

# **iOS 11 App Development**

**11**

**Essentials**

---

# **iOS 11 App Development Essentials**

---

iOS 11 App Development Essentials – First Edition

© 2018 Neil Smyth / Payload Media, Inc. 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

<b>1. Start Here</b> .....	<b>1</b>
1.1 For New iOS Developers .....	1
1.2 For iOS 10 Developers.....	1
1.3 Source Code Download.....	2
1.4 Feedback.....	2
1.5 Errata .....	2
<b>2. Joining the Apple Developer Program</b> .....	<b>3</b>
2.1 Downloading Xcode 9 and the iOS 11 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 .....	4
2.5 Summary .....	5
<b>3. Installing Xcode 9 and the iOS 11 SDK</b> .....	<b>7</b>
3.1 Identifying if you have an Intel or PowerPC based Mac .....	7
3.2 Installing Xcode 9 and the iOS 11 SDK .....	7
3.3 Starting Xcode.....	8
3.4 Adding Your Apple ID to the Xcode Preferences .....	8
3.5 Developer and Distribution Signing Identities .....	9
<b>4. A Guided Tour of Xcode 9</b> .....	<b>11</b>
4.1 Starting Xcode 9.....	11
4.2 Creating the iOS App User Interface.....	15
4.3 Changing Component Properties.....	17
4.4 Adding Objects to the User Interface .....	18
4.5 Building and Running an iOS 11 App in Xcode 9 .....	21
4.6 Running the App on a Physical iOS Device.....	22
4.7 Managing Devices and Simulators .....	22
4.8 Enabling Network Testing .....	23
4.9 Dealing with Build Errors .....	24
4.10 Monitoring Application Performance .....	24
4.11 An Exploded View of the User Interface Layout Hierarchy.....	25
4.12 Summary.....	26
<b>5. An Introduction to Xcode 9 Playgrounds</b> .....	<b>27</b>
5.1 What is a Playground? .....	27
5.2 Creating a New Playground .....	27
5.3 A Basic Swift Playground Example .....	29
5.4 Viewing Results.....	29
5.5 Adding Rich Text Comments.....	31
5.6 Working with Playground Pages .....	32
5.7 Working with UIKit in Playgrounds .....	32
5.8 Adding Resources to a Playground .....	33
5.9 Working with Enhanced Live Views .....	34
5.10 When to Use Playgrounds .....	37
5.11 Summary.....	37

<b>6. Swift Data Types, Constants and Variables .....</b>	<b>39</b>
6.1 Using a Swift Playground .....	39
6.2 Swift Data Types .....	39
6.2.1 Integer Data Types .....	40
6.2.2 Floating Point Data Types .....	40
6.2.3 Bool Data Type .....	41
6.2.4 Character Data Type.....	41
6.2.5 String Data Type.....	41
6.2.6 Special Characters/Escape Sequences.....	42
6.3 Swift Variables .....	43
6.4 Swift Constants .....	43
6.5 Declaring Constants and Variables .....	43
6.6 Type Annotations and Type Inference.....	43
6.7 The Swift Tuple .....	44
6.8 The Swift Optional Type .....	45
6.9 Type Casting and Type Checking .....	48
6.10 Summary.....	50
<b>7. Swift Operators and Expressions .....</b>	<b>51</b>
7.1 Expression Syntax in Swift .....	51
7.2 The Basic Assignment Operator.....	51
7.3 Swift Arithmetic Operators.....	51
7.4 Compound Assignment Operators .....	52
7.5 Comparison Operators .....	52
7.6 Boolean Logical Operators.....	53
7.7 Range Operators.....	54
7.8 The Ternary Operator .....	54
7.9 Bitwise Operators .....	55
7.9.1 Bitwise NOT.....	55
7.9.2 Bitwise AND.....	55
7.9.3 Bitwise OR .....	56
7.9.4 Bitwise XOR .....	56
7.9.5 Bitwise Left Shift.....	57
7.9.6 Bitwise Right Shift .....	57
7.10 Compound Bitwise Operators .....	57
7.11 Summary.....	58
<b>8. Swift Flow Control .....</b>	<b>59</b>
8.1 Looping Flow Control.....	59
8.2 The Swift for-in Statement .....	59
8.2.1 The while Loop.....	60
8.3 The repeat ... while loop .....	60
8.4 Breaking from Loops.....	61
8.5 The continue Statement .....	61
8.6 Conditional Flow Control .....	62
8.7 Using the if Statement.....	62
8.8 Using if ... else ... Statements.....	63
8.9 Using if ... else if ... Statements.....	63

8.10 The guard Statement .....	63
8.11 Summary.....	64
<b>9. The Swift Switch Statement .....</b>	<b>65</b>
9.1 Why Use a switch Statement? .....	65
9.2 Using the switch Statement Syntax .....	65
9.3 A Swift switch Statement Example .....	66
9.4 Combining case Statements .....	66
9.5 Range Matching in a switch Statement .....	67
9.6 Using the where statement .....	67
9.7 Fallthrough.....	68
9.8 Summary.....	69
<b>10. An Overview of Swift 4 Functions, Methods and Closures .....</b>	<b>71</b>
10.1 What is a Function? .....	71
10.2 What is a Method? .....	71
10.3 How to Declare a Swift Function .....	71
10.4 Calling a Swift Function.....	72
10.5 Handling Return Values .....	72
10.6 Local and External Parameter Names.....	72
10.7 Declaring Default Function Parameters.....	73
10.8 Returning Multiple Results from a Function.....	74
10.9 Variable Numbers of Function Parameters .....	74
10.10 Parameters as Variables .....	75
10.11 Working with In-Out Parameters.....	75
10.12 Functions as Parameters.....	76
10.13 Closure Expressions .....	78
10.14 Closures in Swift.....	79
10.15 Summary.....	80
<b>11. The Basics of Object-Oriented Programming in Swift .....</b>	<b>81</b>
11.1 What is an Object?.....	81
11.2 What is a Class? .....	81
11.3 Declaring a Swift Class .....	81
11.4 Adding Instance Properties to a Class.....	82
11.5 Defining Methods .....	82
11.6 Declaring and Initializing a Class Instance.....	83
11.7 Initializing and Deinitializing a Class Instance .....	83
11.8 Calling Methods and Accessing Properties .....	84
11.9 Stored and Computed Properties .....	85
11.10 Using self in Swift.....	86
11.11 Summary.....	87
<b>12. An Introduction to Swift Subclassing and Extensions .....</b>	<b>89</b>
12.1 Inheritance, Classes and Subclasses .....	89
12.2 A Swift Inheritance Example .....	89
12.3 Extending the Functionality of a Subclass.....	90
12.4 Overriding Inherited Methods .....	91
12.5 Initializing the Subclass .....	91

12.6 Using the SavingsAccount Class.....	92
12.7 Swift Class Extensions.....	92
12.8 Summary.....	93
<b>13. Working with Array and Dictionary Collections in Swift.....</b>	<b>95</b>
13.1 Mutable and Immutable Collections .....	95
13.2 Swift Array Initialization.....	95
13.3 Working with Arrays in Swift .....	96
13.3.1 Array Item Count .....	96
13.3.2 Accessing Array Items .....	96
13.4 Appending Items to an Array.....	97
13.4.1 Inserting and Deleting Array Items.....	97
13.4.2 Array Iteration .....	97
13.5 Creating Mixed Type Arrays.....	97
13.6 Swift Dictionary Collections .....	98
13.7 Swift Dictionary Initialization.....	98
13.8 Sequence-based Dictionary Initialization .....	99
13.9 Dictionary Item Count.....	100
13.10 Accessing and Updating Dictionary Items .....	100
13.11 Adding and Removing Dictionary Entries .....	100
13.12 Dictionary Iteration.....	100
13.13 Summary.....	101
<b>14. Understanding Error Handling in Swift 4.....</b>	<b>103</b>
14.1 Understanding Error Handling.....	103
14.2 Declaring Error Types.....	103
14.3 Throwing an Error.....	104
14.4 Calling Throwing Methods and Functions .....	105
14.5 Accessing the Error Object.....	105
14.6 Disabling Error Catching .....	106
14.7 Using the defer Statement .....	106
14.8 Summary.....	107
<b>15. The iOS 11-Application and Development Architecture .....</b>	<b>109</b>
15.1 An Overview of the iOS 11 Operating System Architecture .....	109
15.2 Model View Controller (MVC) .....	110
15.3 The Target-Action pattern, IBOutlets and IBActions .....	110
15.4 Subclassing.....	111
15.5 Delegation.....	111
15.6 Summary.....	111
<b>16. Creating an Interactive iOS 11 App .....</b>	<b>113</b>
16.1 Creating the New Project.....	113
16.2 Creating the User Interface .....	113
16.3 Building and Running the Sample Application.....	116
16.4 Adding Actions and Outlets .....	116
16.5 Building and Running the Finished Application .....	120
16.6 Hiding the Keyboard .....	120
16.7 Summary.....	121



<b>17. Understanding iOS 11 Views, Windows and the View Hierarchy .....</b>	<b>123</b>
17.1 An Overview of Views and the UIKit Class Hierarchy.....	123
17.2 The UIWindow Class .....	123
17.3 The View Hierarchy .....	123
17.4 Viewing Hierarchy Ancestors in Interface Builder .....	125
17.5 View Types .....	125
17.5.1 The Window.....	126
17.5.2 Container Views.....	126
17.5.3 Controls .....	126
17.5.4 Display Views.....	126
17.5.5 Text and Web Views .....	126
17.5.6 Navigation Views and Tab Bars.....	126
17.5.7 Alert Views.....	126
17.6 Summary.....	126
<b>18. An Introduction to Auto Layout in iOS 11 .....</b>	<b>127</b>
18.1 An Overview of Auto Layout .....	127
18.2 Alignment Rects .....	128
18.3 Intrinsic Content Size .....	128
18.4 Content Hugging and Compression Resistance Priorities .....	128
18.5 Safe Area Layout Guide.....	129
18.6 Three Ways to Create Constraints .....	129
18.7 Constraints in more Detail .....	130
18.8 Summary.....	130
<b>19. Working with iOS 11 Auto Layout Constraints in Interface Builder .....</b>	<b>131</b>
19.1 A Simple Example of Auto Layout in Action.....	131
19.2 Working with Constraints .....	131
19.3 The Auto Layout Features of Interface Builder .....	135
19.3.1 Suggested Constraints .....	135
19.3.2 Visual Cues.....	136
19.3.3 Highlighting Constraint Problems.....	137
19.3.4 Viewing, Editing and Deleting Constraints .....	139
19.4 Creating New Constraints in Interface Builder .....	141
19.5 Adding Aspect Ratio Constraints .....	142
19.6 Resolving Auto Layout Problems .....	142
19.7 Summary.....	143
<b>20. An iOS 11 Auto Layout Example .....</b>	<b>145</b>
20.1 Preparing the Project.....	145
20.2 Designing the User Interface .....	145
20.3 Adding Auto Layout Constraints .....	146
20.4 Adjusting Constraint Priorities .....	148
20.5 Testing the Application .....	149
20.6 Summary.....	149
<b>21. Implementing iOS 11 Auto Layout Constraints in Code .....</b>	<b>151</b>
21.1 Creating Constraints in Code .....	151
21.2 Adding a Constraint to a View .....	152

21.3 Turning off Auto Resizing Translation.....	153
21.4 An Example Application .....	153
21.5 Creating the Views.....	153
21.6 Creating and Adding the Constraints.....	154
21.7 Removing Constraints.....	156
21.8 Summary.....	156
<b>22. Implementing Cross-Hierarchy Auto Layout Constraints in iOS 11 .....</b>	<b>157</b>
22.1 The Example Application .....	157
22.2 Establishing Outlets .....	158
22.3 Writing the Code to Remove the Old Constraint.....	159
22.4 Adding the Cross Hierarchy Constraint.....	159
22.5 Testing the Application .....	160
22.6 Summary.....	160
<b>23. Understanding the iOS 11 Auto Layout Visual Format Language.....</b>	<b>161</b>
23.1 Introducing the Visual Format Language.....	161
23.2 Visual Format Language Examples .....	161
23.3 Using the constraints(withVisualFormat:) Method .....	162
23.4 Summary.....	163
<b>24. Using Trait Variations to Design Adaptive iOS 11 User Interfaces .....</b>	<b>165</b>
24.1 Understanding Traits and Size Classes.....	165
24.2 Size Classes in Interface Builder.....	165
24.3 Setting “Any” Defaults.....	166
24.4 Working with Trait Variations in Interface Builder .....	166
24.5 Attributes Inspector Trait Variations .....	167
24.6 Using Vary for Traits Layout Variations .....	168
24.7 An Adaptive User Interface Tutorial .....	169
24.8 Designing the Initial Layout .....	169
24.9 Adding Universal Image Assets.....	170
24.10 Increasing Font Size for iPad Devices.....	171
24.11 Using Vary for Traits .....	172
24.12 Testing the Adaptivity.....	172
24.13 Testing the Application.....	173
24.14 Summary.....	173
<b>25. Using Storyboards in Xcode 9 .....</b>	<b>175</b>
25.1 Creating the Storyboard Example Project .....	175
25.2 Accessing the Storyboard .....	175
25.3 Adding Scenes to the Storyboard .....	177
25.4 Configuring Storyboard Segues .....	178
25.5 Configuring Storyboard Transitions .....	178
25.6 Associating a View Controller with a Scene.....	179
25.7 Passing Data Between Scenes .....	180
25.8 Unwinding Storyboard Segues.....	181
25.9 Triggering a Storyboard Segue Programmatically .....	181
25.10 Summary.....	182
<b>26. Organizing Scenes over Multiple Storyboard Files .....</b>	<b>183</b>

26.1 Organizing Scenes into Multiple Storyboards.....	183
26.2 Establishing a Connection between Different Storyboards.....	185
26.3 Summary.....	185
<b>27. Using Xcode 9 Storyboards to Create an iOS 11 Tab Bar Application .....</b>	<b>187</b>
27.1 An Overview of the Tab Bar .....	187
27.2 Understanding View Controllers in a Multiview Application.....	187
27.3 Setting up the Tab Bar Example Application.....	188
27.4 Reviewing the Project Files .....	188
27.5 Adding the View Controllers for the Content Views.....	188
27.6 Adding the Tab Bar Controller to the Storyboard.....	188
27.7 Designing the View Controller User interfaces.....	190
27.8 Configuring the Tab Bar Items .....	191
27.9 Building and Running the Application.....	192
27.10 Summary.....	192
<b>28. An Overview of iOS 11 Table Views and Xcode 9 Storyboards .....</b>	<b>193</b>
28.1 An Overview of the Table View.....	193
28.2 Static vs. Dynamic Table Views .....	193
28.3 The Table View Delegate and dataSource .....	193
28.4 Table View Styles .....	194
28.5 Self-Sizing Table Cells.....	195
28.6 Dynamic Type .....	195
28.7 Table View Cell Styles .....	196
28.8 Table View Cell Reuse .....	197
28.9 Table View Swipe Actions .....	198
28.10 Summary.....	199
<b>29. Using Xcode 9 Storyboards to Build Dynamic TableViews .....</b>	<b>201</b>
29.1 Creating the Example Project .....	201
29.2 Adding the TableView Controller to the Storyboard .....	201
29.3 Creating the UITableViewController and UITableViewCell Subclasses.....	202
29.4 Declaring the Cell Reuse Identifier.....	203
29.5 Designing a Storyboard UITableView Prototype Cell.....	204
29.6 Modifying the AttractionTableViewCell Class.....	204
29.7 Creating the Table View Datasource .....	205
29.8 Downloading and Adding the Image Files.....	208
29.9 Compiling and Running the Application .....	208
29.10 Handling TableView Swipe Gestures .....	208
29.11 Summary.....	210
<b>30. Implementing iOS 11 TableView Navigation using Storyboards in Xcode 9 .....</b>	<b>211</b>
30.1 Understanding the Navigation Controller.....	211
30.2 Adding the New Scene to the Storyboard .....	211
30.3 Adding a Navigation Controller.....	212
30.4 Establishing the Storyboard Segue .....	213
30.5 Modifying the AttractionDetailViewController Class.....	214
30.6 Using prepare(for segue:) to Pass Data between Storyboard Scenes .....	215
30.7 Testing the Application .....	216

30.8 Customizing the Navigation Title Size.....	217
30.9 Summary.....	218
<b>31. Integrating Search using the iOS UISearchController .....</b>	<b>219</b>
31.1 Introducing the UISearchController Class.....	219
31.2 Adding a Search Controller to the TableViewController Project .....	220
31.3 Implementing the updateSearchResults Method.....	221
31.4 Reporting the Number of Table Rows .....	221
31.5 Modifying the cellForRowAt Method .....	222
31.6 Modifying the Trailing Swipe Delegate Method .....	222
31.7 Modifying the Detail Segue .....	223
31.8 Handling the Search Cancel Button .....	223
31.9 Testing the Search Controller .....	224
31.10 Summary.....	224
<b>32. Working with the iOS 11 Stack View Class .....</b>	<b>225</b>
32.1 Introducing the UIStackView Class .....	225
32.2 Understanding Subviews and Arranged Subviews .....	226
32.3 StackView Configuration Options .....	227
32.3.1 <i>axis</i> .....	227
32.3.2 <i>Distribution</i> .....	227
32.3.3 <i>spacing</i> .....	229
32.3.4 <i>alignment</i> .....	229
32.3.5 <i>baseLineRelativeArrangement</i> .....	232
32.3.6 <i>layoutMarginsRelativeArrangement</i> .....	232
32.4 Creating a Stack View in Code .....	232
32.5 Adding Subviews to an Existing Stack View .....	232
32.6 Hiding and Removing Subviews.....	233
32.7 Summary.....	233
<b>33. An iOS 11 Stack View Tutorial.....</b>	<b>235</b>
33.1 About the Stack View Example App.....	235
33.2 Creating the First Stack View .....	235
33.3 Creating the Banner Stack View .....	237
33.4 Adding the Switch Stack Views .....	238
33.5 Creating the Top Level Stack View .....	239
33.6 Adding the Button Stack View .....	240
33.7 Adding the Final Subviews to the Top Level Stack View .....	241
33.8 Dynamically Adding and Removing Subviews .....	242
33.9 Summary.....	243
<b>34. An iOS 11 Split View Master-Detail Example.....</b>	<b>245</b>
34.1 An Overview of Split View and Popovers.....	245
34.2 About the Example Split View Project .....	245
34.3 Creating the Project.....	246
34.4 Reviewing the Project.....	246
34.5 Configuring Master View Items .....	246
34.6 Configuring the Detail View Controller.....	248
34.7 Connecting Master Selections to the Detail View .....	249

34.8 Modifying the DetailViewController Class .....	250
34.9 Testing the Application .....	250
34.10 Summary .....	251
<b>35. A Guide to Multitasking in iOS 11 .....</b>	<b>253</b>
35.1 Using iPad Multitasking .....	253
35.2 Picture-In-Picture Multitasking .....	255
35.3 iPad Devices with Multitasking Support .....	255
35.4 Multitasking and Size Classes .....	255
35.5 Multitasking and the Master-Detail Split View .....	256
35.6 Handling Multitasking in Code .....	258
35.6.1 <i>willTransition(to newcollection: with coordinator:)</i> .....	258
35.6.2 <i>viewWillTransition(to size: with coordinator:)</i> .....	259
35.6.3 <i>traitCollectionDidChange(_:)</i> .....	259
35.7 Lifecycle Method Calls .....	259
35.8 Opting Out of Multitasking .....	260
35.9 Summary .....	260
<b>36. An iOS 11 Multitasking Example .....</b>	<b>261</b>
36.1 Creating the Multitasking Example Project .....	261
36.2 Adding the Image Files .....	261
36.3 Designing the Regular Width Size Class Layout .....	262
36.4 Designing the Compact Width Size Class .....	263
36.5 Testing the Project in a Multitasking Environment .....	265
36.6 Summary .....	266
<b>37. Working with Directories in Swift on iOS 11 .....</b>	<b>267</b>
37.1 The Application Documents Directory .....	267
37.2 The FileManager, FileHandle and Data Classes .....	267
37.3 Understanding Pathnames in Swift .....	268
37.4 Obtaining a Reference to the Default FileManager Object .....	268
37.5 Identifying the Current Working Directory .....	268
37.6 Identifying the Documents Directory .....	268
37.7 Identifying the Temporary Directory .....	269
37.8 Changing Directory .....	269
37.9 Creating a New Directory .....	270
37.10 Deleting a Directory .....	270
37.11 Listing the Contents of a Directory .....	271
37.12 Getting the Attributes of a File or Directory .....	271
37.13 Summary .....	272
<b>38. Working with Files in Swift on iOS 11 .....</b>	<b>273</b>
38.1 Obtaining a FileManager Instance Reference .....	273
38.2 Checking for the Existence of a File .....	273
38.3 Comparing the Contents of Two Files .....	273
38.4 Checking if a File is Readable/Writable/Executable/Deletable .....	274
38.5 Moving/Renaming a File .....	274
38.6 Copying a File .....	274
38.7 Removing a File .....	275

38.8 Creating a Symbolic Link .....	275
38.9 Reading and Writing Files with FileManager .....	275
38.10 Working with Files using the FileHandle Class .....	275
38.11 Creating a FileHandle Object .....	276
38.12 FileHandle File Offsets and Seeking .....	276
38.13 Reading Data from a File .....	277
38.14 Writing Data to a File .....	277
38.15 Truncating a File .....	278
38.16 Summary .....	278
<b>39. iOS 11 Directory Handling and File I/O in Swift – A Worked Example .....</b>	<b>279</b>
39.1 The Example Application .....	279
39.2 Setting up the Application Project .....	279
39.3 Designing the User Interface .....	279
39.4 Checking the Data File on Application Startup .....	280
39.5 Implementing the Action Method .....	281
39.6 Building and Running the Example .....	281
39.7 Summary .....	282
<b>40. Preparing an iOS 11 App to use iCloud Storage .....</b>	<b>283</b>
40.1 iCloud Data Storage Services .....	283
40.2 Preparing an Application to Use iCloud Storage .....	283
40.3 Enabling iCloud Support for an iOS 11 Application .....	284
40.4 Reviewing the iCloud Entitlements File .....	284
40.5 Accessing Multiple Ubiquity Containers .....	285
40.6 Ubiquity Container URLs .....	285
40.7 Summary .....	285
<b>41. Managing Files using the iOS 11 UIDocument Class .....</b>	<b>287</b>
41.1 An Overview of the UIDocument Class .....	287
41.2 Subclassing the UIDocument Class .....	287
41.3 Conflict Resolution and Document States .....	287
41.4 The UIDocument Example Application .....	288
41.5 Creating a UIDocument Subclass .....	288
41.6 Designing the User Interface .....	288
41.7 Implementing the Application Data Structure .....	289
41.8 Implementing the contents(forType:) Method .....	290
41.9 Implementing the load(fromContents:) Method .....	290
41.10 Loading the Document at App Launch .....	291
41.11 Saving Content to the Document .....	293
41.12 Testing the Application .....	294
41.13 Summary .....	294
<b>42. Using iCloud Storage in an iOS 11 Application .....</b>	<b>295</b>
42.1 iCloud Usage Guidelines .....	295
42.2 Preparing the iCloudStore Application for iCloud Access .....	295
42.3 Configuring the View Controller .....	296
42.4 Implementing the loadFile Method .....	296
42.5 Implementing the metadataQueryDidFinishGathering Method .....	298

42.6 Implementing the saveDocument Method .....	301
42.7 Enabling iCloud Document and Data Storage.....	301
42.8 Running the iCloud Application .....	302
42.9 Reviewing and Deleting iCloud Based Documents .....	302
42.10 Making a Local File Ubiquitous .....	303
42.11 Summary.....	303
<b>43. An Overview of the iOS Document Browser View Controller.....</b>	<b>305</b>
43.1 An Overview of the Document Browser View Controller .....	305
43.2 The Anatomy of a Document Based App .....	306
43.3 Document Browser Project Settings .....	307
43.4 The Document Browser Delegate Methods .....	307
43.4.1 <i>didRequestDocumentCreationWithHandler</i> .....	307
43.4.2 <i>didImportDocumentAt</i> .....	308
43.4.3 <i>didPickDocumentURLs</i> .....	308
43.4.4 <i>failedToImportDocumentAt</i> .....	309
43.5 Customizing the Document Browser .....	309
43.6 Adding Browser Actions.....	309
43.7 Summary.....	310
<b>44. An iOS Document Browser Tutorial .....</b>	<b>311</b>
44.1 Creating the DocumentBrowser Project.....	311
44.2 Declaring the Supported File Types .....	311
44.3 Completing the didRequestDocumentCreationWithHandler Method .....	313
44.4 Finishing the UIDocument Subclass.....	315
44.5 Modifying the Document View Controller.....	315
44.6 Testing the Document Browser App.....	317
44.7 Summary.....	317
<b>45. Synchronizing iOS 11 Key-Value Data using iCloud .....</b>	<b>319</b>
45.1 An Overview of iCloud Key-Value Data Storage .....	319
45.2 Sharing Data Between Applications.....	320
45.3 Data Storage Restrictions .....	320
45.4 Conflict Resolution.....	320
45.5 Receiving Notification of Key-Value Changes .....	320
45.6 An iCloud Key-Value Data Storage Example .....	320
45.7 Enabling the Application for iCloud Key Value Data Storage.....	320
45.8 Designing the User Interface .....	321
45.9 Implementing the View Controller .....	322
45.10 Modifying the viewDidLoad Method .....	322
45.11 Implementing the Notification Method.....	323
45.12 Implementing the saveData Method.....	323
45.13 Testing the Application .....	323
45.14 Summary.....	324
<b>46. iOS 11 Database Implementation using SQLite.....</b>	<b>325</b>
46.1 What is SQLite? .....	325
46.2 Structured Query Language (SQL).....	325
46.3 Trying SQLite on macOS.....	325

46.4 Preparing an iOS Application Project for SQLite Integration .....	327
46.5 SQLite, Swift and Wrappers .....	327
46.6 Key FMDB Classes .....	327
46.7 Creating and Opening a Database .....	328
46.8 Creating a Database Table .....	328
46.9 Extracting Data from a Database Table .....	328
46.10 Closing a SQLite Database .....	329
46.11 Summary .....	329
<b>47. An Example SQLite based iOS 11 Application using Swift and FMDB .....</b>	<b>331</b>
47.1 About the Example SQLite Application .....	331
47.2 Creating and Preparing the SQLite Application Project .....	331
47.3 Checking Out the FMDB Source Code .....	331
47.4 Designing the User Interface .....	333
47.5 Creating the Database and Table .....	334
47.6 Implementing the Code to Save Data to the SQLite Database .....	335
47.7 Implementing Code to Extract Data from the SQLite Database .....	336
47.8 Building and Running the Application .....	337
47.9 Summary .....	337
<b>48. Working with iOS 11 Databases using Core Data .....</b>	<b>339</b>
48.1 The Core Data Stack .....	339
48.2 Persistent Container .....	340
48.3 Managed Objects .....	340
48.4 Managed Object Context .....	340
48.5 Managed Object Model .....	340
48.6 Persistent Store Coordinator .....	341
48.7 Persistent Object Store .....	341
48.8 Defining an Entity Description .....	341
48.9 Initializing the Persistent Container .....	342
48.10 Obtaining the Managed Object Context .....	342
48.11 Getting an Entity Description .....	343
48.12 Setting the Attributes of a Managed Object .....	343
48.13 Saving a Managed Object .....	343
48.14 Fetching Managed Objects .....	343
48.15 Retrieving Managed Objects based on Criteria .....	344
48.16 Accessing the Data in a Retrieved Managed Object .....	344
48.17 Summary .....	344
<b>49. An iOS 11 Core Data Tutorial .....</b>	<b>345</b>
49.1 The Core Data Example Application .....	345
49.2 Creating a Core Data based Application .....	345
49.3 Creating the Entity Description .....	345
49.4 Designing the User Interface .....	346
49.5 Initializing the Persistent Container .....	347
49.6 Saving Data to the Persistent Store using Core Data .....	348
49.7 Retrieving Data from the Persistent Store using Core Data .....	348
49.8 Building and Running the Example Application .....	349
49.9 Summary .....	350



<b>50. An Introduction to CloudKit Data Storage on iOS 11</b>	<b>351</b>
50.1 An Overview of CloudKit	351
50.2 CloudKit Containers	351
50.3 CloudKit Public Database	351
50.4 CloudKit Private Databases	352
50.5 Data Storage and Transfer Quotas	352
50.6 CloudKit Records	352
50.7 CloudKit Record IDs	354
50.8 CloudKit References	354
50.9 CloudKit Assets	355
50.10 Record Zones	355
50.11 CloudKit Sharing	356
50.12 CloudKit Subscriptions	356
50.13 Obtaining iCloud User Information	356
50.14 CloudKit Dashboard	357
50.15 Summary	359
<b>51. An Introduction to CloudKit Sharing</b>	<b>361</b>
51.1 Understanding CloudKit Sharing	361
51.2 Preparing for CloudKit Sharing	361
51.3 The CKShare Class	361
51.4 The UICloudSharingController Class	362
51.5 Accepting a CloudKit Share	365
51.6 Fetching a Shared Record	365
51.7 Summary	366
<b>52. An iOS 11 CloudKit Example</b>	<b>367</b>
52.1 About the Example CloudKit Project	367
52.2 Creating the CloudKit Example Project	367
52.3 Designing the User Interface	368
52.4 Establishing Outlets and Actions	369
52.5 Accessing the Private Database	369
52.6 Hiding the Keyboard	371
52.7 Implementing the selectPhoto method	371
52.8 Saving a Record to the Cloud Database	372
52.9 Implementing the notifyUser Method	374
52.10 Testing the Record Saving Method	374
52.11 Searching for Cloud Database Records	377
52.12 Updating Cloud Database Records	378
52.13 Deleting a Cloud Record	379
52.14 Testing the Application	380
52.15 Summary	380
<b>53. An iOS 11 CloudKit Subscription Example</b>	<b>381</b>
53.1 Push Notifications and CloudKit Subscriptions	381
53.2 Configuring the Project for Remote Notifications	381
53.3 Registering an App to Receive Push Notifications	382
53.4 Configuring a CloudKit Subscription	383
53.5 Handling Remote Notifications	385

53.6 Implementing the <code>didReceiveRemoteNotification</code> Method .....	385
53.7 Fetching a Record From a Cloud Database .....	386
53.8 Completing the <code>didFinishLaunchingWithOptions</code> Method .....	387
53.9 Testing the Application .....	388
53.10 Summary .....	388
<b>54. An iOS 11 CloudKit Sharing Example .....</b>	<b>389</b>
54.1 Preparing the Project for CloudKit Sharing .....	389
54.2 Adding the Share Button .....	389
54.3 Creating the CloudKit Share .....	390
54.4 Accepting a CloudKit Share .....	391
54.5 Fetching the Shared Record .....	391
54.6 Testing the CloudKit Share Example .....	392
54.7 Summary .....	393
<b>55. An Overview of iOS 11 Multitouch, Taps and Gestures .....</b>	<b>395</b>
55.1 The Responder Chain .....	395
55.2 Forwarding an Event to the Next Responder .....	396
55.3 Gestures .....	396
55.4 Taps .....	396
55.5 Touches .....	396
55.6 Touch Notification Methods .....	396
55.6.1 <i>touchesBegan</i> method .....	396
55.6.2 <i>touchesMoved</i> method .....	397
55.6.3 <i>touchesEnded</i> method .....	397
55.6.4 <i>touchesCancelled</i> method .....	397
55.7 Touch Prediction .....	397
55.8 Touch Coalescing .....	397
55.9 3D Touch .....	398
55.10 Summary .....	398
<b>56. An Example iOS 11 Touch, Multitouch and Tap Application .....</b>	<b>399</b>
56.1 The Example iOS Tap and Touch Application .....	399
56.2 Creating the Example iOS Touch Project .....	399
56.3 Designing the User Interface .....	399
56.4 Enabling Multitouch on the View .....	400
56.5 Implementing the <code>touchesBegan</code> Method .....	400
56.6 Implementing the <code>touchesMoved</code> Method .....	401
56.7 Implementing the <code>touchesEnded</code> Method .....	401
56.8 Getting the Coordinates of a Touch .....	401
56.9 Building and Running the Touch Example Application .....	402
56.10 Checking for Touch Predictions .....	402
56.11 Accessing Coalesced Touches .....	403
56.12 Summary .....	403
<b>57. Detecting iOS 11 Touch Screen Gesture Motions .....</b>	<b>405</b>
57.1 The Example iOS 11 Gesture Application .....	405
57.2 Creating the Example Project .....	405
57.3 Designing the Application User Interface .....	405

57.4 Implementing the touchesBegan Method .....	406
57.5 Implementing the touchesMoved Method .....	406
57.6 Implementing the touchesEnded Method.....	407
57.7 Building and Running the Gesture Example .....	407
57.8 Summary.....	407
<b>58. Identifying Gestures using iOS 11 Gesture Recognizers .....</b>	<b>409</b>
58.1 The UIGestureRecognizer Class .....	409
58.2 Recognizer Action Messages.....	410
58.3 Discrete and Continuous Gestures .....	410
58.4 Obtaining Data from a Gesture.....	410
58.5 Recognizing Tap Gestures .....	410
58.6 Recognizing Pinch Gestures .....	410
58.7 Detecting Rotation Gestures.....	410
58.8 Recognizing Pan and Dragging Gestures.....	411
58.9 Recognizing Swipe Gestures .....	411
58.10 Recognizing Long Touch (Touch and Hold) Gestures .....	411
58.11 Summary.....	412
<b>59. An iOS 11 Gesture Recognition Tutorial.....</b>	<b>413</b>
59.1 Creating the Gesture Recognition Project .....	413
59.2 Designing the User Interface .....	413
59.3 Implementing the Action Methods.....	414
59.4 Testing the Gesture Recognition Application .....	415
59.5 Summary.....	415
<b>60. A 3D Touch Force Handling Tutorial .....</b>	<b>417</b>
60.1 Creating the 3D Touch Example Project .....	417
60.2 Adding the UIView Subclass to the Project.....	417
60.3 Locating the draw Method in the UIView Subclass .....	417
60.4 Implementing the Touch Methods .....	418
60.5 Testing the Touch Force App .....	419
60.6 Summary.....	420
<b>61. An iOS 11 3D Touch Quick Actions Tutorial .....</b>	<b>421</b>
61.1 Creating the Quick Actions Example Project .....	421
61.2 Static Quick Action Keys.....	421
61.3 Adding a Static Quick Action to the Project.....	421
61.4 Adding a Dynamic Quick Action .....	423
61.5 Adding, Removing and Changing Dynamic Quick Actions .....	424
61.6 Responding to a Quick Action Selection .....	424
61.7 Testing the Quick Action App.....	425
61.8 Summary.....	426
<b>62. An iOS 11 3D Touch Peek and Pop Tutorial.....</b>	<b>427</b>
62.1 About the Example Project .....	427
62.2 Adding the UIViewControllerPreviewDelegate.....	427
62.3 Implementing the Peek Delegate Method .....	428
62.4 Assigning the Detail Controller Storyboard ID .....	429
62.5 Implementing the Pop Delegate Method .....	430

62.6 Registering the Previewing Delegate.....	430
62.7 Testing the Peek and Pop Behavior .....	431
62.8 Adding Peek Quick Actions .....	432
62.9 Summary.....	434
<b>63. Implementing Touch ID and Face ID Authentication in iOS 11 Apps.....</b>	<b>435</b>
63.1 The Local Authentication Framework.....	435
63.2 Checking for Biometric Authentication Availability .....	435
63.3 Identifying Authentication Options .....	436
63.4 Evaluating Biometric Policy .....	436
63.5 A Biometric Authentication Example Project .....	437
63.6 Checking for Biometric Availability .....	438
63.7 Seeking Biometric Authentication .....	439
63.8 Adding the Face ID Privacy Statement.....	441
63.9 Testing the Application .....	441
63.10 Summary.....	443
<b>64. Drawing iOS 11 2D Graphics with Core Graphics.....</b>	<b>445</b>
64.1 Introducing Core Graphics and Quartz 2D.....	445
64.2 The draw Method .....	445
64.3 Points, Coordinates and Pixels.....	445
64.4 The Graphics Context .....	446
64.5 Working with Colors in Quartz 2D .....	446
64.6 Summary.....	447
<b>65. Interface Builder Live Views and iOS 11 Embedded Frameworks .....</b>	<b>449</b>
65.1 Embedded Frameworks.....	449
65.2 Interface Builder Live Views .....	449
65.3 Creating the Example Project .....	450
65.4 Adding an Embedded Framework .....	451
65.5 Implementing the Drawing Code in the Framework .....	452
65.6 Making the View Designable .....	453
65.7 Making Variables Inspectable.....	454
65.8 Summary.....	454
<b>66. An iOS 11 Graphics Tutorial using Core Graphics and Core Image .....</b>	<b>455</b>
66.1 The iOS Drawing Example Application.....	455
66.2 Creating the New Project.....	455
66.3 Creating the UIView Subclass .....	455
66.4 Locating the draw Method in the UIView Subclass .....	456
66.5 Drawing a Line .....	456
66.6 Drawing Paths.....	458
66.7 Drawing a Rectangle .....	459
66.8 Drawing an Ellipse or Circle .....	459
66.9 Filling a Path with a Color .....	460
66.10 Drawing an Arc .....	461
66.11 Drawing a Cubic Bézier Curve .....	462
66.12 Drawing a Quadratic Bézier Curve .....	462
66.13 Dashed Line Drawing .....	463

66.14 Drawing Shadows .....	464
66.15 Drawing Gradients .....	465
66.16 Drawing an Image into a Graphics Context .....	469
66.17 Image Filtering with the Core Image Framework .....	470
66.18 Summary .....	472
<b>67. iOS 11 Animation using UIViewPropertyAnimator .....</b>	<b>473</b>
67.1 The Basics of UIKit Animation .....	473
67.2 Understanding Animation Curves .....	474
67.3 Performing Affine Transformations .....	474
67.4 Combining Transformations .....	475
67.5 Creating the Animation Example Application .....	475
67.6 Implementing the Variables .....	475
67.7 Drawing in the UIView .....	476
67.8 Detecting Screen Touches and Performing the Animation .....	476
67.9 Building and Running the Animation Application .....	478
67.10 Implementing Spring Timing .....	479
67.11 Summary .....	479
<b>68. iOS 11 UIKit Dynamics – An Overview .....</b>	<b>481</b>
68.1 Understanding UIKit Dynamics .....	481
68.2 The UIKit Dynamics Architecture .....	481
68.2.1 <i>Dynamic Items</i> .....	481
68.2.2 <i>Dynamic Behaviors</i> .....	482
68.2.3 <i>The Reference View</i> .....	482
68.2.4 <i>The Dynamic Animator</i> .....	482
68.3 Implementing UIKit Dynamics in an iOS Application .....	483
68.4 Dynamic Animator Initialization .....	483
68.5 Configuring Gravity Behavior .....	483
68.6 Configuring Collision Behavior .....	484
68.7 Configuring Attachment Behavior .....	485
68.8 Configuring Snap Behavior .....	486
68.9 Configuring Push Behavior .....	486
68.10 The UIDynamicItemBehavior Class .....	487
68.11 Combining Behaviors to Create a Custom Behavior .....	488
68.12 Summary .....	489
<b>69. An iOS 11 UIKit Dynamics Tutorial .....</b>	<b>491</b>
69.1 Creating the UIKit Dynamics Example Project .....	491
69.2 Adding the Dynamic Items .....	491
69.3 Creating the Dynamic Animator Instance .....	492
69.4 Adding Gravity to the Views .....	493
69.5 Implementing Collision Behavior .....	494
69.6 Attaching a View to an Anchor Point .....	495
69.7 Implementing a Spring Attachment Between two Views .....	498
69.8 Summary .....	499
<b>70. An Overview of iOS Collection View and Flow Layout .....</b>	<b>501</b>
70.1 An Overview of Collection Views .....	501

70.2 The UICollectionView Class.....	502
70.3 The UICollectionViewCell Class.....	502
70.4 The UICollectionViewReusableView Class .....	503
70.5 The UICollectionViewFlowLayout Class .....	503
70.6 The UICollectionViewLayoutAttributes Class .....	503
70.7 The UICollectionViewDataSource Protocol .....	504
70.8 The UICollectionViewDelegate Protocol.....	504
70.9 The UICollectionViewDelegateFlowLayout Protocol .....	505
70.10 Cell and View Reuse.....	505
70.11 Summary.....	506
<b>71. An iOS 11 Storyboard-based Collection View Tutorial.....</b>	<b>509</b>
71.1 Creating the Collection View Example Project .....	509
71.2 Removing the Template View Controller .....	509
71.3 Adding a Collection View Controller to the Storyboard .....	509
71.4 Adding the Collection View Cell Class to the Project.....	510
71.5 Designing the Cell Prototype .....	511
71.6 Implementing the Data Model .....	512
71.7 Implementing the Data Source.....	513
71.8 Testing the Application .....	514
71.9 Setting Sizes for Cell Items.....	515
71.10 Changing Scroll Direction.....	516
71.11 Implementing a Supplementary View .....	517
71.12 Implementing the Supplementary View Protocol Methods .....	518
71.13 Summary.....	519
<b>72. Subclassing and Extending the Collection View Flow Layout.....</b>	<b>521</b>
72.1 About the Example Layout Class.....	521
72.2 Subclassing the UICollectionViewFlowLayout Class .....	521
72.3 Extending the New Layout Class.....	521
72.4 Overriding the layoutAttributesForItem(at indexPath:) Method .....	522
72.5 Overriding the layoutAttributesForElements(in rect:) Method.....	523
72.6 Implementing the modifyLayoutAttributes Method .....	523
72.7 Adding the New Layout and Pinch Gesture Recognizer .....	524
72.8 Implementing the Pinch Recognizer .....	525
72.9 Avoiding Image Clipping .....	526
72.10 Testing the Application .....	527
72.11 Summary.....	527
<b>73. An Introduction to Drag and Drop in iOS 11.....</b>	<b>529</b>
73.1 An Overview of Drag and Drop.....	529
73.2 Drag and Drop Delegates.....	529
73.3 Drag and Drop Interactions .....	529
73.4 The Drag Item .....	530
73.5 The Drag and Drop Lifecycle .....	530
73.6 Spring Loaded Controls.....	530
73.7 Summary.....	531
<b>74. An iOS 11 Drag and Drop Tutorial .....</b>	<b>533</b>

74.1 Creating the Drag and Drop Project.....	533
74.2 Designing the User Interface .....	533
74.3 Testing the Default Behavior .....	535
74.4 Adding Drop Support to the Image View .....	536
74.5 Testing the Drop Behavior .....	539
74.6 Adding Drag Support to the Views.....	540
74.7 Testing the Drag Behavior.....	541
74.8 Customizing the Lift Preview Image.....	541
74.9 Testing the Custom Preview Image .....	544
74.10 Implementing Animation .....	545
74.11 Summary.....	545
<b>75. An iOS 11 Collection View Drag and Drop Tutorial .....</b>	<b>547</b>
75.1 The Example Application .....	547
75.2 Declaring the Drag Delegate .....	547
75.3 Implementing Drag Support .....	548
75.4 Dragging Multiple Items .....	548
75.5 Adding the Drop Delegate .....	549
75.6 Implementing the Delegate Methods.....	550
75.7 Adding Drag and Drop Animation .....	552
75.8 Adding the Move Behavior .....	554
75.9 TableView Drag and Drop .....	556
75.10 Summary.....	556
<b>76. Integrating Maps into iOS 11 Applications using MKMapItem.....</b>	<b>557</b>
76.1 MKMapItem and MKPlacemark Classes .....	557
76.2 An Introduction to Forward and Reverse Geocoding .....	557
76.3 Creating MKPlacemark Instances .....	559
76.4 Working with MKMapItem .....	559
76.5 MKMapItem Options and Configuring Directions.....	560
76.6 Adding Item Details to an MKMapItem .....	561
76.7 Summary.....	563
<b>77. An Example iOS 11 MKMapItem Application .....</b>	<b>565</b>
77.1 Creating the MapItem Project .....	565
77.2 Designing the User Interface .....	565
77.3 Converting the Destination using Forward Geocoding.....	566
77.4 Launching the Map .....	567
77.5 Building and Running the Application.....	568
77.6 Summary.....	569
<b>78. Getting Location Information using the iOS 11 Core Location Framework.....</b>	<b>571</b>
78.1 The Core Location Manager.....	571
78.2 Requesting Location Access Authorization .....	571
78.3 Configuring the Desired Location Accuracy .....	572
78.4 Configuring the Distance Filter .....	572
78.5 Continuous Background Location Updates.....	573
78.6 The Location Manager Delegate .....	574
78.7 Starting and Stopping Location Updates .....	574

78.8 Obtaining Location Information from CLLocation Objects .....	575
78.8.1 Longitude and Latitude .....	575
78.8.2 Accuracy .....	575
78.8.3 Altitude.....	575
78.9 Getting the Current Location.....	575
78.10 Calculating Distances .....	576
78.11 Summary.....	576
<b>79. An Example iOS 11 Location Application.....</b>	<b>577</b>
79.1 Creating the Example iOS 11 Location Project .....	577
79.2 Designing the User Interface .....	577
79.3 Configuring the CLLocationManager Object.....	579
79.4 Setting up the Usage Description Keys .....	579
79.5 Implementing the startWhenInUse Method .....	580
79.6 Implementing the startAlways Method.....	580
79.7 Implementing the resetDistance Method .....	580
79.8 Implementing the Application Delegate Methods .....	580
79.9 Building and Running the Location Application.....	581
79.10 Adding Continuous Background Location Updates .....	583
79.11 Summary.....	584
<b>80. Working with Maps on iOS 11 with MapKit and the MKMapView Class .....</b>	<b>585</b>
80.1 About the MapKit Framework .....	585
80.2 Understanding Map Regions .....	585
80.3 Getting Transit ETA Information.....	585
80.4 About the MKMapView Tutorial.....	586
80.5 Creating the Map Project.....	586
80.6 Adding the Navigation Controller .....	587
80.7 Creating the MKMapView Instance and Toolbar .....	587
80.8 Obtaining Location Information Permission .....	590
80.9 Setting up the Usage Description Keys .....	590
80.10 Configuring the Map View .....	590
80.11 Changing the MapView Region.....	591
80.12 Changing the Map Type.....	591
80.13 Testing the MapView Application.....	592
80.14 Updating the Map View based on User Movement .....	592
80.15 Summary.....	593
<b>81. Working with MapKit Local Search in iOS 11 .....</b>	<b>595</b>
81.1 An Overview of iOS 11 Local Search .....	595
81.2 Adding Local Search to the MapSample Application .....	596
81.3 Adding the Local Search Text Field .....	596
81.4 Performing the Local Search.....	598
81.5 Testing the Application .....	600
81.6 Customized Annotation Markers.....	600
81.7 Annotation Marker Clustering .....	604
81.8 Summary.....	605
<b>82. Using MKDirections to get iOS 11 Map Directions and Routes.....</b>	<b>607</b>



82.1 An Overview of MKDirections.....	607
82.2 Adding Directions and Routes to the MapSample Application.....	609
82.3 Adding the New Classes to the Project.....	609
82.4 Configuring the Results Table View .....	609
82.5 Implementing the Result Table View Segue .....	611
82.6 Adding the Route Scene.....	612
82.7 Identifying the User’s Current Location.....	613
82.8 Getting the Route and Directions .....	614
82.9 Establishing the Route Segue.....	616
82.10 Testing the Application .....	616
82.11 Summary.....	617
<b>83. An iOS 11 MapKit Flyover Tutorial.....</b>	<b>619</b>
83.1 MKMapView Flyover Map Types .....	619
83.2 The MKMapCamera Class .....	620
83.3 An MKMapKit Flyover Example .....	620
83.4 Designing the User Interface .....	621
83.5 Configuring the Map View and Camera.....	622
83.6 Animating Camera Changes.....	623
83.7 Testing the Map Flyover App.....	624
83.8 Summary.....	624
<b>84. Accessing the iOS 11 Camera and Photo Library .....</b>	<b>625</b>
84.1 The UIImagePickerController Class.....	625
84.2 Creating and Configuring a UIImagePickerController Instance .....	625
84.3 Configuring the UIImagePickerController Delegate.....	626
84.4 Detecting Device Capabilities .....	627
84.5 Saving Movies and Images.....	627
84.6 Summary.....	628
<b>85. An Example iOS 11 Camera Application.....</b>	<b>629</b>
85.1 An Overview of the Application .....	629
85.2 Creating the Camera Project.....	629
85.3 Designing the User Interface .....	629
85.4 Implementing the Action Methods.....	631
85.5 Writing the Delegate Methods .....	632
85.6 Seeking Camera and Photo Library Access .....	633
85.7 Building and Running the Application.....	634
85.8 Summary.....	635
<b>86. iOS 11 Video Playback using AVPlayer and AVPlayerViewController .....</b>	<b>637</b>
86.1 The AVPlayer and AVPlayerViewController Classes.....	637
86.2 The iOS Movie Player Example Application .....	637
86.3 Adding a Security Exception for an HTTP Connection .....	637
86.4 Designing the User Interface .....	638
86.5 Initializing Video Playback.....	639
86.6 Build and Run the Application .....	640
86.7 Creating an AVPlayerViewController Instance from Code.....	640
86.8 Summary.....	641

<b>87. An iOS 11 Multitasking Picture in Picture Tutorial .....</b>	<b>643</b>
87.1 An Overview of Picture in Picture Multitasking.....	643
87.2 Adding Picture in Picture Support to the AVPlayerDemo App .....	644
87.3 Adding the Navigation Controller .....	644
87.4 Setting the Audio Session Category .....	644
87.5 Implementing the Delegate .....	646
87.6 Opting Out of Picture in Picture Support.....	647
87.7 Additional Delegate Methods.....	647
87.8 Summary.....	648
<b>88. An Introduction to Extensions in iOS 11.....</b>	<b>649</b>
88.1 iOS Extensions – An Overview .....	649
88.2 Extension Types .....	649
88.2.1 Today Extension .....	650
88.2.2 Share Extension .....	650
88.2.3 Action Extension .....	651
88.2.4 Photo Editing Extension.....	652
88.2.5 Document Provider Extension .....	653
88.2.6 Custom Keyboard Extension .....	653
88.2.7 Audio Unit Extension .....	653
88.2.8 Shared Links Extension .....	653
88.2.9 Content Blocking Extension .....	653
88.2.10 Sticker Pack Extension .....	653
88.2.11 iMessage Extension .....	653
88.2.12 Intents Extension .....	653
88.3 Creating Extensions .....	654
88.4 Summary.....	654
<b>89. An iOS 11 Today Extension Widget Tutorial .....</b>	<b>655</b>
89.1 About the Example Extension Widget .....	655
89.2 Creating the Example Project .....	655
89.3 Adding the Extension to the Project .....	655
89.4 Reviewing the Extension Files.....	658
89.5 Designing the Widget User Interface.....	658
89.6 Setting the Preferred Content Size in Code .....	660
89.7 Modifying the Widget View Controller .....	660
89.8 Testing the Extension .....	662
89.9 Opening the Containing App from the Extension .....	662
89.10 Summary.....	664
<b>90. Creating an iOS 11 Photo Editing Extension .....</b>	<b>665</b>
90.1 Creating a Photo Editing Extension.....	665
90.2 Accessing the Photo Editing Extension .....	666
90.3 Configuring the Info.plist File.....	668
90.4 Designing the User Interface .....	668
90.5 The PHContentEditingController Protocol.....	669
90.6 Photo Extensions and Adjustment Data .....	670
90.7 Receiving the Content .....	670
90.8 Implementing the Filter Actions .....	671

90.9 Returning the Image to the Photos App .....	674
90.10 Testing the Application .....	676
90.11 Summary.....	677
<b>91. Creating an iOS 11 Action Extension .....</b>	<b>679</b>
91.1 An Overview of Action Extensions .....	679
91.2 About the Action Extension Example.....	680
91.3 Creating the Action Extension Project .....	680
91.4 Adding the Action Extension Target .....	680
91.5 Changing the Extension Display Name .....	681
91.6 Designing the Action Extension User Interface.....	681
91.7 Receiving the Content.....	682
91.8 Returning the Modified Data to the Host App.....	684
91.9 Testing the Extension.....	685
91.10 Declaring the Supported Content Type .....	687
91.11 Summary.....	688
<b>92. Receiving Data from an iOS 11 Action Extension .....</b>	<b>689</b>
92.1 Creating the Example Project .....	689
92.2 Designing the User Interface .....	689
92.3 Importing the Mobile Core Services Framework .....	690
92.4 Adding an Action Button to the Application .....	690
92.5 Receiving Data from an Extension .....	692
92.6 Testing the Application .....	693
92.7 Summary.....	693
<b>93. An Introduction to Building iOS 11 Message Apps .....</b>	<b>695</b>
93.1 Introducing Message Apps.....	695
93.2 Types of Message App .....	696
93.3 The Key Messages Framework Classes .....	696
93.3.1 <i>MSMessagesAppViewController</i> .....	697
93.3.2 <i>MSConversation</i> .....	697
93.3.3 <i>MSMessage</i> .....	698
93.3.4 <i>MSMessageTemplateLayout</i> .....	698
93.4 Sending Simple Messages .....	699
93.5 Creating an <i>MSMessage</i> Message .....	700
93.6 Receiving a Message .....	700
93.7 Supported Message App Platforms .....	701
93.8 Summary.....	701
<b>94. An iOS 11 Interactive Message App Tutorial .....</b>	<b>703</b>
94.1 About the Example Message App Project.....	703
94.2 Creating the MessageApp Project .....	703
94.3 Designing the MessageApp User Interface.....	705
94.4 Creating the Outlet Collection .....	707
94.5 Creating the Game Model.....	708
94.6 Responding to Button Selections.....	709
94.7 Preparing the Message URL.....	710
94.8 Preparing and Inserting the Message .....	711

94.9 Message Receipt Handling.....	712
94.10 Setting the Message Image .....	713
94.11 Implementing a Session.....	715
94.12 Summary.....	716
<b>95. An Introduction to SiriKit .....</b>	<b>717</b>
95.1 Siri and SiriKit.....	717
95.2 SiriKit Domains.....	717
95.3 SiriKit Intents .....	718
95.4 How SiriKit Integration Works .....	718
95.5 Resolving Intent Parameters .....	719
95.6 The Confirm Method .....	720
95.7 The Handle Method.....	720
95.8 Custom Vocabulary.....	721
95.9 The Siri User Interface .....	721
95.10 Summary.....	721
<b>96. An iOS 11 Example SiriKit Messaging Extension .....</b>	<b>723</b>
96.1 Creating the Example Project .....	723
96.2 Enabling the Siri Entitlement .....	723
96.3 Seeking Siri Authorization.....	723
96.4 Adding the Extensions .....	724
96.5 Supported Intents.....	725
96.6 Using the Default User Interface .....	725
96.7 Trying the Example .....	726
96.8 Specifying a Default Phrase .....	726
96.9 Reviewing the Intent Handler .....	727
96.10 Summary.....	728
<b>97. Customizing the SiriKit Intent User Interface .....</b>	<b>729</b>
97.1 Modifying the UI Extension .....	729
97.2 Using the configure Method.....	729
97.3 Designing the Siri Snippet Scene .....	730
97.4 Adding the configure Method .....	731
97.5 Overriding Siri Content .....	732
97.6 Using the configureView Method.....	733
97.7 Implementing a configureView Custom UI.....	734
97.8 Summary.....	738
<b>98. An iOS 11 SiriKit Photo Search Tutorial .....</b>	<b>739</b>
98.1 About the SiriKit Photo Search Project .....	739
98.2 Creating the SiriPhoto Project .....	739
98.3 Enabling the Siri Entitlement .....	739
98.4 Obtaining Siri Authorization .....	739
98.5 Designing the App User Interface .....	741
98.6 Adding the Intents Extension to the Project.....	741
98.7 Reviewing the Default Intents Extension.....	742
98.8 Modifying the Supported Intents .....	742
98.9 Modifying the IntentHandler Implementation .....	743

98.10 Implementing the Resolve Methods.....	743
98.11 Implementing the Confirmation Method .....	745
98.12 Handling the Intent.....	745
98.13 Testing the App .....	747
98.14 Handling the NSUserActivity Object .....	747
98.15 Testing the Completed App .....	748
98.16 Summary.....	748
<b>99. An iOS 11 Local Notification Tutorial .....</b>	<b>749</b>
99.1 Creating the Local Notification App Project.....	749
99.2 Requesting Notification Authorization .....	749
99.3 Designing the User Interface .....	750
99.4 Creating the Message Content .....	751
99.5 Specifying a Notification Trigger .....	752
99.6 Creating the Notification Request .....	752
99.7 Adding the Request .....	753
99.8 Testing the Notification .....	753
99.9 Receiving Notifications in the Foreground .....	753
99.10 Adding Notification Actions .....	754
99.11 Handling Notification Actions .....	756
99.12 Hidden Notification Content.....	757
99.13 Managing Notifications.....	759
99.14 Summary.....	760
<b>100. Playing Audio on iOS 11 using AVAudioPlayer .....</b>	<b>761</b>
100.1 Supported Audio Formats.....	761
100.2 Receiving Playback Notifications .....	761
100.3 Controlling and Monitoring Playback .....	762
100.4 Creating the Audio Example Application .....	762
100.5 Adding an Audio File to the Project Resources.....	762
100.6 Designing the User Interface .....	762
100.7 Implementing the Action Methods.....	763
100.8 Creating and Initializing the AVAudioPlayer Object.....	764
100.9 Implementing the AVAudioPlayerDelegate Protocol Methods .....	765
100.10 Building and Running the Application.....	765
100.11 Summary .....	765
<b>101. Recording Audio on iOS 11 with AVAudioRecorder .....</b>	<b>767</b>
101.1 An Overview of the AVAudioRecorder Tutorial .....	767
101.2 Creating the Recorder Project .....	767
101.3 Configuring the Microphone Usage Description .....	767
101.4 Designing the User Interface .....	768
101.5 Creating the AVAudioRecorder Instance .....	769
101.6 Implementing the Action Methods.....	770
101.7 Implementing the Delegate Methods.....	771
101.8 Testing the Application .....	772
101.9 Summary.....	772
<b>102. An iOS 11 Speech Recognition Tutorial.....</b>	<b>773</b>

102.1 An Overview of Speech Recognition in iOS.....	773
102.2 Speech Recognition Authorization .....	774
102.3 Transcribing Recorded Audio.....	774
102.4 Transcribing Live Audio.....	774
102.5 An Audio File Speech Recognition Tutorial.....	774
102.6 Modifying the User Interface.....	774
102.7 Adding the Speech Recognition Permission .....	775
102.8 Seeking Speech Recognition Authorization .....	776
102.9 Performing the Transcription .....	777
102.10 Testing the App.....	778
102.11 Summary.....	778
<b>103. An iOS 11 Real-Time Speech Recognition Tutorial.....</b>	<b>779</b>
103.1 Creating the Project.....	779
103.2 Designing the User Interface .....	779
103.3 Adding the Speech Recognition Permission .....	780
103.4 Requesting Speech Recognition Authorization .....	780
103.5 Declaring and Initializing the Speech and Audio Objects .....	781
103.6 Starting the Transcription.....	782
103.7 Implementing the stopTranscribing Method.....	785
103.8 Testing the App.....	785
103.9 Summary.....	786
<b>104. iOS 11 Multitasking, Background Transfer Service and Fetching .....</b>	<b>787</b>
104.1 Understanding iOS Application States .....	787
104.2 A Brief Overview of the Multitasking Application Lifecycle.....	788
104.3 Checking for Multitasking Support .....	788
104.4 Enabling Multitasking for an iOS Application .....	789
104.5 Supported Forms of Background Execution .....	789
104.6 An Overview of Background Fetch .....	789
104.7 An Overview of Remote Notifications .....	791
104.8 An Overview of Local Notifications.....	792
104.9 An Overview of Background Transfer Service .....	792
104.10 The Rules of Background Execution.....	792
104.11 Summary.....	792
<b>105. An Overview of iOS 11 Application State Preservation and Restoration .....</b>	<b>793</b>
105.1 The Preservation and Restoration Process .....	793
105.2 Opting In to Preservation and Restoration .....	794
105.3 Assigning Restoration Identifiers.....	794
105.4 Default Preservation Features of UIKit .....	795
105.5 Saving and Restoring Additional State Information.....	795
105.6 Understanding the Restoration Process .....	796
105.7 Saving General Application State.....	797
105.8 Summary.....	797
<b>106. An iOS 11 State Preservation and Restoration Tutorial.....</b>	<b>799</b>
106.1 Creating the Example Application .....	799
106.2 Trying the Application without State Preservation .....	799

106.3	Opting-in to State Preservation .....	799
106.4	Setting Restoration Identifiers .....	800
106.5	Encoding and Decoding View Controller State .....	800
106.6	Adding a Navigation Controller to the Storyboard .....	802
106.7	Adding the Third View Controller .....	803
106.8	Creating the Restoration Class .....	804
106.9	Summary .....	805
<b>107.</b>	<b>An Introduction to iOS 11 Sprite Kit Programming .....</b>	<b>807</b>
107.1	What is Sprite Kit?.....	807
107.2	The Key Components of a Sprite Kit Game .....	807
107.2.1	<i>Sprite Kit View</i> .....	807
107.2.2	<i>Scenes</i> .....	808
107.2.3	<i>Nodes</i> .....	808
107.2.4	<i>Physics Bodies</i> .....	808
107.2.5	<i>Physics World</i> .....	809
107.2.6	<i>Actions</i> .....	809
107.2.7	<i>Transitions</i> .....	809
107.2.8	<i>Texture Atlas</i> .....	809
107.2.9	<i>Constraints</i> .....	809
107.3	An Example Sprite Kit Game Hierarchy.....	810
107.4	The Sprite Kit Game Rendering Loop .....	810
107.5	The Sprite Kit Level Editor .....	811
107.6	Summary.....	811
<b>108.</b>	<b>An iOS 11 Sprite Kit Level Editor Game Tutorial.....</b>	<b>813</b>
108.1	About the Sprite Kit Demo Game .....	813
108.2	Creating the SpriteKitDemo Project.....	814
108.3	Reviewing the SpriteKit Game Template Project.....	814
108.4	Restricting Interface Orientation .....	815
108.5	Modifying the GameScene SpriteKit Scene File .....	815
108.6	Creating the Archery Scene .....	817
108.7	Transitioning to the Archery Scene.....	818
108.8	Adding the Texture Atlas .....	819
108.9	Designing the Archery Scene .....	820
108.10	Preparing the Archery Scene .....	822
108.11	Preparing the Animation Texture Atlas .....	822
108.12	Creating the Named Action Reference .....	824
108.13	Testing Actions in an Action File .....	824
108.14	Triggering the Named Action from the Code.....	825
108.15	Creating the Arrow Sprite Node .....	825
108.16	Shooting the Arrow.....	826
108.17	Adding the Ball Sprite Node.....	827
108.18	Summary .....	828
<b>109.</b>	<b>An iOS 11 Sprite Kit Collision Handling Tutorial .....</b>	<b>829</b>
109.1	Defining the Category Bit Masks.....	829
109.2	Assigning the Category Masks to the Sprite Nodes .....	829
109.3	Configuring the Collision and Contact Masks .....	830

109.4 Implementing the Contact Delegate .....	831
109.5 Game Over.....	832
109.6 Summary.....	834
<b>110. An iOS 11 Sprite Kit Particle Emitter Tutorial .....</b>	<b>835</b>
110.1 What is the Particle Emitter?.....	835
110.2 The Particle Emitter Editor .....	835
110.3 The SKEmitterNode Class.....	835
110.4 Using the Particle Emitter Editor .....	836
110.5 Particle Emitter Node Properties.....	837
110.5.1 Background .....	837
110.5.2 Particle Texture .....	837
110.5.3 Particle Birthrate .....	838
110.5.4 Particle Life Cycle.....	838
110.5.5 Particle Position Range.....	838
110.5.6 Angle .....	838
110.5.7 Particle Speed.....	838
110.5.8 Particle Acceleration.....	838
110.5.9 Particle Scale .....	838
110.5.10 Particle Rotation.....	838
110.5.11 Particle Color .....	838
110.5.12 Particle Blend Mode .....	839
110.6 Experimenting with the Particle Emitter Editor.....	839
110.7 Bursting a Ball using Particle Emitter Effects .....	840
110.8 Adding the Burst Particle Emitter Effect.....	841
110.9 Adding an Audio Action .....	842
110.10 Summary.....	843
<b>111. Making Store Purchases with the SKStoreProductViewController Class.....</b>	<b>845</b>
111.1 The SKStoreProductViewController Class.....	845
111.2 Creating the Example Project .....	846
111.3 Creating the User Interface .....	846
111.4 Displaying the Store Kit Product View Controller .....	847
111.5 Implementing the Delegate Method .....	848
111.6 Testing the Application.....	848
111.7 Summary.....	849
<b>112. Building In-App Purchasing into iOS 11 Applications.....</b>	<b>851</b>
112.1 In-App Purchase Options .....	851
112.2 Uploading App Store Hosted Content .....	851
112.3 Configuring In-App Purchase Items .....	852
112.4 Sending a Product Request.....	852
112.5 Accessing the Payment Queue .....	853
112.6 The Transaction Observer Object .....	853
112.7 Initiating the Purchase.....	853
112.8 The Transaction Process .....	853
112.9 Transaction Restoration Process .....	855
112.10 Testing In-App Purchases.....	855
112.11 Promoting In-App Purchases .....	856



112.12 Requesting App Reviews.....	856
112.13 Summary.....	857
<b>113. Preparing an iOS 11 Application for In-App Purchases.....</b>	<b>859</b>
113.1 About the Example Application .....	859
113.2 Creating the Xcode Project .....	859
113.3 Registering and Enabling the App ID for In App Purchasing .....	859
113.4 Configuring the Application in iTunes Connect.....	860
113.5 Creating an In-App Purchase Item .....	860
113.6 Summary.....	861
<b>114. An iOS 11 In-App Purchase Tutorial .....</b>	<b>863</b>
114.1 The Application User Interface .....	863
114.2 Designing the Storyboard .....	863
114.3 Configuring the View Controller Class .....	865
114.4 Initiating and Handling the Purchase.....	866
114.5 Testing the Application .....	868
114.6 Troubleshooting.....	869
114.7 Promoting the In-App Purchase.....	869
114.8 Summary.....	871
<b>115. Configuring and Creating App Store Hosted Content for iOS 11 In-App Purchases.....</b>	<b>873</b>
115.1 Configuring an Application for In-App Purchase Hosted Content .....	873
115.2 The Anatomy of an In-App Purchase Hosted Content Package .....	873
115.3 Creating an In-App Purchase Hosted Content Package .....	874
115.4 Archiving the Hosted Content Package.....	874
115.5 Validating the Hosted Content Package .....	875
115.6 Uploading the Hosted Content Package .....	875
115.7 Summary.....	876
<b>116. Preparing and Submitting an iOS 11 Application to the App Store.....</b>	<b>877</b>
116.1 Verifying the iOS Distribution Certificate.....	877
116.2 Adding App Icons .....	879
116.3 Designing the Launch Screen .....	880
116.4 Assign the Project to a Team .....	880
116.5 Archiving the Application for Distribution .....	881
116.6 Configuring the Application in iTunes Connect.....	881
116.7 Validating and Submitting the Application .....	883
116.8 Configuring and Submitting the App for Review .....	885
<b>Index.....</b>	<b>887</b>



## 1. Start Here

The goal of this book is to teach the skills necessary to create iOS applications using the iOS 11 SDK, Xcode 9 and the Swift 4 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 10 and need to get up to speed on the features of iOS 11 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 11 and programming in Swift 4 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, graphics drawing and animation are also covered, as are touch screen handling, gesture recognition, multitasking, location management, local notifications, camera access and video playback support. Other features are also covered including Auto Layout, local map search, user interface animation using UIKit dynamics, Siri integration, iMessage app development, CloudKit sharing and biometric authentication.

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

The key new features of iOS 11 and Xcode 9 are also covered in detail, including Swift 4, drag and drop integration and the document browser.

The aim of this book, therefore, is to teach you the skills necessary to build your own apps for iOS 11. Assuming you are ready to download the iOS 11 SDK and Xcode 9, have an Intel-based Mac and ideas for some apps to develop, you are ready to get started.

### 1.2 For iOS 10 Developers

If you have already read the iOS 10 edition of this book, or have experience with the iOS 10 SDK then you might prefer to go directly to the new chapters in this iOS 11 edition of the book.

All chapters have been updated to reflect the changes and features introduced as part of iOS 11, Swift 4 and Xcode 9. Chapters included in this edition that were not contained in the previous edition, or have been significantly rewritten for iOS 11 and Xcode 9 are as follows:

- *Integrating Search using the iOS UISearchController*
- *An Overview of the iOS Document Browser View Controller*
- *An iOS Document Browser Tutorial*
- *Implementing Touch ID and Face ID Authentication in iOS 11 Apps*
- *An Overview of iOS Collection View and Flow Layout*
- *An iOS 11 Storyboard-based Collection View Tutorial*
- *Subclassing and Extending the Collection View Flow Layout*

Start Here

- *An Introduction to Drag and Drop in iOS 11*
- *An iOS 11 Drag and Drop Tutorial*
- *An iOS 11 Collection View Drag and Drop Tutorial*
- *Customizing the SiriKit Intent User Interface*

In addition, the following changes have also been made:

- All chapters have been updated where necessary to reflect the changes made to Xcode 9.
- All chapters and examples have been rewritten where necessary to use Swift 4 syntax.
- *A Guided Tour of Xcode 9* has been updated to include network-based testing of apps via Wi-Fi connection.
- The *Using Xcode 9 Storyboards to Build Dynamic TableViews* chapter has been updated to include the implementation of swipe actions.
- The *Implementing iOS 11 TableView Navigation using Storyboards in Xcode 9* chapter has been extended to cover the new support of larger titles in navigation bars.
- The *An iOS 11 Local Notification Tutorial* chapter has been updated for the new APIs and to cover hidden notification content.
- Location and Map chapters have been updated for the new location tracking usage permissions and the latest annotation and annotation clustering options.
- The in-app purchasing chapters have been extended to cover promotion of in-app items and requesting user reviews.

## 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/retail/ios11/>

## 1.4 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](mailto:feedback@ebookfrenzy.com).

## 1.5 Errata

While 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/ios11.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](mailto:feedback@ebookfrenzy.com).

## 2. Joining the Apple Developer Program

The first step in the process of learning to develop iOS 11 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 9 and the iOS 11 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 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.

While 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:

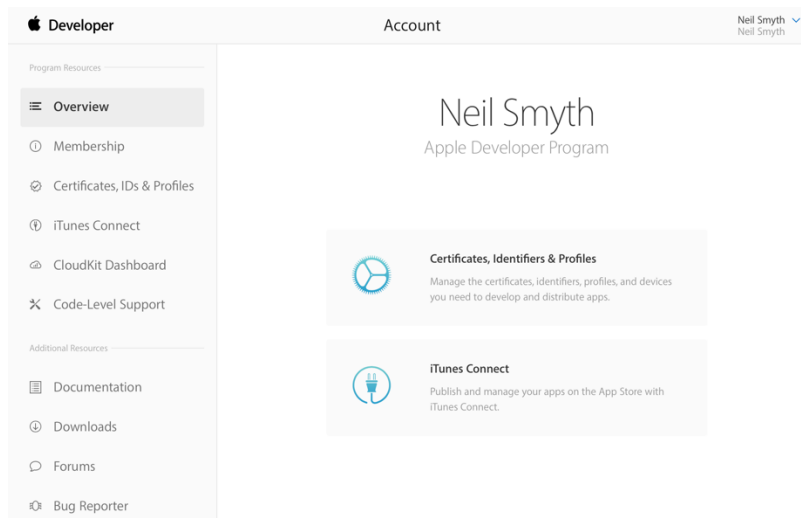


Figure 2-1

## 2.5 Summary

An important early step in the iOS 11 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 11 SDK and Xcode 9 development environment.





## 3. Installing Xcode 9 and the iOS 11 SDK

iOS apps are developed using the iOS SDK in conjunction with Apple's Xcode development environment. Xcode is an integrated development environment (IDE) within which you will code, compile, test and debug your iOS applications. The Xcode 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 and the iOS 11 SDK on macOS.

### 3.1 Identifying if you have an Intel or PowerPC based Mac

Only Intel based macOS 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 Xcode 9 environment requires that the version of macOS running on the system be version 10.12.6 or later. If the "About This Mac" dialog does not indicate that macOS 10.12.6 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 9 and the iOS 11 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 macOS system, enter Xcode into the search box and click on the *Get* 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 11 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, select *Apple ID* from the resulting panel and click on the *Continue* button. When prompted, enter your Apple ID and associated password and click on the *Sign In* button to add the account to the preferences.

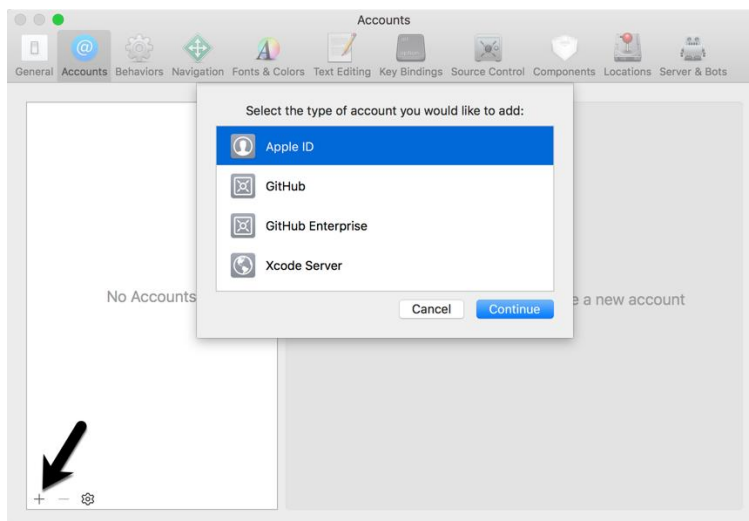


Figure 3-3

### 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 *Manage Certificates...* 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 + button highlighted in Figure 3-4 and make the appropriate selection from the menu:

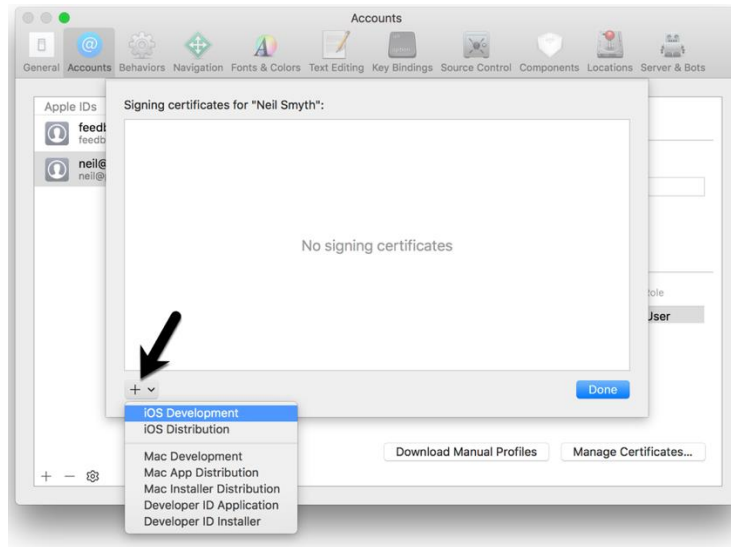


Figure 3-4

If the Apple ID has been used to enroll in the Apple Developer program, the option to create an *iOS Distribution* certificate will, when clicked, generate the signing identity required to submit the app to the Apple App Store.

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



## 4. A Guided Tour of Xcode 9

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 9

As with all iOS examples in this book, the development of our example will take place within the Xcode 9 development environment. If you have not already installed this tool together with the latest iOS SDK refer first to the *Installing Xcode 9 and the iOS 11 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 macOS 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:

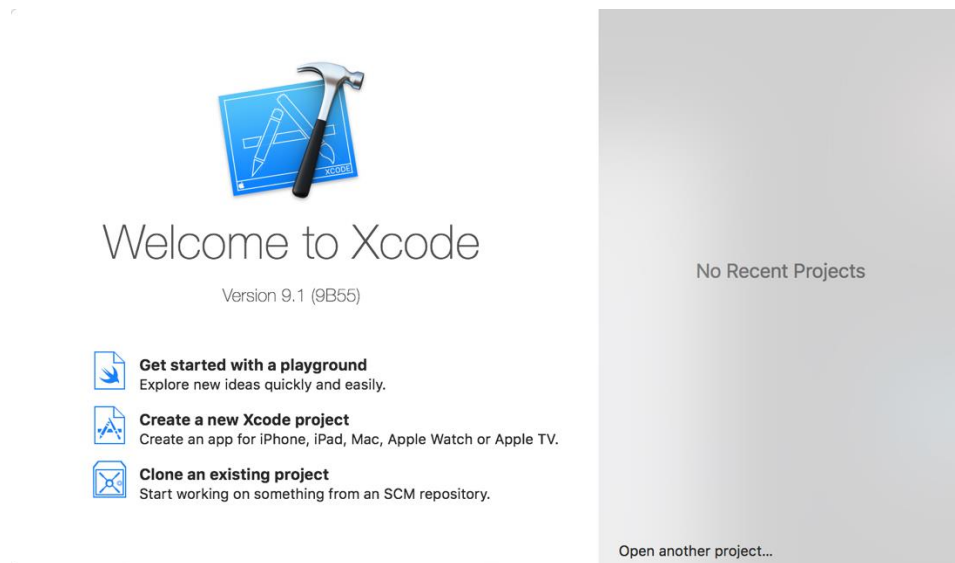


Figure 4-1

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 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:

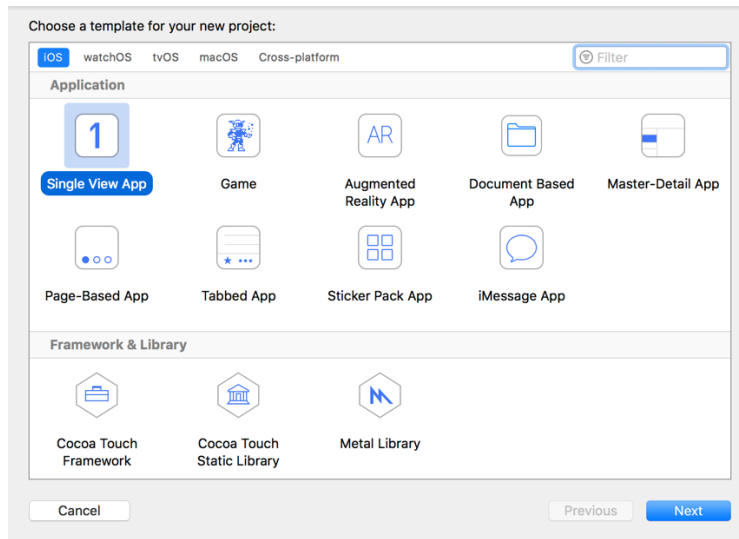


Figure 4-2

The toolbar located on the top edge of the window allows for the selection of the target platform, providing options to develop an application for iOS, watchOS, tvOS or macOS.

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.
- **iMessage Application** – iMessage apps are extensions to the built-in iOS Messages app that allow users to send interactive messages such as games to other users. Once created, iMessage apps are made available for purchase through the Message App Store.
- **Sticker Pack Application** – Allows a sticker pack application to be created and sold within the Message App Store. Sticker pack apps allow additional images to be made available for inclusion in messages sent via the iOS Messages app.
- **Augmented Reality App** – Creates a template project pre-configured to make use of ARKit to integrate augmented reality support into an iOS app.

- **Document Based App** – Creates a project intended for making use of the iOS document browser. The document browser provides a visual environment in which the user can navigate and manage both local and cloud-based files from within an iOS app.

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:

Choose options for your new project:

Product Name: HelloWorld

Team: Neil Smyth

Organization Name: eBookFrenzy

Organization Identifier: com.ebookfrenzy

Bundle Identifier: com.ebookfrenzy.HelloWorld

Language: Swift

Use Core Data

Include Unit Tests

Include UI Tests

Cancel Previous Next

Figure 4-3

On this screen, enter a Product name for the application that is going to be created, in this case “HelloWorld” and then select the development team to which this project is to be assigned. If you have already signed up to the Apple developer program, select your account from the menu, otherwise leave the option set to None. 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.

Apple supports two programming languages for the development of iOS apps in the form of *Objective-C* and *Swift*. While 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. This panel also provides the option to place the project under Git source code control. Source code control systems such as Git allow different revisions of the project to be managed and restored, and for changes made over the development lifecycle of the project to be tracked. Since this is typically used for larger projects, or those involving more than one developer, this option can be turned off for this and the other projects created in the book.

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

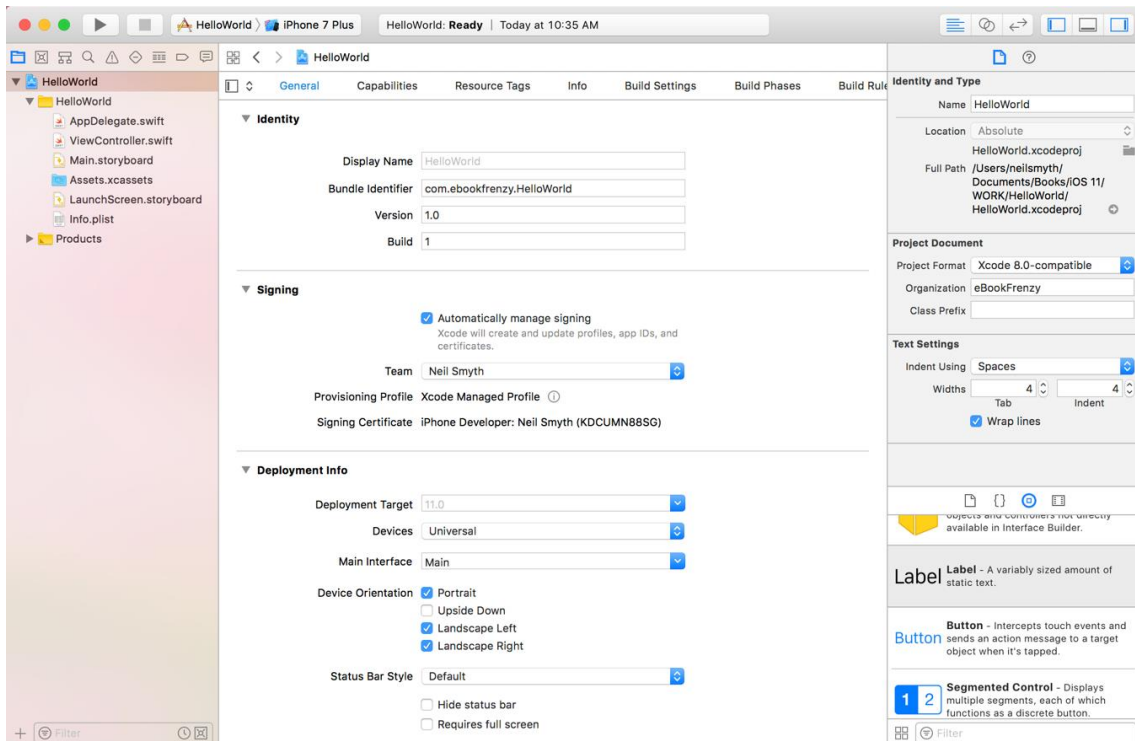


Figure 4-4

Before proceeding we should take some time to look at what Xcode has done for us. First, 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.

The Signing section provides the option to select an Apple identity to use when building the app. This ensures that the app is signed with a certificate when it is compiled. If you have registered your Apple ID with Xcode



using the Preferences screen as outlined in the previous chapter, select that identity now using the Team menu. If no team is selected, it will not be possible to test apps on physical devices, though the simulator environment may still be used.

The Deployment Info section of the screen also includes a setting to specify the device types on which the completed app is intended to run as highlighted in Figure 4-5:

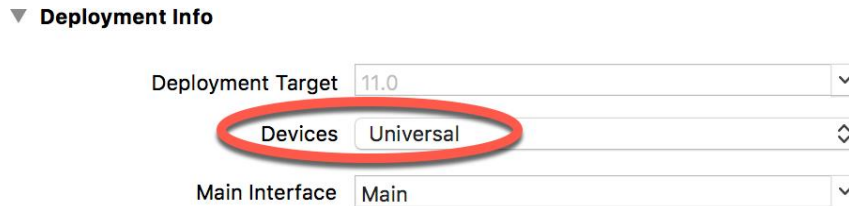


Figure 4-5

The iOS ecosystem now includes a variety of devices and screen sizes. When developing a 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 Plus range of devices 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 9 and iOS 11 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*.

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. While 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 4-6:

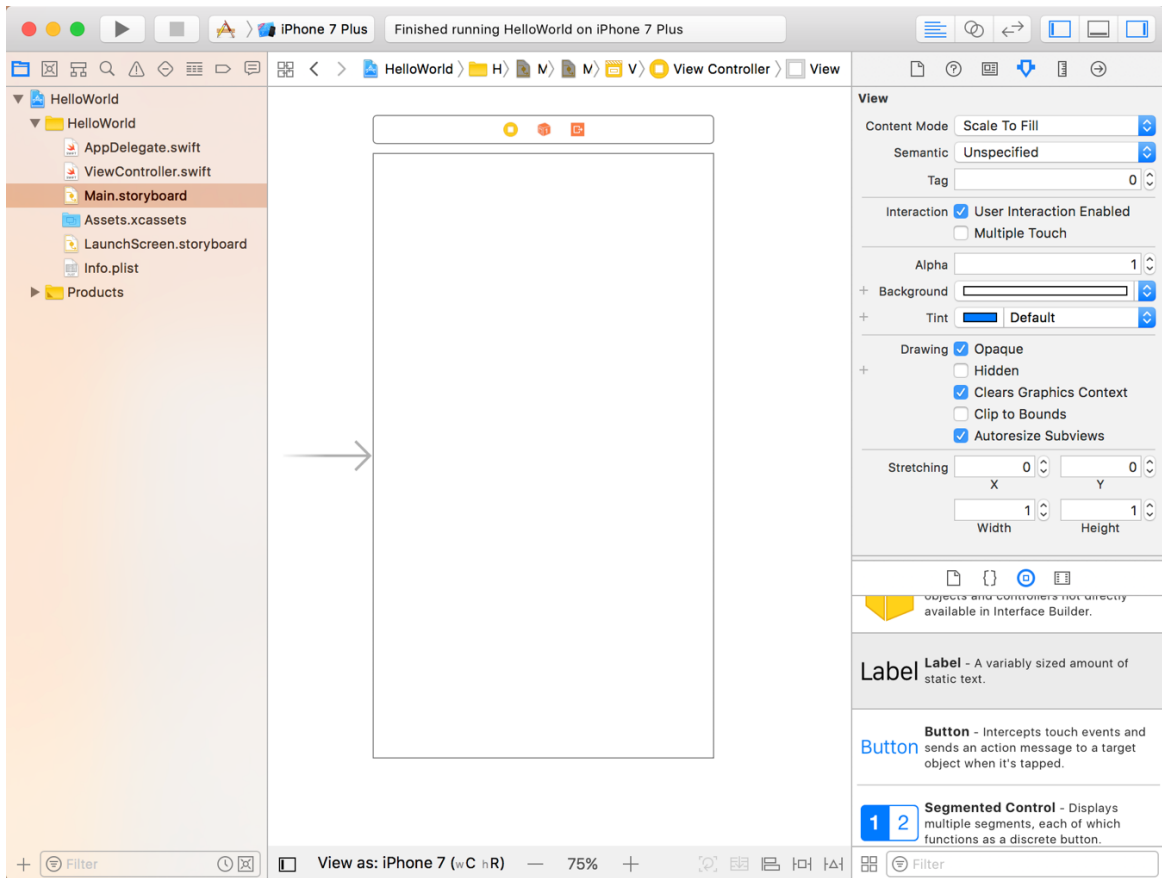


Figure 4-6

In the center panel a visual representation of the user interface of the application is displayed. Initially this consists solely of a *View Controller* (*UIViewController*) containing a single *View* (*UIView*) object. This layout 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 4-7

The Utilities panel, once displayed, will appear as illustrated in Figure 4-8:

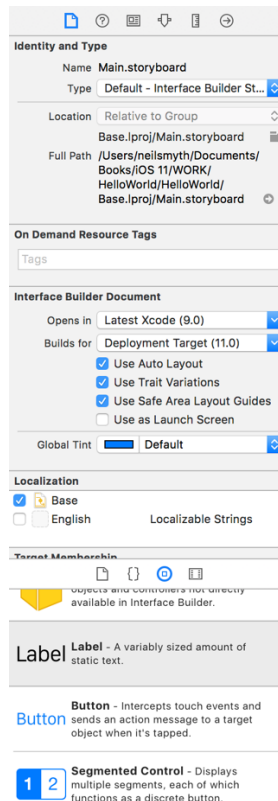


Figure 4-8

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 typically 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 4-9, enter *Label* into the search box beneath the panel:

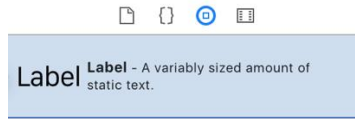


Figure 4-9

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.

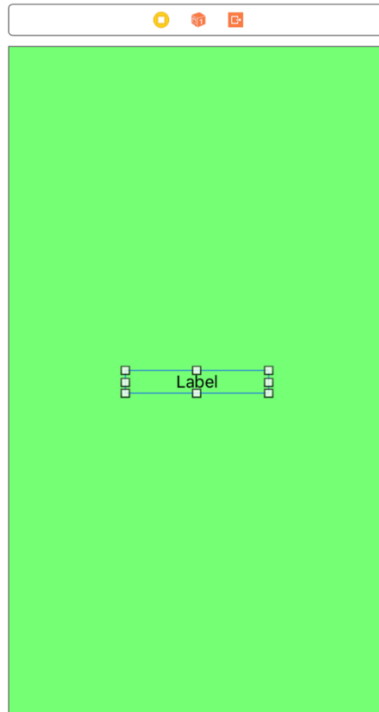


Figure 4-10

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 4-11:

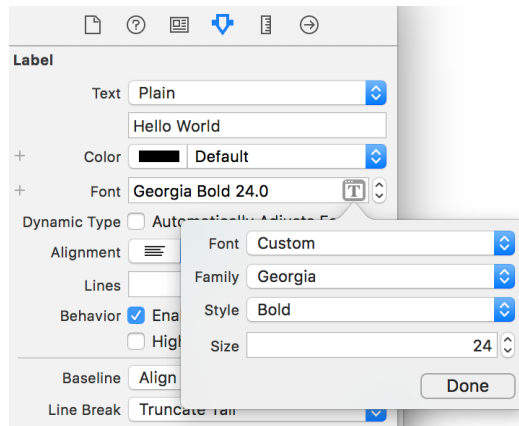


Figure 4-11

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 4-12 and enable both the *Horizontally in Container* and *Vertically in Container* options with offsets of 0 before clicking on the *Add 2 Constraints* button.

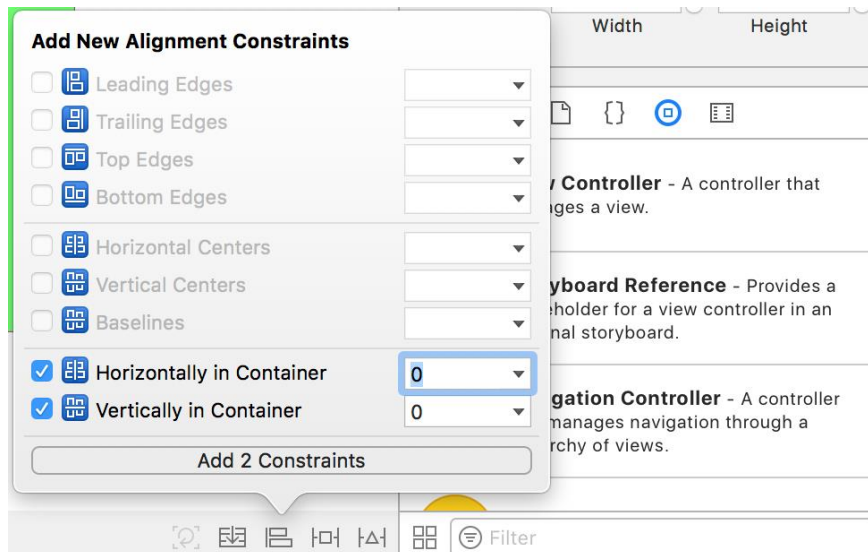


Figure 4-12

At this point, your View window will hopefully appear as outlined in Figure 4-13 (allowing, of course, for differences in your color and font choices).

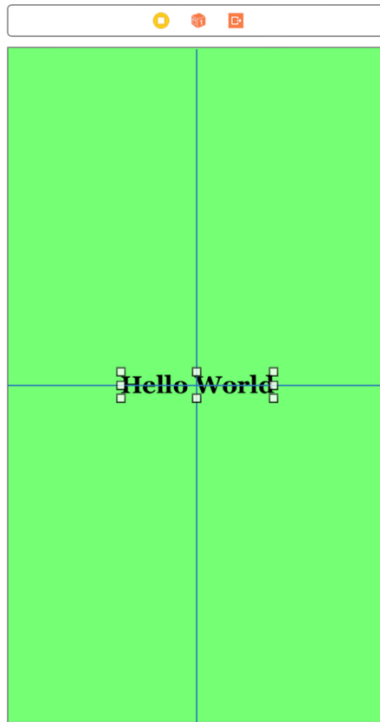


Figure 4-13

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 4-14) of the Interface Builder panel.



Figure 4-14

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.

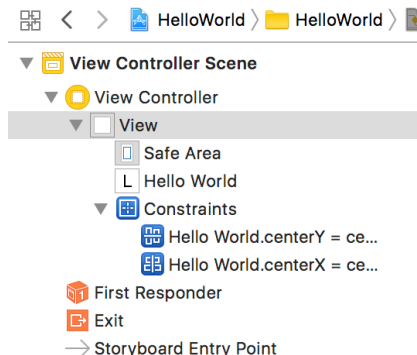


Figure 4-15

## 4.5 Building and Running an iOS 11 App in Xcode 9

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 project window, make sure that the menu located in the top left-hand corner of the window (marked C in Figure 4-16) has the *iPhone 7 Plus* simulator option selected:

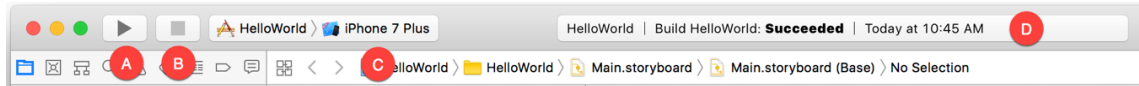


Figure 4-16

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 4-17

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 7 Plus 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.

If you have entered your Apple ID in the Xcode preferences screen as outlined in the previous chapter and selected a development team for the project, it is possible to run the app on a physical device simply by connecting it to the development Mac system with a USB cable 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. Switch to the physical device by selecting this menu and changing it to the device name as shown in Figure 4-18:

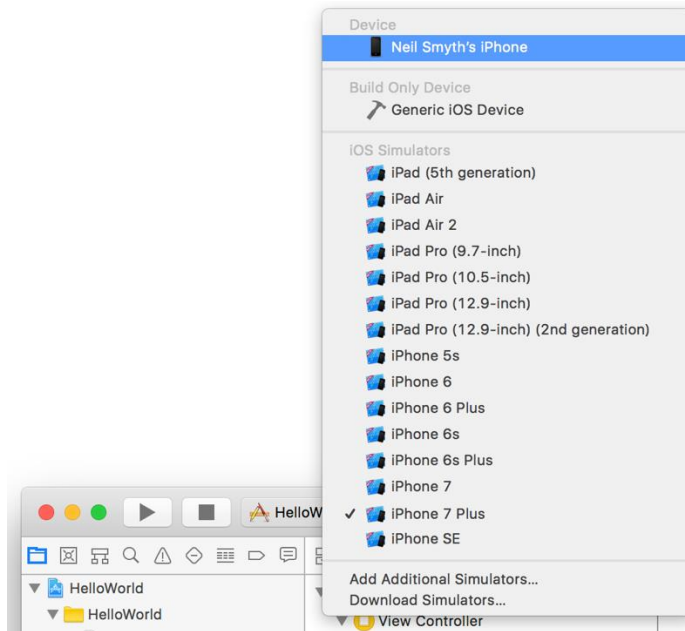


Figure 4-18

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. As will be discussed later in this chapter, a physical device may also be configured for network testing, whereby apps are installed and tested on the device via a network connection without the need to have the device connected by a USB cable.

## 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 and Simulators* menu option. Figure 4-19, for example, shows a typical Device screen on a system where an iPhone has been detected:



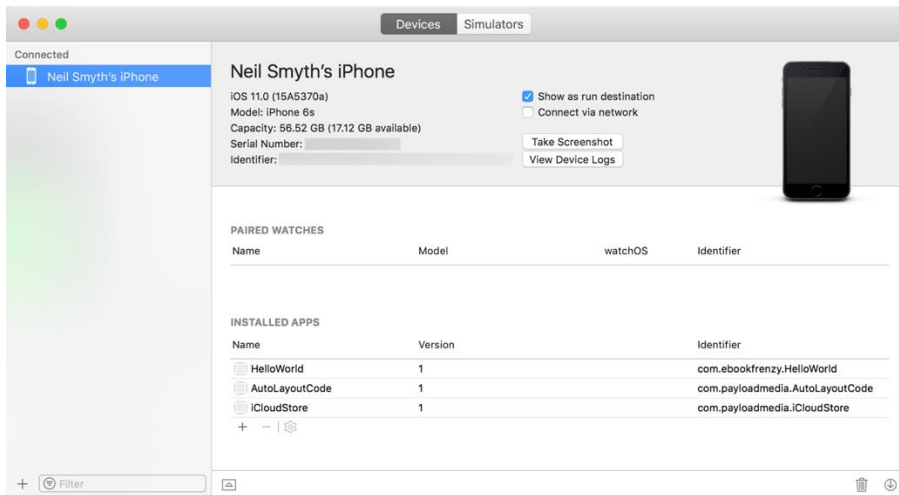


Figure 4-19

A wide range of simulator configurations are set up within Xcode by default and can be view by selecting the *Simulators* tab at the top of the dialog. 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 renamed or removed from the Xcode run target menu.

## 4.8 Enabling Network Testing

Earlier in this chapter, the example app was installed and run on a physical device connected to the development system via a USB cable. Xcode 9 also supports testing via a network connection. This option is enabled on a per device basis within the Devices and Simulators dialog introduced in the previous section. With the device connected via the USB cable, display this dialog, select the device from the list and enable the *Connect via network* option as highlighted in Figure 4-20:

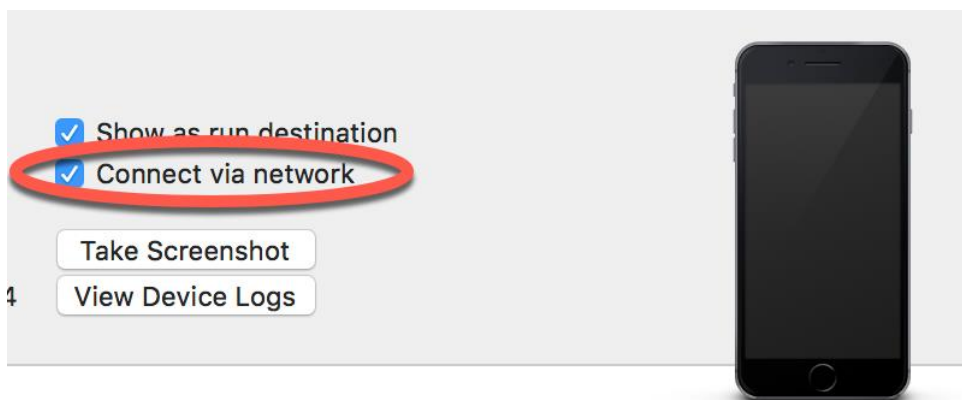


Figure 4-20

Once the setting has been enabled, the device may continue to be used as the run target for the app even when the USB cable is disconnected. The only requirement being that both the device and development computer be connected to the same Wi-Fi network. Assuming this requirement has been met, clicking on the

run button with the device selected in the run menu will install and launch the app over the network connection.

## 4.9 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.

## 4.10 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 4-21. 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, energy efficiency, network activity and iCloud storage access.

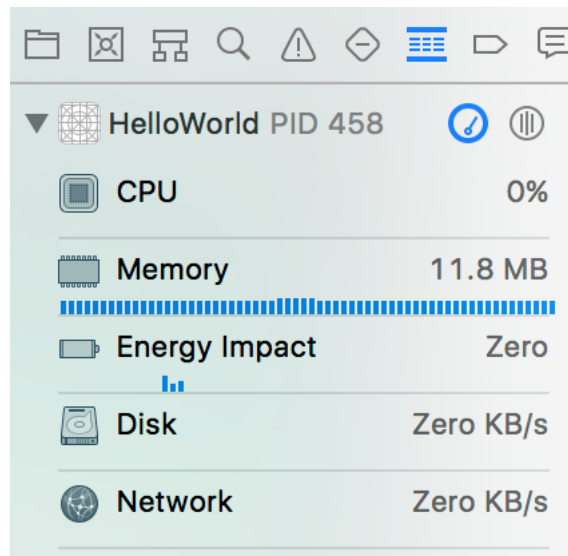


Figure 4-21

When one of these categories is selected, the main panel (Figure 4-22) updates to provide additional information about that particular aspect of the application’s performance:

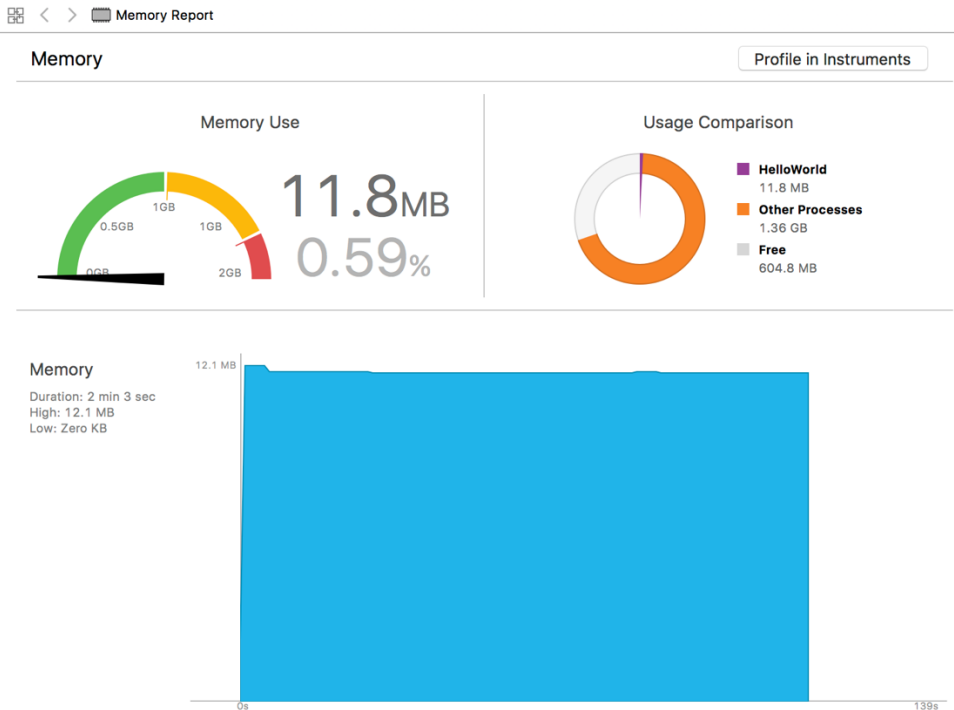


Figure 4-22

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

## 4.11 An Exploded View of the User Interface Layout Hierarchy

Xcode 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 4-23:

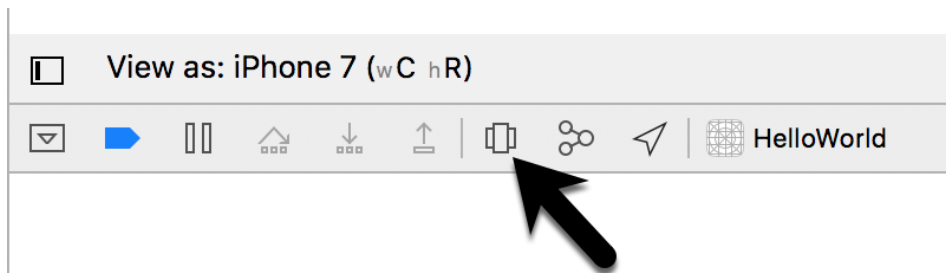


Figure 4-23

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 4-24 to switch to 3D mode:

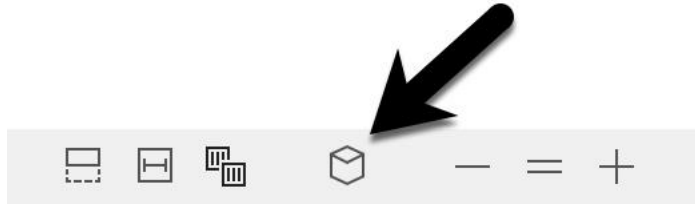


Figure 4-24

Figure 4-25 shows an example layout in this mode for a more complex user interface than that created in this chapter:

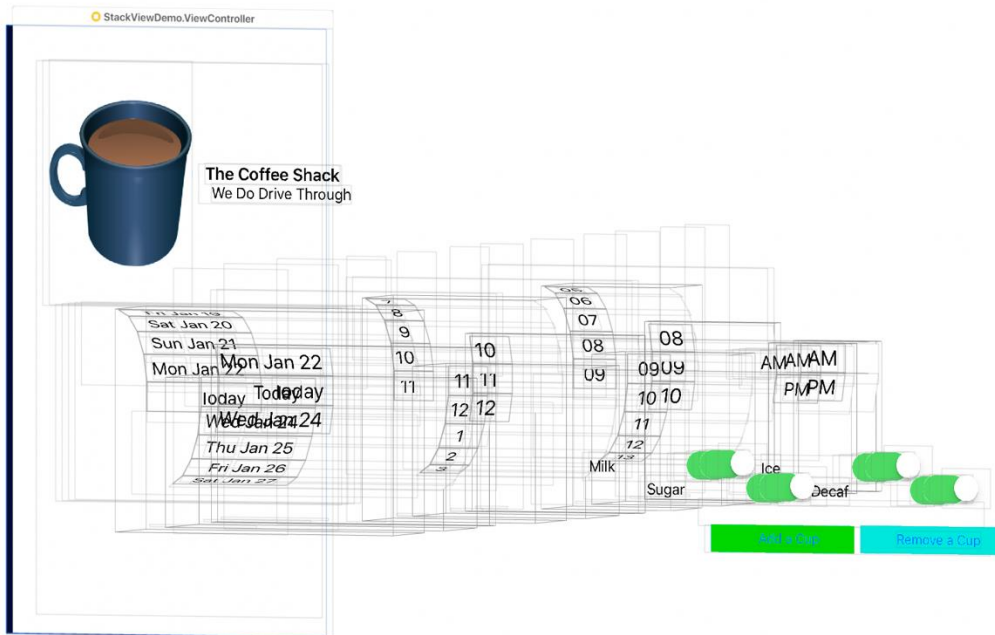


Figure 4-25

## 4.12 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 9. Many more features and capabilities of Xcode and Interface Builder will be covered in subsequent chapters of the book.

# 5. An Introduction to Xcode 9 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 of Xcode designed to make learning Swift and experimenting with the iOS 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. Choose the iOS option on the resulting panel and select the Blank template.

The Blank template is useful for trying out Swift coding. The Single View template, on the other hand, provides a view controller environment for trying out code that requires a user interface layout. The game and map templates provide preconfigured playgrounds that allow you to experiment with the iOS MapKit and SpriteKit frameworks respectively.

On the next screen, name the playground *LearnSwift* and choose a suitable file system location into which the playground should be saved before clicking on the *Create* button.

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

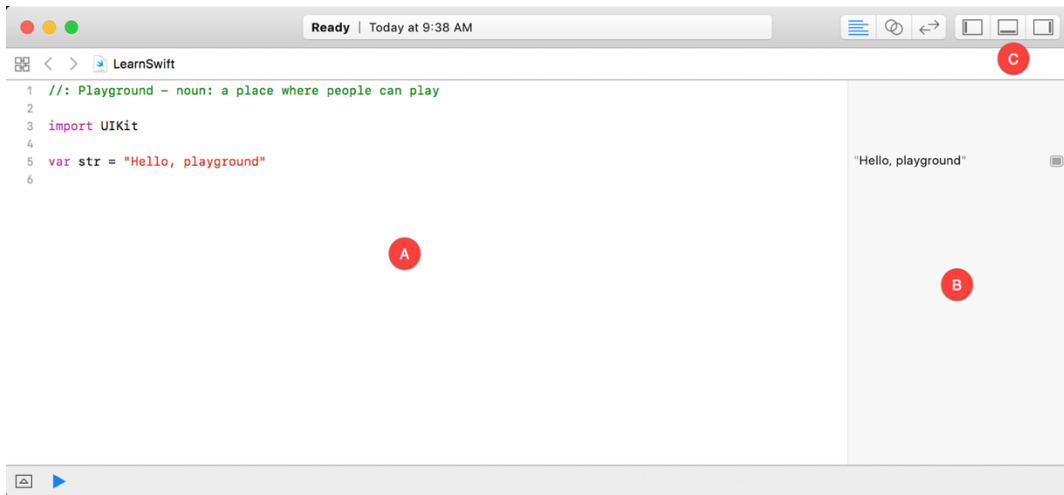


Figure 5-1

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.

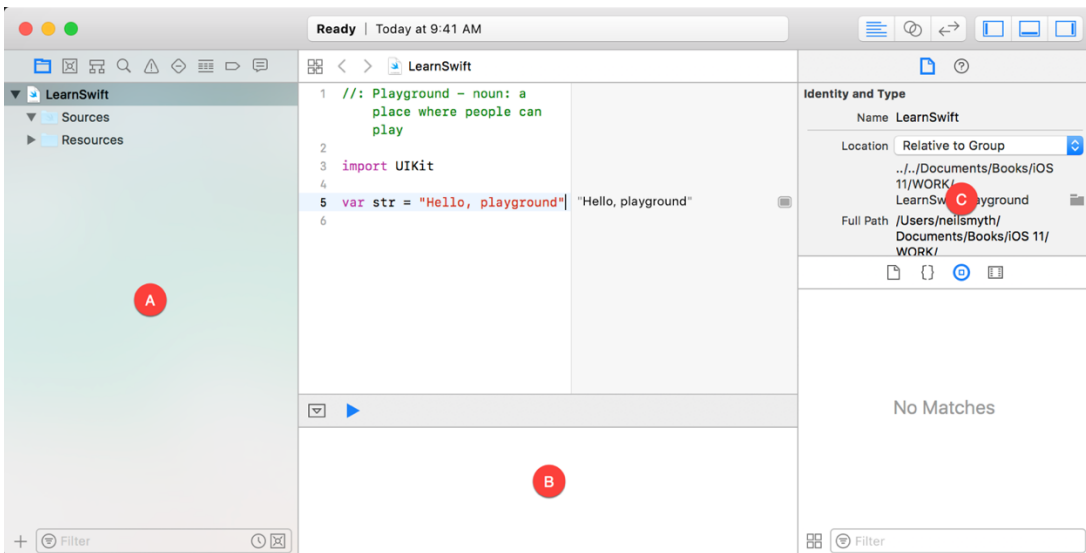


Figure 5-2

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. Note that the output also appears in the debug panel:

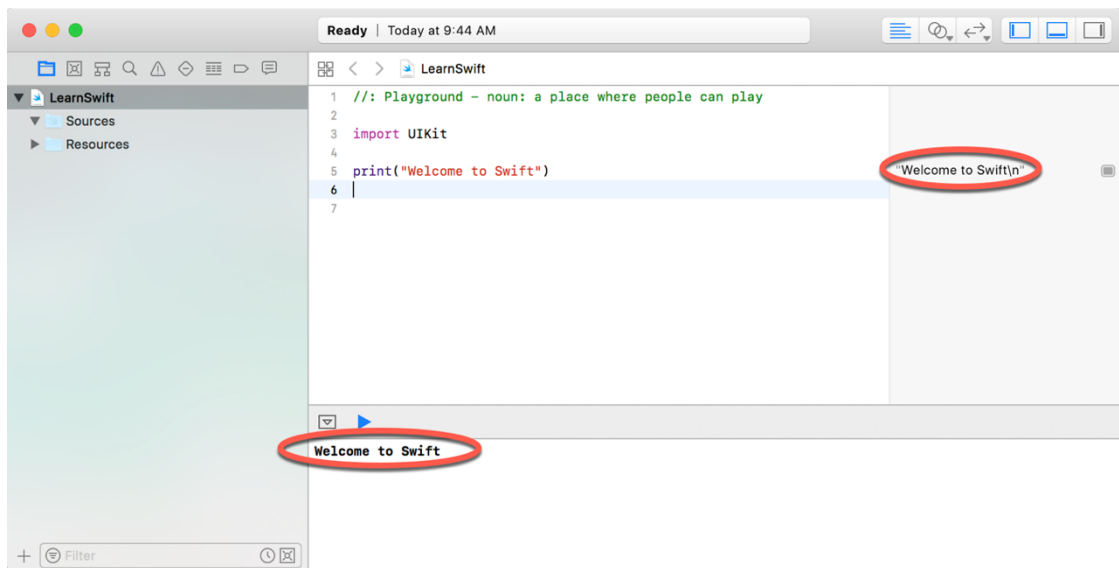


Figure 5-3

### 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
    x -= 1
    print(y)
}
```

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:

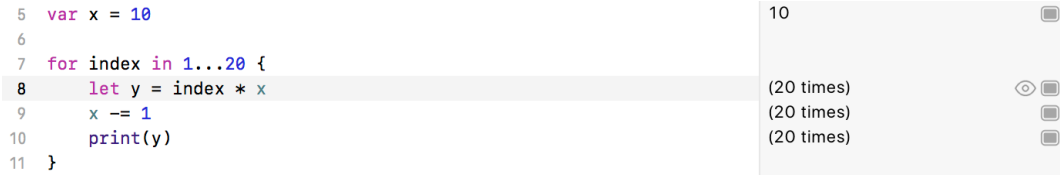


Figure 5-4

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:

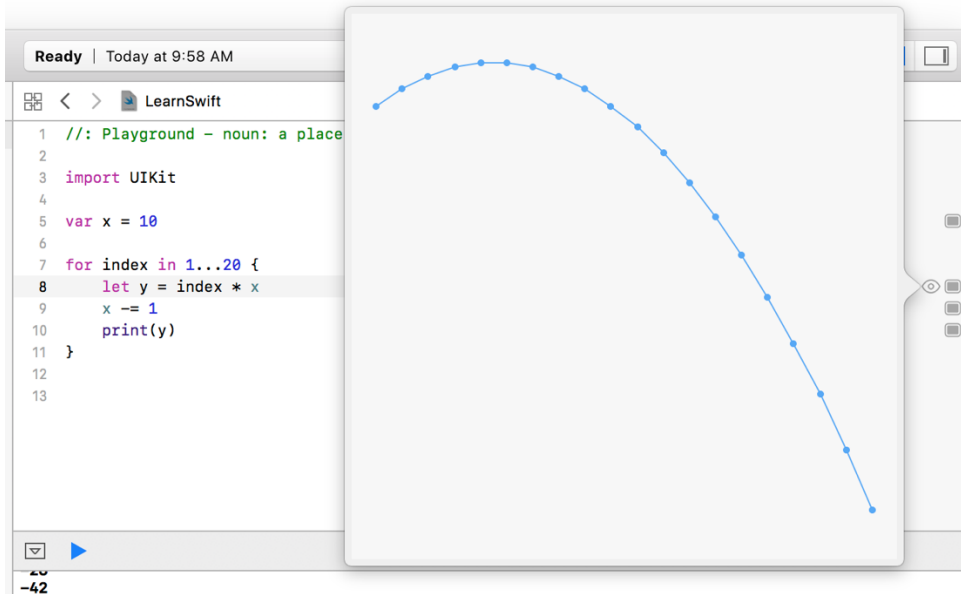


Figure 5-5

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

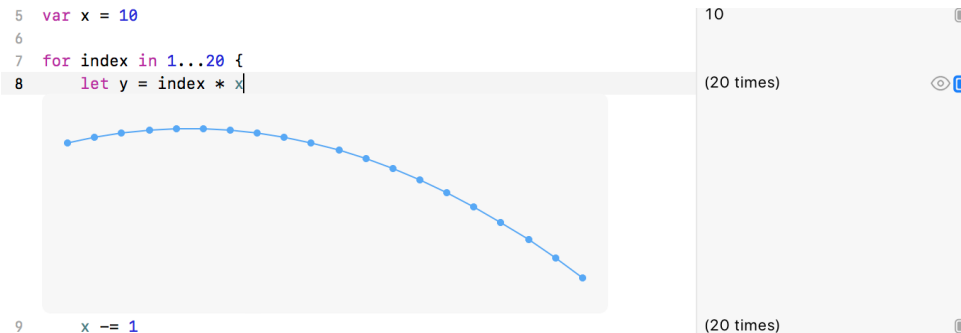


Figure 5-6



## 5.5 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, either select the *Editor -> Show Rendered Markup* menu option, or enable the *Render Documentation* option located under *Playground Settings* in the Utilities panel (marked C in Figure 5-2). Once rendered, the above rich text should appear as illustrated in Figure 5-7:

```
3 import UIKit
4
5 print("Welcome to Swift")
```

# 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

Figure 5-7

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

[https://developer.apple.com/library/content/documentation/Xcode/Reference/xcode\\_markup\\_formatting\\_ref/index.html](https://developer.apple.com/library/content/documentation/Xcode/Reference/xcode_markup_formatting_ref/index.html)

## 5.6 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 5-8:

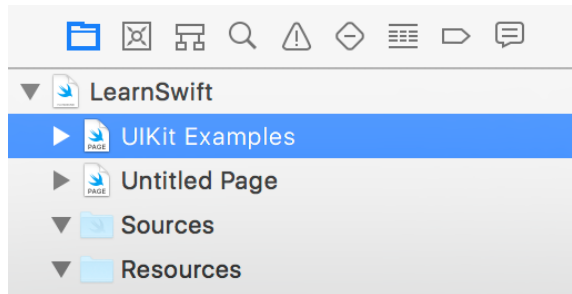


Figure 5-8

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

## 5.7 Working with UIKit in Playgrounds

The playground environment is not restricted to simple Swift code statements. Much of the power of the iOS 11 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 11 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

let myLabel = UILabel(frame: CGRect(x: 0, y: 0, width: 200, height: 50))

myLabel.backgroundColor = UIColor.red
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:

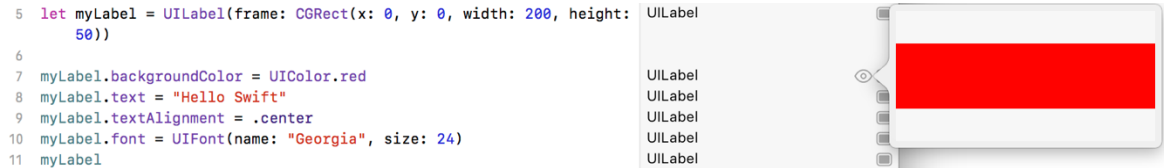


Figure 5-9

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:

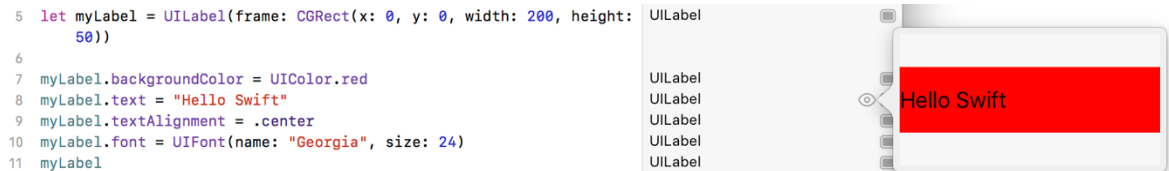


Figure 5-10

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

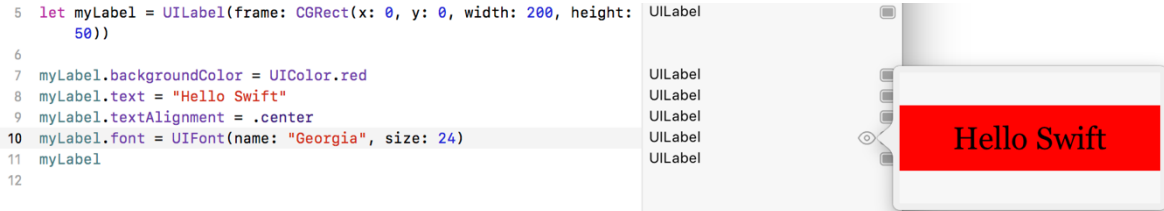


Figure 5-11

## 5.8 Adding Resources to a Playground

Another feature of playgrounds 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 (known as a *disclosure arrow*) to the left of the UIKit Examples page entry to unfold the page contents (Figure 5-12) and note the presence of a folder named *Resources*:

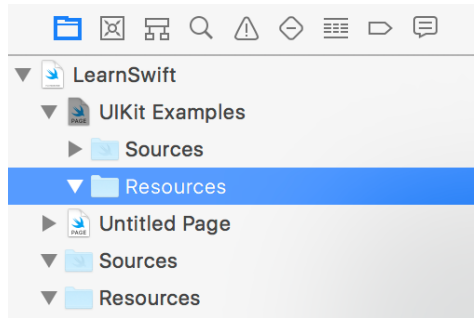


Figure 5-12

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

<http://www.ebookfrenzy.com/retail/ios11/>

Open a Finder window, navigate to the *playground\_images* 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:

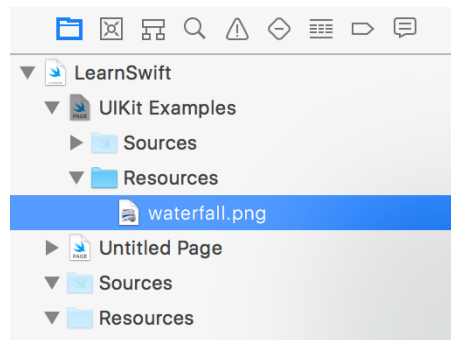


Figure 5-13

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 either the Quick Look or inline option to view the results of the code:



Figure 5-14

## 5.9 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 in-line features. It is, however, also possible to test dynamic user interface behavior within

a playground using the Xcode 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 PlaygroundSupport:

```
import UIKit
import PlaygroundSupport
```

The PlaygroundSupport 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 PlaygroundSupport

let container = UIView(frame: CGRect(x: 0,y: 0,width: 200,height: 200))
container.backgroundColor = UIColor.white
let square = UIView(frame: CGRect(x: 50,y: 50,width: 100,height: 100))
square.backgroundColor = UIColor.red

container.addSubview(square)

UIView.animate(withDuration: 5.0, animations: {
    square.backgroundColor = UIColor.blue
    let rotation = CGAffineTransform(rotationAngle: 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 PlaygroundSupport module includes a class named PlaygroundPage that allows playground code to interact with the pages that make up a playground. This is achieved through a range of methods and properties of the class, one of which is the *current* property. This property, in turn, provides access to the current playground page. In order to execute the live view within the playground timeline, the *liveView* property of the current page needs to be set to our new container. To display the timeline, click on the toolbar button containing the interlocking circles as highlighted in Figure 5-15:



Figure 5-15

When clicked, this button displays the Assistant Editor panel containing the timeline. Once the timeline is visible, add the code to assign the container to the live view of the current page as follows:

```
import UIKit
import PlaygroundSupport

let container = UIView(frame: CGRect(x: 0,y: 0,width: 200,height: 200))

PlaygroundPage.current.liveView = container

container.backgroundColor = UIColor.white
let square = UIView(frame: CGRect(x: 50,y: 50,width: 100,height: 100))
square.backgroundColor = UIColor.red

container.addSubview(square)

UIView.animate(withDuration: 5.0, animations: {
    square.backgroundColor = UIColor.blue
    let rotation = CGAffineTransform(rotationAngle: 3.14)
    square.transform = rotation
})
```

Once the call has been added, the views should appear in the timeline (Figure 5-16). During the 5 second animation duration, the red square should rotate through 360 degrees while gradually changing color to blue:

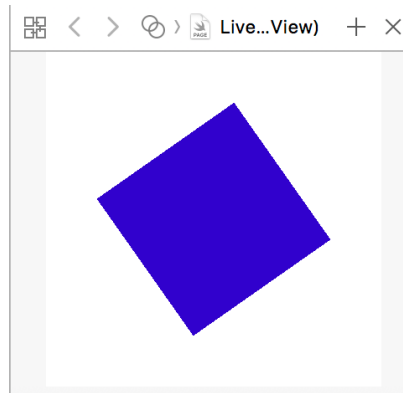


Figure 5-16

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.

## 5.10 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 11 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.

## 5.11 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 11 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 Swift programming language.

Due entirely to the popularity of iOS, Objective-C had become one of the more widely used programming languages. 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 iOS, macOS, watchOS and tvOS 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 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
```

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 scalars 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 in Swift are represented internally as collections of characters (where a character is, as previously discussed, comprised of one or more Unicode scalar values).

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.
```

A multiline string literal may be declared by encapsulating the string within triple quotes as follows:

```
var multiline = """

    The console glowed with flashing warnings.
    Clearly time was running out.
```

```
"I thought you said you knew how to fly this!" yelled Mary.  
  
"It was much easier on the simulator" replied her brother,  
trying not to sound scared.  
  
""  
  
print(multiline)
```

The above code will generate the following output when run:

```
The console glowed with flashing warnings.  
Clearly time was running out.  
  
"I thought you said you knew how to fly this!" yelled Mary.  
  
"It was much easier on the simulator" replied her brother,  
trying to keep the panic out of his voice.
```

The amount by which each line is indented within a multiline literal is calculated as the number of characters by which the line is indented minus the number of characters by which the closing triple quote line is indented. If, for example, the fourth line in the above example had a 10 character indentation and the closing triple quote was indented by 5 characters, the actual indentation of the fourth line within the string would be 5 characters. This allows multiline literals to be formatted tidily within Swift code while still allowing control over indentation of individual lines.

### 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 11 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 11 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 while 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. Second, 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. For this reason, there is no conflict in using the same temporary name as that assigned to the optional. The following is, for example, valid code:

```

.
.
if let index = index {
    print(treeArray[index])
} else {
.
.

```

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, let constname2 = optName2,
    let optName3 = ..., <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, let secondPet = pet2 {
    print(firstPet)
    print(secondPet)
} else {
    print("insufficient pets")
}

```

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

```

if let firstPet = pet1, let secondPet = pet2, petCount > 1 {
    print(firstPet)
    print(secondPet)
} else {
    print("insufficient 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 *object(forKey:)* method needs to be treated as a String type:

```
let myValue = record.object(forKey: "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:

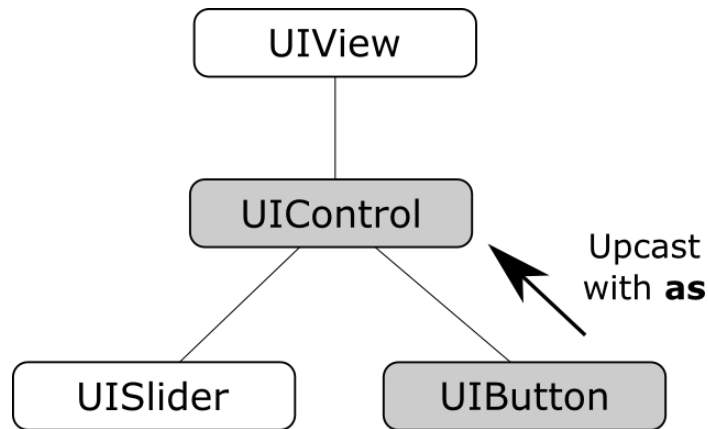


Figure 6-1

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:

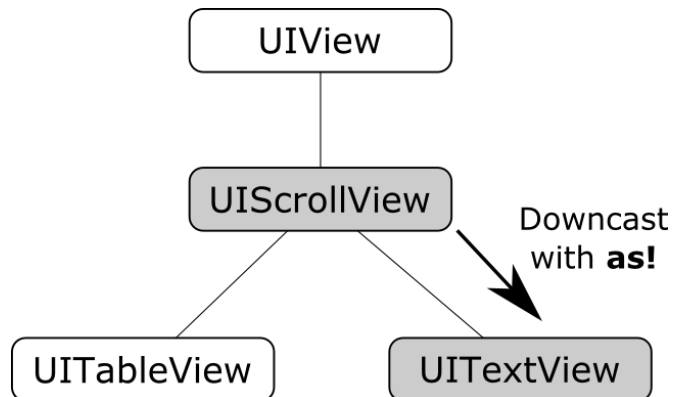


Figure 6-2

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 UITextView using the *guaranteed conversion* or *upcast* approach:

```
let myScrollView: UIScrollView = UIScrollView()
```

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

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 necessarily 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 could instead be forced using the *as!* annotation:

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

Now the code will compile without an error. As an example of the dangers of downcasting, however, the above code will crash on execution stating that UIScrollView cannot be cast to UITextView. Forced downcasting should, therefore, be used with caution.

A safer 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 myTextView = myScrollView 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
<b>x += y</b>	Add x to y and place result in x
<b>x -= y</b>	Subtract y from x and place result in x
<b>x *= y</b>	Multiply x by y and place result in x
<b>x /= y</b>	Divide x by y and place result in x
<b>x %= y</b>	Perform Modulo on x and y and place result in x

## 7.5 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
<b>x == y</b>	Returns true if x is equal to y
<b>x &gt; y</b>	Returns true if x is greater than y
<b>x &gt;= y</b>	Returns true if x is greater than or equal to y
<b>x &lt; y</b>	Returns true if x is less than y
<b>x &lt;= y</b>	Returns true if x is less than or equal to y
<b>x != y</b>	Returns true if x is not equal to y

## 7.6 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")  
}
```

### 7.7 Range Operators

Swift includes several 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-open 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.

Finally, the *one-sided range operator* specifies a range that can extend as far as possible in a specified range direction until the natural beginning or end of the range is reached (or until some other condition is met). A one-sided range is declared by omitting the number from one side of the range declaration, for example:

`x...`

or

`...y`

The previous chapter, for example, explained that a String in Swift is actually a collection of individual characters. A range to specify the characters in a string starting with the character at position 2 through to the last character in the string (regardless of string length) would be declared as follows:

`2...`

Similarly, to specify a range that begins with the first character and ends with the character at position 6, the range would be specified as follows:

`...6`

### 7.8 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
```

## 7.9 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. First, the decimal number 171 is represented in binary as:

```
10101011
```

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

```
0000011
```

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

### 7.9.1 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:

```
0000011 NOT
=====
1111100
```

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

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

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

### 7.9.2 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
0000011
```

```
=====  
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)")
```

### 7.9.3 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)")
```

### 7.9.4 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)")
```

### 7.9.5 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.

### 7.9.6 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.

## 7.10 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
<code>x &amp;= y</code>	Perform a bitwise AND of x and y and assign result to x

<b>x  = y</b>	Perform a bitwise OR of x and y and assign result to x
<b>x ^= y</b>	Perform a bitwise XOR of x and y and assign result to x
<b>x &lt;&lt;= n</b>	Shift x left by n places and assign result to x
<b>x &gt;&gt;= n</b>	Shift x right by n places and assign result to x

### 7.11 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.