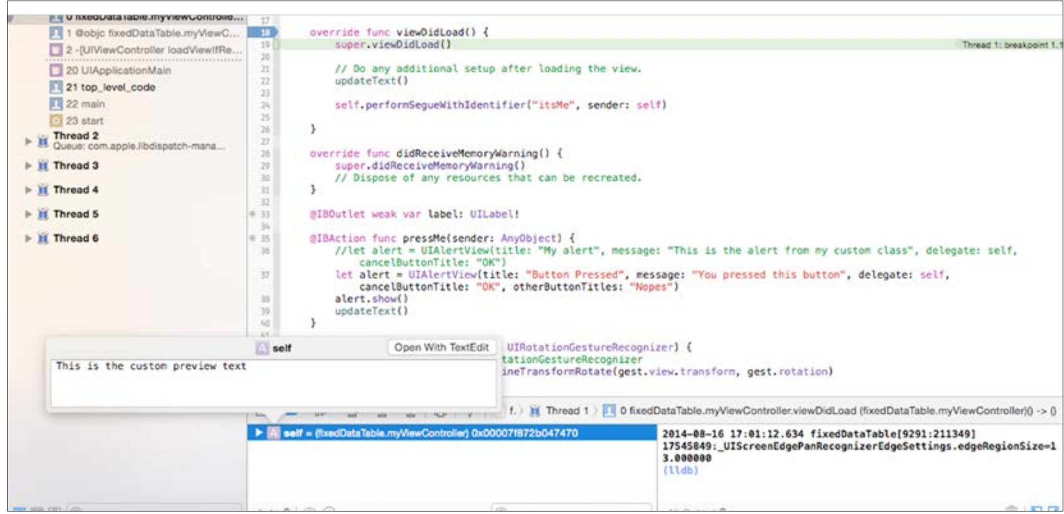
Visualizing variables with the Quick Look functionality

If you have worked on a Mac, you would be aware of the Quick Look functionality. It provides a way to quickly see a file's contents without having to open it in the associated application. As you saw earlier in *Chapter 3*, *Playgrounds*, with Xcode you can now visualize objects and structs such as `CGRect`, `CGPoint` and so on. However, when it comes to your own classes, Xcode does not know how to render these. Your class can be an image, a text, or a combination of these. For example, if our object is an employee class, should it display the image, the employee number, or the employee name? The choices or options are endless. However, you are the best judge to decide the best way to visualize the employee class. So, Xcode provides us with a function that we can use to visualize the class as per our specifications.

Let us see how Quick Look works with our custom class that we created in the previous chapter. After opening the project, click on the `myViewController.swift` file and add the following code before the last curly bracket:

```
func debugQuickLookObject() -> AnyObject {
    return "This is the custom preview text"
}
```

Create a breakpoint on the `viewDidLoad` function. Now, if you run the application, you can see it in the `self` variable in the watch list, as shown in the following screenshot:



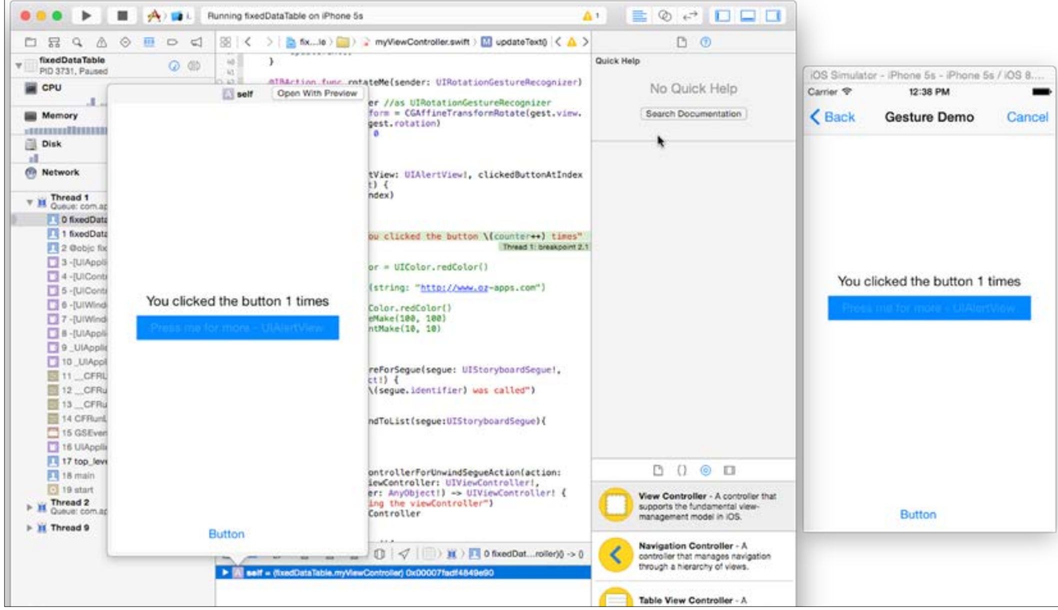
Creating a breakpoint on the viewDidLoad function

In the context of this example, the `self` variable refers to `myViewController`. You can click on it and then click on the little icon that looks like an eye. This displays a pop over with the visual representation of the custom class. In this case, since we return a string value, it displays the text in the pop-up, as shown in the preceding screenshot.

Using images with quick view

We can alternatively return a `UIImage` object or draw on one and return it. We can also render the current view to show what we have on the screen. This is relatively simple. Replace the `debugQuickLookObject` function with the following code:

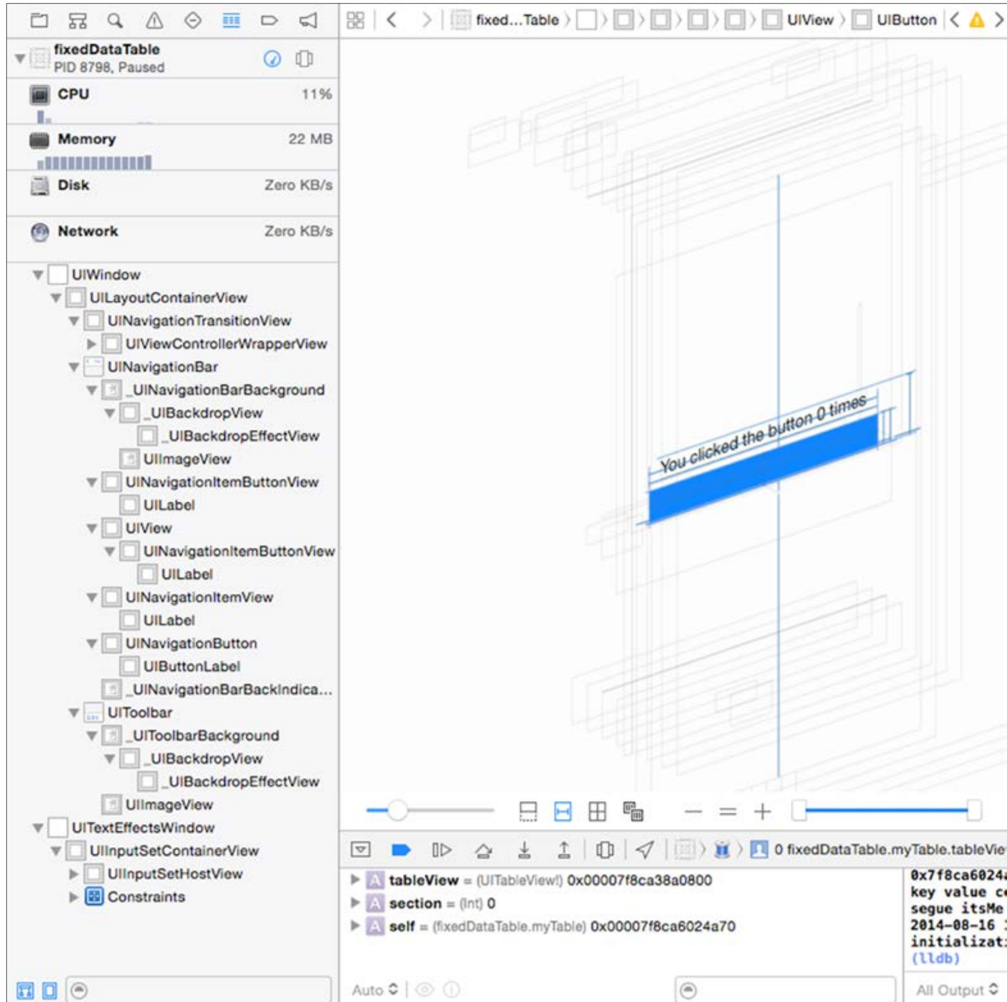
```
func debugQuickLookObject() -> AnyObject {
    let theView = self.view
    UIGraphicsBeginImageContextWithOptions(theView.bounds.size, false,
0)

    if !theView.drawViewHierarchyInRect(theView.bounds,
afterScreenUpdates:true) {
        theView.layer.renderInContext(UIGraphicsGetCurrentContext())
    }
    let image = UIGraphicsGetImageFromCurrentImageContext()
    UIGraphicsEndImageContext()
    return image
}
```



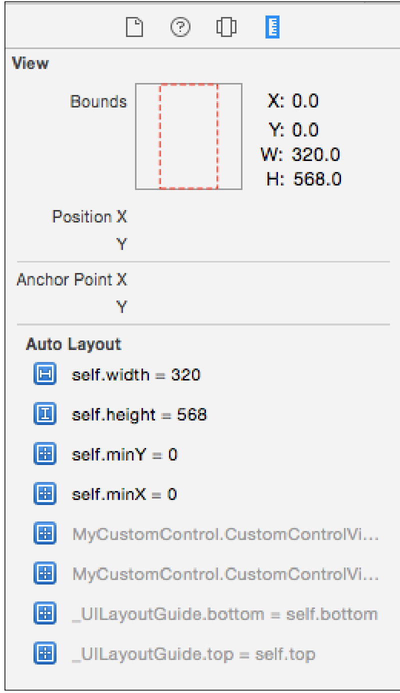
Using images with quick view

Debugging the view hierarchy

When an application is running, it is made up of a hierarchy of windows and views. As you create elements, they are combinations of views and subviews. This is similar to what we saw in the previous chapter where we subclassed a UIView with our custom class to create the custom control.

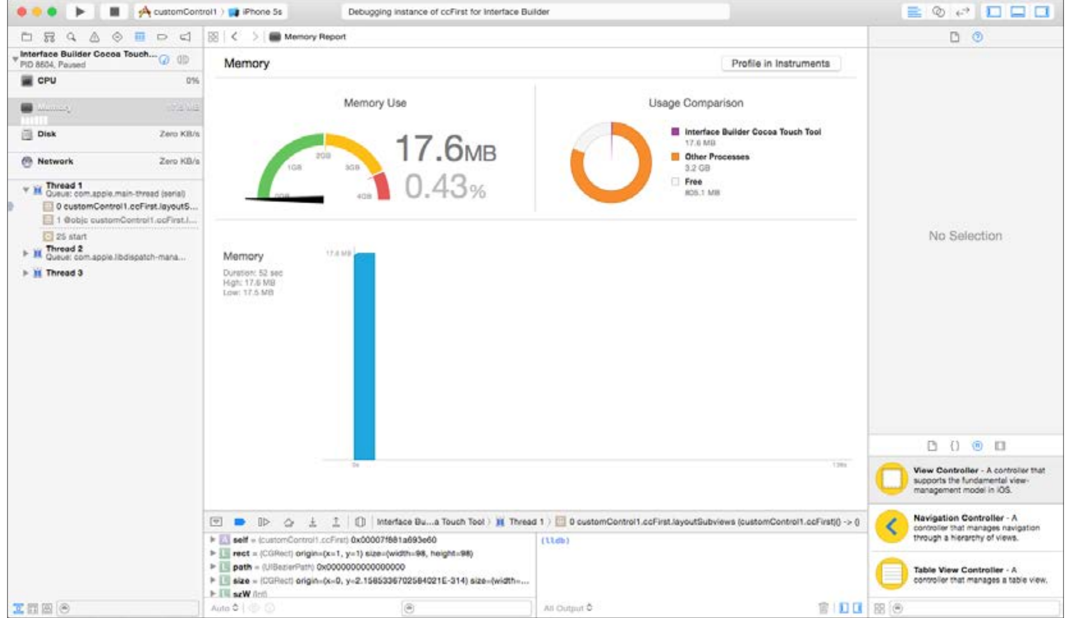


While debugging, you can inspect the variables, the code, and so on, and investigate any code issues. However, if a particular view is presented in an unexpected order other than and gets covered, it will not be seen. You could spend a lot of time trying to debug the code before you discover that your text or image was covered by another one. When you use the hierarchy view, you can see all the views exploded and you can also rotate the view to display all of them. Apple introduced this feature in Xcode 6 and it offers options to display the views along with their attributes. These attributes can be inspected in the inspectors.



Introducing debug gauges

When you start an application, the navigation viewer changes to the debug view where you can see some gauges. These gauges provide you with information about features such as CPU usage, memory usage, and so on. This also displays all the threads running with the functions:



Debug gauges

The CPU report displays information about the process that is being run. In the preceding screenshot, the **Memory** report is displayed and you can see that **17.6 MB** of memory is in use by the application, 0.43 percent of the total available memory. The **Usage Comparison** report shows the memory in use by other processes. This is clearly running in the simulator on a Mac with 4 GB of RAM. When you are testing and optimizing, you must use a device.

Similarly, there are gauges for CPU usage, disk usage, and network usage. Therefore, I you added functionality for other features such as iCloud, and so on, then the appropriate gauges for these would also be displayed.

Summary

In this chapter, you saw that, despite reducing errors at precompile time, errors can still slip in and crash or prevent your code from doing what is expected. Apple provides a powerful debugger and features that allow you to debug your code, both visually and functionally. You also saw how you can create custom Quick Look previews for your own classes. Lastly, you also received a brief introduction to the debug console and its commands. For the scope of this book, we had a brief look at the relevant features in Xcode that you might use for debugging. Debugging by itself is vast and can warrant an entire book.

In the next chapter, we will compile and package everything to create our application ready for deployment to the testers or the store.

7
Building and Running

In this chapter, we will cover the following topics:

- Simulator
- Organizer
- Code signing
- Archiving

In the last few chapters, we have explored some of the features of Xcode. From writing the code to debugging it and ironing out bugs (if any). However, we can't give our application to our testers and/or upload it to the App Store as-it-is in source-code form. The first step in running our application is simply achieved by clicking on the big **Run** button on the top-left corner of the Xcode window. This compiles and runs our code into a simulator (by default) or the target as specified.

Simulator

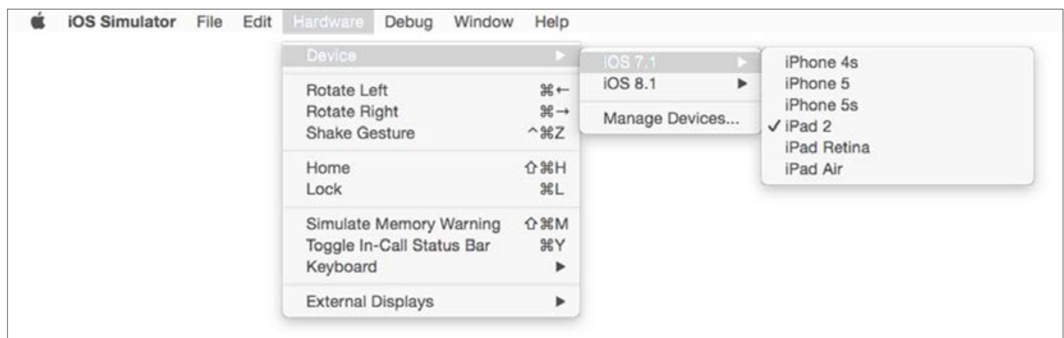
Apple provides a simulator that can be configured to simulate several iOS devices. With Xcode 6, Apple has provided the iOS7 and iOS8 simulators (though you have to download the iOS7 simulator). There are a few things to note about the simulator:

- It is a simulator, not an emulator
- Most of the hardware-specific features are not supported
- Some of the features do not work on the simulator with iOS8 simulation

For speed-testing and optimization, the code should be run on a device. The simulator is no stranger to us, as we have been running our code and testing it in the simulator.

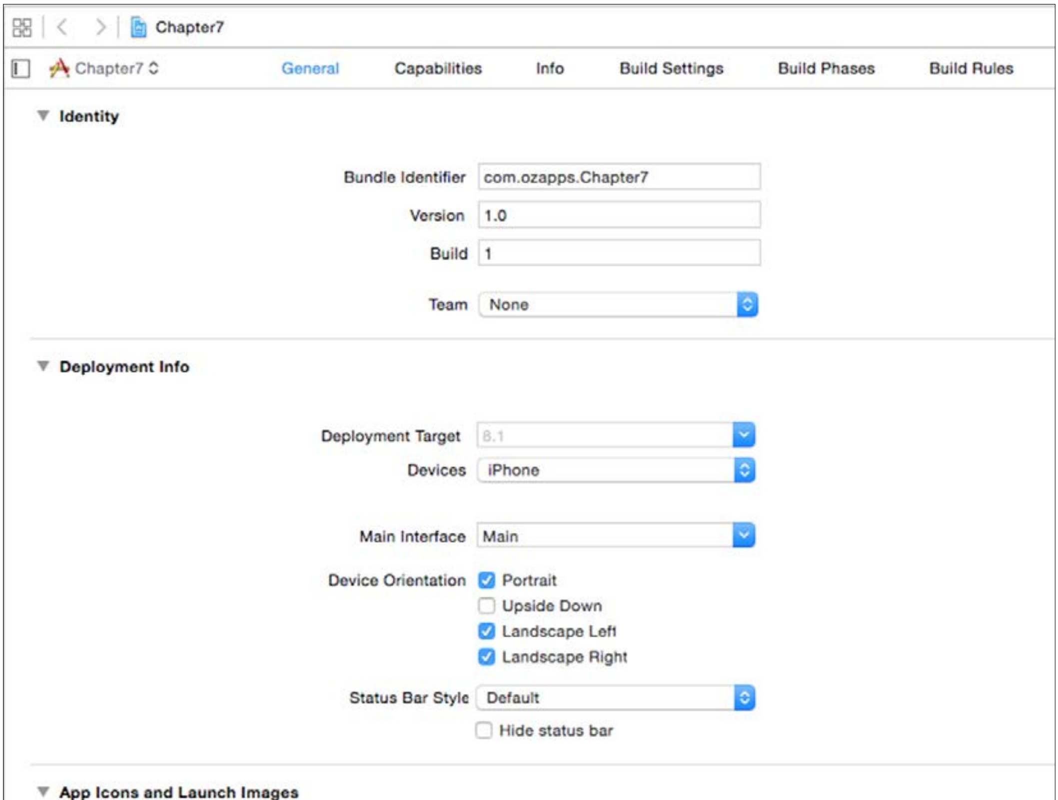
Choosing the device

When testing, you might want to test the application on different devices such as the iPhone or the iPad to check how the application renders on different devices. You might also want to choose how the application runs on normal iPad and retina iPad devices. It would be unwise to submit an application to the App store without testing on a device. To give an example, say you have an application that requires a phone-specific feature. You have tested against a physical phone, and it works all right. Now, your user downloads the same onto an iPod Touch. There is a chance that the application will crash as that phone-specific feature would be missing. It is best to check against a majority of hardware as possible and also check for device capabilities that you might want to use in your application.



Preparing for distribution

Once you have satisfactorily tested your application with the simulator and/or the device, you need to prepare it for distribution. The first stop to do so is by clicking on the application name in the project list that brings up the screen, displaying details about the application. Let's create a single view application called `Chapter 7`. We will use this template application (it does nothing for now) and prepare it for distribution. We will customize the different aspects that one might need at the end of their application.



Configuring the project settings

The fist tab you can see is the **General** tab. Under this tab, you will find major options listed in the following subsections. In the most basic scenarios, all one needs is present in this tab—for example, would the device support the various orientations; should the status bar (displaying the time and battery level) be visible?

The Identity section

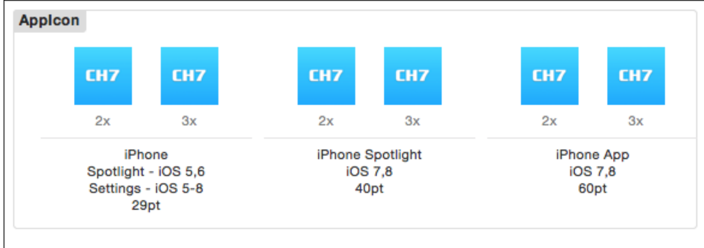
This helps to set up attributes such as **Bundle Identifier**, **Version**, **Build**, and **Team**. Once you have your Apple Developer account set up, Team will show your team name (generally your developer apple ID). If you have a new installation without any account, the **Add an account** option helps you to set up your account for use. You can also view your account details at any time via Preferences (*CMD* + ,).

The Deployment Info section

The **Deployment Info** section is useful to set the various aspects such as the orientation and the devices on which the application should run. The first option is **Deployment Target**. This is the setting that specifies the minimum iOS version. The next option is **Devices** that specifies the devices on which the application can run. This can be **iPhone**, **iPad** or **Universal (both)**. The **Device Orientation** option specifies the orientations that the device would adapt to/support. The **Status Bar Style** option specifies the type of status bar, light, or default and the status bar can also be hidden.

Application icons and launch images

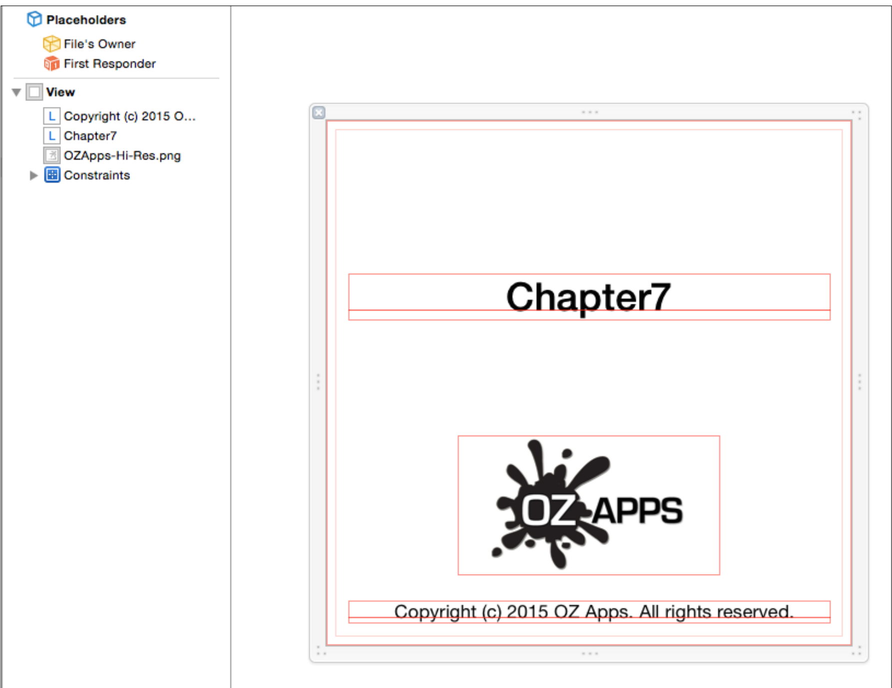
This is where you can specify the icons that will show up on the main screen from where the application is launched. The little arrow next to the **AppIcon** option takes you to `images.xcassets` that allows us to specify all the icons required as per the various resolutions and sizes:



If you have worked with or read about Xcode, you might have read about the filename, `filename@2x`. This was introduced allowing developers to add retina capable assets to their projects. With the minimum entry barrier raising to an iPhone 4S, the devices are all retina enabled and hence you do not create the 1x assets any more.

The launch image is the image that displays when the applications starts, also called the splash screen. Apple's suggestion for Splash Screens was to make it like the empty UI of your application. So, as far as the user was concerned, the application started instantly from the time they saw the splash screen. However, many developers would want to utilize this to display their logos, their publishers' logos, and so on. These were static images called `Default.png`; you could, like icons, have multiple splash screens for each device type and orientation. So you could have two Splash screens for the iPad, one in portrait and another in landscape. Then, you could have another two for Retina iPad and so on.

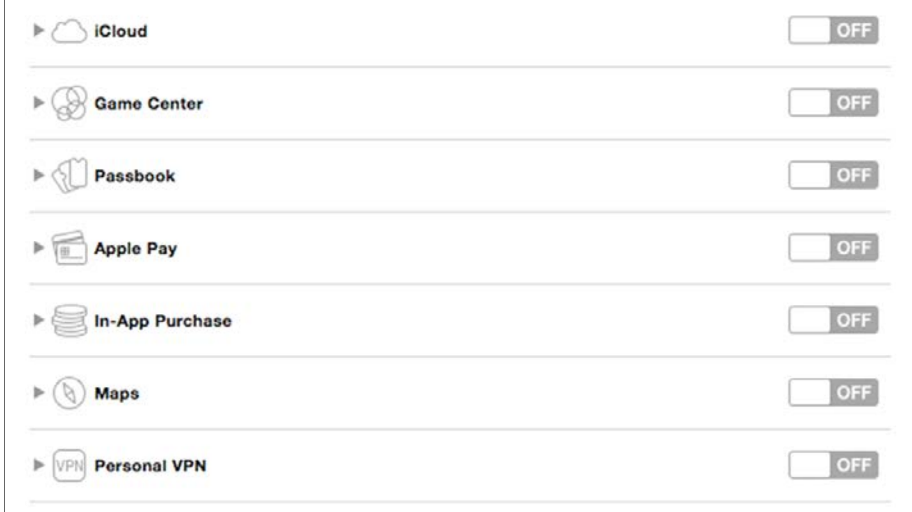
With Xcode 6, Apple has allowed the use of a XIB or storyboard file as a launch image. This is a single view like our application that is displayed as the Splash Screen. This could be used to display an animated Splash Screen, which is a custom loading screen. The following is the image of `LaunchScreen.xib` that is used as the launch image for the sample application. Note that this was automatically created by XCode 6.



The advantage of using a launch screen is that this is code driven and can include dynamism as compared to a static image.

Linked frameworks and libraries

This allows you to add any libraries or frameworks that you need to your application.



The Capabilities tab

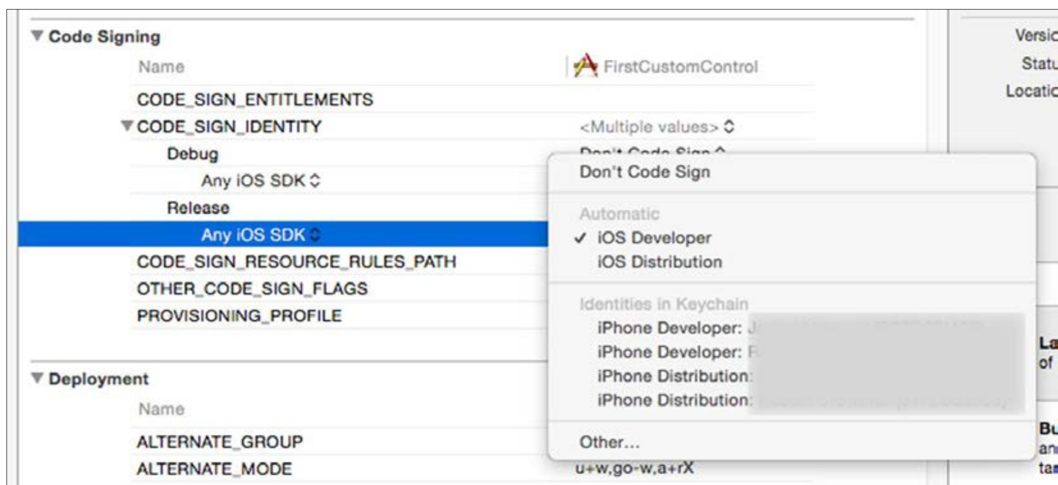
Under the Capabilities section, you can visually specify the options supported by your application. The way to set or unset any option is via a checkbox that specifies **ON** or **OFF**. The good part of this section is that it takes away the arduous task of linking frameworks and setting a key in the `info.plist` file. When you unset, it also removes those settings. If you have ever used any of these features in earlier versions of Xcode, you will know how painful and ugly this process could become. With Xcode 6, it automatically generates the required entitlements and provisioning certificates.

The Info tab

This is the section that displays the `info.plist` file that is used by the application to determine its capabilities and settings. Apart from the custom `plist` settings, it also allows you to set up the document types and custom UTI and URLs that the application will support. This is useful to allow your application to be able to handle a particular file type or handle a URL. Even with a simple application, you might have to visit this tab to add a key or tweak a setting. The most common scenario would be to specify the setting for location access authorization.

The Build tabs

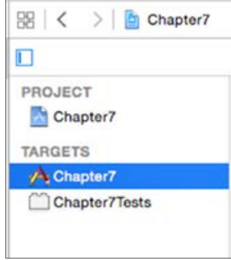
There are three additional sections, **Build Settings**, **Build Phases**, and **Build Rules**. We will not go into detail about these. It could be very rare for you to get to these sections. However, you could use this to fine-tune the compilation and linking flags. Earlier, you could specify the certificates and so on to use for signing the application. Now, in Xcode 6.1, Apple makes it easy for you and automatically sets the appropriate certificates and so on. As a beginner, you could get away without ever having to peek under the bonnet and tweak any settings in these tabs. Going through these settings in detail is a bit out of scope of this book. However, note that you can, if you want to, change a specific provisioning profile under the **Code Signing** section. In a real-life scenario, you might be developing with your own certificates but, when you want to deploy the application, you might want to use a client certificate. You can change from the automatic value set by Xcode here.



The Targets tab

Not all projects are applications; you might want to create a framework, a test, a library, or any other type. You could do this because you might not want to distribute the source code to anyone and instead offer them a binary that they could use in their application (via the embedded binary option).

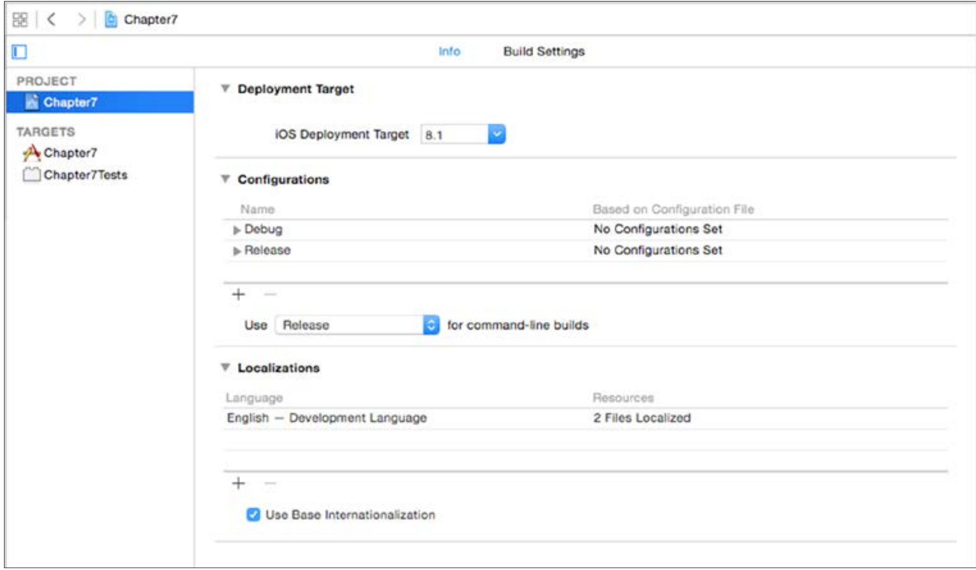
Each target has its own set of options associated with it; when you click on the different ones, you get different options. There are two sections namely: **Project** and **Targets**. Under **Targets**, by default, Xcode adds two options; one is the application and the other is the set of tests for that application.



Setting project-wide properties

The project settings page allows you to quickly configure the settings for the project—things such as the iOS deployment target. The settings we looked at earlier were specific for each target. This setting allows you to specify the minimum iOS version installed on the device for the application to run. Please note that the minimum requirement to run a Swift application is iOS 7.1; thus, if your application is written with or has a Swift component, setting a value lower than 7.1 will not work and the application will crash on launch.

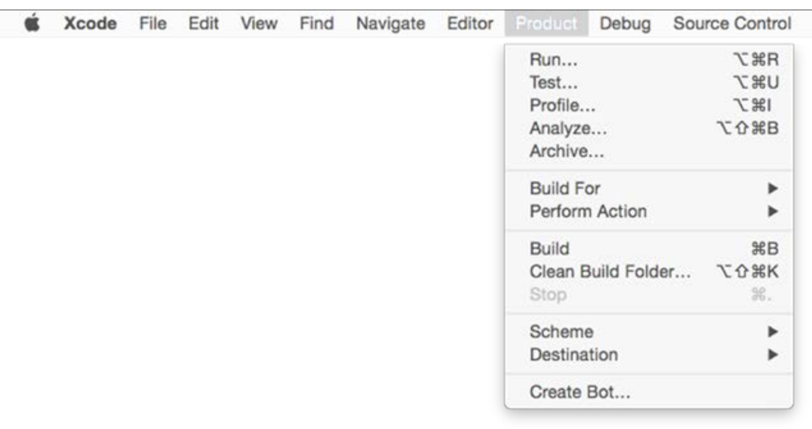
This also allows you to specify the build type, debug, or release.



Code signing

After you have finished testing and trying out the various settings, you need to build a binary that you can upload to the App Store. This is where a lot of new newcomers to Xcode get stuck. The general expectation is to receive a binary file that can be uploaded to the App Store. The earlier workflow in Xcode was exactly that simple. Start the build process and, it would compile and generate the final binary. You could pick up the relevant one from either the `Debug` directory or the `Release` directory. However, there is no such option available anymore. There is a small trick to it to get a release build for the App Store. I call it a trick because the natural assumption is that, when you compile the code, you will be presented with an end result as per the settings chosen. Perform the following steps:

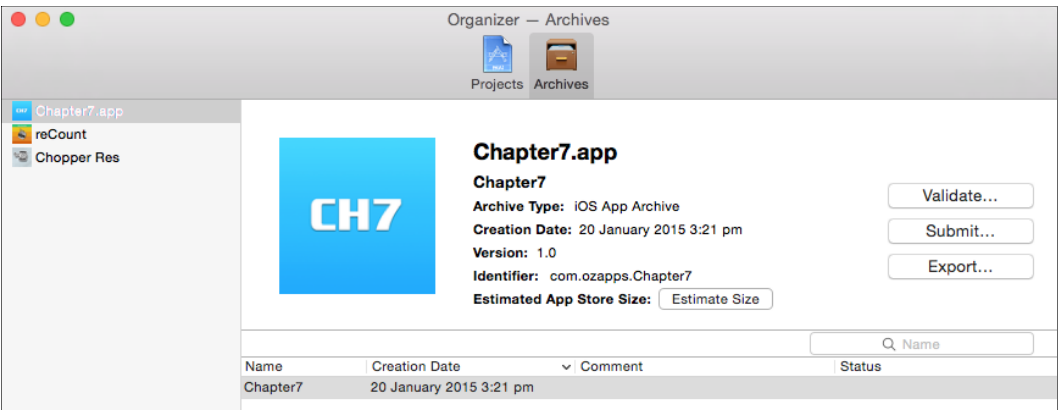
1. Ensure that Bundle Identifiers, assets, and so on are all set
2. Ensure that, under the **Build Settings** section, under **Code Signing**, the **CODE_SIGN_IDENTITY** under **Release** has iPhone **Distribution** and the **Any iOS SDK** has your team ID; if you are compiling for a client, you select their distribution identity and not your own, or vice-versa.
3. Also ensure that the **PROVISIONING_PROFILE** setting has the appropriate certificate that you want to use for distributing this application.
4. Change **Destination** under **Product** to **iOS Device**. The literal value of the iOS device not a connected iOS device or a simulator device.
5. Now, if you look at the **Product** menu, there is an option that was disabled earlier and is now enabled. This is the **Archive** option. Select the **Archive** option from the menu:



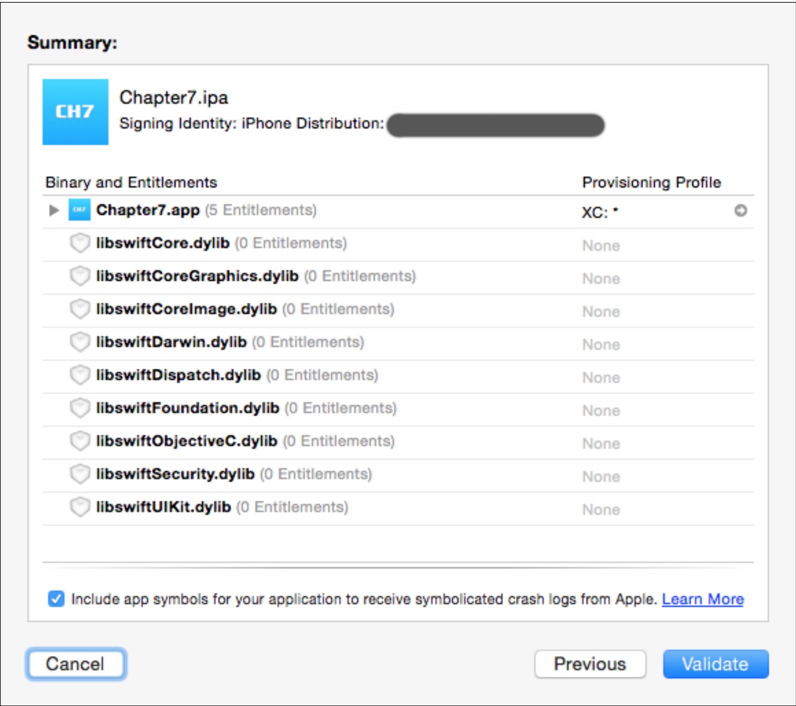
6. This will compile, code-sign, and create an archive of your application. This archive can be found in **Organizer**. To display the **Organizer** window, from the Xcode menu navigate to **Window** | **Organizer**.

Organizer

The organizer is the place where you can navigate and find all of your projects and archives. Under the projects, you can find all of the projects that you opened on that machine. You can also create snapshots, a saved version of your code like a versioning system. This is sometimes useful to check and fix bugs if your users provide you with crash reports. The organizer also houses the archives; this is of interest to us as this is how Apple has provided a workflow to upload your application to the App Store.



As you can see in the preceding screenshot, there are three projects that have been archived including our sample Project `Chapter7`. Since there was no icon specified for this, you are seeing the default blank template used for icons. From here, you can run the **Validate**, **Submit**, or **Export** commands. Validate checks for all of the libraries, dependencies including related files, and the certificate used for code signing.



This is quite handy as it checks for the preliminary issues that could cause your application to be rejected by Apple. However, this assumes that you have created an entry on the iTunes App Store for your application. Post validation, the application can also be submitted to the App store right from the **Organizer** window. Once it is uploaded, you wait for Apple to review and approve your application. Once it is approved, you can log into iTunes Connect and make it available for sale (if you set a later date), control the price, and so on.

Summary

Xcode has seen a lot of improvements and changes. Creating an application has become easier and more integrated than it used to be a couple of years ago. Certificates were a big hassle, which has now been integrated into the Xcode workflow. Some developers might feel that the way release binaries are created are a bit strange and rather inconvenient, whereas on the other hand, you no longer have to search for it, zip it, and so on. It is all integrated. Creating an application with Xcode is much faster and easier, add to that the introduction of the Swift language makes it simply a breeze. Hope this book has been able to introduce you to Xcode and get you started and look forward to seeing what you can achieve with this knowledge.

Conditional Execution and Interface Designing

"*The only thing that is constant is change*"

– Heraclitus

From the time Swift was introduced on stage to the first public release, there have been many changes. This is mainly because Swift has been thrust into the limelight due to being an Apple technology. This has also added the pressure for it to become a mature language. There will definitely be changes in the way Swift works in future. So bear that in mind that the tools around it will reflect the dynamic nature of change. It is not only about Swift; other aspects such as the editor, storyboard, and the debugger could also start to reflect the new changes.

Coming back to the present time, there are a few things that were important to discuss with respect to developing with Xcode, some little gotchas that needed a little getting around. This section actually came up as a result of the rapid changes that were introduced in Xcode 6.

More?.Swift!

From the version of Swift demonstrated at WWDC, to the current version that is available with Xcode 6, Swift has seen changes. Some syntax changed and in some cases the way the command/keyword worked changed. However, there are some important concepts that will always be important.

Each language has its own peculiarities that confuse or keep a beginner at a distance. These include `+(void)` or `-(void)` found with Objective-C, or `object.method()`, and `object:method()` found in Lua or **`?`** and **`!`** found in Swift.

What do those two symbols mean in the context of Swift code? The answer to that in a sentence is: `!` and `?` are used to identify optionals. We did discuss that Swift is a typesafe language, which means that the compiler ensures that every variable is of a particular type; if you assign the wrong type, the compiler complains, thereby avoiding a runtime error or a crash.

Consider the following lines of code:

```
var name:String = "Jayant Varma"
name = "Alexander"    // reassignment works
name = 123.45        // does not work
name = nil        // does not work
```

We can assign a new value of the same type but not of a different type. Even if the variable was of type `Any`, you cannot assign `nil` to it. It is important to note that `nil` indicates the absence of a value.

Let's look at another scenario; if we were to declare our variables, due to Swift being a typesafe language, you are expected to specify a type. So you cannot simply have code that says:

```
var theName    //Type annotation missing in pattern
```

You have to specify it like this:

```
var theName:String    // Class <classname> has no initializers
```

However, when you try to compile it, Swift's compiler will throw errors. The compiler expects that the variable is assigned a value via the initialization functions.

With primitives, you can specify default values, such as a blank string or a 0 for numbers:

```
var theOldBookName:String = ""
var theOldVersionNumber = 0
var theBookName:String = "Xcode Essentials"
var theVersionNumber = 6
```

Even though we have `theOldBookName` assigned the value; of an empty string or `theoldVersionNumber` with 0, it indicates the presence of a value.

With objects, you simply wouldn't create an object to assign a value, you might want to assign a `nil` value to the variable and then check if the variable was initialized with a value or not. If you try to assign a `nil` as we saw earlier, an error is thrown. Say we have a function that returns the number of books sold when we pass it the book ISBN. We store that result in a variable like this:

```
var booksSold = getBooksSoldByISBN("9781430246626")
```

The only issue with this code is that, if the function could not find the ISBN number or does not have any data on the number of books sold, it returns `nil`. If we try to assign `nil` to the variable `booksSold`, we get an error as the variable cannot hold `nil` values. This is where optionals come in handy. We can mark a variable with `?` at the end of the variable type to tell the compiler that that variable could now have or not have a value (that is, nil):

```
var booksSold as Int?
booksSold = getBooksSoldByISBN("9781430246626")
```

We can also display the number of books sold using the value of this variable:

```
let message = "We sold \(booksSold!) books"
```

The compiler instantly starts to complain because the initializer for string cannot accept `nils`. We need to unwrap the variable. Unwrapping simply means that it transfers the variable from being of the optional type to the underlying value type. To unwrap, we use the `!` symbol:

```
let message = "We sold \(booksSold!) books"
```

If the `!` symbol is used at the beginning of the variable name, it acts as Boolean negation; it unwraps an optional only if used at the end of the variable name.

This is called forced unwrapping. Like the `?` symbol informs the compiler that a value *might* be missing, the `!` symbol tells the compiler that a value exists. If you use a `!` symbol and the value is `nil`, you will definitely have a crash. If you run this line in the playground, you will see that an error that occurred as `booksSold` was `nil`. Set the value of `booksSold` to `50000` and you will see:

```
booksSold = 500000    // getBooksSoldByISBN("9781430246626")
let message = "We sold \(booksSold!) books"
```

It is advised that you should check for `nil` values before using them, so your code might look something like this:

```
booksSold = 500000    // getBooksSoldByISBN("9781430246626")
if booksSold != nil {
    let message = "We sold \(booksSold!) books"
    println(message)
    }
```

Now if you comment the `booksSold` to leave it as `nil`, it will be fine but that adds a lot of redundant code and checking. Swift offers a better way:

```
booksSold = 500000    // getBooksSoldByISBN("9781430246626")
if  let numberBooksSold = booksSold {
    let message = "We sold \(booksSold!) books"
    println(message)
    }
```

This assignment using the `let` statement is called *Optional Binding*. It serves two purposes: it assigns an unwrapped value to the variable and it works with the `if` Boolean conditional statement. To the compiler, it instructs:

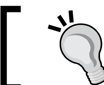
1. Unwrap the value of `booksSold`.
2. Assign it to the `numberBooksSold` variable.
3. If the value was `nil` or not present, then skip that block; otherwise execute the statements in that block.

We could simplify it further and use the return value from the function as follows:

```
if let numberBooksSold = getBooksSoldByISBN("9781430246626") {
  let message = "We sold \(booksSold!) books"
  println(message)
}
```

There is one more interesting operator used with Swift, the `??` (double question marks). This is also used with optionals and is called the *nil coalescing operator*:

```
var myNilVariable:Int?
println("The  value of myNilVariable is \(myNilVariable ?? 100)")
```



This is used to provide a default value if the variable does not have a value (that is nil).

You will see that the output displays 100.

Let's say we have another function called `getSalesForBookWithISBN` that returns a class object. The members of this class that we are interested in are `salesNumbers` and `bookPrice`. We could use them like this:

```
if let bookSales = getSalesForBookWithISBN("9781430246626") {
    if let salesNumbers = bookSales.salesNumbers {
      println("We sold \(salesNumbers) books")
    }
}
```

We could use optional chaining as follows:

```
if let salesNumbers = getSalesForBookWithISBN("9781430246626")?.
salesNumbers {
    println("We sold \(salesNumbers) books")
    }
}
```

Note that we have a `?` symbol at the end of the function; this indicates that, if the value returned is `nil`, then do not check for the remaining properties. So you do not have to first check if a valid class object was returned and then if `salesNumbers` was not `nil`. This helps avoid a big mess of nested `if let` conditional statements. Apple suggests the usage of optional chaining instead of using forced unwrapping, as optional chaining fails gracefully, whereas forced unwrapping can cause runtime errors due to `nil` values.

Here is another example of how your code can work gracefully with Swift using optionals:

```
class theClass{
  var students:[String]?
  var className:String?
  var classCode:String?
}
var cp5290 = theClass()
```

Everything is fine till here, we have an instantiated class object of the `theClass` type but all of its properties are still nil. Pay special attention to the array called students. If we try to get the number of students from the array using the count function, we could simply use something like this:

```
if var studentNumbers = cp5290.students?.count {
    println("We have \(studentNumbers) students in our class")
}
```

One last thing to note about declaring optional variables with respect to using a `?` or `!` symbol is:

```
var name1:String!
var name2:String?
name1 = "Moe"
name2 = "Larry"
println("The name is \(name1) and \(name2)")
```

This outputs `The name is Moe and Optional("Larry").`

All optionals that are declared with the `?` symbol need to be unwrapped for use whereas the variables declared with the `!` symbol do not need forced unwrapping. The preceding code line can be used as follows:

```
println("The name is \(name1) and \(name2!)")
```

This outputs `The name is Moe and Larry`.

With iOS development, you will invariably be using `TableViews` and you will see a lot of `!` and `?`; hopefully, you will be able to understand it better after reading this section.

As you use and get familiar with Swift, you will notice the beauty and ease of use that Swift offers. Also note that Swift is an organically growing language and the concepts discussed could change in future versions.

Differences between iOS 7.x and 8.x SDK

There are some differences between the iOS 7.x and 8.x SDK in the way things work. Here are some examples and resolutions to get them to work.

Location services

Let's look at a simple application that gets the location fix and displays it to the console. The steps for that are quite simple:

1. In a project that you want to add location capabilities to, we need to import `CoreLocation` and declare a couple of functions:

   ```
   import CoreLocation
   ```

2. Next, we need to add the `CLLocationManagerDelegate` protocol to the class definition, which for the `ViewController.swift` file would look something like this:

   ```
   class ViewController: UIViewController, CLLocationManagerDelegate
   {
   ```

3. Now, declare the variable to hold a reference to the location manager:

   ```
   var locManager:CLLocationManager!
   ```

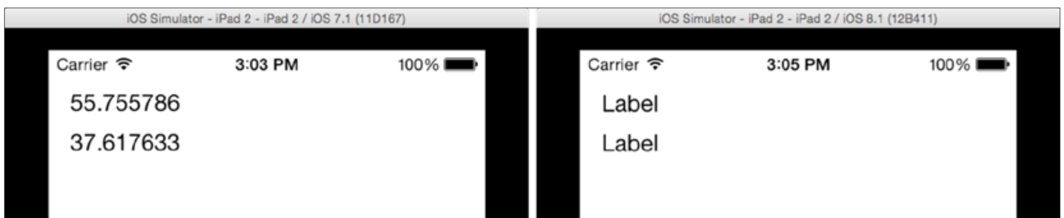
4. Also declare the same variable in a function such as `viewDidLoad` or where you might want to start the updates:

```
if locManager == nil {
  locManager = CLLocationManager()
  locManager.delegate = self
  locManager.desiredAccuracy = kCLLocationAccuracyBest
  locManager.distanceFilter = 10.0
}
locManager.startUpdatingLocation()
```

This would be fine, however; when position updates are received, we need a function such as the following to handle them:

```
func locationManager(manager: CLLocationManager!, didUpdateLocations
locations: [AnyObject]!) {
  let arrLocations = locations as NSArray
  let newLoc = aLocations.lastObject as CLLocation!

  let lat = \(newLoc.coordinate.latitude.description)
  let lon = \(newLoc.coordinate.longitude.description)

     manager.stopUpdatingLocation()
  println("We got \(lat), \(lon)")
}
```

Now run this on the iOS 7.x simulator or device; everything works fine. Run this on an iOS 8.x simulator or device and… nothing works:



You will find that, due to the new privacy policies on iOS 8.x, you have to first obtain permission from the user to be able to access the location services. There are two functions available that allow you to do so:

- `requestWhenInUseAuthorization`: This tries to get permission every time it has to use the application in the foreground—that is, as an active application
- `requestAlwaysAuthorization`: This tries to get the permission for access even when the application is in the background—that is it is not the active application

When you request either of these, it invokes the following function:

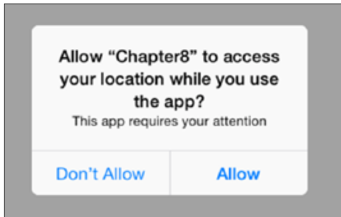
```
locationManager(manager: CLLocationManager!,
didChangeAuthorizationStatus status: CLAuthorizationStatus)
```

The status is one among: `NotDetermined`, `Authorized`, `Denied`, `Restricted`, or `AuthorizedWhenInUse`.

Now, you run the application with the request for authorization functions added as:

```
locManger.requestWhenInUseAuthorization()
```

Just before the `locManager.startUpdatingLocation()` function, it still doesn't do anything; it does not even ask for the authorization dialog. This is because there is a key that you need to set in the `plist` file. The authorization functions will only work as expected if this key is set. For `requestWhenInUseAuthorization`, the key is `NSLocationWhenInUseUsageDescription` and, for `requestAlwaysAuthorization`, the key is `NSLocationAlwaysUsageDescription`. The values for these keys are a message that you can set.



Conditional execution

Whenever you have backwards compatibility in your application, you will have to check for the device iOS version to ensure that you do not call functions that do not exist; otherwise, the app will crash. The easiest way to manage this is to use a conditional `if` statement. First, we can determine the iOS version on which the app is running, as follows:

```
let device = UIDevice.currentDevice()
let iosVersion = device.systemVersion
let OS_8_OR_LATER = NSString(string: iosVersion).doubleValue >= 8.0
```

After this, with a simple `if` statement, we can check the following:

```
if OS_8_OR_LATER {
 println("Only when we run the app on iOS 8.x")
} else {
 println("This would be iOS 7.x")
}
```

If you are uncertain and do not want to have version-specific code, you can also check for the existence of a particular function with `respondsToSelector`. In our preceding example, you could write the code that would work on both iOS 7.x and iOS 8.x without errors:

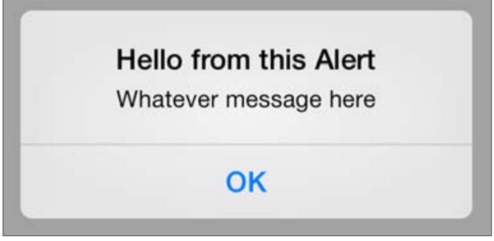
```
if locManager.respondsToSelector("requestWhenInUseAuthorization") {
  locManager.requestWhenInUseAuthorization() // iOS 8.x only
}
```

Displaying alerts

If you have developed for iOS or searched some code samples on the Internet, you will find a solution in the form of something called `UIAlertView`. This is a class that allows you to quickly display an alert on the screen:

```
let vue = UIAlertView()
vue.title = "Hello from this Alert"
vue.message = "Whatever message here"
vue.addButtonWithTitle("OK")
vue.show()
```

The following is the alert screen:



However, to be able to detect and perform some action when the button is clicked, you need to also add `UIAlertViewDelegate` and set `delegate` to `self`, and then write the function to handle the button tap:

```
alertView(_ alertView: UIAlertView,
   clickedButtonAtIndex buttonIndex: Int)
```

With iOS 8.0, `UIAlertView` and `UIActionSheet` have been deprecated and replaced by `UIAlertController`. The `AlertViews` displays a message in the center of its view while `ActionSheet` is displayed anchored to the bottom of the screen. The new `AlertController` can display either style but note that this is available from iOS 8.x onwards only.

Showing an alert with ActionController

You can display an alert using `UIAlertController` and add the actions like this:

```
let actController = UIAlertController(title:"Hello Fom this Alert",
message:"Whatever message", preferredStyle: .Alert)
let actOK = UIAlertAction(title: "OK", style: .Default) {
 (action) in
  // Code to handle the click here
}
actController.addAction(actOK)
self.presentViewController(actController, animated:true) {}
```

Showing ActionSheet with AlertController

You could display `ActionSheet` using `UIAlertController` by setting its `preferredStyle` value to `.ActionSheet` like this:

```
let actController = UIAlertController(title:"Hello from the Alert",
message:"This is the ActionSheet alert", preferredStyle: .ActionSheet)
let actCancel = UIAlertAction(title: "Cancel", style: .Cancel) {
(action) in }
actController.addAction(actCancel)
let actOK = UIAlertAction(title: "OK", style: .Default) { (action) in
}
actController.addAction(actOK)
let actDestroy = UIAlertAction(title: "End of the World", style:
.Destructive) { (action) in }
actController.addAction(actDestroy)

self.presentViewController(actController, animated:true) {}
```

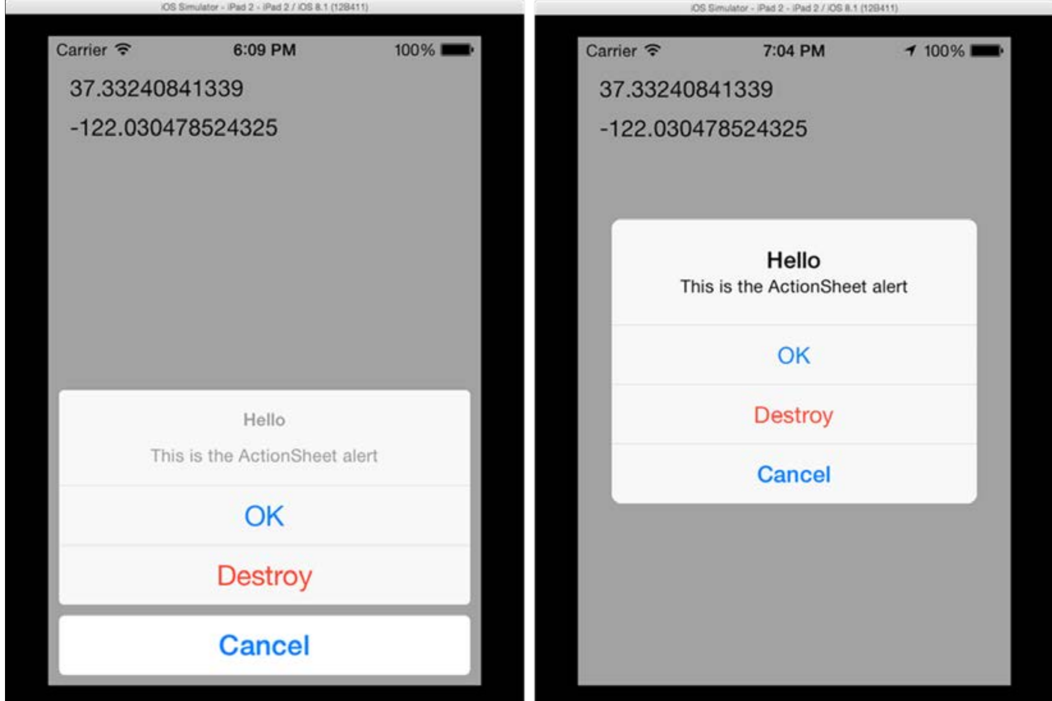
In case you are wondering what those funny `{ (action) in }` lines are all about, they are inline handlers or blocks, as those familiar with Objective-C will know them. There are two ways to pass handlers, either as inline code (seen in the preceding code) or as a reference to the handler, like this:

```
let actOKHandler = { (action:UIAlertAction!) in println("This is the
OK Handler") }
```

Then, add an OK action as follows:

```
actController.addAction(UIAlertAction(title:"OK", style: .Default,
handler: actOKHandler))
```

You can see that the rest of the code is mostly the same. If you were to change `preferredStyle` from `.ActionSheet` to `.Alert`, the same code would work but would display the alert on the screen differently:



Action styles can be one of three button types, namely:

- `.Default`
- `.Cancel`
- `.Destructive`

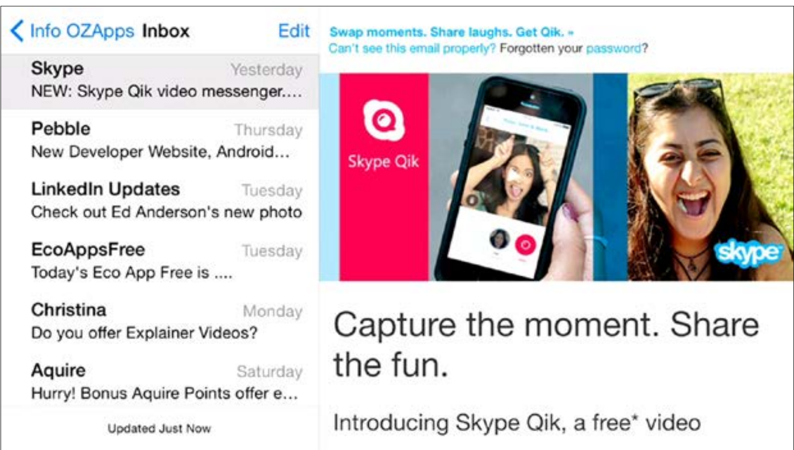
You can also add other UI elements onto `ActionController`; however, we are not going to cover that in this book.

Designing in Interface Builder

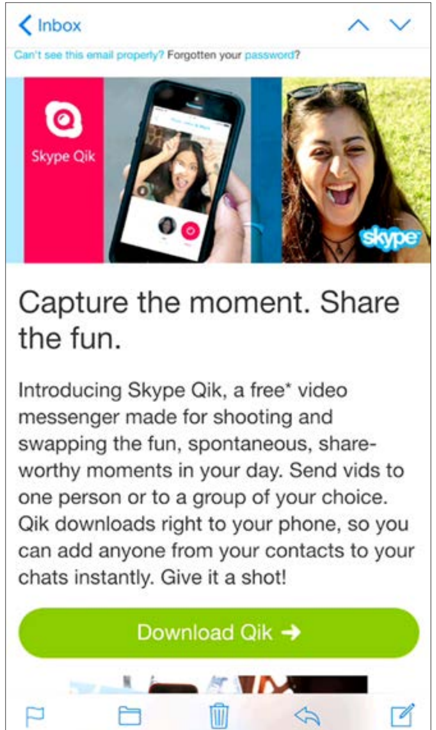
We did discuss Interface Builder in detail in *Chapter 4*, *Interface Builder*; however, there are a couple of tricks and tips that you could use. With the new UI changes since iOS7, buttons have stopped being grey rectangles with a border and some text. They are now transparent areas with just the text sitting there. It is difficult to see their size on the screen, as it would not show up other than as text. This can quickly be resolved by a simple setting, by going to **Editor** | **Canvas** | **Show Layout Rectangles** | **Show Bounds Rectangles**. When these are turned on, you will see the elements and their size.

Adaptive UI

In a previous chapter, we have not even skimmed the surface of what auto layout and size classes can do for your projects. The advantage of these classes is that your UI's adapt to devices based on characteristics as supplied by the class sizes. If you have seen the iPhone 6+, it is the first iOS phone that adapts the screen to both portrait and landscape mode. Here is a screenshot from the mail application. You can see that it has a master-detail view like we are used to on iPads:



From that point on, portrait mode does not have the sidebar, just the e-mail like we are used to on iPhones:



This is because of the new class called `UISplitViewController` that adapts and provides this functionality.

Summary

We have nearly come to the end of the chapter and the book. In this short book, we have tried to cover as much as possible while not subjecting you to a whole lot of information. The content consists of what you might encounter or need to know to get started, make your own apps, and have them App Store-ready. The mileage and learning that each reader gets from the book could be different but we would like to hear from you. We wish you all the best with your applications and your journey with Xcode.

Index

A

Adaptive UI 64-67
AnyObject type
using 38
Any type
using 38
Apple development languages
about 23
Swift 23
application, Interface Builder
autolayout 79
coding 77
connections, managing 83
elements, adding for different dimensions 76
elements, adding for different orientations 76
gesture recognizers, adding 84
items, adding to Table View 71-75
quick help 82
subclassing 80-82
array objects
about 33
initializing 33
values, appending 34
values, enumerating 35
values, inserting 34
values, iterating 35
values, removing 34
arrays 32
assistant editor, Xcode 20
associative index. *See* **dictionaries**
attributes inspector 21
autolayout 79

B

basics, custom controls
about 89
class, creating 90, 91
enhancing 93, 94
framework 96
framework, creating 96-98
properties 91, 92
properties, modifying from inspector 93
Booleans 32
Bool type 32
breakpoint navigator 18
breakpoints
about 102
listing 103
navigating 103
types 102

C

canvas 61
classes
about 51
properties 52
closures
using 50
code signing 123
code snippets library 21
commands, console
b filename:line 105
b function_name 105
br 105
br del num 105
br l 105
br m –c "condition" 105

frame select n 105
thread backtrace 105
thread info 104
components, Xcode environment
debugger 10
editor 10
File View 10
Interface Builder 10
versioning 10
conditional execution
about 134
ActionSheet, displaying with AlertController 136, 137
alerts, displaying 135
alerts, displaying with ActionController 136
connections
about 86
managing 83
connections inspector 21
console 104
console output printing, Swift 29
control flow options, Swift
about 39
for loops 40
if..elseif..else statement 39
if..else statement 39
if statement 39
switch condition 42-46
ternary operator 40
While loop 42
Create a new project option
about 14
project options, setting 15
project properties, setting 15
project type, selecting 14
custom controls
about 89
debugging 98
custom controls, enhancing
about 93, 94
text, adding 95

D

debug gauges 112
debugger 10
debug information 105-107
debug navigator 17
dictionaries 33-35
dictionary object, values
adding 36
iterating 36
removing 36
replacing 36
Double variable 31

E

editor 10
elements, Interface Builder
about 62-64
Exit 63
First Responder 63
View Controller 63
Elvis Operators. *See* **ternary operator**
enhancements, Xcode 6
improved debugger 12
interactive coding, with Playgrounds 11
live design 11
Mac OS X storyboards 11
responsive UI 11
visual debugging 11
enumerations 54, 55
Exit element 63
extensions, Swift
about 55
operator overloading 56

F

file inspector 20
file template library 21
File View 10
First Responder element 63
Float variable 31
for-increment loop 40
for-in loop 40
for loops
about 40
for-in 40
for-increment 40
stepped ranges 41
using, with numeric ranges 41
using, with strings 41

functions
about 47
parameters 48
return values 47

G

gesture recognizers
adding 84

I

identity inspector 21
if..elseif..else statement 39
if..else statement 39
if statement 39
images
using, with quick view 109
infix operator 57
inspector section, Xcode overview
attributes inspector 21
connections inspector 21
file inspector 20
identity inspector 21
quick help inspector 20
size inspector 21
Interface Builder
about 10, 59, 60
Adaptive UI 64-67
application, building 71
connections 86
elements 62-64
interface, creating 61
scenes, adding 68, 69
segues 86
View Controllers, navigating between 69, 70
Int variable 31
iOS 7.x, versus 8.x SDK
about 132
Adaptive UI 138, 139
conditional execution 134
designing, in Interface Builder 138
location services 132-134
issues navigator 17

K

key-value pairs. *See* **dictionaries**

L

location services 132-134
log navigator 18

M

Mac OS X storyboards 11
media library 22
Model-View-Controller (MVC) 77
multiple parameter function 48
multiple parameters
passing 49
multiple-return values function 47

N

named parameters 48
no-return value function 47
numeric ranges
for loops, using with 41

O

object library 22
objects
about 51
class 51
enumerations 54, 55
operator overloading
about 56
infix operator 57
postfix operator 56
optional parameters, default values 49
options, breakpoint
hierarchy view 103
location 104
step into 103
step out 103
step over 103
organizer 124, 125

P

parameters, function
about 48
multiple parameter function 48
multiple parameters, passing 49
named parameters 48
optional parameters, with default values 49
single parameter function 48
Playgrounds
about 11, 26
UI 26-28
postfix operator 56
prefix operator 56
println function 29
project navigator 17
project section, Xcode
about 17
breakpoint navigator 18
debug navigator 17
issues navigator 17
log navigator 18
project navigator 17
search navigator 17
symbol navigator 17
tests navigator 17
project settings, template application
about 117
application icons 118
Deployment Info section 118
Identity section 117
launch images 119
Linked frameworks and libraries section 120
project-wide properties
code signing 123
organizer 124, 125
setting 122
properties 52

Q

quick help inspector 20
quick look functionality
variables, visualizing with 107, 108
quick view
images, using with 109

R

REPL mode (Read-Evaluate-Print-Loop) 105
return values, function
about 47
multiple-return values function 47
no-return value function 47
single-return value function 47

S

scenes
adding 68, 69
search navigator 17
segues 59, 86
simulator
about 115
device, selecting 116
single parameter function 48
single-return value function 47
size inspector 21
source code editors, Xcode overview
features 18, 19
stepped ranges, for loops 41
storyboards 59
strings
about 37
appending 38
for loops, using with 41
formatting 38
Swift
about 10, 23
concepts 127-132
console output printing 29
control flow options 39
extensions 55
features 23
functions 47
learning 29
objects 51
URL 23
variables 29
switch condition 42-46
symbol navigator 17

T

template application, preparing for distribution
about 116
Build tabs 121
Capabilities tab 120
Info tab 120
project settings, configuring 117
Targets tab 121, 122
ternary operator 40
tests navigator 17
tuples
about 37
members, accessing of 37
named access, to tuple members 37

V

variables
visualizing, with quick look functionality 107, 108
variables, Swift
about 29
AnyObject type 38
Any type 38
arrays 32
Booleans 32
declaring 30
dictionaries 33
different types, adding 31, 32
Double 31
Float 31
Int 31
strings 37
tuples 37
types 30
versioning 10
View Controller
about 63
navigating between 69, 70
view hierarchy
debugging 110, 111
views 59

W

While loop 42

X

Xcode
about 7
features 10
installing 7-9
requisites 7-9
starting 13
Xcode 6
enhancements 10-12
Xcode environment
components 10
Xcode overview
about 15, 16
assistant editor section 18, 20
project section 17, 18
source code editors 18
source code editors, features 18
utility/inspector section 20-22

Thank you for buying

Xcode 6 Essentials

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at `www.packtpub.com`.

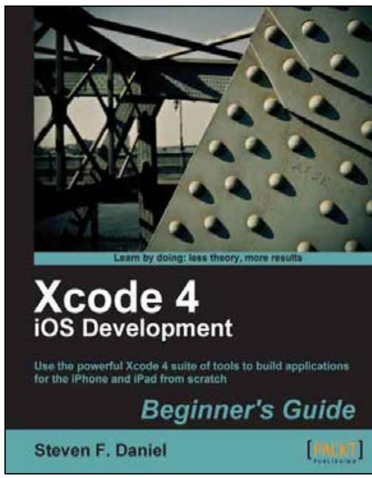
About Packt Enterprise

In 2010, Packt launched two new brands, Packt Enterprise and Packt Open Source, in order to continue its focus on specialization. This book is part of the Packt Enterprise brand, home to books published on enterprise software – software created by major vendors, including (but not limited to) IBM, Microsoft, and Oracle, often for use in other corporations. Its titles will offer information relevant to a range of users of this software, including administrators, developers, architects, and end users.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

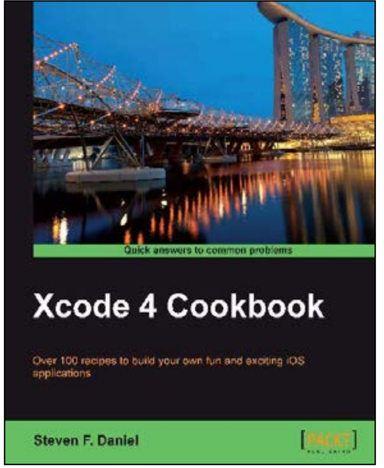


Xcode 4 iOS Development Beginner's Guide

ISBN: 978-1-84969-130-7 Paperback: 432 pages

Use the powerful Xcode 4 suite of tools to build applications for the iPhone and iPad from scratch

1. Learn how to use Xcode 4 to build simple, yet powerful applications with ease.
2. Each chapter builds on what you have learned already.
3. Learn to add audio and video playback to your applications.
4. Plentiful step-by-step examples, images, and diagrams to get you up to speed in no time with helpful hints along the way.



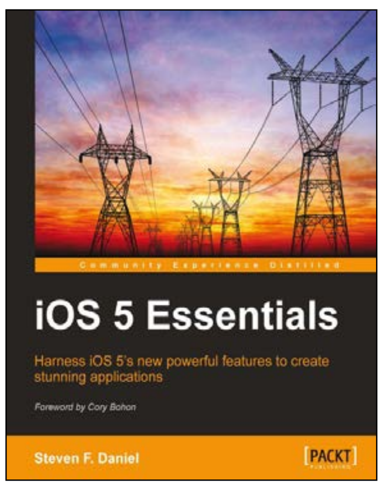
Xcode 4 Cookbook

ISBN: 978-1-84969-334-9 Paperback: 402 pages

Over 100 recipes to build your own fun and exciting iOS applications

1. Learn how to go about developing some simple, yet powerful applications with ease using recipes and example code.
2. Teaches how to use the features of iOS 6 to integrate Facebook, Twitter, iCloud, and Airplay into your applications.
3. Lots of step-by-step recipe examples with ample screenshots right through to application deployment to the Apple App Store to get you up to speed in no time, with helpful hints along the way.

Please check **www.PacktPub.com** for information on our titles

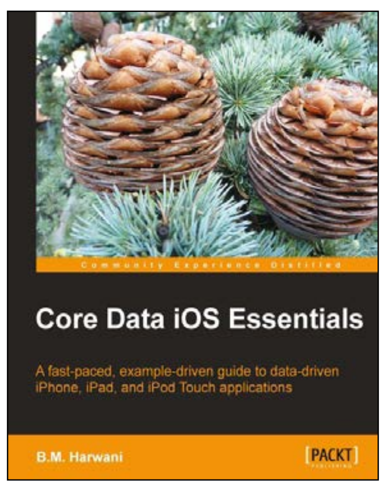


iOS 5 Essentials

ISBN: 978-1-84969-226-7 Paperback: 252 pages

Harness iOS 5's new powerful features to create stunning applications

1. Integrate iCloud, Twitter, and AirPlay into your applications.
2. Lots of step-by-step examples, images, and diagrams to get you up to speed in no time with helpful hints along the way.
3. Each chapter explains iOS 5's new features in-depth, while providing you with enough practical examples to help incorporate these features in your apps.
4. From the author of Xcode 4 iOS development.



Core Data iOS Essentials

ISBN: 978-1-84969-094-2 Paperback: 340 pages

A fast-paced, example-driven guide to data-driven iPhone, iPad, and iPod Touch applications

1. Covers the essential skills you need for working with Core Data in your applications.
2. Particularly focused on developing fast, light weight data-driven iOS applications.
3. Builds a complete example application. Every technique is shown in context.
4. Completely practical with clear, step-by-step instructions.

Please check **www.PacktPub.com** for information on our titles