

iOS 9 App Development



Essentials

iOS 9 App Development Essentials

iOS 9 App Development Essentials – First Edition

© 2015 Neil Smyth/eBookFrenzy. All Rights Reserved.

This book is provided for personal use only. Unauthorized use, reproduction and/or distribution strictly prohibited. All rights reserved.

The content of this book is provided for informational purposes only. Neither the publisher nor the author offers any warranties or representation, express or implied, with regard to the accuracy of information contained in this book, nor do they accept any liability for any loss or damage arising from any errors or omissions.

This book contains trademarked terms that are used solely for editorial purposes and to the benefit of the respective trademark owner. The terms used within this book are not intended as infringement of any trademarks.

Rev: 1.0



Table of Contents

1. Start Here	1
1.1 For New iOS Developers	1
1.2 For iOS 8 Developers	1
1.3 Source Code Download	2
1.4 Learn to Develop watchOS Apps	2
1.5 Feedback.....	2
1.6 Errata	2
2. Joining the Apple Developer Program	3
2.1 Downloading Xcode 7 and the iOS 9 SDK	3
2.2 Apple Developer Program	3
2.3 When to Enroll in the Apple Developer Program?	3
2.4 Enrolling in the Apple Developer Program	3
2.5 Summary.....	4
3. Installing Xcode 7 and the iOS 9 SDK.....	5
3.1 Identifying if you have an Intel or PowerPC based Mac	5
3.2 Installing Xcode 7 and the iOS 9 SDK	5
3.3 Starting Xcode.....	5
3.4 Adding Your Apple ID to the Xcode Preferences	6
3.5 Developer and Distribution Signing Identities	6
4. A Guided Tour of Xcode 7	9
4.1 Starting Xcode 7.....	9
4.2 Creating the iOS App User Interface.....	12
4.3 Changing Component Properties.....	14
4.4 Adding Objects to the User Interface	14
4.5 Building and Running an iOS 9 App in Xcode 7.....	17
4.6 Running the App on a Physical iOS Device	18
4.7 Managing Devices and Simulators.....	18
4.8 Dealing with Build Errors	19
4.9 Monitoring Application Performance	19
4.10 An Exploded View of the User Interface Layout Hierarchy	19
4.11 Summary.....	20
5. An Introduction to Xcode 7 Playgrounds	21
5.1 What is a Playground?	21
5.2 Creating a New Playground	21
5.3 A Basic Swift Playground Example	22
5.4 Viewing Results.....	23
5.5 Enabling the Timeline Slider	23
5.6 Adding Rich Text Comments.....	24
5.7 Working with Playground Pages.....	25
5.8 Working with UIKit in Playgrounds	25
5.9 Adding Resources to a Playground	26
5.10 Working with Enhanced Live Views	27
5.11 When to Use Playgrounds	29
5.12 Summary.....	29
6. Swift Data Types, Constants and Variables	31
6.1 Using a Swift Playground	31
6.2 Swift Data Types	31
6.2.1 Integer Data Types	32
6.2.2 Floating Point Data Types	32
6.2.3 Bool Data Type	32
6.2.4 Character Data Type.....	32

6.2.5 String Data Type	32
6.2.6 Special Characters/Escape Sequences	33
6.3 Swift Variables	33
6.4 Swift Constants	33
6.5 Declaring Constants and Variables	34
6.6 Type Annotations and Type Inference	34
6.7 The Swift Tuple	34
6.8 The Swift Optional Type	35
6.9 Type Casting and Type Checking	37
6.10 Summary	39
7. Swift Operators and Expressions	41
7.1 Expression Syntax in Swift	41
7.2 The Basic Assignment Operator	41
7.3 Swift Arithmetic Operators	41
7.4 Compound Assignment Operators	42
7.5 Increment and Decrement Operators	42
7.6 Comparison Operators	43
7.7 Boolean Logical Operators	43
7.8 Range Operators	44
7.9 The Ternary Operator	44
7.10 Bitwise Operators	44
7.10.1 Bitwise NOT	44
7.10.2 Bitwise AND	45
7.10.3 Bitwise OR	45
7.10.4 Bitwise XOR	45
7.10.5 Bitwise Left Shift	46
7.10.6 Bitwise Right Shift	46
7.11 Compound Bitwise Operators	46
7.12 Summary	47
8. Swift Flow Control	49
8.1 Looping Flow Control	49
8.2 The Swift for Statement	49
8.2.1 The Condition-Increment for Loop	49
8.2.2 The for-in Loop	49
8.2.3 The while Loop	50
8.3 The repeat ... while loop	51
8.4 Breaking from Loops	51
8.5 The continue Statement	51
8.6 Conditional Flow Control	52
8.7 Using the if Statement	52
8.8 Using if ... else ... Statements	52
8.9 Using if ... else if ... Statements	53
8.10 The guard Statement	53
8.11 Summary	54
9. The Swift Switch Statement	55
9.1 Why Use a switch Statement?	55
9.2 Using the switch Statement Syntax	55
9.3 A Swift switch Statement Example	55
9.4 Combining case Statements	56
9.5 Range Matching in a switch Statement	56
9.6 Using the where statement	57
9.7 Fallthrough	57
9.8 Summary	58
10. An Overview of Swift 2 Functions, Methods and Closures	59
10.1 What is a Function?	59

10.2 What is a Method?	59
10.3 How to Declare a Swift Function	59
10.4 Calling a Swift Function.....	60
10.5 Local and External Parameter Names.....	60
10.6 Declaring Default Function Parameters.....	61
10.7 Returning Multiple Results from a Function.....	61
10.8 Variable Numbers of Function Parameters	61
10.9 Parameters as Variables	62
10.10 Working with In-Out Parameters.....	62
10.11 Functions as Parameters.....	63
10.12 Closure Expressions	64
10.13 Closures in Swift	65
10.14 Summary.....	66
11. The Basics of Object Oriented Programming in Swift	67
11.1 What is an Object?.....	67
11.2 What is a Class?	67
11.3 Declaring a Swift Class	67
11.4 Adding Instance Properties to a Class.....	67
11.5 Defining Methods	68
11.6 Declaring and Initializing a Class Instance	69
11.7 Initializing and Deinitializing a Class Instance	69
11.8 Calling Methods and Accessing Properties	70
11.9 Stored and Computed Properties	70
11.10 Using self in Swift.....	71
11.11 Summary.....	72
12. An Introduction to Swift Subclassing and Extensions	73
12.1 Inheritance, Classes and Subclasses	73
12.2 A Swift Inheritance Example	73
12.3 Extending the Functionality of a Subclass	74
12.4 Overriding Inherited Methods.....	74
12.5 Initializing the Subclass	75
12.6 Using the SavingsAccount Class.....	75
12.7 Swift Class Extensions.....	76
12.8 Summary.....	76
13. Working with Array and Dictionary Collections in Swift.....	77
13.1 Mutable and Immutable Collections	77
13.2 Swift Array Initialization.....	77
13.3 Working with Arrays in Swift	78
13.3.1 Array Item Count	78
13.3.2 Accessing Array Items	78
13.4 Appending Items to an Array.....	78
13.4.1 Inserting and Deleting Array Items.....	78
13.4.2 Array Iteration	78
13.5 Creating Mixed Type Arrays.....	79
13.6 Swift Dictionary Collections	79
13.7 Swift Dictionary Initialization.....	80
13.7.1 Dictionary Item Count	80
13.7.2 Accessing and Updating Dictionary Items	80
13.7.3 Adding and Removing Dictionary Entries	81
13.7.4 Dictionary Iteration	81
13.8 Summary.....	81
14. Understanding Error Handling in Swift 2.....	83
14.1 Understanding Error Handling.....	83
14.2 Declaring Error Types.....	83
14.3 Throwing an Error.....	83

14.4 Calling Throwing Methods and Functions	84
14.5 Accessing the Error Object.....	85
14.6 Disabling Error Catching	85
14.7 Using the defer Statement	85
14.8 Summary.....	86
15. The iOS 9 Application and Development Architecture	87
15.1 An Overview of the iOS 9 Operating System Architecture	87
15.2 Model View Controller (MVC)	87
15.3 The Target-Action pattern, IBOutlet and IBActions	88
15.4 Subclassing.....	88
15.5 Delegation.....	88
15.6 Summary.....	89
16. Creating an Interactive iOS 9 App	91
16.1 Creating the New Project.....	91
16.2 Creating the User Interface	91
16.3 Building and Running the Sample Application.....	93
16.4 Adding Actions and Outlets	93
16.5 Building and Running the Finished Application	96
16.6 Hiding the Keyboard	96
16.7 Summary.....	97
17. Understanding iOS 9 Views, Windows and the View Hierarchy	99
17.1 An Overview of Views and the UIKit Class Hierarchy	99
17.2 The UIWindow Class	99
17.3 The View Hierarchy.....	99
17.4 Viewing Hierarchy Ancestors in Interface Builder	100
17.5 View Types.....	101
17.5.1 The Window	101
17.5.2 Container Views.....	101
17.5.3 Controls	101
17.5.4 Display Views.....	101
17.5.5 Text and Web Views.....	101
17.5.6 Navigation Views and Tab Bars.....	101
17.5.7 Alert Views	101
17.6 Summary.....	101
18. An Introduction to Auto Layout in iOS 9	103
18.1 An Overview of Auto Layout.....	103
18.2 Alignment Rects.....	104
18.3 Intrinsic Content Size	104
18.4 Content Hugging and Compression Resistance Priorities.....	104
18.5 Three Ways to Create Constraints.....	104
18.6 Constraints in more Detail.....	104
18.7 Summary.....	105
19. Working with iOS 9 Auto Layout Constraints in Interface Builder	107
19.1 A Simple Example of Auto Layout in Action	107
19.2 Enabling and Disabling Auto Layout in Interface Builder.....	107
19.3 The Auto Layout Features of Interface Builder.....	111
19.3.1 Suggested Constraints	111
19.3.2 Visual Cues	112
19.3.3 Highlighting Constraint Problems	112
19.3.4 Viewing, Editing and Deleting Constraints	114
19.4 Creating New Constraints in Interface Builder	115
19.5 Adding Aspect Ratio Constraints	116
19.6 Resolving Auto Layout Problems	116
19.7 Summary.....	117

20. An iOS 9 Auto Layout Example.....	119
20.1 Preparing the Project.....	119
20.2 Designing the User Interface	119
20.3 Adding Auto Layout Constraints	120
20.4 Adjusting Constraint Priorities	121
20.5 Testing the Application	123
20.6 Summary.....	123
21. Implementing iOS 9 Auto Layout Constraints in Code.....	125
21.1 Creating Constraints in Code	125
21.2 Adding a Constraint to a View	126
21.3 Turning off Auto Resizing Translation.....	126
21.4 An Example Application	127
21.5 Creating the Views.....	127
21.6 Creating and Adding the Constraints.....	127
21.7 Removing Constraints.....	129
21.8 Summary.....	129
22. Implementing Cross-Hierarchy Auto Layout Constraints in iOS 9	131
22.1 The Example Application	131
22.2 Establishing Outlets	132
22.3 Writing the Code to Remove the Old Constraint.....	132
22.4 Adding the Cross Hierarchy Constraint.....	132
22.5 Testing the Application	133
22.6 Summary.....	133
23. Understanding the iOS 9 Auto Layout Visual Format Language	135
23.1 Introducing the Visual Format Language.....	135
23.2 Visual Format Language Examples	135
23.3 Using the constraintsWithVisualFormat Method.....	136
23.4 Summary.....	137
24. Using Size Classes to Design Adaptable Universal iOS User Interfaces	139
24.1 Understanding Size Classes	139
24.2 Size Classes in Interface Builder.....	139
24.3 Setting “Any” Defaults	139
24.4 Working with Size Classes in Interface Builder	140
24.5 A Universal User Interface Tutorial.....	141
24.6 Designing the iPad Layout	141
24.7 Adding Universal Image Assets.....	143
24.8 Designing the iPhone Layout	143
24.9 Adding a Size Class Specific Image File	144
24.10 Removing Redundant Constraints	145
24.11 Testing the Application.....	145
24.12 Summary.....	145
25. Using Storyboards in Xcode 7	147
25.1 Creating the Storyboard Example Project	147
25.2 Accessing the Storyboard	147
25.3 Adding Scenes to the Storyboard	148
25.4 Configuring Storyboard Segues	149
25.5 Configuring Storyboard Transitions	149
25.6 Associating a View Controller with a Scene.....	150
25.7 Passing Data Between Scenes	150
25.8 Unwinding Storyboard Segues.....	151
25.9 Triggering a Storyboard Segue Programmatically	152
25.10 Summary.....	152
26. Organizing Scenes over Multiple Storyboard Files	153

26.1 Organizing Scenes into Multiple Storyboards.....	153
26.2 Establishing a Connection between Different Storyboards.....	155
26.3 Summary.....	155
27. Using Xcode 7 Storyboards to Create an iOS 9 Tab Bar Application	157
27.1 An Overview of the Tab Bar.....	157
27.2 Understanding View Controllers in a Multiview Application.....	157
27.3 Setting up the Tab Bar Example Application	157
27.4 Reviewing the Project Files.....	158
27.5 Adding the View Controllers for the Content Views.....	158
27.6 Adding the Tab Bar Controller to the Storyboard.....	158
27.7 Designing the View Controller User interfaces.....	159
27.8 Configuring the Tab Bar Items	160
27.9 Building and Running the Application	160
27.10 Summary.....	161
28. An Overview of iOS 9 Table Views and Xcode 7 Storyboards	163
28.1 An Overview of the Table View	163
28.2 Static vs. Dynamic Table Views.....	163
28.3 The Table View Delegate and dataSource	163
28.4 Table View Styles	163
28.5 Self-Sizing Table Cells.....	164
28.6 Dynamic Type	164
28.7 Table View Cell Styles	165
28.8 Table View Cell Reuse.....	166
28.9 Summary.....	166
29. Using Xcode 7 Storyboards to Build Dynamic TableViews.....	167
29.1 Creating the Example Project	167
29.2 Adding the TableView Controller to the Storyboard	167
29.3 Creating the UITableViewController and UITableViewCell Subclasses.....	168
29.4 Declaring the Cell Reuse Identifier	168
29.5 Designing a Storyboard UITableView Prototype Cell.....	169
29.6 Modifying the AttractionTableViewCell Class.....	169
29.7 Creating the Table View Datasource	169
29.8 Downloading and Adding the Image Files	171
29.9 Compiling and Running the Application	172
29.10 Summary.....	172
30. Implementing iOS 9 TableView Navigation using Storyboards in Xcode 7.....	173
30.1 Understanding the Navigation Controller	173
30.2 Adding the New Scene to the Storyboard	173
30.3 Adding a Navigation Controller	174
30.4 Establishing the Storyboard Segue	174
30.5 Modifying the AttractionDetailViewController Class	174
30.6 Using prepareForSegue to Pass Data between Storyboard Scenes.....	175
30.7 Testing the Application	176
30.8 Summary.....	176
31. Working with the iOS 9 Stack View Class	177
31.1 Introducing the UIStackView Class	177
31.2 Understanding Subviews and Arranged Subviews	178
31.3 StackView Configuration Options.....	179
31.3.1 <i>axis</i>	179
31.3.2 <i>Distribution</i>	179
31.3.3 <i>spacing</i>	180
31.3.4 <i>alignment</i>	180
31.3.5 <i>baseLineRelativeArrangement</i>	183
31.3.6 <i>layoutMarginsRelativeArrangement</i>	183

31.4 Creating a Stack View in Code	183
31.5 Adding Subviews to an Existing Stack View	183
31.6 Hiding and Removing Subviews	183
31.7 Summary.....	184
32. An iOS 9 Stack View Tutorial.....	185
32.1 About the Stack View Example App.....	185
32.2 Creating the First Stack View	185
32.3 Creating the Banner Stack View	187
32.4 Adding the Switch Stack Views	188
32.5 Creating the Top Level Stack View.....	188
32.6 Adding the Button Stack View	189
32.7 Adding the Final Subviews to the Top Level Stack View	190
32.8 Dynamically Adding and Removing Subviews	192
32.9 Summary.....	193
33. An iOS 9 Split View Master-Detail Example	195
33.1 An Overview of Split View and Popovers.....	195
33.2 About the Example Split View Project	195
33.3 Creating the Project.....	195
33.4 Reviewing the Project.....	195
33.5 Configuring Master View Items	196
33.6 Configuring the Detail View Controller.....	198
33.7 Connecting Master Selections to the Detail View	198
33.8 Modifying the DetailViewController Class.....	198
33.9 Testing the Application	199
33.10 Summary.....	199
34. A Guide to Multitasking in iOS 9	201
34.1 Using iPad Multitasking	201
34.2 Picture-In-Picture Multitasking.....	202
34.3 iPad Devices with Multitasking Support	202
34.4 Multitasking and Size Classes	203
34.5 Multitasking and the Master-Detail Split View.....	204
34.6 Handling Multitasking in Code.....	204
34.6.1 <i>willTransitionToTraitCollection(_:withTransitionCoordinator:)</i>	205
34.6.2 <i>viewWillTransitionToSize(_:withTransitionCoordinator:)</i>	205
34.6.3 <i>traitCollectionDidChange(_:)</i>	205
34.7 Lifecycle Method Calls	206
34.8 Enabling Multitasking Support	206
34.9 Opting Out of Multitasking	206
34.10 Summary.....	207
35. An iOS 9 Multitasking Example	209
35.1 Creating the Multitasking Example Project	209
35.2 Adding the Image Files.....	209
35.3 Designing the Regular Width Size Class Layout	209
35.4 Designing the Compact Width Size Class	211
35.5 Testing the Project in a Multitasking Environment	213
35.6 Summary.....	214
36. Implementing a Page based iOS 9 Application with UIPageViewController	215
36.1 The UIPageViewController Class.....	215
36.2 The UIPageViewController DataSource	215
36.3 Navigation Orientation	215
36.4 Spine Location	215
36.5 The UIPageViewController Delegate Protocol.....	216
36.6 Summary.....	216
37. An Example iOS 9 UIPageViewController Application	217

37.1 The Xcode Page-based Application Template	217
37.2 Creating the Project.....	217
37.3 Adding the Content View Controller	217
37.4 Creating the Data Model	218
37.5 Initializing the UINavigationController	221
37.6 Running the UINavigationController Application	222
37.7 Summary.....	223
38. Working with Directories in Swift on iOS 9	225
38.1 The Application Documents Directory.....	225
38.2 The NSFileManager, NSFileHandle and NSData Classes	225
38.3 Understanding Pathnames in Swift	225
38.4 Obtaining a Reference to the Default NSFileManager Object.....	225
38.5 Identifying the Current Working Directory	226
38.6 Identifying the Documents Directory	226
38.7 Identifying the Temporary Directory	226
38.8 Changing Directory	226
38.9 Creating a New Directory	227
38.10 Deleting a Directory.....	227
38.11 Listing the Contents of a Directory	227
38.12 Getting the Attributes of a File or Directory.....	228
39. Working with Files in Swift on iOS 9	231
39.1 Obtaining an NSFileManager Instance Reference	231
39.2 Checking for the Existence of a File	231
39.3 Comparing the Contents of Two Files.....	231
39.4 Checking if a File is Readable/Writable/Executable/Deletable	231
39.5 Moving/Renaming a File.....	232
39.6 Copying a File.....	232
39.7 Removing a File.....	232
39.8 Creating a Symbolic Link.....	232
39.9 Reading and Writing Files with NSFileManager.....	232
39.10 Working with Files using the NSFileHandle Class	233
39.11 Creating an NSFileHandle Object.....	233
39.12 NSFileHandle File Offsets and Seeking	233
39.13 Reading Data from a File.....	234
39.14 Writing Data to a File	234
39.15 Truncating a File	234
39.16 Summary.....	234
40. iOS 9 Directory Handling and File I/O in Swift – A Worked Example	237
40.1 The Example Application	237
40.2 Setting up the Application Project.....	237
40.3 Designing the User Interface	237
40.4 Checking the Data File on Application Startup	238
40.5 Implementing the Action Method	238
40.6 Building and Running the Example	239
41. Preparing an iOS 9 App to use iCloud Storage.....	241
41.1 iCloud Data Storage Services	241
41.2 Preparing an Application to Use iCloud Storage.....	241
41.3 Enabling iCloud Support for an iOS 9 Application.....	241
41.4 Reviewing the iCloud Entitlements File	242
41.5 Accessing Multiple Ubiquity Containers	242
41.6 Ubiquity Container URLs.....	242
41.7 Summary.....	242
42. Managing Files using the iOS 9 UIDocument Class	243
42.1 An Overview of the UIDocument Class.....	243

42.2 Subclassing the UIDocument Class	243
42.3 Conflict Resolution and Document States	243
42.4 The UIDocument Example Application	243
42.5 Creating a UIDocument Subclass	244
42.6 Designing the User Interface	244
42.7 Implementing the Application Data Structure.....	244
42.8 Implementing the contentsForType Method	245
42.9 Implementing the loadFromContents Method	245
42.10 Loading the Document at App Launch	245
42.11 Saving Content to the Document	247
42.12 Testing the Application	248
42.13 Summary.....	248
43. Using iCloud Storage in an iOS 9 Application	249
43.1 iCloud Usage Guidelines	249
43.2 Preparing the iCloudStore Application for iCloud Access	249
43.3 Configuring the View Controller	249
43.4 Implementing the viewDidLoad Method.....	250
43.5 Implementing the metadataQueryDidFinishGathering Method	251
43.6 Implementing the saveDocument Method	253
43.7 Enabling iCloud Document and Data Storage.....	254
43.8 Running the iCloud Application	254
43.9 Reviewing and Deleting iCloud Based Documents	254
43.10 Making a Local File Ubiquitous	255
43.11 Summary.....	255
44. Synchronizing iOS 9 Key-Value Data using iCloud	257
44.1 An Overview of iCloud Key-Value Data Storage	257
44.2 Sharing Data Between Applications.....	257
44.3 Data Storage Restrictions	258
44.4 Conflict Resolution.....	258
44.5 Receiving Notification of Key-Value Changes	258
44.6 An iCloud Key-Value Data Storage Example	258
44.7 Enabling the Application for iCloud Key Value Data Storage.....	258
44.8 Designing the User Interface	258
44.9 Implementing the View Controller	259
44.10 Modifying the viewDidLoad Method	259
44.11 Implementing the Notification Method	260
44.12 Implementing the saveData Method.....	260
44.13 Testing the Application	260
45. iOS 9 Database Implementation using SQLite	263
45.1 What is SQLite?.....	263
45.2 Structured Query Language (SQL)	263
45.3 Trying SQLite on MacOS X	263
45.4 Preparing an iOS Application Project for SQLite Integration	264
45.5 SQLite, Swift and Wrappers	264
45.6 Key FMDB Classes	265
45.7 Creating and Opening a Database	265
45.8 Creating a Database Table	265
45.9 Extracting Data from a Database Table	265
45.10 Closing a SQLite Database	266
45.11 Summary.....	266
46. An Example SQLite based iOS 9 Application using Swift and FMDB	267
46.1 About the Example SQLite Application.....	267
46.2 Creating and Preparing the SQLite Application Project	267
46.3 Checking Out the FMDB Source Code	267
46.4 Designing the User Interface	268

46.5 Creating the Database and Table	269
46.6 Implementing the Code to Save Data to the SQLite Database	270
46.7 Implementing Code to Extract Data from the SQLite Database	271
46.8 Building and Running the Application	271
46.9 Summary.....	272
47. Working with iOS 9 Databases using Core Data	273
47.1 The Core Data Stack.....	273
47.2 Managed Objects.....	273
47.3 Managed Object Context.....	273
47.4 Managed Object Model	274
47.5 Persistent Store Coordinator	274
47.6 Persistent Object Store	274
47.7 Defining an Entity Description	274
47.8 Obtaining the Managed Object Context.....	275
47.9 Getting an Entity Description	275
47.10 Generating a Managed Object Subclass	275
47.11 Setting the Attributes of a Managed Object.....	276
47.12 Saving a Managed Object	276
47.13 Fetching Managed Objects	276
47.14 Retrieving Managed Objects based on Criteria	276
47.15 Accessing the Data in a Retrieved Managed Object	277
47.16 Summary.....	277
48. An iOS 9 Core Data Tutorial	279
48.1 The Core Data Example Application	279
48.2 Creating a Core Data based Application	279
48.3 Creating the Entity Description.....	279
48.4 Generating the Managed Object Subclass.....	280
48.5 Designing the User Interface	281
48.6 Accessing the Managed Object Context.....	281
48.7 Saving Data to the Persistent Store using Core Data.....	282
48.8 Retrieving Data from the Persistent Store using Core Data	282
48.9 Building and Running the Example Application.....	283
48.10 Summary.....	283
49. An Introduction to CloudKit Data Storage on iOS 9	285
49.1 An Overview of CloudKit.....	285
49.2 CloudKit Containers	285
49.3 CloudKit Public Database	285
49.4 CloudKit Private Databases.....	285
49.5 Data Storage and Transfer Quotas	285
49.6 CloudKit Records.....	286
49.7 CloudKit Record IDs	287
49.8 CloudKit References.....	287
49.9 CloudKit Assets	287
49.10 Record Zones	287
49.11 CloudKit Subscriptions	288
49.12 Obtaining iCloud User Information.....	288
49.13 CloudKit Dashboard	289
49.14 Summary.....	289
50. An iOS 9 CloudKit Example	291
50.1 About the Example CloudKit Project	291
50.2 Creating the CloudKit Example Project.....	291
50.3 Designing the User Interface	291
50.4 Establishing Outlets and Actions	292
50.5 Accessing the Public Database.....	293
50.6 Hiding the Keyboard	293

50.7 Implementing the selectPhoto method.....	294
50.8 Saving a Record to the Cloud Database.....	295
50.9 Implementing the notifyUser Method.....	296
50.10 Testing the Record Saving Method.....	296
50.11 Searching for Cloud Database Records.....	296
50.12 Updating Cloud Database Records.....	298
50.13 Deleting a Cloud Record.....	298
50.14 Testing the Application.....	299
50.15 Summary.....	299
51. An iOS 9 CloudKit Subscription Example.....	301
51.1 Push Notifications and CloudKit Subscriptions.....	301
51.2 Registering an App to Receive Push Notifications.....	301
51.3 Configuring a CloudKit Subscription.....	302
51.4 Handling Remote Notifications.....	303
51.5 Implementing the didReceiveRemoteNotification Method.....	303
51.6 Fetching a Record From a Cloud Database.....	304
51.7 Completing the didFinishLaunchingWithOptions Method.....	304
51.8 Testing the Application.....	305
51.9 Summary.....	305
52. An Overview of iOS 9 Multitouch, Taps and Gestures.....	307
52.1 The Responder Chain.....	307
52.2 Forwarding an Event to the Next Responder.....	307
52.3 Gestures.....	307
52.4 Taps.....	308
52.5 Touches.....	308
52.6 Touch Notification Methods.....	308
52.6.1 touchesBegan method.....	308
52.6.2 touchesMoved method.....	308
52.6.3 touchesEnded method.....	308
52.6.4 touchesCancelled method.....	308
52.7 Touch Prediction.....	308
52.8 Touch Coalescing.....	308
52.9 3D Touch.....	309
52.10 iPad Pro and the Apple Pencil Stylus.....	309
52.11 Summary.....	309
53. An Example iOS 9 Touch, Multitouch and Tap Application.....	311
53.1 The Example iOS 9 Tap and Touch Application.....	311
53.2 Creating the Example iOS Touch Project.....	311
53.3 Designing the User Interface.....	311
53.4 Enabling Multitouch on the View.....	311
53.5 Implementing the touchesBegan Method.....	312
53.6 Implementing the touchesMoved Method.....	312
53.7 Implementing the touchesEnded Method.....	312
53.8 Getting the Coordinates of a Touch.....	313
53.9 Building and Running the Touch Example Application.....	313
53.10 Checking for Touch Predictions.....	313
53.11 Accessing Coalesced Touches.....	314
53.12 Summary.....	314
54. Detecting iOS 9 Touch Screen Gesture Motions.....	315
54.1 The Example iOS 9 Gesture Application.....	315
54.2 Creating the Example Project.....	315
54.3 Designing the Application User Interface.....	315
54.4 Implementing the touchesBegan Method.....	316
54.5 Implementing the touchesMoved Method.....	316
54.6 Implementing the touchesEnded Method.....	316

54.7 Building and Running the Gesture Example	317
54.8 Summary.....	317
55. Identifying Gestures using iOS 9 Gesture Recognizers	319
55.1 The UIGestureRecognizer Class	319
55.2 Recognizer Action Messages	319
55.3 Discrete and Continuous Gestures	319
55.4 Obtaining Data from a Gesture.....	319
55.5 Recognizing Tap Gestures.....	320
55.6 Recognizing Pinch Gestures	320
55.7 Detecting Rotation Gestures	320
55.8 Recognizing Pan and Dragging Gestures	320
55.9 Recognizing Swipe Gestures	320
55.10 Recognizing Long Touch (Touch and Hold) Gestures.....	321
55.11 Summary.....	321
56. An iOS 9 Gesture Recognition Tutorial.....	323
56.1 Creating the Gesture Recognition Project	323
56.2 Designing the User Interface	323
56.3 Implementing the Action Methods	324
56.4 Testing the Gesture Recognition Application	324
56.5 Summary.....	325
57. A 3D Touch Force Handling Tutorial.....	327
57.1 Creating the 3D Touch Example Project	327
57.2 Adding the UIView Subclass to the Project.....	327
57.3 Locating the drawRect Method in the UIView Subclass	327
57.4 Implementing the Touch Methods	328
57.5 Testing the Touch Force App	328
57.6 Summary.....	329
58. An iOS 9 3D Touch Quick Actions Tutorial.....	331
58.1 Creating the Quick Actions Example Project	331
58.2 Static Quick Action Keys	331
58.3 Adding a Static Quick Action to the Project.....	331
58.4 Adding a Dynamic Quick Action.....	332
58.5 Adding, Removing and Changing Dynamic Quick Actions	333
58.6 Responding to a Quick Action Selection	334
58.7 Testing the Quick Action App	334
58.8 Summary.....	335
59. An iOS 9 3D Touch Peek and Pop Tutorial.....	337
59.1 About the Example Project.....	337
59.2 Adding the UIViewControllerPreviewDelegate	337
59.3 Implementing the Peek Delegate Method	338
59.4 Assigning the Detail Controller Storyboard ID.....	339
59.5 Implementing the Pop Delegate Method	339
59.6 Registering the Previewing Delegate.....	339
59.7 Testing the Peek and Pop Behavior	340
59.8 Adding Peek Quick Actions	341
59.9 Summary.....	343
60. Implementing TouchID Authentication in iOS 9 Apps	345
60.1 The Local Authentication Framework.....	345
60.2 Checking for TouchID Availability	345
60.3 Evaluating TouchID Policy.....	345
60.4 A TouchID Example Project.....	346
60.5 Checking for TouchID Availability	346
60.6 Seeking TouchID Authentication	347

60.7 Testing the Application	349
60.8 Summary.....	349
61. Drawing iOS 9 2D Graphics with Core Graphics.....	351
61.1 Introducing Core Graphics and Quartz 2D	351
61.2 The drawRect Method	351
61.3 Points, Coordinates and Pixels.....	351
61.4 The Graphics Context	351
61.5 Working with Colors in Quartz 2D	352
61.6 Summary.....	352
62. Interface Builder Live Views and iOS 9 Embedded Frameworks	353
62.1 Embedded Frameworks	353
62.2 Interface Builder Live Views	353
62.3 Creating the Example Project	353
62.4 Adding an Embedded Framework	354
62.5 Implementing the Drawing Code in the Framework	355
62.6 Making the View Designable	355
62.7 Making Variables Inspectable.....	356
62.8 Summary.....	357
63. An iOS 9 Graphics Tutorial using Core Graphics and Core Image	359
63.1 The iOS Drawing Example Application.....	359
63.2 Creating the New Project.....	359
63.3 Creating the UIView Subclass	359
63.4 Locating the drawRect Method in the UIView Subclass	359
63.5 Drawing a Line	360
63.6 Drawing Paths.....	361
63.7 Drawing a Rectangle	362
63.8 Drawing an Ellipse or Circle	362
63.9 Filling a Path with a Color	363
63.10 Drawing an Arc	364
63.11 Drawing a Cubic Bézier Curve	365
63.12 Drawing a Quadratic Bézier Curve	366
63.13 Dashed Line Drawing	366
63.14 Drawing Shadows	367
63.15 Drawing Gradients	368
63.16 Drawing an Image into a Graphics Context	371
63.17 Image Filtering with the Core Image Framework	372
63.18 Summary.....	373
64. Basic iOS 9 Animation using Core Animation	375
64.1 UIView Core Animation Blocks	375
64.2 Understanding Animation Curves.....	375
64.3 Receiving Notification of Animation Completion	375
64.4 Performing Affine Transformations.....	376
64.5 Combining Transformations	376
64.6 Creating the Animation Example Application	376
64.7 Implementing the Variables	376
64.8 Drawing in the UIView	377
64.9 Detecting Screen Touches and Performing the Animation	377
64.10 Building and Running the Animation Application	378
64.11 Summary.....	379
65. iOS 9 UIKit Dynamics – An Overview.....	381
65.1 Understanding UIKit Dynamics	381
65.2 The UIKit Dynamics Architecture	381
65.2.1 <i>Dynamic Items</i>	381
65.2.2 <i>Dynamic Behaviors</i>	381

65.2.3 The Reference View	382
65.2.4 The Dynamic Animator	382
65.3 Implementing UIKit Dynamics in an iOS 9 Application	382
65.4 Dynamic Animator Initialization	382
65.5 Configuring Gravity Behavior	382
65.6 Configuring Collision Behavior	383
65.7 Configuring Attachment Behavior	384
65.8 Configuring Snap Behavior	385
65.9 Configuring Push Behavior	385
65.10 The UIDynamicItemBehavior Class	386
65.11 Combining Behaviors to Create a Custom Behavior	386
65.12 Summary	387
66. An iOS 9 UIKit Dynamics Tutorial	389
66.1 Creating the UIKit Dynamics Example Project	389
66.2 Adding the Dynamic Items	389
66.3 Creating the Dynamic Animator Instance	390
66.4 Adding Gravity to the Views	390
66.5 Implementing Collision Behavior	391
66.6 Attaching a View to an Anchor Point	392
66.7 Implementing a Spring Attachment Between two Views	394
66.8 Summary	394
67. An Introduction to iOS 9 Sprite Kit Programming	397
67.1 What is Sprite Kit?	397
67.2 The Key Components of a Sprite Kit Game	397
67.2.1 Sprite Kit View	397
67.2.2 Scenes	397
67.2.3 Nodes	397
67.2.4 Physics Bodies	398
67.2.5 Physics World	398
67.2.6 Actions	398
67.2.7 Transitions	398
67.2.8 Texture Atlas	399
67.2.9 Constraints	399
67.3 An Example Sprite Kit Game Hierarchy	399
67.4 The Sprite Kit Game Rendering Loop	399
67.5 The Sprite Kit Level Editor	400
67.6 Summary	400
68. An iOS 9 Sprite Kit Level Editor Game Tutorial	401
68.1 About the Sprite Kit Demo Game	401
68.2 Creating the SpriteKitDemo Project	402
68.3 Reviewing the SpriteKit Game Template Project	402
68.4 Restricting Interface Orientation	403
68.5 Modifying the GameScene SpriteKit Scene File	403
68.6 Creating the Archery Scene	405
68.7 Transitioning to the Archery Scene	406
68.8 Adding the Texture Atlas	406
68.9 Designing the Archery Scene	407
68.10 Preparing the Archery Scene	409
68.11 Preparing the Animation Texture Atlas	409
68.12 Creating the Named Action Reference	410
68.13 Testing Actions in an Action File	411
68.14 Triggering the Named Action from the Code	412
68.15 Creating the Arrow Sprite Node	412
68.16 Shooting the Arrow	412
68.17 Adding the Ball Sprite Node	413
68.18 Summary	414

69. An iOS 9 Sprite Kit Collision Handling Tutorial	415
69.1 Defining the Category Bit Masks	415
69.2 Assigning the Category Masks to the Sprite Nodes	415
69.3 Configuring the Collision and Contact Masks	416
69.4 Implementing the Contact Delegate	416
69.5 Game Over	417
69.6 Summary.....	418
70. An iOS 9 Sprite Kit Particle Emitter Tutorial	419
70.1 What is the Particle Emitter?.....	419
70.2 The Particle Emitter Editor	419
70.3 The SKEmitterNode Class.....	419
70.4 Using the Particle Emitter Editor	420
70.5 Particle Emitter Node Properties.....	420
70.5.1 Background	421
70.5.2 Particle Texture	421
70.5.3 Particle Birthrate	421
70.5.4 Particle Life Cycle.....	421
70.5.5 Particle Position Range.....	421
70.5.6 Angle	421
70.5.7 Particle Speed.....	421
70.5.8 Particle Acceleration.....	421
70.5.9 Particle Scale	421
70.5.10 Particle Rotation.....	421
70.5.11 Particle Color	421
70.5.12 Particle Blend Mode	422
70.6 Experimenting with the Particle Emitter Editor.....	422
70.7 Bursting a Ball using Particle Emitter Effects.....	422
70.8 Adding the Burst Particle Emitter Effect.....	423
70.9 Adding an Audio Action	424
70.10 Summary.....	425
71. Integrating iAds into an iOS 9 App	427
71.1 Preparing to Run iAds within an Application	427
71.2 iAd Advertisement Formats.....	427
71.2.1 Banner Ads	427
71.2.2 Interstitial Ads	428
71.2.3 Medium Rectangle Ads.....	428
71.2.4 Pre-Roll Video Ads	429
71.3 Creating an Example iAds Application	430
71.4 Adding the iAds Framework to the Xcode Project.....	430
71.5 Enabling Banner Ads.....	430
71.6 Adding a Medium Rectangle Ad	430
71.7 Implementing an Interstitial Ad.....	431
71.8 Configuring iAds Test Settings	433
71.9 Summary.....	433
72. iOS 9 Multitasking, Background Transfer Service and Fetching	435
72.1 Understanding iOS Application States.....	435
72.2 A Brief Overview of the Multitasking Application Lifecycle.....	435
72.3 Checking for Multitasking Support	436
72.4 Enabling Multitasking for an iOS Application	436
72.5 Supported Forms of Background Execution	436
72.6 An Overview of Background Fetch	437
72.7 An Overview of Remote Notifications	438
72.8 An Overview of Local Notifications.....	438
72.9 An Overview of Background Transfer Service	438
72.10 The Rules of Background Execution	438

72.11 Summary.....	439
73. Scheduling iOS 9 Local Notifications	441
73.1 Creating the Local Notification App Project	441
73.2 Adding a Sound File to the Project	441
73.3 Requesting Permission to Trigger Alerts.....	441
73.4 Locating the Application Delegate Method	442
73.5 Scheduling the Local Notification	442
73.6 Testing the Application	443
73.7 Cancelling Scheduled Notifications.....	443
73.8 Immediate Triggering of a Local Notification	443
73.9 Summary.....	443
74. An Overview of iOS 9 Application State Preservation and Restoration	445
74.1 The Preservation and Restoration Process	445
74.2 Opting In to Preservation and Restoration	445
74.3 Assigning Restoration Identifiers	446
74.4 Default Preservation Features of UIKit	446
74.5 Saving and Restoring Additional State Information.....	446
74.6 Understanding the Restoration Process	447
74.7 Saving General Application State.....	448
74.8 Summary.....	448
75. An iOS 9 State Preservation and Restoration Tutorial	449
75.1 Creating the Example Application	449
75.2 Trying the Application without State Preservation	449
75.3 Opting-in to State Preservation	449
75.4 Setting Restoration Identifiers.....	449
75.5 Encoding and Decoding View Controller State	450
75.6 Adding a Navigation Controller to the Storyboard	451
75.7 Adding the Third View Controller	452
75.8 Creating the Restoration Class.....	453
75.9 Summary.....	454
76. Integrating Maps into iOS 9 Applications using MKMapItem	455
76.1 MKMapItem and MKPlacemark Classes	455
76.2 An Introduction to Forward and Reverse Geocoding	455
76.3 Creating MKPlacemark Instances	456
76.4 Working with MKMapItem	457
76.5 MKMapItem Options and Configuring Directions	457
76.6 Adding Item Details to an MKMapItem	458
76.7 Summary.....	458
77. An Example iOS 9 MKMapItem Application	461
77.1 Creating the MapItem Project	461
77.2 Designing the User Interface	461
77.3 Converting the Destination using Forward Geocoding.....	462
77.4 Launching the Map	463
77.5 Building and Running the Application	463
77.6 Summary.....	463
78. Getting Location Information using the iOS 9 Core Location Framework	465
78.1 The Core Location Manager	465
78.2 Requesting Location Access Authorization	465
78.3 Configuring the Desired Location Accuracy	465
78.4 Configuring the Distance Filter	466
78.5 The Location Manager Delegate.....	466
78.6 Starting Location Updates	466
78.7 Obtaining Location Information from CLLocation Objects	467

78.7.1 Longitude and Latitude	467
78.7.2 Accuracy	467
78.7.3 Altitude.....	467
78.8 Getting the Current Location.....	467
78.9 Calculating Distances	467
78.10 Location Information and Multitasking	467
78.11 Summary.....	468
79. An Example iOS 9 Location Application.....	469
79.1 Creating the Example iOS 9 Location Project	469
79.2 Designing the User Interface	469
79.3 Configuring the CLLocationManager Object.....	470
79.4 Setting up the Usage Description Key.....	470
79.5 Implementing the Action Method	470
79.6 Implementing the Application Delegate Methods	471
79.7 Building and Running the Location Application.....	471
80. Working with Maps on iOS 9 with MapKit and the MKMapView Class	473
80.1 About the MapKit Framework	473
80.2 Understanding Map Regions	473
80.3 Getting Transit ETA Information.....	473
80.4 About the MKMapView Tutorial.....	474
80.5 Creating the Map Project.....	474
80.6 Adding the Navigation Controller	474
80.7 Creating the MKMapView Instance and Toolbar	474
80.8 Obtaining Location Information Permission	476
80.9 Setting up the Usage Description Key.....	476
80.10 Configuring the Map View	477
80.11 Changing the MapView Region.....	477
80.12 Changing the Map Type.....	477
80.13 Testing the MapView Application.....	477
80.14 Updating the Map View based on User Movement	478
80.15 Summary.....	478
81. Working with MapKit Local Search in iOS 9	479
81.1 An Overview of iOS 9 Local Search	479
81.2 Adding Local Search to the MapSample Application	480
81.3 Adding the Local Search Text Field	480
81.4 Performing the Local Search.....	481
81.5 Testing the Application	482
81.6 Summary.....	482
82. Using MKDirections to get iOS 9 Map Directions and Routes.....	483
82.1 An Overview of MKDirections.....	483
82.2 Adding Directions and Routes to the MapSample Application	484
82.3 Adding the New Classes to the Project.....	484
82.4 Configuring the Results Table View	484
82.5 Implementing the Result Table View Segue	486
82.6 Adding the Route Scene	487
82.7 Identifying the User's Current Location.....	488
82.8 Getting the Route and Directions	489
82.9 Establishing the Route Segue	490
82.10 Testing the Application	490
82.11 Summary.....	491
83. An iOS 9 MapKit Flyover Tutorial	493
83.1 MKMapView Flyover Map Types	493
83.2 The MKMapCamera Class	493
83.3 An MKMapKit Flyover Example	494

83.4 Designing the User Interface	494
83.5 Configuring the Map View and Camera	496
83.6 Animating Camera Changes.....	496
83.7 Testing the Map Flyover App.....	497
83.8 Summary.....	497
84. An Introduction to Extensions in iOS 9.....	499
84.1 iOS Extensions – An Overview	499
84.2 Extension Types	499
84.2.1 Today Extension	499
84.2.2 Share Extension	500
84.2.3 Action Extension	501
84.2.4 Photo Editing Extension.....	501
84.2.5 Document Provider Extension	502
84.2.6 Custom Keyboard Extension.....	502
84.2.7 Audio Unit Extension	502
84.2.8 Shared Links Extension	502
84.2.9 Content Blocking Extension	502
84.3 Creating Extensions	502
84.4 Summary.....	503
85. An iOS 9 Today Extension Widget Tutorial	505
85.1 About the Example Extension Widget	505
85.2 Creating the Example Project	505
85.3 Adding the Extension to the Project	505
85.4 Reviewing the Extension Files.....	507
85.5 Designing the Widget User Interface.....	507
85.6 Setting the Preferred Content Size in Code	508
85.7 Modifying the Widget View Controller	508
85.8 Testing the Extension	510
85.9 Opening the Containing App from the Extension	510
85.10 Summary.....	511
86. Creating an iOS 9 Photo Editing Extension	513
86.1 Creating a Photo Editing Extension.....	513
86.2 Accessing the Photo Editing Extension	513
86.3 Configuring the Info.plist File.....	515
86.4 Designing the User Interface	515
86.5 The PHContentEditingController Protocol.....	516
86.6 Photo Extensions and Adjustment Data	516
86.7 Receiving the Content	517
86.8 Implementing the Filter Actions	518
86.9 Returning the Image to the Photos App	519
86.10 Testing the Application	521
86.11 Summary.....	522
87. Creating an iOS 9 Action Extension	523
87.1 An Overview of Action Extensions	523
87.2 About the Action Extension Example	523
87.3 Creating the Action Extension Project	524
87.4 Adding the Action Extension Target	524
87.5 Changing the Extension Display Name	524
87.6 Designing the Action Extension User Interface	524
87.7 Receiving the Content	525
87.8 Returning the Modified Data to the Host App	527
87.9 Testing the Extension	527
87.10 Declaring the Supported Content Type	528
87.11 Summary.....	529
88. Receiving Data from an iOS 9 Action Extension	531

88.1 Creating the Example Project	531
88.2 Designing the User Interface	531
88.3 Importing the Mobile Core Services Framework	531
88.4 Adding a Share Button to the Application	532
88.5 Receiving Data from an Extension	533
88.6 Testing the Application	533
88.7 Summary.....	533
89. Using iOS 9 Event Kit to Create Date and Location Based Reminders	535
89.1 An Overview of the Event Kit Framework	535
89.2 The EKEventStore Class	535
89.3 Accessing Calendars in the Database	536
89.4 Creating Reminders	536
89.5 Creating Alarms	537
89.6 Creating the Example Project	537
89.7 Designing the User Interface for the Date/Time Based Reminder Screen	537
89.8 Implementing the Reminder Code	538
89.9 Hiding the Keyboard	539
89.10 Designing the Location-based Reminder Screen	540
89.11 Creating a Location-based Reminder.....	540
89.12 Setting up the Usage Description Key.....	542
89.13 Testing the Application	543
89.14 Summary.....	543
90. Accessing the iOS 9 Camera and Photo Library	545
90.1 The UIImagePickerController Class.....	545
90.2 Creating and Configuring a UIImagePickerController Instance	545
90.3 Configuring the UIImagePickerController Delegate	546
90.4 Detecting Device Capabilities	546
90.5 Saving Movies and Images.....	547
90.6 Summary.....	547
91. An Example iOS 9 Camera Application	549
91.1 An Overview of the Application	549
91.2 Creating the Camera Project	549
91.3 Designing the User Interface	549
91.4 Implementing the Action Methods	550
91.5 Writing the Delegate Methods	551
91.6 Building and Running the Application	552
92. iOS 9 Video Playback using AVPlayer and AVPlayerViewController	553
92.1 The AVPlayer and AVPlayerViewController Classes	553
92.2 The iOS Movie Player Example Application	553
92.3 Adding a Security Exception for an HTTP Connection	553
92.4 Designing the User Interface	554
92.5 Initializing Video Playback	554
92.6 Build and Run the Application	555
92.7 Creating AVPlayerViewController Instance from Code	555
92.8 Summary.....	555
93. An iOS 9 Multitasking Picture in Picture Tutorial	557
93.1 An Overview of Picture in Picture Multitasking.....	557
93.2 Adding Picture in Picture Support to the AVPlayerDemo App	557
93.3 Adding the Navigation Controller	558
93.4 Setting the Audio Session Category.....	558
93.5 Implementing the Delegate	559
93.6 Opting Out of Picture in Picture Support.....	560
93.7 Additional Delegate Methods.....	561
93.8 Summary.....	561

94. Playing Audio on iOS 9 using AVAudioPlayer	563
94.1 Supported Audio Formats.....	563
94.2 Receiving Playback Notifications	563
94.3 Controlling and Monitoring Playback	563
94.4 Creating the Audio Example Application	564
94.5 Adding an Audio File to the Project Resources.....	564
94.6 Designing the User Interface	564
94.7 Implementing the Action Methods	565
94.8 Creating and Initializing the AVAudioPlayer Object	565
94.9 Implementing the AVAudioPlayerDelegate Protocol Methods	565
94.10 Building and Running the Application	566
94.11 Summary.....	566
95. Recording Audio on iOS 9 with AVAudioRecorder	567
95.1 An Overview of the AVAudioRecorder Tutorial.....	567
95.2 Creating the Recorder Project	567
95.3 Designing the User Interface	567
95.4 Creating the AVAudioRecorder Instance	568
95.5 Implementing the Action Methods	569
95.6 Implementing the Delegate Methods	569
95.7 Testing the Application	570
96. Integrating Twitter and Facebook into iOS 9 Applications	571
96.1 The UINavigationController class.....	571
96.2 The Social Framework.....	571
96.3 Accounts Framework.....	571
96.4 Using the UINavigationController Class	572
96.5 Using the SLComposeViewController Class	573
96.6 Summary.....	574
97. An iOS 9 Social Media Integration Tutorial using UINavigationController	575
97.1 Creating the Facebook Social App	575
97.2 Designing the User Interface	575
97.3 Creating Outlets and Actions.....	575
97.4 Implementing the selectImage and Delegate Methods	576
97.5 Hiding the Keyboard	576
97.6 Posting the Message.....	577
97.7 Running the Social Application	577
97.8 Summary.....	578
98. iOS 9 Facebook and Twitter Integration using SLRequest	579
98.1 Using SLRequest and the Account Framework.....	579
98.2 Twitter Integration using SLRequest	579
98.3 Facebook Integration using SLRequest.....	581
98.4 Summary.....	583
99. An iOS 9 Twitter Integration Tutorial using SLRequest.....	585
99.1 Creating the TwitterApp Project.....	585
99.2 Designing the User Interface	585
99.3 Modifying the View Controller Class	586
99.4 Accessing the Twitter API	586
99.5 Calling the getTimeLine Method	588
99.6 The Table View Delegate Methods.....	588
99.7 Building and Running the Application	589
99.8 Summary.....	589
100. Making Store Purchases with the SKStoreProductViewController Class.....	591
100.1 The SKStoreProductViewController Class.....	591
100.2 Creating the Example Project	591

100.3 Creating the User Interface	591
100.4 Displaying the Store Kit Product View Controller	592
100.5 Implementing the Delegate Method	593
100.6 Testing the Application	593
100.7 Summary	594
101. Building In-App Purchasing into iOS 9 Applications	595
101.1 In-App Purchase Options	595
101.2 Uploading App Store Hosted Content	595
101.3 Configuring In-App Purchase Items	595
101.4 Sending a Product Request	595
101.5 Accessing the Payment Queue	596
101.6 The Transaction Observer Object	596
101.7 Initiating the Purchase	597
101.8 The Transaction Process	597
101.9 Transaction Restoration Process	598
101.10 Testing In-App Purchases	598
101.11 Summary	598
102. Preparing an iOS 9 Application for In-App Purchases	599
102.1 About the Example Application	599
102.2 Creating the Xcode Project	599
102.3 Registering and Enabling the App ID for In App Purchasing	599
102.4 Configuring the Application in iTunes Connect	599
102.5 Creating an In-App Purchase Item	600
102.6 Summary	600
103. An iOS 9 In-App Purchase Tutorial	601
103.1 The Application User Interface	601
103.2 Designing the Storyboard	601
103.3 Creating the Purchase View Controller Class	602
103.4 Storing the Home View Controller in the App Delegate Class	603
103.5 Completing the ViewController Class	603
103.6 Completing the PurchaseViewController Class	604
103.7 Testing the Application	606
103.8 Troubleshooting	606
103.9 Summary	606
104. Configuring and Creating App Store Hosted Content for iOS 9 In-App Purchases	607
104.1 Configuring an Application for In-App Purchase Hosted Content	607
104.2 The Anatomy of an In-App Purchase Hosted Content Package	607
104.3 Creating an In-App Purchase Hosted Content Package	607
104.4 Archiving the Hosted Content Package	608
104.5 Validating the Hosted Content Package	608
104.6 Uploading the Hosted Content Package	609
104.7 Summary	609
105. Preparing and Submitting an iOS 9 Application to the App Store	611
105.1 Verifying the iOS Distribution Certificate	611
105.2 Adding App Icons	612
105.3 Designing the Launch Screen	613
105.4 Assign the Project to a Team	613
105.5 Archiving the Application for Distribution	613
105.6 Configuring the Application in iTunes Connect	614
105.7 Validating and Submitting the Application	614
105.8 Configuring and Submitting the App for Review	616
Index	617

1. Start Here

The goal of this book is to teach the skills necessary to create iOS applications using the iOS 9 SDK, Xcode 7 and the Swift 2 programming language.

How you make use of this book will depend to a large extent on whether you are new to iOS development, or have worked with iOS 8 and need to get up to speed on the features of iOS 9 and the latest version of the Swift programming language. Rest assured, however, that the book is intended to address both category of reader.

1.1 For New iOS Developers

If you are entirely new to iOS development then the entire contents of the book will be relevant to you.

Beginning with the basics, this book provides an outline of the steps necessary to set up an iOS development environment. An introduction to the architecture of iOS 9 and programming in Swift is provided, followed by an in-depth look at the design of iOS applications and user interfaces. More advanced topics such as file handling, database management, in-app purchases, graphics drawing and animation are also covered, as are touch screen handling, gesture recognition, multitasking, iAds integration, location management, local notifications, camera access and video and audio playback support. Other features are also covered including Auto Layout, Twitter and Facebook integration, App Store hosted in-app purchase content, Sprite Kit-based game development, local map search and user interface animation using UIKit dynamics.

Additional features of iOS development using Xcode 7 are also covered, including Swift playgrounds, universal user interface design using size classes, app extensions, Interface Builder Live Views, embedded frameworks, CloudKit data storage and TouchID authentication.

The key new features of iOS 9 and Xcode 7 are also covered in detail, including new error handling in Swift 2, designing Stack View based user interfaces, multiple storyboard support, iPad multitasking, map flyover support, 3D Touch and Picture-in-Picture media playback.

The aim of this book, therefore, is to teach you the skills necessary to build your own apps for iOS 9. Assuming you are ready to download the iOS 9 SDK and Xcode 7, have an Intel-based Mac and ideas for some apps to develop, you are ready to get started.

1.2 For iOS 8 Developers

If you have already read the iOS 8 edition of this book, or have experience with the iOS 8 SDK then you might prefer to go directly to the new chapters in this iOS 9 edition of the book.

All chapters have been updated to reflect the changes and features introduced as part of iOS 9, Swift 2 and Xcode 7. Chapters included in this edition that were not contained in the previous edition, or have been significantly rewritten for iOS 9 and Xcode 7 are as follows:

- *An Introduction to Xcode 7 Playgrounds*
- *Joining the Apple Developer Program*
- *An Introduction to Swift Subclassing and Extensions*
- *Understanding Error Handling in Swift 2*
- *Organizing Scenes over Multiple Storyboard Files*
- *Working with the iOS 9 Stack View Class*
- *An iOS 9 Stack View Tutorial*
- *A Guide to Multitasking in iOS 9*
- *An iOS 9 Multitasking Example*
- *A 3D Touch Force Handling Tutorial*
- *An iOS 9 3D Touch Quick Actions Tutorial*
- *An iOS 9 3D Touch Peek and Pop Tutorial*
- *An iOS 9 MapKit Flyover Tutorial*
- *An iOS 9 Multitasking Picture in Picture Tutorial*

In addition the following changes have also been made:

- All chapters have been updated where necessary to reflect the changes made to Xcode 7.
- All chapters and examples have been rewritten where necessary to use Swift 2 syntax.

Start Here

- The Multitouch handling chapters have been updated to cover the new predictive touches and touch coalescing features of iOS 9.
- Map handling topics covered in the book now include steps to obtain Transit ETA Information.
- The SpriteKit chapters have been updated to use the new Xcode 7 Live Animation and Action Editors.
- The Index has been improved with more than double the number of entries.

1.3 Source Code Download

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

<http://www.ebookfrenzy.com/direct/ios9/>

1.4 Learn to Develop watchOS Apps

If you are planning to develop WatchKit apps for the Apple Watch we would like to recommend *watchOS 2 App Development Essentials*.

watchOS 2 App Development



Essentials

This is the companion book to iOS 9 App Development Essentials and is available for purchase now in both print and eBook formats:

http://www.ebookfrenzy.com/ebookpages/watchos2_ebook.html

1.5 Feedback

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

1.6 Errata

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

<http://www.ebookfrenzy.com/errata/ios9.html>

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

2. Joining the Apple Developer Program

The first step in the process of learning to develop iOS 9 based applications involves gaining an understanding of the advantages of enrolling in the Apple Developer Program and deciding the point at which it makes sense to pay to join. With these goals in mind, this chapter will outline the costs and benefits of joining the developer program and, finally, walk through the steps involved in enrolling.

2.1 Downloading Xcode 7 and the iOS 9 SDK

The latest versions of both the iOS SDK and Xcode can be downloaded free of charge from the Mac App Store. Since the tools are free, this raises the question of whether to enroll in the Apple Developer Program, or to wait until it becomes necessary later in your app development learning curve.

2.2 Apple Developer Program

Membership in the Apple Developer Program currently costs \$99 per year to enroll as an individual developer. Organization level membership is also available.

Prior to the introduction of iOS 9 and Xcode 7, one of the key advantages of the developer program was that it permitted the creation of certificates and provisioning profiles to test your applications on physical iOS devices. Fortunately this is no longer the case and all that is now required to test apps on physical iOS devices is an Apple ID.

Clearly much can be achieved without the need to pay to join the Apple Developer program. There are, however, areas of app development which cannot be fully tested without program membership. Of particular significance is the fact that iCloud access, Apple Pay, Game Center and In-App Purchasing can only be enabled and tested with Apple Developer Program membership.

Of further significance is the fact that Apple Developer Program members have access to technical support from Apple's iOS support engineers (though the annual fee initially covers the submission of only two support incident reports more can be purchased) and membership of the Apple Developer forums which can be an invaluable resource for obtaining assistance and guidance from other iOS developers and for finding solutions to problems that others have encountered and subsequently resolved.

Program membership also provides early access to the pre-release Beta versions of both Xcode and iOS.

By far the most important aspect of the Apple Developer Program is that membership is a mandatory requirement in order to publish an application for sale or download in the App Store.

Clearly, program membership is going to be required at some point before your application reaches the App Store. The only question remaining is when exactly to sign up.

2.3 When to Enroll in the Apple Developer Program?

Clearly, there are many benefits to Apple Developer Program membership and, eventually, membership will be necessary to begin selling applications. As to whether or not to pay the enrollment fee now or later will depend on individual circumstances. If you are still in the early stages of learning to develop iOS applications or have yet to come up with a compelling idea for an application to develop then much of what you need is provided without program membership. As your skill level increases and your ideas for applications to develop take shape you can, after all, always enroll in the developer program at a later date.

If, on the other hand, you are confident that you will reach the stage of having an application ready to publish or know that you will need access to more advanced features such as iCloud, In-App Purchasing and Apple Pay then it is worth joining the developer program sooner rather than later.

2.4 Enrolling in the Apple Developer Program

If your goal is to develop iOS applications for your employer then it is first worth checking whether the company already has membership. That being the case, contact the program administrator in your company and ask them to send you an invitation from within the Apple Developer Program Member Center to join the team. Once they have done so, Apple will send you an email entitled *You Have Been Invited to Join an Apple Developer Program* containing a link to activate your membership. If you or your company is not already a program member, you can enroll online at:

<https://developer.apple.com/programs/enroll/>

Apple provides enrollment options for businesses and individuals. To enroll as an individual you will need to provide credit card information in order to verify your identity. To enroll as a company you must have legal signature authority (or access to someone

Joining the Apple Developer Program

who does) and be able to provide documentation such as a Dun & Bradstreet D-U-N-S number and documentation confirming legal entity status.

Acceptance into the developer program as an individual member typically takes less than 24 hours with notification arriving in the form of an activation email from Apple. Enrollment as a company can take considerably longer (sometimes weeks or even months) due to the burden of the additional verification requirements.

Whilst awaiting activation you may log into the Member Center with restricted access using your Apple ID and password at the following URL:

<http://developer.apple.com/membercenter>

Once logged in, clicking on the *Your Account* tab at the top of the page will display the prevailing status of your application to join the developer program as *Enrollment Pending*. Once the activation email has arrived, log into the Member Center again and note that access is now available to a wide range of options and resources as illustrated in Figure 2-1:

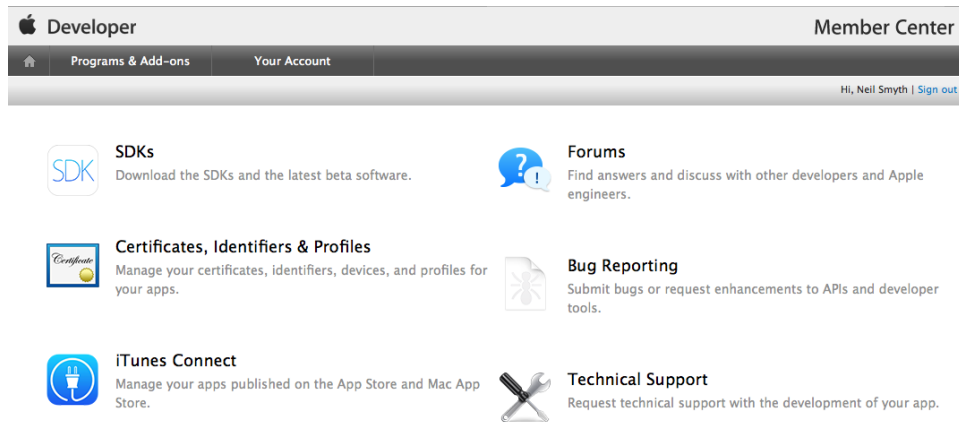


Figure 2-1

2.5 Summary

An important early step in the iOS 9 application development process involves identifying the best time to enroll in the Apple Developer Program. This chapter has outlined the benefits of joining the program, provided some guidance to keep in mind when considering developer program membership and walked briefly through the enrollment process. The next step is to download and install the iOS 9 SDK and Xcode 7 development environment.

3. Installing Xcode 7 and the iOS 9 SDK

iOS apps are developed using the iOS SDK in conjunction with Apple's Xcode 7 development environment. Xcode 7 is an integrated development environment (IDE) within which you will code, compile, test and debug your iOS applications. The Xcode 7 environment also includes a feature called Interface Builder which enables you to graphically design the user interface of your application using the components provided by the UIKit Framework.

In this chapter we will cover the steps involved in installing both Xcode 7 and the iOS 9 SDK on Mac OS X.

3.1 Identifying if you have an Intel or PowerPC based Mac

Only Intel based Mac OS X systems can be used to develop applications for iOS. If you have an older, PowerPC based Mac then you will need to purchase a new system before you can begin your iOS app development project. If you are unsure of the processor type inside your Mac, you can find this information by clicking on the Apple menu in the top left hand corner of the screen and selecting the *About This Mac* option from the menu. In the resulting dialog check the *Processor* line. Figure 3-1 illustrates the results obtained on an Intel based system.

If the dialog on your Mac does not reflect the presence of an Intel based processor then your current system is, sadly, unsuitable as a platform for iOS app development.

In addition, the iOS 9 SDK with Xcode 7 environment requires that the version of Mac OS X running on the system be version 10.10.4 or later. If the "About This Mac" dialog does not indicate that Mac OS X 10.10.4 or later is running, click on the *Software Update...* button to download and install the appropriate operating system upgrades.



Figure 3-1

3.2 Installing Xcode 7 and the iOS 9 SDK

The best way to obtain the latest versions of Xcode and the iOS SDK is to download them from the Apple Mac App Store. Launch the App Store on your Mac OS X system, enter Xcode into the search box and click on the *Free* button to initiate the installation.

3.3 Starting Xcode

Having successfully installed the SDK and Xcode, the next step is to launch it so that we can create a sample iOS 9 application. To start up Xcode, open the Finder and search for *Xcode*. Since you will be making frequent use of this tool take this opportunity to drag and drop it into your dock for easier access in the future. Click on the Xcode icon in the dock to launch the tool. The first time Xcode runs you may be prompted to install additional components. Follow these steps, entering your username and password when prompted to do so.

Once Xcode has loaded, and assuming this is the first time you have used Xcode on this system, you will be presented with the *Welcome* screen from which you are ready to proceed:



Figure 3-2

3.4 Adding Your Apple ID to the Xcode Preferences

Regardless of whether or not you choose to enroll in the Apple Developer Program it is worth adding your Apple ID to Xcode now that it is installed and running. Select the *Xcode -> Preferences...* menu option and select the *Accounts* tab. On the Accounts screen, click on the + button highlighted in Figure 3-3 and select *Add Apple ID...* from the resulting menu:

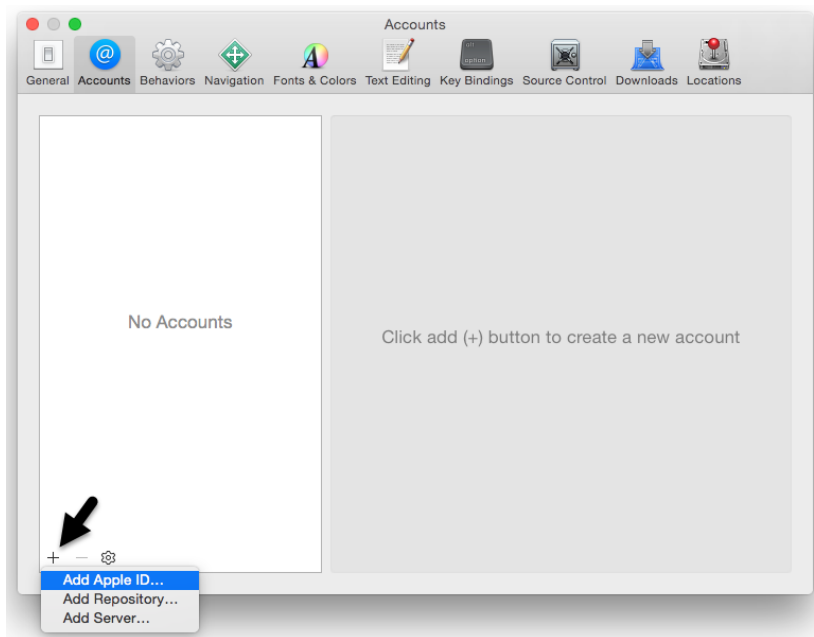


Figure 3-3

When prompted, enter your Apple ID and associated password and click on the *Add* button to add the account to the preferences.

3.5 Developer and Distribution Signing Identities

Once the Apple ID has been entered the next step is to generate signing identities. To view the current signing identities, select the newly added Apple ID in the Accounts panel and click on the *View Details...* button at which point a list of available signing identities will be listed. If you have not yet enrolled in the Apple Developer Program it will only be possible to create iOS and Mac Development identities. To create the iOS Development signing identity, simply click on the *Create* button highlighted in Figure 3-4:

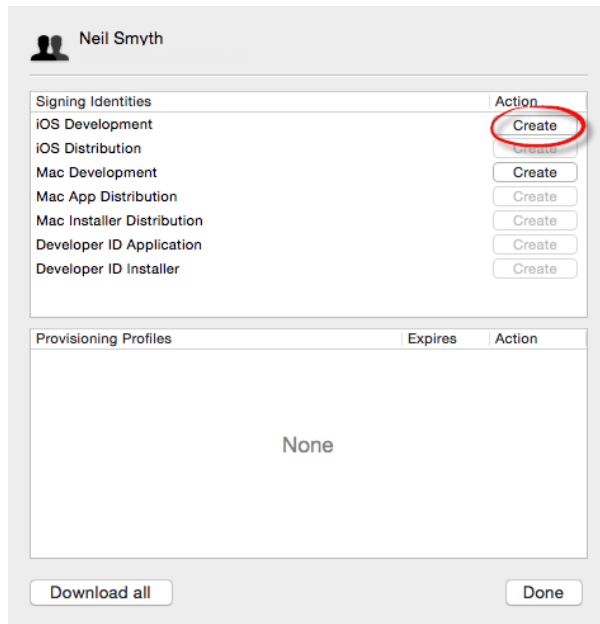


Figure 3-4

If the Apple ID has been used to enroll in the Apple Developer program, the *Create* button next to the *iOS Distribution* entry will also be active and, when clicked, will generate the signing identity required to submit the app to the Apple App Store.

Having installed the iOS 9 SDK and successfully launched Xcode 7 we can now look at Xcode in more detail.

4. A Guided Tour of Xcode 7

Just about every activity related to developing and testing iOS applications involves the use of the Xcode environment. This chapter is intended to serve two purposes. Primarily it is intended to provide an overview of many of the key areas that comprise the Xcode development environment. In the course of providing this overview, the chapter will also work through the creation of a very simple iOS application project designed to display a label which reads “Hello World” on a colored background.

By the end of this chapter you will have a basic familiarity with Xcode and your first running iOS application.

4.1 Starting Xcode 7

As with all iOS examples in this book, the development of our example will take place within the Xcode 7 development environment. If you have not already installed this tool together with the latest iOS SDK refer first to the *Installing Xcode 7 and the iOS 9 SDK* chapter of this book. Assuming that the installation is complete, launch Xcode either by clicking on the icon on the dock (assuming you created one) or use the Mac OS X Finder to locate Xcode in the Applications folder of your system.

When launched for the first time, and until you turn off the *Show this window when Xcode launches* toggle, the screen illustrated in Figure 4-1 will appear by default:



Figure 4-1

If you do not see this window, simply select the *Window -> Welcome to Xcode* menu option to display it. From within this window, click on the option to *Create a new Xcode project*. This will display the main Xcode 7 project window together with the *project template* panel where we are able to select a template matching the type of project we want to develop:

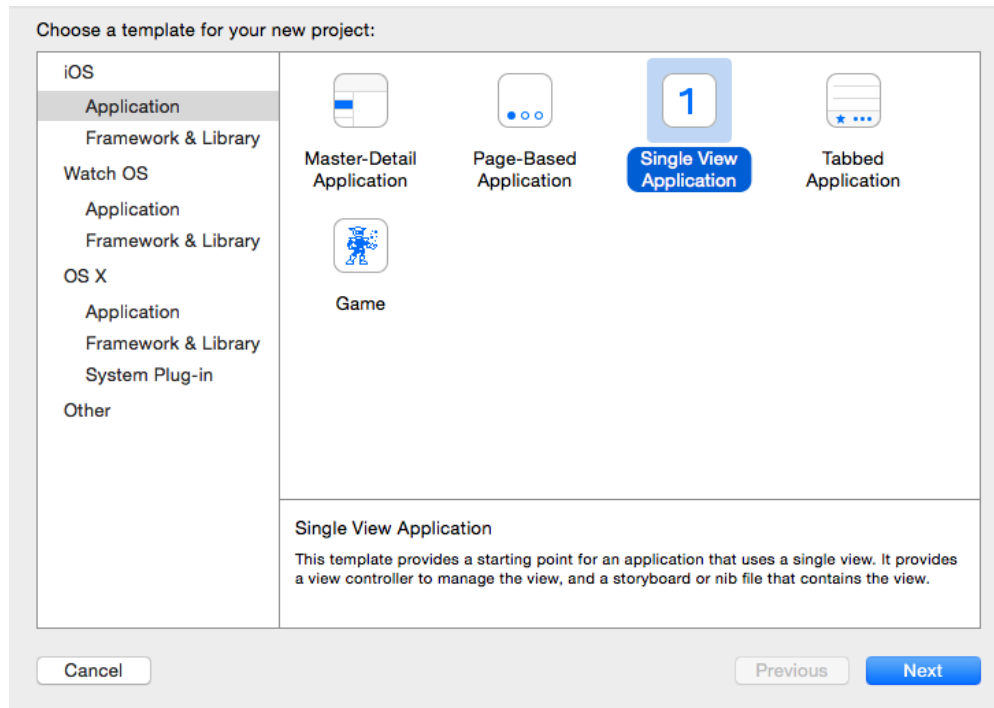


Figure 4-2

The panel located on the left hand side of the window allows for the selection of the target platform, providing options to develop an application either for iOS and watchOS based devices or Mac OS X.

Begin by making sure that the *Application* option located beneath *iOS* is selected. The main panel contains a list of templates available to use as the basis for an application. The options available are as follows:

- **Master-Detail Application** – Used to create a list based application. Selecting an item from a master list displays a detail view corresponding to the selection. The template then provides a *Back* button to return to the list. You may have seen a similar technique used for news based applications, whereby selecting an item from a list of headlines displays the content of the corresponding news article. When used for an iPad based application this template implements a basic split-view configuration.
- **Page-based Application** – Creates a template project using the page view controller designed to allow views to be transitioned by turning pages on the screen.
- **Tabbed Application** – Creates a template application with a tab bar. The tab bar typically appears across the bottom of the device display and can be programmed to contain items that, when selected, change the main display to different views. The iPhone's built-in *Phone* user interface, for example, uses a tab bar to allow the user to move between favorites, contacts, keypad and voicemail.
- **Single View Application** – Creates a basic template for an application containing a single view and corresponding view controller.
- **Game** – Creates a project configured to take advantage of Sprite Kit, Scene Kit, OpenGL ES and Metal for the development of 2D and 3D games.

For the purposes of our simple example, we are going to use the *Single View Application* template so select this option from the new project window and click *Next* to configure some more project options:

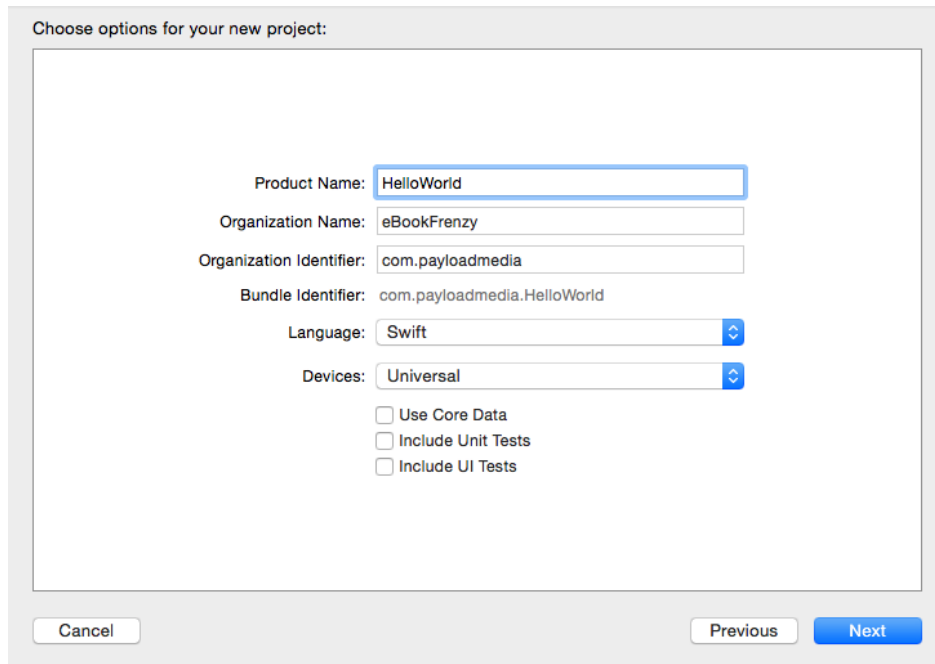


Figure 4-3

On this screen, enter a Product name for the application that is going to be created, in this case “HelloWorld”. The text entered into the Organization Name field will be placed within the copyright comments of all of the source files that make up the project.

The company identifier is typically the reversed URL of your company’s website, for example “com.mycompany”. This will be used when creating provisioning profiles and certificates to enable testing of advanced features of iOS on physical devices. It also serves to uniquely identify the app within the Apple App Store when the app is published.

The iOS ecosystem now includes a variety of devices and screen sizes. When creating a new project it is possible to indicate that the project is intended to target either the iPhone or iPad family of devices. With the gap between iPad and iPhone screen sizes now reduced by the introduction of the iPad Mini and iPhone 6 Plus it no longer makes sense to create a project that targets just one device family. A much more sensible approach is to create a single project that addresses all device types and screen sizes. In fact, as will be shown in later chapters, Xcode 7 and iOS 9 include a number of features designed specifically to make the goal of *universal* application projects easy to achieve. With this in mind, make sure that the *Devices* menu is set to *Universal*.

Apple supports two programming languages for the development of iOS apps in the form of *Objective-C* and *Swift*. Whilst it is still possible to program using the older Objective-C language, Apple considers Swift to be the future of iOS development. All the code examples in this book are written in Swift, so make sure that the *Language* menu is set accordingly before clicking on the *Next* button.

On the final screen, choose a location on the file system for the new project to be created and click on *Create*.

Once the new project has been created, the main Xcode window will appear as illustrated in Figure 4-4:

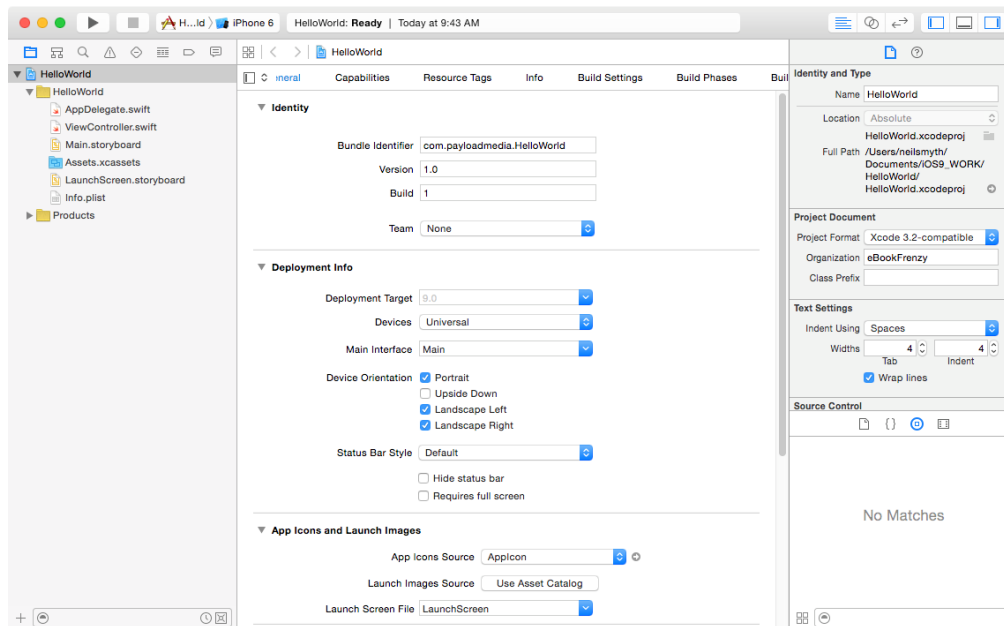


Figure 4-4

Before proceeding we should take some time to look at what Xcode has done for us. Firstly it has created a group of files that we will need to create our application. Some of these are Swift source code files (with a `.swift` extension) where we will enter the code to make our application work.

In addition, the *Main.storyboard* file is the save file used by the Interface Builder tool to hold the user interface design we will create. A second Interface Builder file named *LaunchScreen.storyboard* will also have been added to the project. This contains the user interface layout design for the screen which appears on the device while the application is loading.

Also present will be one or more files with a `.plist` file extension. These are *Property List* files which contain key/value pair information. For example, the *Info.plist* file contains resource settings relating to items such as the language, executable name and app identifier and, as will be shown in later chapters, is the location where a number of properties are stored to configure the capabilities of the project (for example to configure access to the user's current geographical location). The list of files is displayed in the *Project Navigator* located in the left hand panel of the main Xcode project window. A toolbar at the top of this panel contains options to display other information such as build and run history, breakpoints and compilation errors.

By default, the center panel of the window shows a general summary of the settings for the application project. This includes the identifier specified during the project creation process and the target device. Options are also provided to configure the orientations of the device that are to be supported by the application together with options to upload icons (the small images the user selects on the device screen to launch the application) and launch screen images (displayed to the user while the application loads) for the application.

In addition to the General screen, tabs are provided to view and modify additional settings consisting of Capabilities, Info, Build Settings, Build Phases and Build Rules. As we progress through subsequent chapters of this book we will explore some of these other configuration options in greater detail. To return to the project settings panel at any future point in time, make sure the *Project Navigator* is selected in the left hand panel and select the top item (the application name) in the navigator list.

When a source file is selected from the list in the navigator panel, the contents of that file will appear in the center panel where it may then be edited. To open the file in a separate editing window, simply double click on the file in the list.

4.2 Creating the iOS App User Interface

Simply by the very nature of the environment in which they run, iOS apps are typically visually oriented. As such, a key component of just about any app involves a user interface through which the user will interact with the application and, in turn, receive feedback. Whilst it is possible to develop user interfaces by writing code to create and position items on the screen, this is a complex and error prone process. In recognition of this, Apple provides a tool called Interface Builder which allows a user interface to be visually constructed by dragging and dropping components onto a canvas and setting properties to configure the appearance and behavior of those components.

As mentioned in the preceding section, Xcode pre-created a number of files for our project, one of which has a `.storyboard` filename extension. This is an Interface Builder storyboard save file and the file we are interested in for our HelloWorld project is named *Main.storyboard*. To load this file into Interface Builder simply select the file name in the list in the left hand panel. Interface Builder will subsequently appear in the center panel as shown in Figure 4-5:

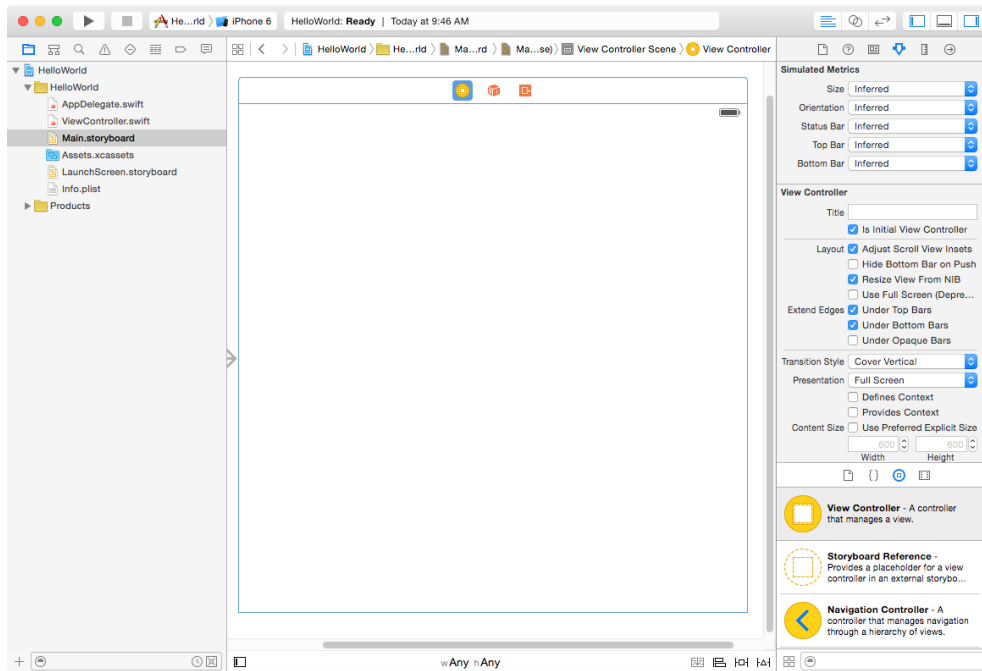


Figure 4-5

In the center panel a visual representation of the user interface of the application is displayed. Initially this consists solely of the `UIView` object. This `UIView` object was added to our design by Xcode when we selected the Single View Application option during the project creation phase. We will construct the user interface for our HelloWorld app by dragging and dropping user interface objects onto this `UIView` object. Designing a user interface consists primarily of dragging and dropping visual components onto the canvas and setting a range of properties. In order to access objects and property settings it is necessary to display the Xcode right hand panel (if it is not already displayed). This panel is referred to as the *Utilities panel* and can be displayed by selecting the right hand button in the right hand section of the Xcode toolbar:



Figure 4-6

The Utilities panel, once displayed, will appear as illustrated in Figure 4-7:

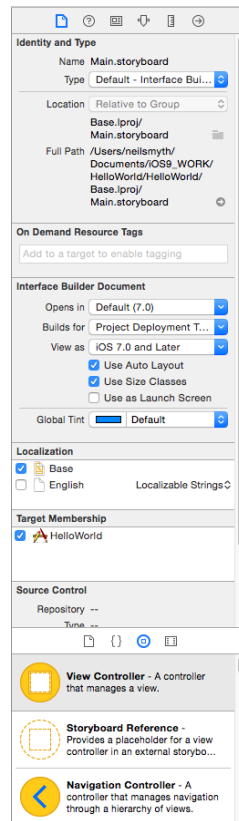


Figure 4-7

Along the top edge of the panel is a row of buttons which change the settings displayed in the upper half of the panel. By default the *File Inspector* is displayed. Options are also provided to display quick help, the *Identity Inspector*, *Attributes Inspector*, *Size Inspector* and *Connections Inspector*. Before proceeding, take some time to review each of these selections to gain some familiarity with the configuration options each provides. Throughout the remainder of this book extensive use of these inspectors will be made.

The lower section of the panel may default to displaying the file template library. Above this panel is another toolbar containing buttons to display other categories. Options include frequently used code snippets to save on typing when writing code, the Object Library and the Media Library. For the purposes of this tutorial we need to display the Object Library so click on the appropriate toolbar button (represented by the circle with a small square in the center). This will display the UI components that can be used to construct our user interface. Move the cursor to the line above the lower toolbar and click and drag to increase the amount of space available for the library if required. The layout of the items in the library may also be switched from a single column of objects with descriptions to multiple columns without descriptions by clicking on the button located in the bottom left hand corner of the panel and to the left of the search box.

4.3 Changing Component Properties

With the property panel for the View selected in the main panel, we will begin our design work by changing the background color of this view. Start by making sure the View is selected and that the *Attributes Inspector* (*View -> Utilities -> Show Attributes Inspector*) is displayed in the Utilities panel. Click on the white rectangle next to the *Background* label to invoke the *Colors* dialog. Using the color selection tool, choose a visually pleasing color and close the dialog. You will now notice that the view window has changed from white to the new color selection.

4.4 Adding Objects to the User Interface

The next step is to add a Label object to our view. To achieve this, either scroll down the list of objects in the Object Library panel to locate the Label object or, as illustrated in Figure 4-8, enter *Label* into the search box beneath the panel:



Figure 4-8

Having located the Label object, click on it and drag it to the center of the view so that the vertical and horizontal center guidelines appear. Once it is in position release the mouse button to drop it at that location. We have now added an instance of the UILabel class to the scene. Cancel the Object Library search by clicking on the “x” button on the right hand edge of the search field. Select the newly added label and stretch it horizontally so that it is approximately three times the current width. With the Label still selected, click on the centered alignment button in the Attributes Inspector (*View -> Utilities -> Show Attributes Inspector*) to center the text in the middle of the label view.

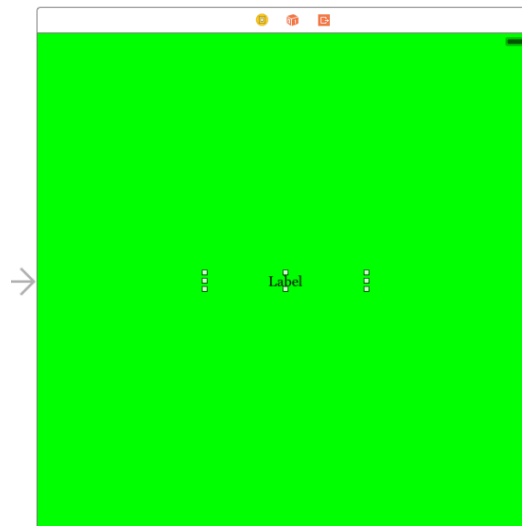


Figure 4-9

Double click on the text in the label that currently reads “Label” and type in “Hello World”. Locate the font setting property in the Attributes Inspector panel and click on the “T” button next to the font name to display the font selection menu. Change the Font setting from *System – System* to *Custom* and choose a larger font setting, for example a Georgia bold typeface with a size of 24 as shown in Figure 4-10:

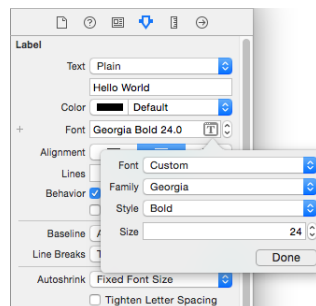


Figure 4-10

The final step is to add some layout constraints to ensure that the label remains centered within the containing view regardless of the size of screen on which the application ultimately runs. This involves the use of the Auto Layout capabilities of iOS, a topic which will be covered extensively in later chapters. For this example, simply select the Label object, display the Align menu as shown in Figure 4-11 and enable both the *Horizontally in Container* and *Vertically in Container* options with offsets of 0 before clicking on the *Add 2 Constraints* button.

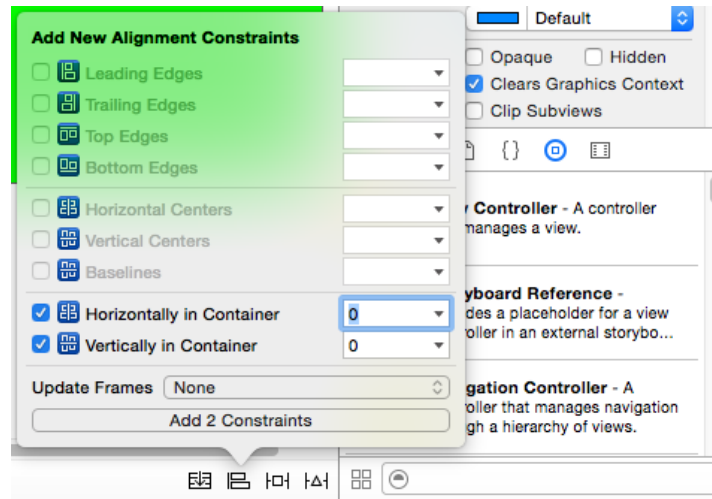


Figure 4-11

At this point, your View window will hopefully appear as outlined in Figure 4-12 (allowing, of course, for differences in your color and font choices).

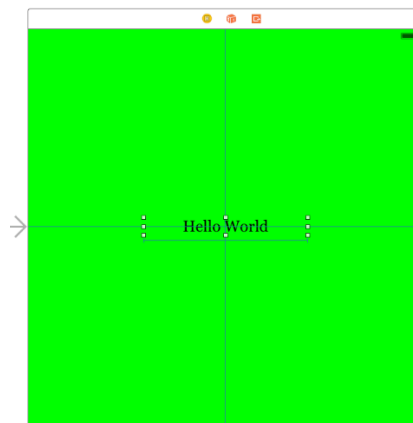


Figure 4-12

Before building and running the project it is worth taking a short detour to look at the Xcode *Document Outline* panel. This panel appears by default to the left of the Interface Builder panel and is controlled by the small button in the bottom left hand corner (indicated by the arrow in Figure 4-13) of the Interface Builder panel.

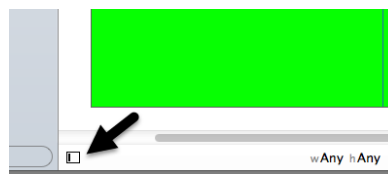


Figure 4-13

When displayed, the document outline shows a hierarchical overview of the elements that make up a user interface layout together with any constraints that have been applied to views in the layout.

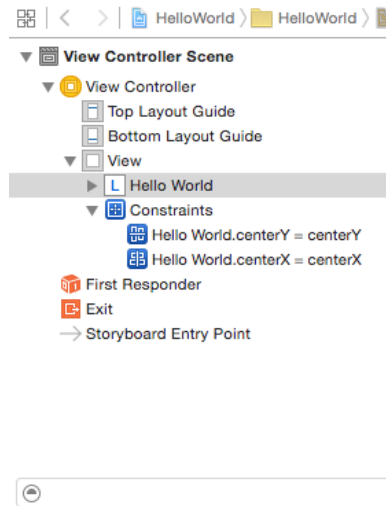


Figure 4-14

4.5 Building and Running an iOS 9 App in Xcode 7

Before an app can be run it must first be compiled. Once successfully compiled it may be run either within a simulator or on a physical iPhone, iPad or iPod Touch device. For the purposes of this chapter, however, it is sufficient to run the app in the simulator.

Within the main Xcode 7 project window, make sure that the menu located in the top left hand corner of the window (marked C in Figure 4-15) has the *iPhone 6* simulator option selected:

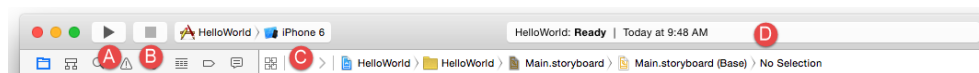


Figure 4-15

Click on the *Run* toolbar button (A) to compile the code and run the app in the simulator. The small panel in the center of the Xcode toolbar (D) will report the progress of the build process together with any problems or errors that cause the build process to fail. Once the app is built, the simulator will start and the HelloWorld app will run:



Figure 4-16

Note that the user interface appears as designed in the Interface Builder tool. Click on the stop button (B), change the target menu from iPhone 6 to iPad Air 2 and run the application again. Once again, the label will appear centered in the screen even with the larger screen size. Finally, verify that the layout is correct in landscape orientation by using the *Hardware* -> *Rotate Left* menu option. This indicates that the Auto Layout constraints are working and that we have designed a *universal* user interface for the project.

4.6 Running the App on a Physical iOS Device

Although the Simulator environment provides a useful way to test an app on a variety of different iOS device models, it is important to also test on a physical iOS device. Regardless of whether or not you have joined the Apple Developer Program at this point, it is possible to run the app on a physical device simply by connecting it to the development Mac system and selecting it as the run target within Xcode.

With a device connected to the development system, and an application ready for testing, refer to the device menu located in the Xcode toolbar. There is a reasonable chance that this will have defaulted to one of the iOS Simulator configurations (in the case of Figure 4-17, this is the iPhone 6s simulator).

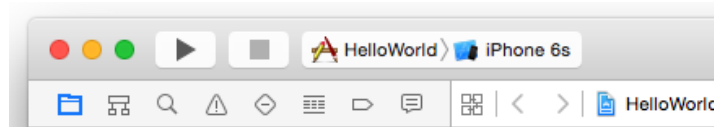


Figure 4-17

Switch to the physical device by selecting this menu and changing it to the device name as shown in Figure 4-18:

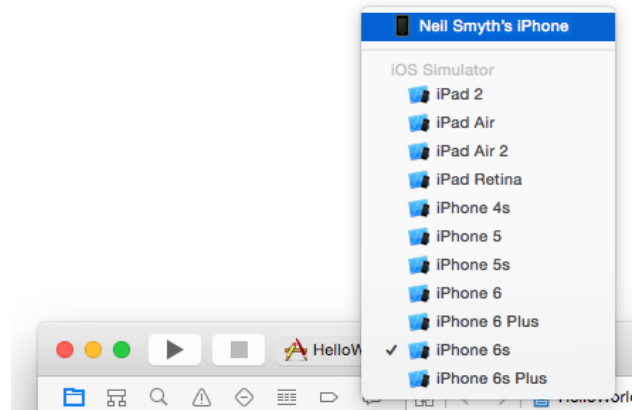


Figure 4-18

With the target device selected, make sure the device is unlocked and click on the run button at which point Xcode will install and launch the app on the device.

4.7 Managing Devices and Simulators

Currently connected iOS devices and the simulators configured for use with Xcode can be viewed and managed using the Xcode Devices window which is accessed via the *Window -> Devices* menu option. Figure 4-19, for example, shows a typical Device screen on a system where an iPhone and an Apple Watch have been detected:

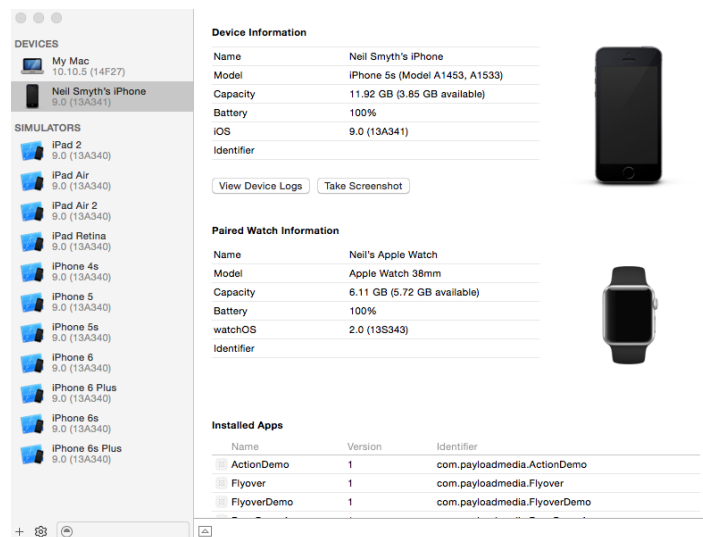


Figure 4-19

A wide range of simulator configurations are set up within Xcode by default. Other simulator configurations can be added by clicking on the + button located in the bottom left hand corner of the window. Once selected, a dialog will appear allowing the simulator to be configured in terms of device, iOS version and name.

The button displaying the gear icon in the bottom left corner allows simulators to be deleted, renamed or removed from the Xcode run target menu.

4.8 Dealing with Build Errors

As we have not actually written or modified any code in this chapter it is unlikely that any errors will be detected during the build and run process. In the unlikely event that something did get inadvertently changed thereby causing the build to fail it is worth taking a few minutes to talk about build errors within the context of the Xcode environment.

If for any reason a build fails, the status window in the Xcode toolbar will report that an error has been detected by displaying “Build” together with the number of errors detected and any warnings. In addition, the left hand panel of the Xcode window will update with a list of the errors. Selecting an error from this list will take you to the location in the code where corrective action needs to be taken.

4.9 Monitoring Application Performance

Another useful feature of Xcode is the ability to monitor the performance of an application while it is running. This information is accessed by displaying the *Debug Navigator*.

When Xcode is launched, the project navigator is displayed in the left hand panel by default. Along the top of this panel is a bar with a range of other options. The sixth option from the left displays the debug navigator when selected as illustrated in Figure 4-20. When displayed, this panel shows a number of real-time statistics relating to the performance of the currently running application such as memory, CPU usage, disk access, network activity and iCloud storage access.

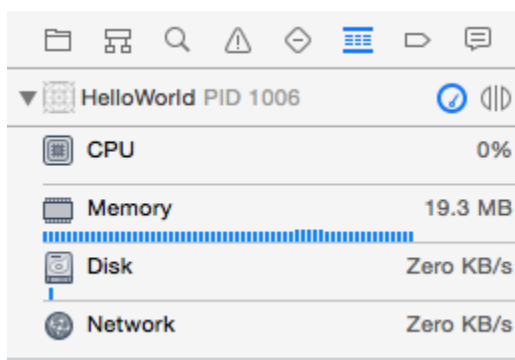


Figure 4-20

When one of these categories is selected, the main panel (Figure 4-21) updates to provide additional information about that particular aspect of the application’s performance:

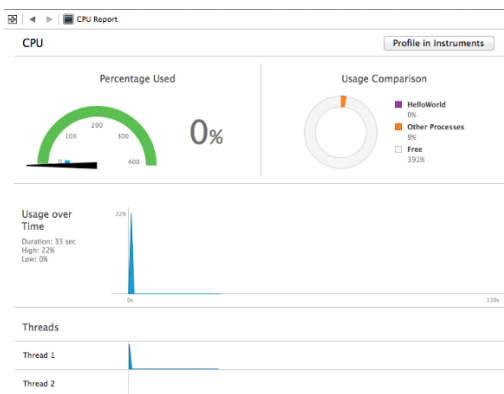


Figure 4-21

Yet more information can be obtained by clicking on the *Profile in Instruments* button in the top right hand corner of the panel.

4.10 An Exploded View of the User Interface Layout Hierarchy

Xcode 7 also provides an option to break the user interface layout out into a rotatable 3D view that shows how the view hierarchy for a user interface is constructed. This can be particularly useful for identifying situations where one view object is obscured by another appearing on top of it or a layout is not appearing as intended. To access the View Hierarchy in this mode, run the application and click on the *Debug View Hierarchy* button highlighted in Figure 4-22:

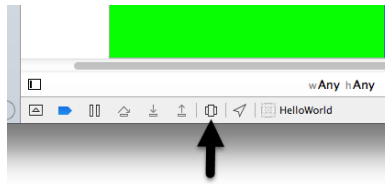


Figure 4-22

Once activated, a 3D “exploded” view of the layout will appear. Note that it may be necessary to click on the *Orient to 3D* button highlighted in Figure 4-23 to switch to 3D mode:



Figure 4-23

Figure 4-24 shows an example layout in this mode for a slightly more complex user interface than that created in this chapter:

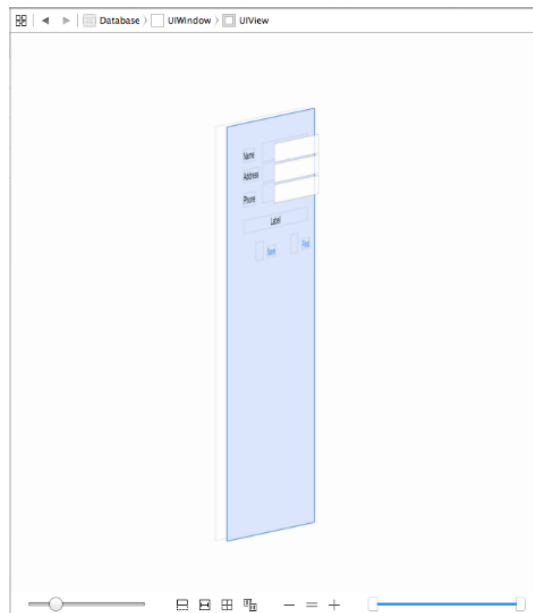


Figure 4-24

4.11 Summary

Applications are primarily created within the Xcode development environment. This chapter has served to provide a basic overview of the Xcode environment and to work through the creation of a very simple example application. Finally, a brief overview was provided of some of the performance monitoring features in Xcode 7. Many more features and capabilities of Xcode and Interface Builder will be covered in subsequent chapters of the book.

Chapter 5

5. An Introduction to Xcode 7 Playgrounds

Before introducing the Swift programming language in the chapters that follow, it is first worth learning about a feature of Xcode known as *Playgrounds*. Playgrounds are a feature introduced in Xcode 6 and enhanced in Xcode 7 that make learning Swift and experimenting with the iOS 9 SDK much easier. The concepts covered in this chapter can be put to use when experimenting with many of the introductory Swift code examples contained in the chapters that follow and will be of continued use in future when experimenting with many of the features of UIKit framework when designing dynamic user interfaces.

5.1 What is a Playground?

A playground is an interactive environment where Swift code can be entered and executed with the results appearing in real-time. This makes an ideal environment in which to learn the syntax of Swift and the visual aspects of iOS app development without the need to work continuously through the edit/compile/run/debug cycle that would ordinarily accompany a standard Xcode iOS project. With support for rich text comments, playgrounds are also a good way to document code as a teaching environment.

5.2 Creating a New Playground

To create a new Playground, start Xcode and select the *Get started with a playground* option from the welcome screen or select the *File -> New -> Playground* menu option. On the resulting options screen, name the playground *LearnSwift* and set the Platform menu to *iOS*. Click *Next* and choose a suitable file system location into which the playground should be saved.

Once the playground has been created, the following screen will appear ready for Swift code to be entered:

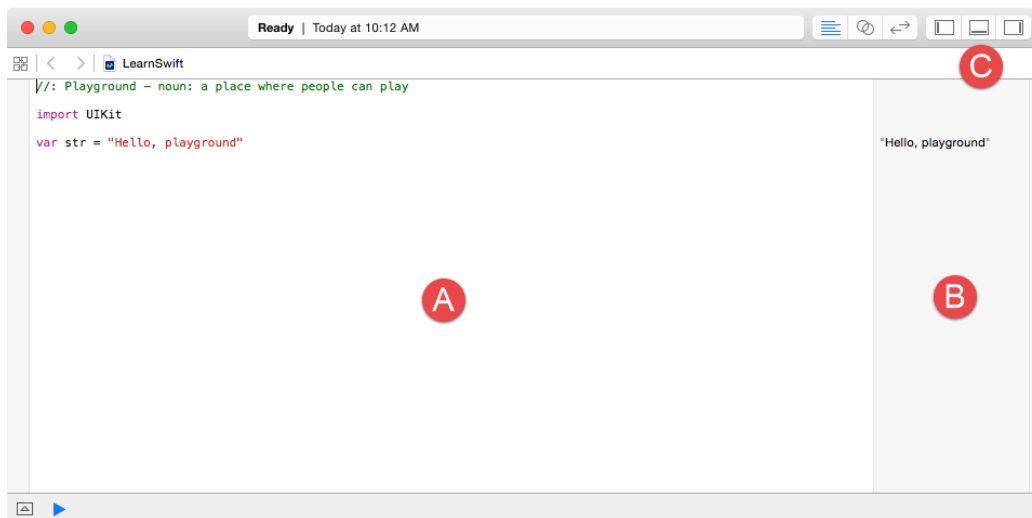


Figure 5-1

The panel on the left hand side of the window (marked A in Figure 5-1) is the *playground editor* where the lines of Swift code are entered. The right hand panel (marked B) is referred to as the *results panel* and is where the results of each Swift expression entered into the playground editor panel are displayed.

The cluster of three buttons at the right-hand side of the toolbar (marked C) are used to hide and display other panels within the playground window. The left most button displays the Navigator panel which provides access to the folders and files that make up the playground (marked A in Figure 5-2 below). The middle button, on the other hand, displays the Debug view (B) which displays code output and information about coding or runtime errors. The right most button displays the Utilities panel (C) where a variety of properties relating to the playground may be configured.

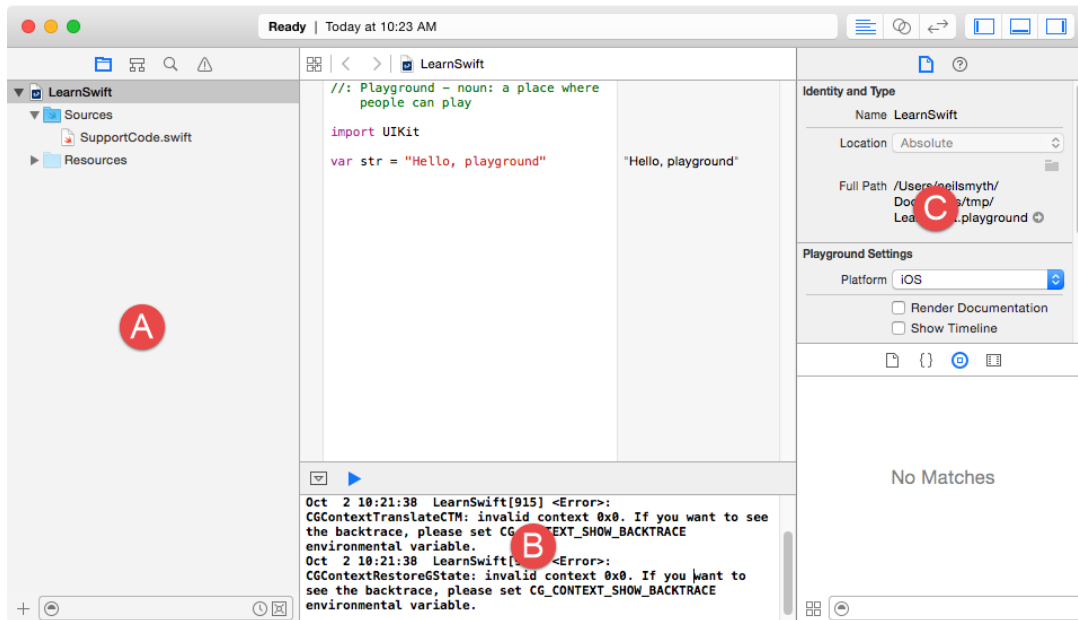


Figure 5-2

By far the quickest way to gain familiarity with the playground environment is to work through some simple examples.

5.3 A Basic Swift Playground Example

Perhaps the simplest of examples in any programming language (that at least does something tangible) is to write some code to output a single line of text. Swift is no exception to this rule so, within the playground window, begin by deleting the current Swift expression from the editor panel:

```
var str = "Hello, playground"
```

Next, enter a line of Swift code that reads as follows:

```
print("Welcome to Swift")
```

All that the code does is make a call to the built-in Swift *print* function which takes as a parameter a string of characters to be displayed on the console. Those familiar with other programming languages will note the absence of a semi-colon at the end of the line of code. In Swift, semi-colons are optional and generally only used as a separator when multiple statements occupy the same line of code.

Note that after entering the line of code, the results panel to the right of the editing panel is now showing the output from the print call as highlighted in Figure 5-3:

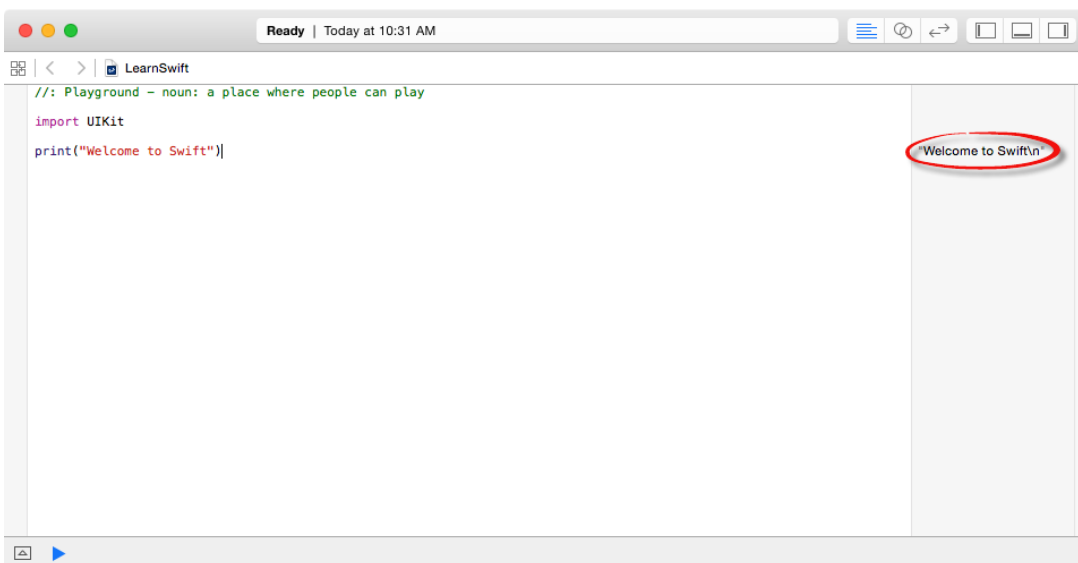


Figure 5-3

5.4 Viewing Results

Playgrounds are particularly useful when working and experimenting with Swift algorithms. This can be useful when combined with the Quick Look feature. Remaining within the playground editor, enter the following lines of code beneath the existing print statement:

```
var x = 10

for index in 1...20 {
    let y = index * x--
}
```

This expression repeats a loop 20 times, performing an arithmetic expression on each iteration of the loop. Once the code has been entered into the editor, the playground will execute the loop and display in the results panel the number of times the loop was performed. More interesting information, however, may be obtained by hovering the mouse pointer over the results line so that two additional buttons appear as shown in Figure 5-4:



Figure 5-4

The left most of the two buttons is the *Quick Look* button which, when selected, will show a popup panel displaying the results as shown in Figure 5-5:

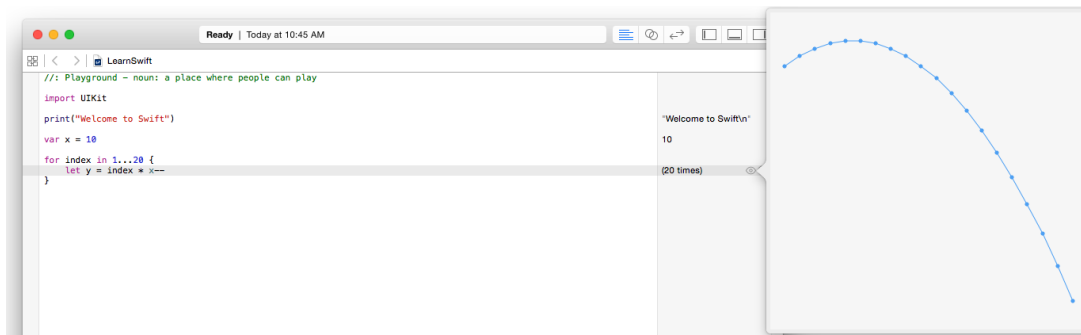


Figure 5-5

The right-most button is the *Show Result* button which, when selected, displays the results in-line with the code:



Figure 5-6

5.5 Enabling the Timeline Slider

A useful tool when inspecting the results of a code sequence is the timeline slider. Switched off by default, the slider can be enabled by displaying the Utilities panel (Marked C in Figure 5-2) and enabling the *Show Timeline* check box as illustrated in Figure 5-7:

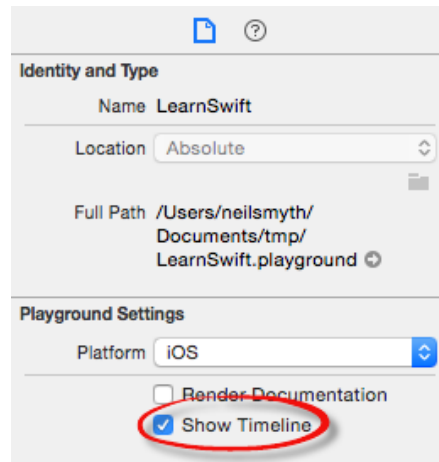


Figure 5-7

Once enabled, the timeline appears as a slider located along the bottom edge of the playground panel and can be moved to view the prevailing results at different points in the value history. Sliding it to the left, for example, will highlight and display the different values in the graph:

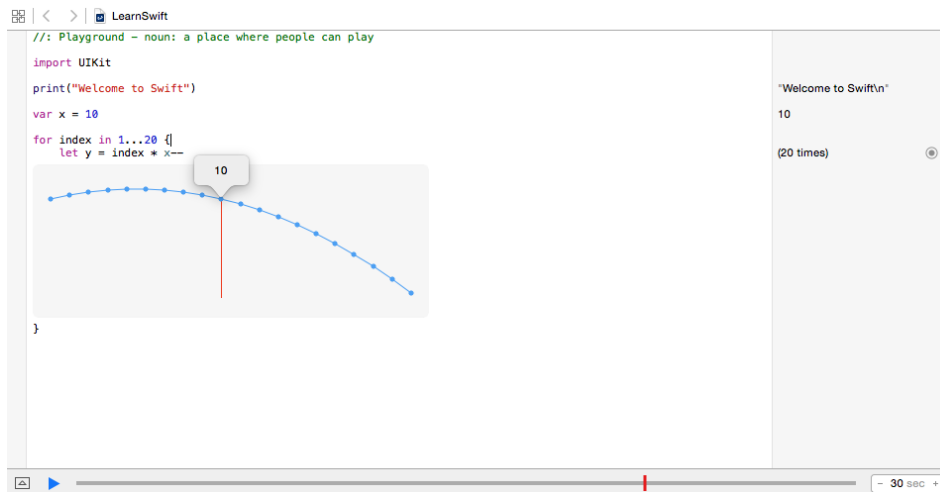


Figure 5-8

Clicking on the blue run button located to the left of the timeline slider will re-run the code within the playground.

5.6 Adding Rich Text Comments

Rich text comments allow the code within a playground to be documented in a way that is easy to format and read. A single line of text can be marked as being rich text by preceding it with a `/// marker. For example:`

```
/// This is a single line of documentation text
```

Blocks of text can be added by wrapping the text in `/*:` and `*/` comment markers:

```
/*:
This is a block of documentation text that is intended
to span multiple lines
*/
```

The rich text uses the Markdown markup language and allows text to be formatted using a lightweight and easy to use syntax. A heading, for example, can be declared by prefixing the line with a `#` character while text is displayed in italics when wrapped in `'*'` characters. Bold text, on the other hand, involves wrapping the text in `'**'` character sequences. It is also possible to configure bullet points by prefixing each line with a single `'*'`. Among the many other features of Markdown are the ability to embed images and hyperlinks into the content of a rich text comment.

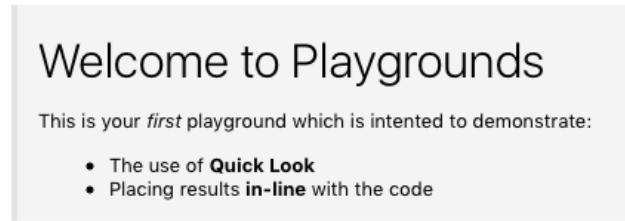
To see rich text comments in action, enter the following markdown content into the playground editor immediately after the `print("Welcome to Swift")` line of code:


```
/*:
# Welcome to Playgrounds
This is your first playground which is intended to demonstrate:
* The use of Quick Look
* Placing results in-line with the code
*/
```

As the comment content is added it is said to be displayed in *raw markup* format. To display in *rendered markup* format, select the *Editor -> Show Rendered Markup* menu option. Once rendered, the above rich text should appear as illustrated in Figure 5-9:

```
import UIKit

print("Welcome to Swift")
```



```
var x = 10
```

Figure 5-9

Detailed information about the Markdown syntax can be found online at the following URL:

https://developer.apple.com/library/ios/documentation/Swift/Reference/Playground_Ref/Chapters/MarkupReference.html

5.7 Working with Playground Pages

A playground can consist of multiple pages, with each page containing its own code, resources and rich text comments. So far, the playground used in this chapter contains a single page. Add an additional page to the playground now by selecting the *File -> New -> Playground Page* menu option. Once added, click on the left most of the three view buttons (marked C in Figure 5-1) to display the Navigator panel. Note that two pages are now listed in the Navigator named “Untitled Page” and “Untitled Page 2”. Select and then click a second time on the “Untitled Page 2” entry so that the name becomes editable and change the name to *UIKit Examples* as outlined in Figure 5-10:

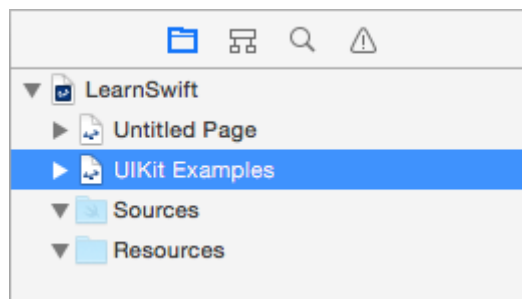


Figure 5-10

Note that the newly added page has Markdown links which, when clicked, navigate to the previous or next page in the playground.

5.8 Working with UIKit in Playgrounds

The playground environment is not restricted to simple Swift code statements. Much of the power of the iOS 9 SDK is also available for experimentation within a playground.

When working with UIKit within a playground page it is necessary to import the iOS UIKit Framework. The UIKit Framework contains most of the classes necessary to implement user interfaces for iOS applications and is an area which will be covered in significant detail throughout the book. An extremely powerful feature of playgrounds is that it is also possible to work with UIKit along with many of the other frameworks that comprise the iOS 9 SDK.

The following code, for example, imports the UIKit framework, creates a UILabel instance and sets color, text and font properties on it:

```
import UIKit
```

An Introduction to Xcode 7 Playgrounds

```
let myLabel = UILabel(frame: CGRectMake(0, 0, 200, 50))
myLabel.backgroundColor = UIColor.redColor()
myLabel.text = "Hello Swift"
myLabel.textAlignment = .Center
myLabel.font = UIFont(name: "Georgia", size: 24)
myLabel
```

Enter this code into the playground editor on the UIKit Examples page (the line importing the Foundation framework can be removed) and note that this is a good example of how the Quick Look feature can be useful. Each line of the example Swift code configures a different aspect of the appearance of the UILabel instance. Clicking on the Quick Look button for the first line of code will display an empty view (since the label exists but has yet to be given any visual attributes). Clicking on the Quick Look button in the line of code which sets the background color, on the other hand, will show the red label:

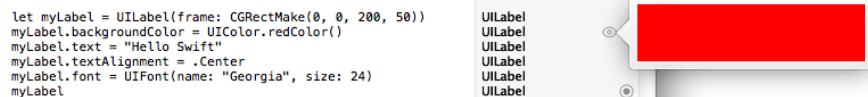


Figure 5-11

Similarly, the quick look view for the line where the text property is set will show the red label with the "Hello Swift" text left aligned:

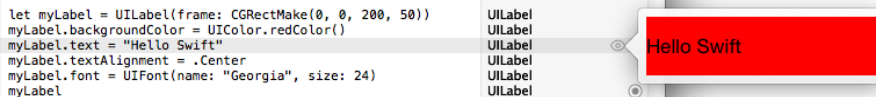


Figure 5-12

The font setting quick look view on the other hand displays the UILabel with centered text and the larger Georgia font:



Figure 5-13

5.9 Adding Resources to a Playground

A new feature of playgrounds which was introduced with Xcode 7 is the ability to bundle and access resources such as image files in a playground. Within the Navigator panel, click on the right facing arrow to the left of the UIKit Examples page entry to unfold the page contents (Figure 5-14) and note the presence of a folder named *Resources*:

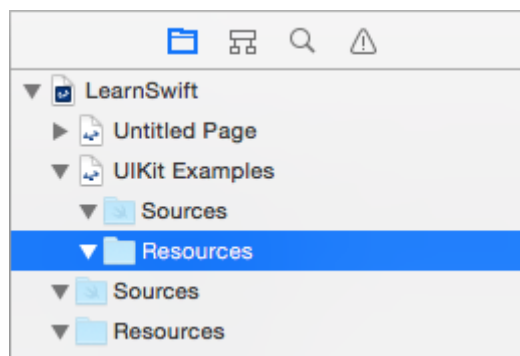


Figure 5-14

If you have not already done so, download and unpack the code samples archive from the following URL:

<http://www.ebookfrenzy.com/direct/ios9/>

Open a Finder window, navigate to the *playground_image* folder within the code samples folder and drag and drop the image file named *waterfall.png* onto the *Resources* folder beneath the UIKit Examples page in the Playground Navigator panel:

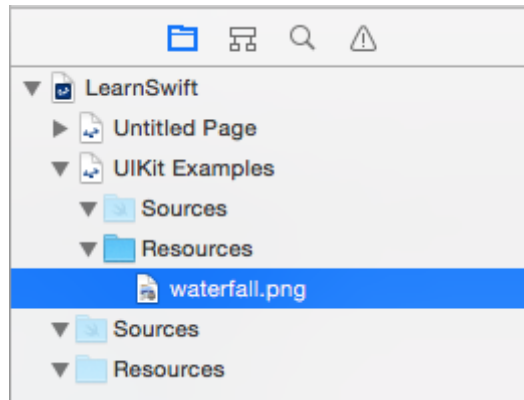


Figure 5-15

With the image added to the resources, add code to the page to create an image object and display the waterfall image on it:

```
let image = UIImage(named: "waterfall")
```

With the code added, use the Quick Look option to view the results of the code:

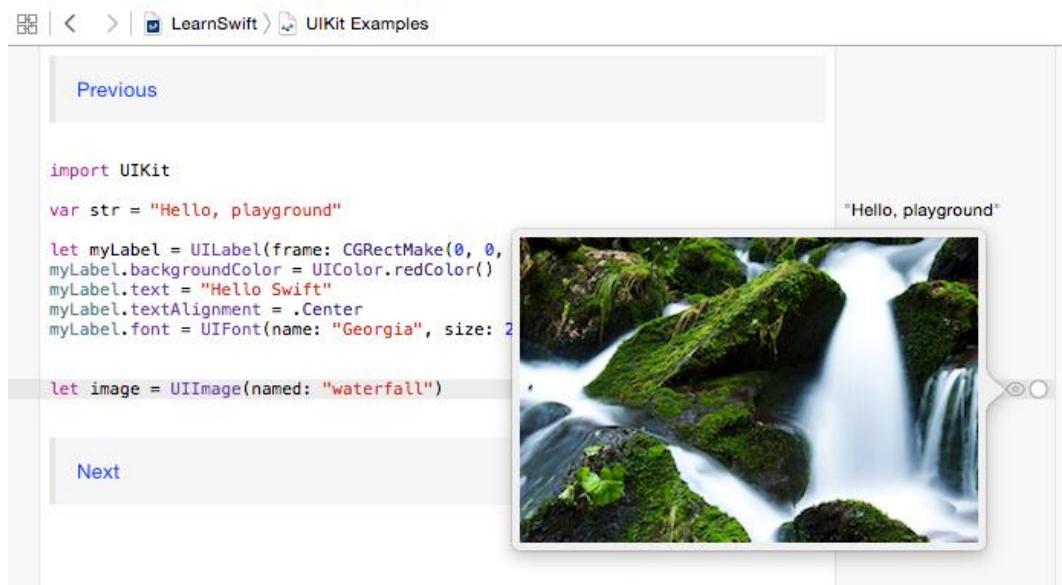


Figure 5-16

5.10 Working with Enhanced Live Views

So far in this chapter, all of the UIKit examples have involved presenting static user interface elements using the Quick Look and inline features. It is, however, also possible to test dynamic user interface behavior within a playground using the Xcode 7 Enhanced Live Views feature. To demonstrate live views in action, create a new page within the playground named *Live View Example*. Within the newly added page, remove the existing lines of Swift code before adding import statements for the UIKit framework and an additional playground module named XCPlayground:

```
import UIKit
import XCPlayground
```

The XCPlayground module provides a number of useful features for playgrounds including the ability to present a live view within the playground timeline.

Beneath the import statements, add the following code:

```
import UIKit
import XCPlayground

let container = UIView(frame: CGRectMake(0,0,200,200))
container.backgroundColor = UIColor.whiteColor()
let square = UIView(frame: CGRectMake(50,50,100,100))
square.backgroundColor = UIColor.redColor()
```

```

container.addSubview(square)

UIView.animateWithDuration(5.0, animations: {
    square.backgroundColor = UIColor.blueColor()
    let rotation = CGAffineTransformMakeRotation(3.14)
    square.transform = rotation
})

```

The code creates a `UIView` object to act as a container view and assigns it a white background color. A smaller view is then drawn positioned in the center of the container view and colored red. The second view is then added as a child of the container view. An animation is then used to change the color of the smaller view to blue and to rotate it through 360 degrees. If you are new to iOS programming rest assured that these areas will be covered in detail in later chapters. At this point the code is simply provided to highlight the capabilities of live views.

Clicking on any of the Quick Look buttons will show a snapshot of the views at each stage in the code sequence. None of the quick look views, however, show the dynamic animation. To see how the animation code works it will be necessary to use the live view playground feature.

The `XCPlayground` module includes a method named `XCPShowView` which can be used to execute the live view within the playground timeline. To display the timeline, click on the toolbar button containing the interlocking circles as highlighted in Figure 5-17:



Figure 5-17

When clicked, this button displays the Assistant Editor panel containing the timeline. Once the timeline is visible, add the call to `XCPShowView` as follows:

```

import UIKit
import XCPlayground

let container = UIView(frame: CGRectMake(0,0,200,200))

XCPShowView("My View", view: container)

container.backgroundColor = UIColor.whiteColor()
let square = UIView(frame: CGRectMake(50,50,100,100))
square.backgroundColor = UIColor.redColor()

container.addSubview(square)

UIView.animateWithDuration(5.0, animations: {
    square.backgroundColor = UIColor.blueColor()
    let rotation = CGAffineTransformMakeRotation(3.14)
    square.transform = rotation
})

```

Once the call has been added, the views should appear in the timeline (Figure 5-18). During the 5 second animation duration, the red square should rotate through 360 degrees while gradually changing color to blue:

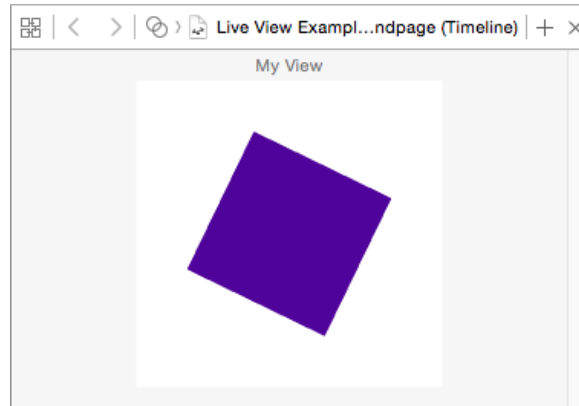


Figure 5-18

To repeat the execution of the code in the playground page, select the *Editor -> Execute Playground* menu option or click on the blue run button located next to the timeline slider. If the square stop button is currently displayed in place of the run button, click on it to stop execution and redisplay the run button. The different stages of the animation may also be viewed by moving the timeline slider located along the bottom edge of the playground window. Since the animation only lasts 5 seconds the length of time covered by the slider may also be reduced to 5 seconds using the control located at the end of the slider:

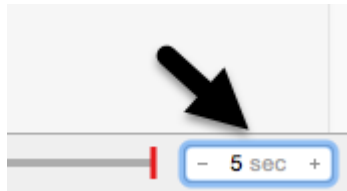


Figure 5-19

5.11 When to Use Playgrounds

Clearly Swift Playgrounds provide an ideal environment for learning to program using the Swift programming language and the use of playgrounds in the Swift introductory chapters that follow is recommended.

It is also important to keep in mind that playgrounds will remain useful long after the basics of Swift have been learned and will become increasingly useful when moving on to more advanced areas of iOS development.

The iOS 9 SDK is a vast collection of Frameworks and classes and it is not unusual for even experienced developers to need to experiment with unfamiliar aspects of iOS development before adding code to a project. Historically this has involved creating a temporary iOS Xcode project and then repeatedly looping through the somewhat cumbersome edit, compile, run cycle to arrive at a programming solution. Rather than fall into this habit, consider having a playground on standby to carry out experiments during your project development work.

5.12 Summary

This chapter has introduced the concept of playgrounds. Playgrounds provide an environment in which Swift code can be entered and the results of that code viewed dynamically. This provides an excellent environment both for learning the Swift programming language and for experimenting with many of the classes and APIs included in the iOS 9 SDK without the need to create Xcode projects and repeatedly edit, compile and run code.

6. Swift Data Types, Constants and Variables

Prior to the introduction of iOS 8, the stipulated programming language for the development of iOS applications was Objective-C. When Apple announced iOS 8, however, the company also introduced an alternative to Objective-C in the form of the new Swift programming language.

Due entirely to the popularity of iOS, Objective-C has become one of the most widely used programming languages today. With origins firmly rooted in the 40 year-old C Programming Language, however, and in spite of recent efforts to modernize some aspects of the language syntax, Objective-C was beginning to show its age.

Swift, on the other hand, is an entirely new programming language designed specifically to make programming easier, faster and less prone to programmer error. Starting with a clean slate and no burden of legacy, Swift is a new and innovative language with which to develop applications for both iOS and Mac OS X with the advantage that much of the syntax will be familiar to those with experience of other programming languages.

The introduction of Swift aside, it is still perfectly acceptable to continue to develop applications using Objective-C. Indeed, it is also possible to mix both Swift and Objective-C within the same application code base. That being said, Apple clearly sees the future of iOS and Mac OS X development in terms of Swift rather than Objective-C. In recognition of this fact, all of the examples in this book are implemented using Swift. Before moving on to those examples, however, the next several chapters will provide an overview and introduction to Swift programming. The intention of these chapters is to provide enough information so that you can begin to confidently program using Swift. For an exhaustive and in-depth guide to all the features, intricacies and capabilities of Swift, some time spent reading Apple's excellent book entitled "The Swift Programming Language" (available free of charge from within the Apple iBookStore) is strongly recommended.

6.1 Using a Swift Playground

Both this and the following few chapters are intended to introduce the basics of the Swift programming language. As outlined in the previous chapter, entitled *An Introduction to Swift Playgrounds* the best way to learn Swift is to experiment within a Swift playground environment. Before starting this chapter, therefore, create a new playground and use it to try out the code in both this and the other Swift introduction chapters that follow.

6.2 Swift Data Types

When we look at the different types of software that run on computer systems and mobile devices, from financial applications to graphics intensive games, it is easy to forget that computers are really just binary machines. Binary systems work in terms of 0 and 1, true or false, set and unset. All the data sitting in RAM, stored on disk drives and flowing through circuit boards and buses are nothing more than sequences of 1s and 0s. Each 1 or 0 is referred to as a *bit* and bits are grouped together in blocks of 8, each group being referred to as a *byte*. When people talk about 32-bit and 64-bit computer systems they are talking about the number of bits that can be handled simultaneously by the CPU bus. A 64-bit CPU, for example, is able to handle data in 64-bit blocks, resulting in faster performance than a 32-bit based system.

Humans, of course, don't think in binary. We work with decimal numbers, letters and words. In order for a human to easily (easily being a relative term in this context) program a computer, some middle ground between human and computer thinking is needed. This is where programming languages such as Swift come into play. Programming languages allow humans to express instructions to a computer in terms and structures we understand, and then compile that down to a format that can be executed by a CPU.

One of the fundamentals of any program involves data, and programming languages such as Swift define a set of *data types* that allow us to work with data in a format we understand when programming. For example, if we want to store a number in a Swift program we could do so with syntax similar to the following:

```
var mynumber = 10
```

In the above example, we have created a variable named *mynumber* and then assigned to it the value of 10. When we compile the source code down to the machine code used by the CPU, the number 10 is seen by the computer in binary as:

```
1010
```

Similarly, we can express a letter, the visual representation of a digit ('0' through to '9') or punctuation mark (referred to in computer terminology as *characters*) using the following syntax:

```
var myletter = "c"
```

Once again, this is understandable by a human programmer, but gets compiled down to a binary sequence for the CPU to understand. In this case, the letter 'c' is represented by the decimal number 99 using the ASCII table (an internationally recognized standard that assigns numeric values to human readable characters). When converted to binary, it is stored as:

```
10101100011
```

Now that we have a basic understanding of the concept of data types and why they are necessary we can take a closer look at some of the more commonly used data types supported by Swift.

6.2.1 Integer Data Types

Swift integer data types are used to store whole numbers (in other words a number with no decimal places). Integers can be *signed* (capable of storing positive, negative and zero values) or *unsigned* (positive and zero values only).

Swift provides support for 8, 16, 32 and 64 bit integers (represented by the `Int8`, `Int16`, `Int32` and `Int64` types respectively). The same variants are also available for unsigned integers (`UInt8`, `UInt16`, `UInt32` and `UInt64`).

In general, Apple recommends using the `Int` data type rather than one of the above specifically sized data types. The `Int` data type will use the appropriate integer size for the platform on which the code is running.

All integer data types contain bounds properties which can be accessed to identify the minimum and maximum supported values of that particular type. The following code, for example, outputs the minimum and maximum bounds for the 32-bit signed integer data type:

```
print("Int32 Min = \(Int32.min) Int32 Max = \(Int32.max)")
```

When executed, the above code will generate the following output:

```
Int32 Min = -2147483648 Int32 Max = 2147483647
```

6.2.2 Floating Point Data Types

The Swift floating point data types are able to store values containing decimal places. For example, 4353.1223 would be stored in a floating point data type. Swift provides two floating point data types in the form of *Float* and *Double*. Which type to use depends on the size of value to be stored and the level of precision required. The `Double` type can be used to store up to 64-bit floating point numbers with a level of precision of 15 decimal places or greater. The `Float` data type, on the other hand, is limited to 32-bit floating point numbers and offers a level of precision as low as 6 decimal places depending on the native platform on which the code is running.

6.2.3 Bool Data Type

Swift, like other languages, includes a data type for the purpose of handling true or false (1 or 0) conditions. Two Boolean constant values (*true* and *false*) are provided by Swift specifically for working with Boolean data types.

6.2.4 Character Data Type

The Swift Character data type is used to store a single character of rendered text such as a letter, numerical digit, punctuation mark or symbol. Internally characters in Swift are stored in the form of *grapheme clusters*. A grapheme cluster is made of two or more Unicode code points that are combined to represent a single visible character.

The following lines assign a variety of different characters to Character type variables:

```
var myChar1 = "f"
var myChar2 = ":"
var myChar3 = "X"
```

Characters may also be referenced using Unicode code points. The following example assigns the 'X' character to a variable using Unicode:

```
var myChar4 = "\u{0058}"
```

6.2.5 String Data Type

The String data type is a sequence of characters that typically make up a word or sentence. In addition to providing a storage mechanism, the String data type also includes a range of string manipulation features allowing strings to be searched, matched, concatenated and modified.

Strings can also be constructed using combinations of strings, variables, constants, expressions, and function calls using a concept referred to as *string interpolation*. For example, the following code creates a new string from a variety of different sources using string interpolation before outputting it to the console:

```
var userName = "John"
var inboxCount = 25
let maxCount = 100

var message = "\(userName) has \(inboxCount) message. Message capacity remaining is
\((maxCount - inboxCount))"

print(message)
```

When executed, the code will output the following message:

```
John has 25 messages. Message capacity remaining is 75 messages.
```

6.2.6 Special Characters/Escape Sequences

In addition to the standard set of characters outlined above, there is also a range of *special characters* (also referred to as *escape sequences*) available for specifying items such as a new line, tab or a specific Unicode value within a string. These special characters are identified by prefixing the character with a backslash (a concept referred to as *escaping*). For example, the following assigns a new line to the variable named `newline`:

```
var newline = "\n"
```

In essence, any character that is preceded by a backslash is considered to be a special character and is treated accordingly. This raises the question as to what to do if you actually want a backslash character. This is achieved by *escaping* the backslash itself:

```
var backslash = "\\"
```

Commonly used special characters supported by Swift are as follows:

- `\n` - New line
- `\r` - Carriage return
- `\t` - Horizontal tab
- `\\` - Backslash
- `\"` - Double quote (used when placing a double quote into a string declaration)
- `\'` - Single quote (used when placing a single quote into a string declaration)
- `\u{nn}` – Single byte Unicode scalar where `nn` is replaced by two hexadecimal digits representing the Unicode character.
- `\u{nnnn}` – Double byte Unicode scalar where `nnnn` is replaced by four hexadecimal digits representing the Unicode character.
- `\U{nnnnnnnn}` – Four byte Unicode scalar where `nnnnnnnn` is replaced by eight hexadecimal digits representing the Unicode character.

6.3 Swift Variables

Variables are essentially locations in computer memory reserved for storing the data used by an application. Each variable is given a name by the programmer and assigned a value. The name assigned to the variable may then be used in the Swift code to access the value assigned to that variable. This access can involve either reading the value of the variable, or changing the value. It is, of course, the ability to change the value of variables which gives them the name *variable*.

6.4 Swift Constants

A constant is similar to a variable in that it provides a named location in memory to store a data value. Constants differ in one significant way in that once a value has been assigned to a constant it cannot subsequently be changed.

Constants are particularly useful if there is a value which is used repeatedly throughout the application code. Rather than use the value each time, it makes the code easier to read if the value is first assigned to a constant which is then referenced in the code. For example, it might not be clear to someone reading your Swift code why you used the value 5 in an expression. If, instead of the value 5, you use a constant named `interestRate` the purpose of the value becomes much clearer. Constants also have the advantage that if the programmer needs to change a widely used value, it only needs to be changed once in the constant declaration and not each time it is referenced.

As with variables, constants have a type, a name and a value. Unlike variables, however, once a value has been assigned to a constant, that value cannot subsequently be changed.

6.5 Declaring Constants and Variables

Variables are declared using the *var* keyword and may be initialized with a value at creation time. If the variable is declared without an initial value it must be declared as being *optional* (a topic which will be covered later in this chapter). The following, for example, is a typical variable declaration:

```
var userCount = 10
```

Constants are declared using the *let* keyword.

```
let maxUserCount = 20
```

For greater code efficiency and execution performance, Apple recommends the use of constants rather than variables whenever possible.

6.6 Type Annotations and Type Inference

Swift is categorized as a *type safe* programming language. This essentially means that once the data type of a variable has been identified, that variable cannot subsequently be used to store data of any other type without inducing a compilation error. This contrasts to *loosely typed* programming languages where a variable, once declared, can subsequently be used to store other data types.

There are two ways in which the type of a constant or variable will be identified. One approach is to use a *type annotation* at the point the variable or constant is declared in the code. This is achieved by placing a colon after the constant or variable name followed by the type declaration. The following line of code, for example, declares a variable named `userCount` as being of type `Int`:

```
var userCount: Int = 10
```

In the absence of a type annotation in a declaration, the Swift compiler uses a technique referred to as *type inference* to identify the type of the constant or variable. When relying on type inference, the compiler looks to see what type of value is being assigned to the constant or variable at the point that it is initialized and uses that as the type. Consider, for example, the following variable and constant declarations:

```
var signalStrength = 2.231
let companyName = "My Company"
```

During compilation of the above lines of code, Swift will infer that the `signalStrength` variable is of type `Double` (type inference in Swift defaults to `Double` for all floating point numbers) and that the `companyName` constant is of type `String`.

When a constant is declared without a type annotation it must be assigned a value at the point of declaration:

```
let bookTitle = "iOS 9 App Development Essentials"
```

If a type annotation is used when the constant is declared, however, the value can be assigned later in the code. For example:

```
let bookTitle: String
·
·
if iosBookType {
    bookTitle = "iOS 9 App Development Essentials"
} else {
    bookTitle = "Android Studio Development Essentials"
}
```

It is important to note that a value may only be assigned to a constant once. A second attempt to assign a value to a constant will result in a syntax error.

6.7 The Swift Tuple

Before proceeding, now is a good time to introduce the Swift tuple. The tuple is perhaps one of the simplest, yet most powerful features of the Swift programming language. A tuple is, quite simply, a way to temporarily group together multiple values into a single entity. The items stored in a tuple can be of any type and there are no restrictions requiring that those values all be of the same type. A tuple could, for example, be constructed to contain an `Int` value, a `Float` value and a `String` as follows:

```
let myTuple = (10, 432.433, "This is a String")
```

The elements of a tuple can be accessed using a number of different techniques. A specific tuple value can be accessed simply by referencing the index position (with the first value being at index position 0). The code below, for example, extracts the string resource (at index position 2 in the tuple) and assigns it to a new string variable:

```
let myTuple = (10, 432.433, "This is a String")
let myString = myTuple.2
print(myString)
```

Alternatively, all of the values in a tuple may be extracted and assigned to variables or constants in a single statement:

```
let (myInt, myFloat, myString) = myTuple
```

This same technique can be used to extract selected values from a tuple whilst ignoring others by replacing the values to be ignored with an underscore character. The following code fragment extracts the integer and string values from the tuple and assigns them to variables, but ignores the floating point value:

```
var (myInt, _, myString) = myTuple
```

When creating a tuple it is also possible to assign a name to each value:

```
let myTuple = (count: 10, length: 432.433, message: "This is a String")
```

The names assigned to the values stored in a tuple may then be used to reference those values in code. For example to output the *message* string value from the *myTuple* instance, the following line of code could be used:

```
print(myTuple.message)
```

Perhaps the most powerful use of tuples is, as will be seen in later chapters, the ability to return multiple values from a function.

6.8 The Swift Optional Type

The Swift optional data type is a new concept that does not exist in most other programming languages. The purpose of the optional type is to provide a safe and consistent approach to handling situations where a variable or constant may not have any value assigned to it.

Variables are declared as being optional by placing a ? character after the type declaration. The following code declares an optional *Int* variable named *index*:

```
var index: Int?
```

The variable *index* can now either have an integer value assigned to it, or have nothing assigned to it. Behind the scenes, and as far as the compiler and runtime are concerned, an optional with no value assigned to it actually has a value of *nil*.

An optional can easily be tested (typically using an *if* statement) to identify whether or not it has a value assigned to it as follows:

```
var index: Int?

if index != nil {
    // index variable has a value assigned to it
} else {
    // index variable has no value assigned to it
}
```

If an optional has a value assigned to it, that value is said to be “wrapped” within the optional. The value wrapped in an optional may be accessed using a concept referred to as *forced unwrapping*. This simply means that the underlying value is extracted from the optional data type, a procedure that is performed by placing an exclamation mark (!) after the optional name.

To explore this concept of unwrapping optional types in more detail, consider the following code:

```
var index: Int?

index = 3

var treeArray = ["Oak", "Pine", "Yew", "Birch"]

if index != nil {
    print(treeArray[index!])
}
```

```

} else {
    print("index does not contain a value")
}

```

The code simply uses an optional variable to hold the index into an array of strings representing the names of tree types (Swift arrays will be covered in more detail in the chapter entitled *Working with Array and Dictionary Collections in Swift*). If the index optional variable has a value assigned to it, the tree name at that location in the array is printed to the console. Since the index is an optional type, the value has been unwrapped by placing an exclamation mark after the variable name:

```
print(treeArray[index!])
```

Had the index not been unwrapped (in other words the exclamation mark omitted from the above line), the compiler would have issued an error similar to the following:

```
Value of optional type Int? not unwrapped
```

As an alternative to forced unwrapping, the value assigned to an optional may be allocated to a temporary variable or constant using *optional binding*, the syntax for which is as follows:

```

if let constantname = optionalName {
}

if var variablename = optionalName {
}

```

The above constructs perform two tasks. In the first instance, the statement ascertains whether or not the designated optional contains a value. Secondly, in the event that the optional has a value, that value is assigned to the declared constant or variable and the code within the body of the statement is executed. The previous forced unwrapping example could, therefore, be modified as follows to use optional binding instead:

```

var index: Int?

index = 3

var treeArray = ["Oak", "Pine", "Yew", "Birch"]

if let myvalue = index {
    print(treeArray[myvalue])
} else {
    print("index does not contain a value")
}

```

In this case the value assigned to the index variable is unwrapped and assigned to a temporary constant named *myvalue* which is then used as the index reference into the array. Note that the *myvalue* constant is described as temporary since it is only available within the scope of the if statement. Once the if statement completes execution, the constant will no longer exist.

Optional binding may also be used to unwrap multiple optionals and include a Boolean test condition, the syntax for which is as follows:

```

if let constname1 = optName1, constname2 = optName2, optName3 = ... where <boolean statement> {
}

```

The following code, for example, uses optional binding to unwrap two optionals within a single statement:

```

var pet1: String?
var pet2: String?

pet1 = "cat"
pet2 = "dog"

```

```

if let firstPet = pet1, secondPet = pet2 {
    print(firstPet)
    print(secondPet)
} else {
    print("insufficient pets")
}

```

The code fragment below, on the other hand, also makes use of the *where* clause to test a Boolean condition:

```

if let firstPet = pet1, secondPet = pet2 where petCount > 1 {
    print(firstPet)
    print(secondPet)
} else {
    print("no pets")
}

```

In the above example, the optional binding will not be attempted unless the value assigned to *petCount* is greater than 1.

It is also possible to declare an optional as being *implicitly unwrapped*. When an optional is declared in this way, the underlying value can be accessed without having to perform forced unwrapping or optional binding. An optional is declared as being implicitly unwrapped by replacing the question mark (?) with an exclamation mark (!) in the declaration. For example:

```

var index: Int! // Optional is now implicitly unwrapped

index = 3

var treeArray = ["Oak", "Pine", "Yew", "Birch"]

if index != nil {
    print(treeArray[index])
} else {
    print("index does not contain a value")
}

```

With the index optional variable now declared as being implicitly unwrapped, it is no longer necessary to unwrap the value when it is used as an index into the array in the above print call.

One final observation with regard to optionals in Swift is that only optional types are able to have no value or a value of nil assigned to them. In Swift it is not, therefore, possible to assign a nil value to a non-optional variable or constant. The following declarations, for instance, will all result in errors from the compiler since none of the variables are declared as optional:

```

var myInt = nil // Invalid code
var myString: String = nil // Invalid Code
let myConstant = nil // Invalid code

```

6.9 Type Casting and Type Checking

When writing Swift code, situations will occur where the compiler is unable to identify the specific type of a value. This is often the case when a value of ambiguous or unexpected type is returned from a method or function call. In this situation it may be necessary to let the compiler know the type of value that your code is expecting or requires using the *as* keyword (a concept referred to as *type casting*).

The following code, for example, lets the compiler know that the value returned from the *objectForKey* method needs to be treated as a String type:

```

let myValue = record.objectForKey("comment") as! String

```

In fact, there are two types of casting which are referred to as *upcasting* and *downcasting*. Upcasting occurs when an object of a particular class is cast to one of its superclasses. Upcasting is performed using the *as* keyword and is also referred to as *guaranteed conversion* since the compiler can tell from the code that the cast will be successful. The UIButton class, for example, is a subclass of the UIControl class as shown in the fragment of the UIKit class hierarchy shown in Figure 6-1:

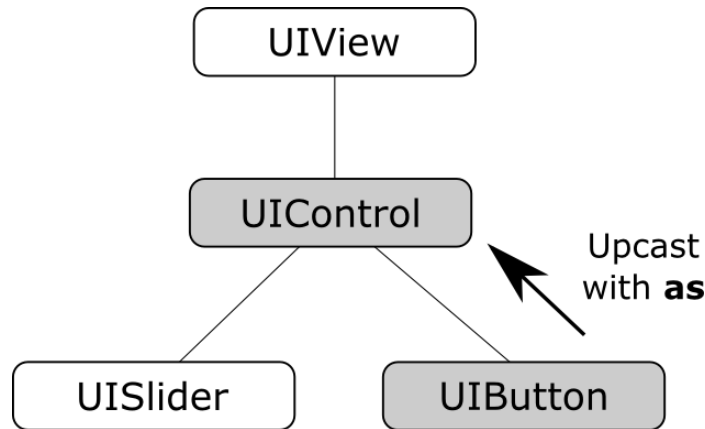


Figure 6-1

Since UIButton is a subclass of UIControl, the object can be safely upcast as follows:

```
let myButton: UIButton = UIButton()

let myControl = myButton as UIControl
```

Downcasting, on the other hand, occurs when a conversion is made from one class to another where there is no guarantee that the cast can be made safely or that an invalid casting attempt will be caught by the compiler. When an invalid cast is made in downcasting and not identified by the compiler it will most likely lead to an error at runtime.

Downcasting usually involves converting from a class to one of its subclasses. Downcasting is performed using the *as!* keyword syntax and is also referred to as *forced conversion*. Consider, for example, the UIKit UIScrollView class which has as subclasses both the UITableView and UITextView classes as shown in Figure 6-2:

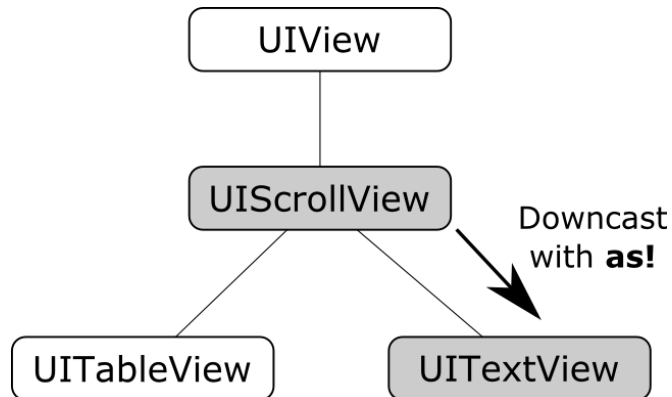


Figure 6-2

In order to convert a UIScrollView object to a UITextView class a downcast operation needs to be performed. The following code attempts to downcast a UIScrollView object to UITableView using the *guaranteed conversion* or *upcast* approach:

```
let myScrollView: UIScrollView = UIScrollView()

let myTextView = myScrollView as UITextView
```

When compiled, the above code will result in the following error:

```
'UIScrollView' is not convertible to 'UITextView'
```

The compiler is indicating that a UIScrollView instance cannot be safely converted to a UITextView class instance. This does not mean that it is incorrect to do so, the compiler is simply stating that it cannot guarantee the safety of the conversion for you. The downcast conversion must instead be forced using the *as!* Annotation:

```
let myTextView = myScrollView as! UITextView
```

Another useful approach to downcasting is to perform an optional binding using *as?*. If the conversion is performed successfully, an optional value of the specified type is returned, otherwise the optional value will be nil:

```
if let classB = classA as? UITextView {
```

```
    print("Type cast to UITextView succeeded")
} else {
    print("Type cast to UITextView failed")
}
```

It is also possible to *type check* a value using the *is* keyword. The following code, for example, checks that a specific object is an instance of a class named `MyClass`:

```
if myobject is MyClass {
    // myobject is an instance of MyClass
}
```

6.10 Summary

This chapter has begun the introduction to Swift by exploring data types together with an overview of how to declare constants and variables. The chapter has also introduced concepts such as type safety, type inference and optionals, each of which is an integral part of Swift programming and designed specifically to make code writing less prone to error.

7. Swift Operators and Expressions

So far we have looked at using variables and constants in Swift and also described the different data types. Being able to create variables is only part of the story however. The next step is to learn how to use these variables and constants in Swift code. The primary method for working with data is in the form of *expressions*.

7.1 Expression Syntax in Swift

The most basic Swift expression consists of an *operator*, two *operands* and an *assignment*. The following is an example of an expression:

```
var myresult = 1 + 2
```

In the above example, the (+) operator is used to add two operands (1 and 2) together. The *assignment operator* (=) subsequently assigns the result of the addition to a variable named *myresult*. The operands could just have easily been variables (or a mixture of constants and variables) instead of the actual numerical values used in the example.

In the remainder of this chapter we will look at the basic types of operators available in Swift.

7.2 The Basic Assignment Operator

We have already looked at the most basic of assignment operators, the = operator. This assignment operator simply assigns the result of an expression to a variable. In essence, the = assignment operator takes two operands. The left hand operand is the variable or constant to which a value is to be assigned and the right hand operand is the value to be assigned. The right hand operand is, more often than not, an expression which performs some type of arithmetic or logical evaluation, the result of which will be assigned to the variable or constant. The following examples are all valid uses of the assignment operator:

```
var x: Int? // Declare an optional Int variable
var y = 10 // Declare and initialize a second Int variable

x = 10 // Assign a value to x
x = x! + y // Assign the result of x + y to x
x = y // Assign the value of y to x
```

7.3 Swift Arithmetic Operators

Swift provides a range of operators for the purpose of creating mathematical expressions. These operators primarily fall into the category of *binary operators* in that they take two operands. The exception is the *unary negative operator* (-) which serves to indicate that a value is negative rather than positive. This contrasts with the *subtraction operator* (-) which takes two operands (i.e. one value to be subtracted from another). For example:

```
var x = -10 // Unary - operator used to assign -10 to variable x
x = x - 5 // Subtraction operator. Subtracts 5 from x
```

The following table lists the primary Swift arithmetic operators:

Operator	Description
-(unary)	Negates the value of a variable or expression
*	Multiplication
/	Division
+	Addition
-	Subtraction
%	Remainder/Modulo

Note that multiple operators may be used in a single expression.

For example:

```
x = y * 10 + z - 5 / 4
```

7.4 Compound Assignment Operators

In an earlier section we looked at the basic assignment operator (=). Swift provides a number of operators designed to combine an assignment with a mathematical or logical operation. These are primarily of use when performing an evaluation where the result is to be stored in one of the operands. For example, one might write an expression as follows:

```
x = x + y
```

The above expression adds the value contained in variable `x` to the value contained in variable `y` and stores the result in variable `x`. This can be simplified using the addition compound assignment operator:

```
x += y
```

The above expression performs exactly the same task as `x = x + y` but saves the programmer some typing.

Numerous compound assignment operators are available in Swift. The most frequently used of which are outlined in the following table:

Operator	Description
<code>x += y</code>	Add <code>x</code> to <code>y</code> and place result in <code>x</code>
<code>x -= y</code>	Subtract <code>y</code> from <code>x</code> and place result in <code>x</code>
<code>x *= y</code>	Multiply <code>x</code> by <code>y</code> and place result in <code>x</code>
<code>x /= y</code>	Divide <code>x</code> by <code>y</code> and place result in <code>x</code>
<code>x %= y</code>	Perform Modulo on <code>x</code> and <code>y</code> and place result in <code>x</code>
<code>x &&= y</code>	Assign to <code>x</code> the result of logical AND operation on <code>x</code> and <code>y</code>
<code>x = y</code>	Assign to <code>x</code> the result of logical OR operation on <code>x</code> and <code>y</code>

7.5 Increment and Decrement Operators

Another useful shortcut can be achieved using the Swift increment and decrement operators (also referred to as *unary operators* because they operate on a single operand). Consider the code fragment below:

```
x = x + 1 // Increase value of variable x by 1
x = x - 1 // Decrease value of variable x by 1
```

These expressions increment and decrement the value of `x` by 1. Instead of using this approach, however, it is quicker to use the `++` and `--` operators. The following examples perform exactly the same tasks as the examples above:

```
x++ // Increment x by 1
x-- // Decrement x by 1
```

These operators can be placed either before or after the variable name. If the operator is placed before the variable name, the increment or decrement operation is performed before any other operations are performed on the variable. For example, in the following code, `x` is incremented before it is assigned to `y`, leaving `y` with a value of 10:

```
var x = 9
var y = ++x
```

In the next example, however, the value of `x` (9) is assigned to variable `y` *before* the decrement is performed. After the expression is evaluated the value of `y` will be 9 and the value of `x` will be 8.

```
var x = 9
var y = x--
```

7.6 Comparison Operators

Swift also includes a set of logical operators useful for performing comparisons. These operators all return a Boolean result depending on the result of the comparison. These operators are *binary operators* in that they work with two operands.

Comparison operators are most frequently used in constructing program flow control logic. For example an *if* statement may be constructed based on whether one value matches another:

```
if x == y {
    // Perform task
}
```

The result of a comparison may also be stored in a *Bool* variable. For example, the following code will result in a *true* value being stored in the variable *result*:

```
var result: Bool?
var x = 10
var y = 20

result = x < y
```

Clearly 10 is less than 20, resulting in a *true* evaluation of the $x < y$ expression. The following table lists the full set of Swift comparison operators:

Operator	Description
$x == y$	Returns true if x is equal to y
$x > y$	Returns true if x is greater than y
$x >= y$	Returns true if x is greater than or equal to y
$x < y$	Returns true if x is less than y
$x <= y$	Returns true if x is less than or equal to y
$x != y$	Returns true if x is not equal to y

7.7 Boolean Logical Operators

Swift also provides a set of so called logical operators designed to return Boolean *true* or *false* values. These operators both return Boolean results and take Boolean values as operands. The key operators are NOT (!), AND (&&) and OR (||).

The NOT (!) operator simply inverts the current value of a Boolean variable, or the result of an expression. For example, if a variable named *flag* is currently true, prefixing the variable with a '!' character will invert the value to false:

```
var flag = true // variable is true
var secondFlag = !flag // secondFlag set to false
```

The OR (||) operator returns true if one of its two operands evaluates to true, otherwise it returns false. For example, the following code evaluates to true because at least one of the expressions either side of the OR operator is true:

```
if (10 < 20) || (20 < 10) {
    print("Expression is true")
}
```

The AND (&&) operator returns true only if both operands evaluate to be true. The following example will return false because only one of the two operand expressions evaluates to true:

```
if (10 < 20) && (20 < 10) {
    print("Expression is true")
}
```

7.8 Range Operators

Swift includes two useful operators that allow ranges of values to be declared. As will be seen in later chapters, these operators are invaluable when working with looping in program logic.

The syntax for the *closed range operator* is as follows:

```
x...y
```

This operator represents the range of numbers starting at *x* and ending at *y* where both *x* and *y* are included within the range. The range operator `5...8`, for example, specifies the numbers 5, 6, 7 and 8.

The *half-closed range operator*, on the other hand uses the following syntax:

```
x..<y
```

In this instance, the operator encompasses all the numbers from *x* up to, but not including, *y*. A half closed range operator `5..<8`, therefore, specifies the numbers 5, 6 and 7.

7.9 The Ternary Operator

Swift supports the *ternary operator* to provide a shortcut way of making decisions within code. The syntax of the ternary operator (also known as the conditional operator) is as follows:

```
condition ? true expression : false expression
```

The way the ternary operator works is that *condition* is replaced with an expression that will return either *true* or *false*. If the result is true then the expression that replaces the *true expression* is evaluated. Conversely, if the result was *false* then the *false expression* is evaluated. Let's see this in action:

```
let x = 10
let y = 20

print("Largest number is \(x > y ? x : y)")
```

The above code example will evaluate whether *x* is greater than *y*. Clearly this will evaluate to false resulting in *y* being returned to the print call for display to the user:

```
Largest number is 20
```

7.10 Bitwise Operators

As previously discussed, computer processors work in binary. These are essentially streams of ones and zeros, each one referred to as a bit. Bits are formed into groups of 8 to form bytes. As such, it is not surprising that we, as programmers, will occasionally end up working at this level in our code. To facilitate this requirement, Swift provides a range of *bit operators*.

Those familiar with bitwise operators in other languages such as C, C++, C#, Objective-C and Java will find nothing new in this area of the Swift language syntax. For those unfamiliar with binary numbers, now may be a good time to seek out reference materials on the subject in order to understand how ones and zeros are formed into bytes to form numbers. Other authors have done a much better job of describing the subject than we can do within the scope of this book.

For the purposes of this exercise we will be working with the binary representation of two numbers. Firstly, the decimal number 171 is represented in binary as:

```
10101011
```

Secondly, the number 3 is represented by the following binary sequence:

```
00000011
```

Now that we have two binary numbers with which to work, we can begin to look at the Swift bitwise operators:

7.10.1 Bitwise NOT

The Bitwise NOT is represented by the tilde character and has the effect of inverting all of the bits in a number. In other words, all the zeros become ones and all the ones become zeros. Taking our example 3 number, a Bitwise NOT operation has the following result:

```
00000011 NOT
=====
11111100
```

The following Swift code, therefore, results in a value of -4:

```
let y = 3
let z = ~y

print("Result is \(z)")
```

7.10.2 Bitwise AND

The Bitwise AND is represented by a single ampersand (&). It makes a bit by bit comparison of two numbers. Any corresponding position in the binary sequence of each number where both bits are 1 results in a 1 appearing in the same position of the resulting number. If either bit position contains a 0 then a zero appears in the result. Taking our two example numbers, this would appear as follows:

```
10101011 AND
00000011
=====
00000011
```

As we can see, the only locations where both numbers have 1s are the last two positions. If we perform this in Swift code, therefore, we should find that the result is 3 (00000011):

```
let x = 171
let y = 3
let z = x & y

print("Result is \(z)")
```

7.10.3 Bitwise OR

The bitwise OR also performs a bit by bit comparison of two binary sequences. Unlike the AND operation, the OR places a 1 in the result if there is a 1 in the first or second operand. The operator is represented by a single vertical bar character (|). Using our example numbers, the result will be as follows:

```
10101011 OR
00000011
=====
10101011
```

If we perform this operation in a Swift example the result will be 171:

```
let x = 171
let y = 3
let z = x | y

print("Result is \(z)")
```

7.10.4 Bitwise XOR

The bitwise XOR (commonly referred to as *exclusive OR* and represented by the caret '^' character) performs a similar task to the OR operation except that a 1 is placed in the result if one or other corresponding bit positions in the two numbers is 1. If both positions are a 1 or a 0 then the corresponding bit in the result is set to a 0. For example:

```
10101011 XOR
00000011
=====
10101000
```

The result in this case is 10101000 which converts to 168 in decimal. To verify this we can, once again, try some Swift code:

```
let x = 171
let y = 3
let z = x ^ y
```

```
print("Result is \(z)")
```

When executed, we get the following output from print:

```
Result is 168
```

7.10.5 Bitwise Left Shift

The bitwise left shift moves each bit in a binary number a specified number of positions to the left. Shifting an integer one position to the left has the effect of doubling the value.

As the bits are shifted to the left, zeros are placed in the vacated right most (low order) positions. Note also that once the left most (high order) bits are shifted beyond the size of the variable containing the value, those high order bits are discarded:

```
10101011 Left Shift one bit
=====
101010110
```

In Swift the bitwise left shift operator is represented by the '<<' sequence, followed by the number of bit positions to be shifted. For example, to shift left by 1 bit:

```
let x = 171
let z = x << 1

print("Result is \(z)")
```

When compiled and executed, the above code will display a message stating that the result is 342 which, when converted to binary, equates to 101010110.

7.10.6 Bitwise Right Shift

A bitwise right shift is, as you might expect, the same as a left except that the shift takes place in the opposite direction. Shifting an integer one position to the right has the effect of halving the value.

Note that since we are shifting to the right there is no opportunity to retain the lower most bits regardless of the data type used to contain the result. As a result the low order bits are discarded. Whether or not the vacated high order bit positions are replaced with zeros or ones depends on whether the *sign bit* used to indicate positive and negative numbers is set or not.

```
10101011 Right Shift one bit
=====
01010101
```

The bitwise right shift is represented by the '>>' character sequence followed by the shift count:

```
let x = 171
let z = x >> 1

print("Result is \(z)")
```

When executed, the above code will report the result of the shift as being 85, which equates to binary 01010101.

7.11 Compound Bitwise Operators

As with the arithmetic operators, each bitwise operator has a corresponding compound operator that allows the operation and assignment to be performed using a single operator:

Operator	Description
x &= y	Perform a bitwise AND of x and y and assign result to x
x = y	Perform a bitwise OR of x and y and assign result to x
x ^= y	Perform a bitwise XOR of x and y and assign result to x
x <<= n	Shift x left by n places and assign result to x
x >>= n	Shift x right by n places and assign result to x

7.12 Summary

Operators and expressions provide the underlying mechanism by which variables and constants are manipulated and evaluated within Swift code. This can take the simplest of forms whereby two numbers are added using the addition operator in an expression and the result stored in a variable using the assignment operator. Operators fall into a range of categories, details of which have been covered in this chapter.

8. Swift Flow Control

Regardless of the programming language used, application development is largely an exercise in applying logic, and much of the art of programming involves writing code that makes decisions based on one or more criteria. Such decisions define which code gets executed, how many times it is executed and, conversely, which code gets by-passed when the program is executing. This is often referred to as *flow control* since it controls the *flow* of program execution. Flow control typically falls into the categories of *looping control* (how often code is executed) and *conditional flow control* (whether or not code is executed). This chapter is intended to provide an introductory overview of both types of flow control in Swift.

8.1 Looping Flow Control

This chapter will begin by looking at flow control in the form of loops. Loops are essentially sequences of Swift statements which are to be executed repeatedly until a specified condition is met. The first looping statement we will explore is the *for* loop.

8.2 The Swift *for* Statement

Swift provides two types of *for* loop, namely the *condition-increment* *for* loop and the *for-in* loop:

8.2.1 The Condition-Increment for Loop

The condition-increment for loop uses the following syntax:

```
for initializer; conditional expression; increment expression {
    // statements to be executed
}
```

The *initializer* typically initializes a variable to act as the counter for the loop. Traditionally the variable name *i* is used for this purpose, though any valid variable name will do. For example:

```
var i = 0
var nameCount = 0
```

The *conditional expression* specifies the test that is to be executed on each loop iteration to verify whether the loop has been performed the required number of times. For example, if a loop is to be performed 10 times:

```
i < 10
```

Finally the *increment expression* specifies the action to be performed on the counter variable. For example to increment by 1:

```
++i
```

The body of statements to be executed on each iteration of the loop is contained within the code block defined by the opening ({} and closing {}) braces.

Bringing this all together we can create a *for* loop to perform a task a specified number of times:

```
for var i = 0; i < 10; ++i {
    print("Value of i is \(i)")
}
```

8.2.2 The *for-in* Loop

The *for-in* loop is used to iterate over a sequence of items contained in a collection or number range and provides a simpler alternative to the condition-increment looping technique previously described.

The syntax of the *for-in* loop is as follows:

```
for constant name in collection or range {
    // code to be executed
}
```

Swift Flow Control

In this syntax, *constant name* is the name to be used for a constant that will contain the current item from the collection or range through which the loop is iterating. The code in the body of the loop will typically use this constant name as a reference to the current item in the loop cycle. The *collection or range* references the item through which the loop is iterating. This could, for example, be an array of string values, a range operator or even a string of characters (the topic of collections will be covered in greater detail within the chapter entitled *Working with Array and Dictionary Collections in Swift*).

Consider, for example, the following for-in loop construct:

```
for index in 1...5 {
    print("Value of index is \(index)")
}
```

The loop begins by stating that the current item is to be assigned to a constant named *index*. The statement then declares a closed range operator to indicate that the for loop is to iterate through a range of numbers, starting at 1 and ending at 5. The body of the loop simply prints out a message to the console panel indicating the current value assigned to the *index* constant, resulting in the following output:

```
Value of index is 1
Value of index is 2
Value of index is 3
Value of index is 4
Value of index is 5
```

As will be demonstrated in the *Working with Array and Dictionary Collections in Swift* chapter of this book, the *for-in* loop is of particular benefit when working with collections such as arrays and dictionaries.

The declaration of a constant name in which to store a reference to the current item is not mandatory. In the event that a reference to the current item is not required in the body of the *for* loop, the constant name in the *for* loop declaration can be replaced by an underscore character. For example:

```
var count = 0

for _ in 1...5 {
    // No reference to the currentvalue is required.
    ++count
}
```

8.2.3 The while Loop

The Swift *for* loop described previously works well when it is known in advance how many times a particular task needs to be repeated in a program. There will, however, be instances where code needs to be repeated until a certain condition is met, with no way of knowing in advance how many repetitions are going to be needed to meet that criteria. To address this need, Swift provides the *while* loop.

Essentially, the *while* loop repeats a set of tasks while a specified condition is met. The *while* loop syntax is defined as follows:

```
while condition {
    // Swift statements go here
}
```

In the above syntax, *condition* is an expression that will return either *true* or *false* and the *// Swift statements go here* comment represents the code to be executed while the *condition* expression is *true*. For example:

```
var myCount = 0

while myCount < 100 {
    ++myCount
}
```

In the above example, the *while* expression will evaluate whether the *myCount* variable is less than 100. If it is already greater than 100, the code in the braces is skipped and the loop exits without performing any tasks.

If, on the other hand, *myCount* is not greater than 100 the code in the braces is executed and the loop returns to the *while* statement and repeats the evaluation of *myCount*. This process repeats until the value of *myCount* is greater than 100, at which point the loop exits.

8.3 The *repeat ... while* loop

The *repeat ... while* loop replaces the Swift 1.x *do .. while* loop. It is often helpful to think of the *repeat ... while* loop as an inverted *while* loop. The *while* loop evaluates an expression before executing the code contained in the body of the loop. If the expression evaluates to *false* on the first check then the code is not executed. The *repeat ... while* loop, on the other hand, is provided for situations where you know that the code contained in the body of the loop will *always* need to be executed at least once. For example, you may want to keep stepping through the items in an array until a specific item is found. You know that you have to at least check the first item in the array to have any hope of finding the entry you need. The syntax for the *repeat ... while* loop is as follows:

```
repeat {
    // Swift statements here
} while conditional expression
```

In the *repeat ... while* example below the loop will continue until the value of a variable named *i* equals 0:

```
var i = 10

repeat {
    --i
} while (i > 0)
```

8.4 Breaking from Loops

Having created a loop, it is possible that under certain conditions you might want to break out of the loop before the completion criteria have been met (particularly if you have created an infinite loop). One such example might involve continually checking for activity on a network socket. Once activity has been detected it will most likely be necessary to break out of the monitoring loop and perform some other task.

For the purpose of breaking out of a loop, Swift provides the *break* statement which breaks out of the current loop and resumes execution at the code directly after the loop. For example:

```
var j = 10

for var i = 0; i < 100; ++i
{
    j += j

    if j > 100 {
        break
    }

    print("j = \(j)")
}
```

In the above example the loop will continue to execute until the value of *j* exceeds 100 at which point the loop will exit and execution will continue with the next line of code after the loop.

8.5 The *continue* Statement

The *continue* statement causes all remaining code statements in a loop to be skipped, and execution to be returned to the top of the loop. In the following example, the print function is only called when the value of variable *i* is an even number:

```
var i = 1

while i < 20
{
    ++i

    if (i % 2) != 0 {
        continue
    }
}
```

Swift Flow Control

```
        print("i = \(i)")
    }
```

The *continue* statement in the above example will cause the print call to be skipped unless the value of *i* can be divided by 2 with no remainder. If the *continue* statement is triggered, execution will skip to the top of the while loop and the statements in the body of the loop will be repeated (until the value of *i* exceeds 19).

8.6 Conditional Flow Control

In the previous chapter we looked at how to use logical expressions in Swift to determine whether something is *true* or *false*. Since programming is largely an exercise in applying logic, much of the art of programming involves writing code that makes decisions based on one or more criteria. Such decisions define which code gets executed and, conversely, which code gets by-passed when the program is executing. This is often referred to as *flow control* since it controls the *flow* of program execution.

8.7 Using the if Statement

The *if* statement is perhaps the most basic of flow control options available to the Swift programmer. Programmers who are familiar with C, Objective-C, C++ or Java will immediately be comfortable using Swift *if* statements.

The basic syntax of the Swift *if* statement is as follows:

```
if boolean expression {
    // Swift code to be performed when expression evaluates to true
}
```

Unlike some other programming languages, it is important to note that the braces ({}) are mandatory in Swift, even if only one line of code is executed after the *if* expression.

Essentially if the *Boolean expression* evaluates to *true* then the code in the body of the statement is executed. The body of the statement is enclosed in braces ({}). If, on the other hand, the expression evaluates to *false* the code in the body of the statement is skipped.

For example, if a decision needs to be made depending on whether one value is greater than another, we would write code similar to the following:

```
let x = 10

if x > 9 {
    print("x is greater than 9!")
}
```

Clearly, *x* is indeed greater than 9 causing the message to appear in the console panel.

8.8 Using if ... else ... Statements

The next variation of the *if* statement allows us to also specify some code to perform if the expression in the *if* statement evaluates to *false*. The syntax for this construct is as follows:

```
if boolean expression {
    // Code to be executed if expression is true
} else {
    // Code to be executed if expression is false
}
```

Using the above syntax, we can now extend our previous example to display a different message if the comparison expression evaluates to be *false*:

```
let x = 10

if x > 9 {
    print("x is greater than 9!")
} else {
    print("x is less than 9!")
}
```

In this case, the second print statement would execute if the value of *x* was less than 9.

8.9 Using if ... else if ... Statements

So far we have looked at *if* statements which make decisions based on the result of a single logical expression. Sometimes it becomes necessary to make decisions based on a number of different criteria. For this purpose, we can use the *if ... else if ...* construct, an example of which is as follows:

```
let x = 9;

if x == 10 {
    print("x is 10")
} else if x == 9 {
    print("x is 9")
} else if x == 8 {
    print("x is 8")
}
```

This approach works well for a moderate number of comparisons, but can become cumbersome for a larger volume of expression evaluations. For such situations, the Swift *switch* statement provides a more flexible and efficient solution. For more details on using the *switch* statement refer to the next chapter entitled *The Swift Switch Statement*.

8.10 The guard Statement

The guard statement is a Swift language feature introduced as part of Swift 2. A guard statement contains a Boolean expression which must evaluate to true in order for the code located *after* the guard statement to be executed. The guard statement must include an *else* clause to be executed in the event that the expression evaluates to false. The code in the else clause must contain a statement to exit the current code flow (i.e. a *return*, *break*, *continue* or *throw* statement). Alternatively the else block may call any other function or method that does not itself return.

The syntax for the guard statement is as follows:

```
guard <boolean expression> else {
    // code to be executed if expression is false
    <exit statement here>
}

// code here is executed if expression is true
```

The guard statement essentially provides an “early exit” strategy from the current function or loop in the event that a specified requirement is not met.

The following code example implements a guard statement within a function:

```
func multiplyByTen(value: Int?) {

    guard let number = value where value < 10 else {
        print("Number is too high")
        return
    }

    let result = number * 10
    print(result)
}
```

The function takes as a parameter an integer value in the form of an optional. The guard statement uses optional binding to unwrap the value and verify that it is less than 10. In the event that the variable could not be unwrapped, or that its value is greater than 9, the else clause is triggered, the error message printed and the return statement executed to exit the function.

In the event that the optional contains a value less than 10, the code after the guard statement executes to multiply the value by 10 and print the result. A particularly important point to note about the above example is that the unwrapped *number* variable is available to the code outside of the guard statement. This would not have been the case had the variable been unwrapped using an *if* statement.

8.11 Summary

The term *flow control* is used to describe the logic that dictates the execution path that is taken through the source code of an application as it runs. This chapter has looked at the two types of flow control provided by Swift (looping and conditional) and explored the various Swift constructs that are available to implement both forms of flow control logic.

9. The Swift Switch Statement

In *Swift Flow Control* we looked at how to control program execution flow using the *if* and *else* statements. Whilst these statement constructs work well for testing a limited number of conditions, they quickly become unwieldy when dealing with larger numbers of possible conditions. To simplify such situations, Swift has inherited the *switch* statement from the C programming language. Those familiar with the switch statement from other programming languages should be aware, however, that the Swift switch statement has some key differences from other implementations. In this chapter we will explore the Swift implementation of the *switch* statement in detail.

9.1 Why Use a switch Statement?

For a small number of logical evaluations of a value the *if ... else if ...* construct is perfectly adequate. Unfortunately, any more than two or three possible scenarios can quickly make such a construct both time consuming to write and difficult to read. For such situations, the *switch* statement provides an excellent alternative.

9.2 Using the switch Statement Syntax

The syntax for a basic Swift *switch* statement implementation can be outlined as follows:

```
switch expression
{
    case match1:
        statements

    case match2:
        statements

    case match3, match4:
        statements

    default:
        statements
}
```

In the above syntax outline, *expression* represents either a value, or an expression which returns a value. This is the value against which the *switch* operates.

For each possible match a *case* statement is provided, followed by a *match* value. Each potential match must be of the same type as the governing expression. Following on from the *case* line are the Swift statements that are to be executed in the event of the value matching the case condition.

Finally, the *default* section of the construct defines what should happen if none of the case statements present a match to the *expression*.

9.3 A Swift switch Statement Example

With the above information in mind we may now construct a simple *switch* statement:

```
let value = 4

switch (value)
{
    case 0:
        print("zero")

    case 1:
```

The Swift Switch Statement

```
    print("one")

    case 2:
        print("two")

    case 3:
        print("three")

    case 4:
        print("four")

    case 5:
        print("five")

    default:
        print("Integer out of range")
}
```

9.4 Combining case Statements

In the above example, each case had its own set of statements to execute. Sometimes a number of different matches may require the same code to be executed. In this case, it is possible to group case matches together with a common set of statements to be executed when a match for any of the cases is found. For example, we can modify the switch construct in our example so that the same code is executed regardless of whether the value is 0, 1 or 2:

```
let value = 1

switch (value)
{
    case 0, 1, 2:
        print("zero, one or two")

    case 3:
        print("three")

    case 4:
        print("four")

    case 5:
        print("five")

    default:
        print("Integer out of range")
}
```

9.5 Range Matching in a switch Statement

The case statements within a switch construct may also be used to implement range matching. The following switch statement, for example, checks a temperature value for matches within three number ranges:

```
let temperature = 83

switch (temperature)
{
    case 0...49:
        print("Cold")

    case 50...79:
```