

iOS 12 App Development

12

Essentials

iOS 12 App Development Essentials

iOS 12 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

1. Start Here	1
1.1 For New iOS Developers	1
1.2 For iOS 11 Developers.....	1
1.3 Source Code Download.....	2
1.4 Feedback.....	2
1.5 Errata	2
2. Joining the Apple Developer Program	3
2.1 Downloading Xcode 10 and the iOS 12 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
3. Installing Xcode 10 and the iOS 12 SDK	7
3.1 Identifying Your macOS Version	7
3.2 Installing Xcode 10 and the iOS 12 SDK	7
3.3 Starting Xcode.....	7
3.4 Adding Your Apple ID to the Xcode Preferences	8
3.5 Developer and Distribution Signing Identities	9
4. A Guided Tour of Xcode 10	11
4.1 Starting Xcode 10.....	11
4.2 Creating the iOS App User Interface	15
4.3 Changing Component Properties.....	18
4.4 Adding Objects to the User Interface	18
4.5 Building and Running an iOS 12 App in Xcode 10	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	23
4.10 Monitoring Application Performance	24
4.11 An Exploded View of the User Interface Layout Hierarchy.....	25
4.12 Summary.....	25
5. An Introduction to Xcode 10 Playgrounds	27
5.1 What is a Playground?	27
5.2 Creating a New Playground	27
5.3 A Basic Swift Playground Example	28
5.4 Viewing Results.....	30
5.5 Adding Rich Text Comments	32
5.6 Working with Playground Pages	33
5.7 Working with UIKit in Playgrounds	33
5.8 Adding Resources to a Playground	35
5.9 Working with Enhanced Live Views	36
5.10 When to Use Playgrounds	38
5.11 Summary.....	38

6. Swift Data Types, Constants and Variables	39
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/ <i>Escape Sequences</i>	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
7. Swift Operators and Expressions	51
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
8. Swift Flow Control	59
8.1 Looping Flow Control.....	59
8.2 The Swift for-in Statement	59
8.2.1 <i>The while Loop</i>	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
9. The Swift Switch Statement	65
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
10. An Overview of Swift 4 Functions, Methods and Closures.....	71
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
11. The Basics of Object-Oriented Programming in Swift	81
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
12. An Introduction to Swift Subclassing and Extensions	89
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
13. Working with Array and Dictionary Collections in Swift.....	95
13.1 Mutable and Immutable Collections	95
13.2 Swift Array Initialization.....	95
13.3 Working with Arrays in Swift	96
13.3.1 <i>Array Item Count</i>	96
13.3.2 <i>Accessing Array Items</i>	96
13.4 Random Items and Shuffling.....	97
13.5 Appending Items to an Array.....	97
13.5.1 <i>Inserting and Deleting Array Items</i>	97
13.6 Array Iteration	97
13.7 Creating Mixed Type Arrays.....	98
13.8 Swift Dictionary Collections	98
13.9 Swift Dictionary Initialization.....	99
13.10 Sequence-based Dictionary Initialization	99
13.11 Dictionary Item Count.....	100
13.12 Accessing and Updating Dictionary Items	100
13.13 Adding and Removing Dictionary Entries	100
13.14 Dictionary Iteration.....	101
13.15 Summary.....	101
14. Understanding Error Handling in Swift 4.....	103
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
15. The iOS 12 Application and Development Architecture	109
15.1 An Overview of the iOS 12 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
16. Creating an Interactive iOS 12 App	113
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
17. Understanding iOS 12 Views, Windows and the View Hierarchy	123
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 <i>The Window</i>	126
17.5.2 <i>Container Views</i>	126
17.5.3 <i>Controls</i>	126
17.5.4 <i>Display Views</i>	126
17.5.5 <i>Text and WebKit Views</i>	126
17.5.6 <i>Navigation Views and Tab Bars</i>	126
17.5.7 <i>Alert Views</i>	126
17.6 Summary	126
18. An Introduction to Auto Layout in iOS 12	127
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	129
18.8 Summary	130
19. Working with iOS 12 Auto Layout Constraints in Interface Builder	131
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	134
19.3.1 <i>Suggested Constraints</i>	135
19.3.2 <i>Visual Cues</i>	135
19.3.3 <i>Highlighting Constraint Problems</i>	136
19.3.4 <i>Viewing, Editing and Deleting Constraints</i>	139
19.4 Creating New Constraints in Interface Builder	141
19.5 Adding Aspect Ratio Constraints	141
19.6 Resolving Auto Layout Problems	142
19.7 Summary	143
20. An iOS 12 Auto Layout Example	145
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
21. Implementing iOS 12 Auto Layout Constraints in Code	151
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
22. Implementing Cross-Hierarchy Auto Layout Constraints in iOS 12	157
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
23. Understanding the iOS 12 Auto Layout Visual Format Language.....	161
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
24. Using Trait Variations to Design Adaptive iOS 12 User Interfaces	165
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
25. Using Storyboards in Xcode 10	175
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

26. Organizing Scenes over Multiple Storyboard Files	183
26.1 Organizing Scenes into Multiple Storyboards.....	183
26.2 Establishing a Connection between Different Storyboards	185
26.3 Summary.....	185
27. Using Xcode 10 Storyboards to Create an iOS 12 Tab Bar Application	187
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
28. An Overview of iOS 12 Table Views and Xcode 10 Storyboards	193
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
29. Using Xcode 10 Storyboards to Build Dynamic TableViews	201
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.....	207
29.9 Compiling and Running the Application	208
29.10 Handling TableView Swipe Gestures	208
29.11 Summary.....	209
30. Implementing iOS 12 TableView Navigation using Storyboards	211
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.....	213
30.6 Using prepare(for segue:) to Pass Data between Storyboard Scenes	215

30.7 Testing the Application	215
30.8 Customizing the Navigation Title Size	216
30.9 Summary.....	217
31. Integrating Search using the iOS UISearchController	219
31.1 Introducing the UISearchController Class	219
31.2 Adding a Search Controller to the TableViewStory 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
32. Working with the iOS 12 Stack View Class	225
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
33. An iOS 12 Stack View Tutorial	235
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
34. An iOS 12 Split View Master-Detail Example.....	245
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
35. A Guide to Multitasking in iOS 12	253
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
36. An iOS 12 Multitasking Example	261
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
37. Working with Directories in Swift on iOS 12	267
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
38. Working with Files in Swift on iOS 12	273
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/Deleteable	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
39. iOS 12 Directory Handling and File I/O in Swift – A Worked Example	279
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
40. Preparing an iOS 12 App to use iCloud Storage	283
40.1 iCloud Data Storage Services	283
40.2 Preparing an Application to Use iCloud Storage.....	284
40.3 Enabling iCloud Support for an iOS 12 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
41. Managing Files using the iOS 12 UIDocument Class	287
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
42. Using iCloud Storage in an iOS 12 Application	295
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
43. Using iCloud Drive Storage in an iOS 12 App	305
43.1 Preparing an App to use iCloud Drive Storage	305
43.2 Making Changes to the NSUbiquitousContainers Key	306
43.3 Creating the iCloud Drive Example Project	306
43.4 Modifying the Info.plist File	306
43.5 Designing the User Interface	307
43.6 Accessing the Ubiquitous Container	308
43.7 Saving the File to iCloud Drive	309
43.8 Testing the App	309
43.9 Summary	310
44. An Overview of the iOS 12 Document Browser View Controller	311
44.1 An Overview of the Document Browser View Controller	311
44.2 The Anatomy of a Document Based App	312
44.3 Document Browser Project Settings	313
44.4 The Document Browser Delegate Methods	313
44.4.1 <i>didRequestDocumentCreationWithHandler</i>	313
44.4.2 <i>didImportDocumentAt</i>	314
44.4.3 <i>didPickDocumentURLs</i>	314
44.4.4 <i>failedToImportDocumentAt</i>	315
44.5 Customizing the Document Browser	315
44.6 Adding Browser Actions	315
44.7 Summary	316
45. An iOS 12 Document Browser Tutorial	317
45.1 Creating the DocumentBrowser Project	317
45.2 Declaring the Supported File Types	318
45.3 Completing the <i>didRequestDocumentCreationWithHandler</i> Method	319
45.4 Finishing the UIDocument Subclass	321
45.5 Modifying the Document View Controller	321
45.6 Testing the Document Browser App	323
45.7 Summary	323
46. Synchronizing iOS 12 Key-Value Data using iCloud	325
46.1 An Overview of iCloud Key-Value Data Storage	325
46.2 Sharing Data Between Applications	326
46.3 Data Storage Restrictions	326
46.4 Conflict Resolution	326
46.5 Receiving Notification of Key-Value Changes	326
46.6 An iCloud Key-Value Data Storage Example	326
46.7 Enabling the Application for iCloud Key Value Data Storage	326

46.8 Designing the User Interface	327
46.9 Implementing the View Controller	328
46.10 Modifying the viewDidLoad Method	328
46.11 Implementing the Notification Method	329
46.12 Implementing the saveData Method.....	329
46.13 Testing the Application	329
46.14 Summary.....	330
47. iOS 12 Database Implementation using SQLite	331
47.1 What is SQLite?.....	331
47.2 Structured Query Language (SQL)	331
47.3 Trying SQLite on macOS.....	331
47.4 Preparing an iOS Application Project for SQLite Integration	333
47.5 SQLite, Swift and Wrappers.....	333
47.6 Key FMDB Classes	333
47.7 Creating and Opening a Database	334
47.8 Creating a Database Table	334
47.9 Extracting Data from a Database Table	334
47.10 Closing a SQLite Database	335
47.11 Summary.....	335
48. An Example SQLite based iOS 12 Application using Swift and FMDB	337
48.1 About the Example SQLite Application	337
48.2 Creating and Preparing the SQLite Application Project	337
48.3 Checking Out the FMDB Source Code	337
48.4 Designing the User Interface	339
48.5 Creating the Database and Table	340
48.6 Implementing the Code to Save Data to the SQLite Database	341
48.7 Implementing Code to Extract Data from the SQLite Database	342
48.8 Building and Running the Application	343
48.9 Summary.....	343
49. Working with iOS 12 Databases using Core Data	345
49.1 The Core Data Stack.....	345
49.2 Persistent Container	346
49.3 Managed Objects.....	346
49.4 Managed Object Context.....	346
49.5 Managed Object Model.....	346
49.6 Persistent Store Coordinator	347
49.7 Persistent Object Store.....	347
49.8 Defining an Entity Description	347
49.9 Initializing the Persistent Container.....	348
49.10 Obtaining the Managed Object Context.....	348
49.11 Getting an Entity Description	349
49.12 Setting the Attributes of a Managed Object.....	349
49.13 Saving a Managed Object	349
49.14 Fetching Managed Objects	349
49.15 Retrieving Managed Objects based on Criteria	350
49.16 Accessing the Data in a Retrieved Managed Object.....	350

49.17 Summary	350
50. An iOS 12 Core Data Tutorial	351
50.1 The Core Data Example Application	351
50.2 Creating a Core Data based Application	351
50.3 Creating the Entity Description.....	351
50.4 Designing the User Interface	352
50.5 Initializing the Persistent Container.....	353
50.6 Saving Data to the Persistent Store using Core Data	353
50.7 Retrieving Data from the Persistent Store using Core Data	354
50.8 Building and Running the Example Application	355
50.9 Summary	355
51. An Introduction to CloudKit Data Storage on iOS 12.....	357
51.1 An Overview of CloudKit	357
51.2 CloudKit Containers	357
51.3 CloudKit Public Database	357
51.4 CloudKit Private Databases.....	358
51.5 Data Storage and Transfer Quotas.....	358
51.6 CloudKit Records.....	358
51.7 CloudKit Record IDs	360
51.8 CloudKit References.....	360
51.9 CloudKit Assets	361
51.10 Record Zones	361
51.11 CloudKit Sharing.....	362
51.12 CloudKit Subscriptions	362
51.13 Obtaining iCloud User Information.....	362
51.14 CloudKit Dashboard	363
51.15 Summary	365
52. An Introduction to CloudKit Sharing	367
52.1 Understanding CloudKit Sharing	367
52.2 Preparing for CloudKit Sharing	367
52.3 The CKShare Class	367
52.4 The UICloudSharingController Class	368
52.5 Accepting a CloudKit Share	371
52.6 Fetching a Shared Record	371
52.7 Summary	372
53. An iOS 12 CloudKit Example	373
53.1 About the Example CloudKit Project.....	373
53.2 Creating the CloudKit Example Project	373
53.3 Designing the User Interface	374
53.4 Establishing Outlets and Actions.....	375
53.5 Implementing the notifyUser Method	375
53.6 Accessing the Private Database	376
53.7 Hiding the Keyboard	377
53.8 Implementing the selectPhoto method.....	377
53.9 Saving a Record to the Cloud Database	379

53.10 Testing the Record Saving Method	380
53.11 Searching for Cloud Database Records	383
53.12 Updating Cloud Database Records	384
53.13 Deleting a Cloud Record	385
53.14 Testing the Application	386
53.15 Summary	386
54. An iOS 12 CloudKit Subscription Example	387
54.1 Push Notifications and CloudKit Subscriptions	387
54.2 Configuring the Project for Remote Notifications	387
54.3 Registering an App to Receive Push Notifications	388
54.4 Configuring a CloudKit Subscription	389
54.5 Handling Remote Notifications	391
54.6 Implementing the <code>didReceiveRemoteNotification</code> Method	391
54.7 Fetching a Record From a Cloud Database	392
54.8 Completing the <code>didFinishLaunchingWithOptions</code> Method	393
54.9 Testing the Application	394
54.10 Summary	394
55. An iOS 12 CloudKit Sharing Example	395
55.1 Preparing the Project for CloudKit Sharing	395
55.2 Adding the Share Button	395
55.3 Creating the CloudKit Share	396
55.4 Accepting a CloudKit Share	397
55.5 Fetching the Shared Record	397
55.6 Testing the CloudKit Share Example	398
55.7 Summary	399
56. An Overview of iOS 12 Multitouch, Taps and Gestures	401
56.1 The Responder Chain	401
56.2 Forwarding an Event to the Next Responder	402
56.3 Gestures	402
56.4 Taps	402
56.5 Touches	402
56.6 Touch Notification Methods	402
56.6.1 <i>touchesBegan</i> method	402
56.6.2 <i>touchesMoved</i> method	403
56.6.3 <i>touchesEnded</i> method	403
56.6.4 <i>touchesCancelled</i> method	403
56.7 Touch Prediction	403
56.8 Touch Coalescing	403
56.9 3D Touch	404
56.10 Summary	404
57. An Example iOS 12 Touch, Multitouch and Tap Application	405
57.1 The Example iOS Tap and Touch Application	405
57.2 Creating the Example iOS Touch Project	405
57.3 Designing the User Interface	405
57.4 Enabling Multitouch on the View	406

57.5 Implementing the touchesBegan Method	406
57.6 Implementing the touchesMoved Method	407
57.7 Implementing the touchesEnded Method.....	407
57.8 Getting the Coordinates of a Touch.....	407
57.9 Building and Running the Touch Example Application	408
57.10 Checking for Touch Predictions	408
57.11 Accessing Coalesced Touches	409
57.12 Summary.....	409
58. Detecting iOS 12 Touch Screen Gesture Motions.....	411
58.1 The Example iOS 12 Gesture Application.....	411
58.2 Creating the Example Project	411
58.3 Designing the Application User Interface	411
58.4 Implementing the touchesBegan Method.....	412
58.5 Implementing the touchesMoved Method	412
58.6 Implementing the touchesEnded Method.....	413
58.7 Building and Running the Gesture Example	413
58.8 Summary.....	413
59. Identifying Gestures using iOS 12 Gesture Recognizers	415
59.1 The UIGestureRecognizer Class	415
59.2 Recognizer Action Messages.....	416
59.3 Discrete and Continuous Gestures	416
59.4 Obtaining Data from a Gesture.....	416
59.5 Recognizing Tap Gestures	416
59.6 Recognizing Pinch Gestures	416
59.7 Detecting Rotation Gestures.....	416
59.8 Recognizing Pan and Dragging Gestures.....	417
59.9 Recognizing Swipe Gestures	417
59.10 Recognizing Long Touch (Touch and Hold) Gestures.....	417
59.11 Summary.....	418
60. An iOS 12 Gesture Recognition Tutorial.....	419
60.1 Creating the Gesture Recognition Project	419
60.2 Designing the User Interface	419
60.3 Implementing the Action Methods.....	420
60.4 Testing the Gesture Recognition Application	421
60.5 Summary.....	421
61. Implementing Touch ID and Face ID Authentication in iOS 12 Apps	423
61.1 The Local Authentication Framework.....	423
61.2 Checking for Biometric Authentication Availability.....	423
61.3 Identifying Authentication Options	424
61.4 Evaluating Biometric Policy.....	424
61.5 A Biometric Authentication Example Project	425
61.6 Checking for Biometric Availability	426
61.7 Seeking Biometric Authentication	427
61.8 Adding the Face ID Privacy Statement.....	429
61.9 Testing the Application	429

61.10 Summary	431
62. Drawing iOS 12 2D Graphics with Core Graphics	433
62.1 Introducing Core Graphics and Quartz 2D	433
62.2 The draw Method	433
62.3 Points, Coordinates and Pixels.....	433
62.4 The Graphics Context	434
62.5 Working with Colors in Quartz 2D	434
62.6 Summary.....	435
63. Interface Builder Live Views and iOS 12 Embedded Frameworks	437
63.1 Embedded Frameworks.....	437
63.2 Interface Builder Live Views	437
63.3 Creating the Example Project	438
63.4 Adding an Embedded Framework	439
63.5 Implementing the Drawing Code in the Framework	440
63.6 Making the View Designable	441
63.7 Making VariablesInspectable	442
63.8 Summary.....	442
64. An iOS 12 Graphics Tutorial using Core Graphics and Core Image	443
64.1 The iOS Drawing Example Application.....	443
64.2 Creating the New Project.....	443
64.3 Creating the UIView Subclass	443
64.4 Locating the draw Method in the UIView Subclass	444
64.5 Drawing a Line	444
64.6 Drawing Paths.....	446
64.7 Drawing a Rectangle	447
64.8 Drawing an Ellipse or Circle	447
64.9 Filling a Path with a Color	448
64.10 Drawing an Arc	449
64.11 Drawing a Cubic Bézier Curve	450
64.12 Drawing a Quadratic Bézier Curve	450
64.13 Dashed Line Drawing	451
64.14 Drawing Shadows	452
64.15 Drawing Gradients	453
64.16 Drawing an Image into a Graphics Context	457
64.17 Image Filtering with the Core Image Framework	458
64.18 Summary.....	460
65. iOS 12 Animation using UIViewPropertyAnimator	461
65.1 The Basics of UIKit Animation	461
65.2 Understanding Animation Curves.....	462
65.3 Performing Affine Transformations	462
65.4 Combining Transformations	463
65.5 Creating the Animation Example Application	463
65.6 Implementing the Variables	463
65.7 Drawing in the UIView	464
65.8 Detecting Screen Touches and Performing the Animation	464

65.9 Building and Running the Animation Application	466
65.10 Implementing Spring Timing	467
65.11 Summary	467
66. iOS 12 UIKit Dynamics – An Overview	469
66.1 Understanding UIKit Dynamics	469
66.2 The UIKit Dynamics Architecture	469
<i>66.2.1 Dynamic Items</i>	469
<i>66.2.2 Dynamic Behaviors</i>	470
<i>66.2.3 The Reference View</i>	470
<i>66.2.4 The Dynamic Animator</i>	470
66.3 Implementing UIKit Dynamics in an iOS Application	471
66.4 Dynamic Animator Initialization	471
66.5 Configuring Gravity Behavior	471
66.6 Configuring Collision Behavior	472
66.7 Configuring Attachment Behavior	473
66.8 Configuring Snap Behavior	474
66.9 Configuring Push Behavior	474
66.10 The UIDynamicItemBehavior Class	475
66.11 Combining Behaviors to Create a Custom Behavior	476
66.12 Summary	476
67. An iOS 12 UIKit Dynamics Tutorial	477
67.1 Creating the UIKit Dynamics Example Project	477
67.2 Adding the Dynamic Items	477
67.3 Creating the Dynamic Animator Instance	478
67.4 Adding Gravity to the Views	479
67.5 Implementing Collision Behavior	480
67.6 Attaching a View to an Anchor Point	481
67.7 Implementing a Spring Attachment Between two Views	483
67.8 Summary	484
68. An Overview of iOS Collection View and Flow Layout	485
68.1 An Overview of Collection Views	485
68.2 The UICollectionView Class	486
68.3 The UICollectionViewCell Class	486
68.4 The UICollectionViewReusableView Class	487
68.5 The UICollectionViewFlowLayout Class	487
68.6 The UICollectionViewLayoutAttributes Class	487
68.7 The UICollectionViewDataSource Protocol	488
68.8 The UICollectionViewDelegate Protocol	488
68.9 The UICollectionViewDelegateFlowLayout Protocol	489
68.10 Cell and View Reuse	489
68.11 Summary	490
69. An iOS 12 Storyboard-based Collection View Tutorial	493
69.1 Creating the Collection View Example Project	493
69.2 Removing the Template View Controller	493
69.3 Adding a Collection View Controller to the Storyboard	493

69.4 Adding the Collection View Cell Class to the Project	494
69.5 Designing the Cell Prototype	495
69.6 Implementing the Data Model	495
69.7 Implementing the Data Source	497
69.8 Testing the Application	498
69.9 Setting Sizes for Cell Items	499
69.10 Changing Scroll Direction	499
69.11 Implementing a Supplementary View	500
69.12 Implementing the Supplementary View Protocol Methods	502
69.13 Summary	502
70. Subclassing and Extending the Collection View Flow Layout.....	503
70.1 About the Example Layout Class.....	503
70.2 Subclassing the UICollectionViewFlowLayout Class	503
70.3 Extending the New Layout Class.....	503
70.4 Overriding the layoutAttributesForItem(at indexPath:) Method	504
70.5 Overriding the layoutAttributesForElements(in rect:) Method.....	505
70.6 Implementing the modifyLayoutAttributes Method	505
70.7 Removing the Item Selection Code	506
70.8 Adding the New Layout and Pinch Gesture Recognizer	507
70.9 Implementing the Pinch Handler	507
70.10 Avoiding Image Clipping	509
70.11 Testing the Application	509
70.12 Summary	509
71. An Introduction to Drag and Drop in iOS 12.....	511
71.1 An Overview of Drag and Drop	511
71.2 Drag and Drop Delegates	511
71.3 Drag and Drop Interactions	511
71.4 The Drag Item	512
71.5 The Drag and Drop Lifecycle	512
71.6 Spring Loaded Controls.....	512
71.7 Summary	513
72. An iOS 12 Drag and Drop Tutorial	515
72.1 Creating the Drag and Drop Project	515
72.2 Designing the User Interface	515
72.3 Testing the Default Behavior	517
72.4 Adding Drop Support to the Image View	518
72.5 Testing the Drop Behavior	521
72.6 Adding Drag Support to the Views	522
72.7 Testing the Drag Behavior	523
72.8 Customizing the Lift Preview Image	523
72.9 Testing the Custom Preview Image	526
72.10 Implementing Animation	527
72.11 Summary	527
73. An iOS 12 Collection View Drag and Drop Tutorial.....	529
73.1 The Example Application	529

73.2 Declaring the Drag Delegate	529
73.3 Implementing Drag Support	530
73.4 Dragging Multiple Items	530
73.5 Adding the Drop Delegate	531
73.6 Implementing the Delegate Methods.....	532
73.7 Adding Drag and Drop Animation.....	534
73.8 Adding the Move Behavior	536
73.9 TableView Drag and Drop	537
73.10 Summary.....	537
74. Integrating Maps into iOS 12 Applications using MKMapItem.....	539
74.1 MKMapItem and MKPlacemark Classes	539
74.2 An Introduction to Forward and Reverse Geocoding	539
74.3 Creating MKPlacemark Instances	541
74.4 Working with MKMapItem	541
74.5 MKMapItem Options and Configuring Directions.....	542
74.6 Adding Item Details to an MKMapItem	543
74.7 Summary.....	544
75. An Example iOS 12 MKMapItem Application	545
75.1 Creating the MapItem Project	545
75.2 Designing the User Interface	545
75.3 Converting the Destination using Forward Geocoding.....	546
75.4 Launching the Map	547
75.5 Building and Running the Application.....	548
75.6 Summary.....	549
76. Getting Location Information using the iOS 12 Core Location Framework.....	551
76.1 The Core Location Manager.....	551
76.2 Requesting Location Access Authorization	551
76.3 Configuring the Desired Location Accuracy	552
76.4 Configuring the Distance Filter	552
76.5 Continuous Background Location Updates.....	553
76.6 The Location Manager Delegate	554
76.7 Starting and Stopping Location Updates	554
76.8 Obtaining Location Information from CLLocation Objects	555
76.8.1 <i>Longitude and Latitude</i>	555
76.8.2 <i>Accuracy</i>	555
76.8.3 <i>Altitude</i>	555
76.9 Getting the Current Location	555
76.10 Calculating Distances	556
76.11 Summary.....	556
77. An Example iOS 12 Location Application	557
77.1 Creating the Example iOS 12 Location Project	557
77.2 Designing the User Interface	557
77.3 Configuring the CLLocationManager Object.....	559
77.4 Setting up the Usage Description Keys	559
77.5 Implementing the startWhenInUse Method	560

77.6 Implementing the startAlways Method	560
77.7 Implementing the resetDistance Method	560
77.8 Implementing the Application Delegate Methods	560
77.9 Building and Running the Location Application	561
77.10 Adding Continuous Background Location Updates	563
77.11 Summary	564
78. Working with Maps on iOS 12 with MapKit and the MKMapView Class	565
78.1 About the MapKit Framework	565
78.2 Understanding Map Regions	565
78.3 Getting Transit ETA Information	565
78.4 About the MKMapView Tutorial	566
78.5 Creating the Map Project	566
78.6 Adding the Navigation Controller	567
78.7 Creating the MKMapView Instance and Toolbar	567
78.8 Obtaining Location Information Permission	570
78.9 Setting up the Usage Description Keys	570
78.10 Configuring the Map View	570
78.11 Changing the MapView Region	571
78.12 Changing the Map Type	571
78.13 Testing the MapView Application	572
78.14 Updating the Map View based on User Movement	572
78.15 Summary	573
79. Working with MapKit Local Search in iOS 12	575
79.1 An Overview of iOS Local Search	575
79.2 Adding Local Search to the MapSample Application	576
79.3 Adding the Local Search Text Field	576
79.4 Performing the Local Search	578
79.5 Testing the Application	579
79.6 Customized Annotation Markers	580
79.7 Annotation Marker Clustering	584
79.8 Summary	585
80. Using MKDirections to get iOS 12 Map Directions and Routes	587
80.1 An Overview of MKDirections	587
80.2 Adding Directions and Routes to the MapSample Application	589
80.3 Adding the New Classes to the Project	589
80.4 Configuring the Results Table View	589
80.5 Implementing the Result Table View Segue	591
80.6 Adding the Route Scene	592
80.7 Identifying the User's Current Location	593
80.8 Getting the Route and Directions	594
80.9 Establishing the Route Segue	596
80.10 Testing the Application	596
80.11 Summary	597
81. An iOS 12 MapKit Flyover Tutorial	599
81.1 MKMapView Flyover Map Types	599

81.2 The MKMapCamera Class	600
81.3 An MKMapKit Flyover Example	600
81.4 Designing the User Interface	601
81.5 Configuring the Map View and Camera	602
81.6 Animating Camera Changes	603
81.7 Testing the Map Flyover App	604
81.8 Summary	604
82. Accessing the iOS 12 Camera and Photo Library	605
82.1 The UIImagePickerController Class	605
82.2 Creating and Configuring a UIImagePickerController Instance	605
82.3 Configuring the UIImagePickerController Delegate	606
82.4 Detecting Device Capabilities	607
82.5 Saving Movies and Images	608
82.6 Summary	608
83. An Example iOS 12 Camera Application	609
83.1 An Overview of the Application	609
83.2 Creating the Camera Project	609
83.3 Designing the User Interface	609
83.4 Implementing the Action Methods	611
83.5 Writing the Delegate Methods	612
83.6 Seeking Camera and Photo Library Access	613
83.7 Building and Running the Application	614
83.8 Summary	615
84. iOS 12 Video Playback using AVPlayer and AVPlayerViewController	617
84.1 The AVPlayer and AVPlayerViewController Classes	617
84.2 The iOS Movie Player Example Application	617
84.3 Designing the User Interface	617
84.4 Initializing Video Playback	618
84.5 Build and Run the Application	619
84.6 Creating an AVPlayerViewController Instance from Code	619
84.7 Adding a Security Exception for an HTTP Connection	620
84.8 Summary	620
85. An iOS 12 Multitasking Picture in Picture Tutorial	621
85.1 An Overview of Picture in Picture Multitasking	621
85.2 Adding Picture in Picture Support to the AVPlayerDemo App	622
85.3 Adding the Navigation Controller	622
85.4 Setting the Audio Session Category	622
85.5 Implementing the Delegate	624
85.6 Opting Out of Picture in Picture Support	625
85.7 Additional Delegate Methods	626
85.8 Summary	626
86. An Introduction to Extensions in iOS 12	627
86.1 iOS Extensions – An Overview	627
86.2 Extension Types	627
86.2.1 Today Extension	628

86.2.2 Share Extension	628
86.2.3 Action Extension	629
86.2.4 Photo Editing Extension	630
86.2.5 Document Provider Extension	631
86.2.6 Custom Keyboard Extension	631
86.2.7 Audio Unit Extension	631
86.2.8 Shared Links Extension	631
86.2.9 Content Blocking Extension	631
86.2.10 Sticker Pack Extension	631
86.2.11 iMessage Extension	631
86.2.12 Intents Extension	631
86.3 Creating Extensions	632
86.4 Summary.....	632
87. An iOS 12 Today Extension Widget Tutorial	633
87.1 About the Example Extension Widget	633
87.2 Creating the Example Project	633
87.3 Adding the Extension to the Project.....	633
87.4 Reviewing the Extension Files.....	636
87.5 Designing the Widget User Interface.....	636
87.6 Setting the Preferred Content Size in Code	638
87.7 Modifying the Widget View Controller.....	638
87.8 Testing the Extension	640
87.9 Opening the Containing App from the Extension	640
87.10 Summary.....	642
88. Creating an iOS 12 Photo Editing Extension	643
88.1 Creating a Photo Editing Extension.....	643
88.2 Accessing the Photo Editing Extension	644
88.3 Configuring the Info.plist File.....	645
88.4 Designing the User Interface	646
88.5 The PHContentEditingController Protocol.....	647
88.6 Photo Extensions and Adjustment Data	648
88.7 Receiving the Content	648
88.8 Implementing the Filter Actions	649
88.9 Returning the Image to the Photos App	652
88.10 Testing the Application	654
88.11 Summary.....	654
89. Creating an iOS 12 Action Extension	655
89.1 An Overview of Action Extensions.....	655
89.2 About the Action Extension Example	656
89.3 Creating the Action Extension Project	656
89.4 Adding the Action Extension Target	656
89.5 Changing the Extension Display Name	657
89.6 Designing the Action Extension User Interface	657
89.7 Receiving the Content	658
89.8 Returning the Modified Data to the Host App	660
89.9 Testing the Extension	661

89.10 Declaring the Supported Content Type	663
89.11 Summary.....	664
90. Receiving Data from an iOS 12 Action Extension	665
90.1 Creating the Example Project	665
90.2 Designing the User Interface	665
90.3 Importing the Mobile Core Services Framework	666
90.4 Adding an Action Button to the Application	666
90.5 Receiving Data from an Extension	668
90.6 Testing the Application	669
90.7 Summary.....	669
91. An Introduction to Building iOS 12 Message Apps.....	671
91.1 Introducing Message Apps.....	671
91.2 Types of Message App	672
91.3 The Key Messages Framework Classes	672
91.3.1 <i>MSMessagesAppViewController</i>	673
91.3.2 <i>MSConversation</i>	673
91.3.3 <i>MSMessage</i>	674
91.3.4 <i>MSMessageTemplateLayout</i>	674
91.4 Sending Simple Messages	675
91.5 Creating an <i>MSMessage</i> Message	676
91.6 Receiving a Message	676
91.7 Supported Message App Platforms	677
91.8 Summary.....	677
92. An iOS 12 Interactive Message App Tutorial.....	679
92.1 About the Example Message App Project.....	679
92.2 Creating the <i>MessageApp</i> Project	679
92.3 Designing the <i>MessageApp</i> User Interface	681
92.4 Creating the Outlet Collection	684
92.5 Creating the Game Model.....	685
92.6 Responding to Button Selections	685
92.7 Preparing the Message URL.....	686
92.8 Preparing and Inserting the Message	687
92.9 Message Receipt Handling.....	688
92.10 Setting the Message Image.....	690
92.11 Implementing a Session	691
92.12 Summary.....	692
93. An Introduction to SiriKit.....	693
93.1 Siri and SiriKit	693
93.2 SiriKit Domains.....	693
93.3 Siri Shortcuts	694
93.4 SiriKit Intents.....	694
93.5 How SiriKit Integration Works	694
93.6 Resolving Intent Parameters.....	695
93.7 The Confirm Method	696
93.8 The Handle Method	696

93.9 Custom Vocabulary.....	697
93.10 The Siri User Interface	697
93.11 Summary.....	697
94. An iOS 12 Example SiriKit Messaging Extension	699
94.1 Creating the Example Project	699
94.2 Enabling the Siri Entitlement	699
94.3 Seeking Siri Authorization.....	699
94.4 Adding the Extensions	700
94.5 Supported Intents	701
94.6 Using the Default User Interface	701
94.7 Trying the Example	702
94.8 Specifying a Default Phrase	702
94.9 Reviewing the Intent Handler	703
94.10 Summary.....	704
95. Customizing the SiriKit Intent User Interface	705
95.1 Modifying the UI Extension	705
95.2 Using the configure Method.....	705
95.3 Designing the Siri Snippet Scene	706
95.4 Adding the configure Method	707
95.5 Overriding Siri Content	708
95.6 Using the configureView Method.....	709
95.7 Implementing a configureView Custom UI.....	710
95.8 Summary.....	714
96. An iOS 12 SiriKit Photo Search Tutorial	715
96.1 About the SiriKit Photo Search Project	715
96.2 Creating the SiriPhoto Project	715
96.3 Enabling the Siri Entitlement	715
96.4 Obtaining Siri Authorization	715
96.5 Designing the App User Interface	717
96.6 Adding the Intents Extension to the Project.....	717
96.7 Reviewing the Default Intents Extension.....	718
96.8 Modifying the Supported Intents	718
96.9 Modifying the IntentHandler Implementation	719
96.10 Implementing the Resolve Methods	719
96.11 Implementing the Confirmation Method	721
96.12 Handling the Intent.....	721
96.13 Testing the App	723
96.14 Handling the NSUserActivity Object	723
96.15 Testing the Completed App	724
96.16 Summary.....	724
97. An Overview of Siri Shortcut App Integration	725
97.1 An Overview of Siri Shortcuts	725
97.2 An Introduction to the Intent Definition File	726
97.3 Automatically Generated Classes	727
97.4 Donating Shortcuts	728

97.5 The Add to Siri Button.....	728
97.6 Summary.....	729
98. Building Siri Shortcut Support into an iOS App	731
98.1 About the Example App	731
98.2 App Groups and UserDefaults	731
98.3 Preparing the Project.....	731
98.4 Enabling Siri Support.....	733
98.5 Adding the Intents Extension.....	733
98.6 Adding the SiriKit Intent Definition File	734
98.7 Configuring the SiriKit Intent Definition File	735
98.8 Adding Intent Parameters.....	736
98.9 Declaring Shortcut Types	736
98.10 Configuring the Intent Response	737
98.11 Configuring Target Membership.....	738
98.12 Modifying the Intent Handler Code	738
98.13 Adding the Confirm Method.....	740
98.14 Donating Shortcuts to Siri	740
98.15 Testing the Shortcuts	741
98.16 Designing the Intent UI	743
98.17 Passing Control to the Host App	744
98.18 Adding an Add to Siri Button	746
98.19 Testing the Add to Siri Button.....	748
98.20 Summary.....	750
99. An Introduction to Machine Learning on iOS.....	751
99.1 Datasets and Machine Learning Models.....	751
99.2 Machine Learning in Xcode and iOS	751
99.3 iOS Machine Learning Frameworks	752
99.4 Summary.....	752
100. Using Create ML to Build an Image Classification Model	753
100.1 About the Dataset.....	753
100.2 Creating the Machine Learning Model	753
100.3 Summary.....	756
101. An iOS Vision and Core ML Image Classification Tutorial	757
101.1 Preparing the Project	757
101.2 Adding the Model	757
101.3 Modifying the User Interface	757
101.4 Initializing the Core ML Request	758
101.5 Handling the Results of the Core ML Request	759
101.6 Making the Classification Request	760
101.7 Testing the App	761
101.8 Summary.....	762
102. A 3D Touch Force Handling Tutorial.....	763
102.1 Creating the 3D Touch Example Project	763
102.2 Adding the UIView Subclass to the Project	763
102.3 Locating the draw Method in the UIView Subclass	763

102.4 Implementing the Touch Methods	764
102.5 Testing the Touch Force App	765
102.6 Summary.....	766
103. An iOS 12 3D Touch Quick Actions Tutorial	767
103.1 Creating the Quick Actions Example Project	767
103.2 Static Quick Action Keys	767
103.3 Adding a Static Quick Action to the Project.....	767
103.4 Adding a Dynamic Quick Action.....	769
103.5 Adding, Removing and Changing Dynamic Quick Actions	770
103.6 Responding to a Quick Action Selection	770
103.7 Testing the Quick Action App	771
103.8 Summary.....	772
104. An iOS 12 3D Touch Peek and Pop Tutorial	773
104.1 About the Example Project	773
104.2 Adding the UIViewControllerPreviewDelegate	773
104.3 Implementing the Peek Delegate Method	774
104.4 Assigning the Detail Controller Storyboard ID.....	775
104.5 Implementing the Pop Delegate Method	776
104.6 Registering the Previewing Delegate.....	776
104.7 Testing the Peek and Pop Behavior	777
104.8 Adding Peek Quick Actions	778
104.9 Summary.....	780
105. An iOS 12 Local Notification Tutorial.....	781
105.1 Creating the Local Notification App Project	781
105.2 Requesting Notification Authorization	781
105.3 Designing the User Interface	782
105.4 Creating the Message Content	783
105.5 Specifying a Notification Trigger.....	784
105.6 Creating the Notification Request	784
105.7 Adding the Request	784
105.8 Testing the Notification	785
105.9 Receiving Notifications in the Foreground	785
105.10 Adding Notification Actions	786
105.11 Handling Notification Actions	788
105.12 Hidden Notification Content.....	788
105.13 Managing Notifications.....	791
105.14 Summary.....	791
106. Playing Audio on iOS 12 using AVAudioPlayer	793
106.1 Supported Audio Formats.....	793
106.2 Receiving Playback Notifications	793
106.3 Controlling and Monitoring Playback	794
106.4 Creating the Audio Example Application	794
106.5 Adding an Audio File to the Project Resources.....	794
106.6 Designing the User Interface	794
106.7 Implementing the Action Methods.....	795

106.8 Creating and Initializing the AVAudioPlayer Object	796
106.9 Implementing the AVAudioPlayerDelegate Protocol Methods	797
106.10 Building and Running the Application.....	797
106.11 Summary.....	797
107. Recording Audio on iOS 12 with AVAudioRecorder	799
107.1 An Overview of the AVAudioRecorder Tutorial	799
107.2 Creating the Recorder Project	799
107.3 Configuring the Microphone Usage Description	799
107.4 Designing the User Interface	800
107.5 Creating the AVAudioRecorder Instance	801
107.6 Implementing the Action Methods.....	802
107.7 Implementing the Delegate Methods.....	803
107.8 Testing the Application	804
107.9 Summary.....	804
108. An iOS 12 Speech Recognition Tutorial	805
108.1 An Overview of Speech Recognition in iOS.....	805
108.2 Speech Recognition Authorization	806
108.3 Transcribing Recorded Audio.....	806
108.4 Transcribing Live Audio	806
108.5 An Audio File Speech Recognition Tutorial	806
108.6 Modifying the User Interface	806
108.7 Adding the Speech Recognition Permission	807
108.8 Seeking Speech Recognition Authorization	808
108.9 Performing the Transcription.....	809
108.10 Testing the App	810
108.11 Summary.....	810
109. An iOS 12 Real-Time Speech Recognition Tutorial	811
109.1 Creating the Project	811
109.2 Designing the User Interface	811
109.3 Adding the Speech Recognition Permission	812
109.4 Requesting Speech Recognition Authorization.....	812
109.5 Declaring and Initializing the Speech and Audio Objects	813
109.6 Starting the Transcription	814
109.7 Implementing the stopTranscribing Method.....	817
109.8 Testing the App	818
109.9 Summary.....	818
110. iOS 12 Multitasking, Background Transfer Service and Fetching.....	819
110.1 Understanding iOS Application States	819
110.2 A Brief Overview of the Multitasking Application Lifecycle.....	820
110.3 Checking for Multitasking Support	820
110.4 Enabling Multitasking for an iOS Application	821
110.5 Supported Forms of Background Execution	821
110.6 An Overview of Background Fetch.....	822
110.7 An Overview of Remote Notifications	823
110.8 An Overview of Local Notifications.....	824

110.9 An Overview of Background Transfer Service	824
110.10 The Rules of Background Execution.....	824
110.11 Summary.....	824
111. An Overview of iOS 12 Application State Preservation and Restoration	825
111.1 The Preservation and Restoration Process	825
111.2 Opting In to Preservation and Restoration	826
111.3 Assigning Restoration Identifiers	826
111.4 Default Preservation Features of UIKit	827
111.5 Saving and Restoring Additional State Information.....	827
111.6 Understanding the Restoration Process	828
111.7 Saving General Application State.....	829
111.8 Summary.....	829
112. An iOS 12 State Preservation and Restoration Tutorial	831
112.1 Creating the Example Application	831
112.2 Trying the Application without State Preservation	831
112.3 Opting-in to State Preservation	831
112.4 Setting Restoration Identifiers.....	832
112.5 Encoding and Decoding View Controller State	832
112.6 Adding a Navigation Controller to the Storyboard	834
112.7 Adding the Third View Controller	835
112.8 Creating the Restoration Class.....	836
112.9 Summary.....	837
113. An Introduction to iOS 12 Sprite Kit Programming.....	839
113.1 What is Sprite Kit?	839
113.2 The Key Components of a Sprite Kit Game.....	839
113.2.1 <i>Sprite Kit View</i>	839
113.2.2 <i>Scenes</i>	840
113.2.3 <i>Nodes</i>	840
113.2.4 <i>Physics Bodies</i>	840
113.2.5 <i>Physics World</i>	841
113.2.6 <i>Actions</i>	841
113.2.7 <i>Transitions</i>	841
113.2.8 <i>Texture Atlas</i>	841
113.2.9 <i>Constraints</i>	841
113.3 An Example Sprite Kit Game Hierarchy.....	842
113.4 The Sprite Kit Game Rendering Loop.....	842
113.5 The Sprite Kit Level Editor.....	843
113.6 Summary.....	843
114. An iOS 12 Sprite Kit Level Editor Game Tutorial	845
114.1 About the Sprite Kit Demo Game	845
114.2 Creating the SpriteKitDemo Project	846
114.3 Reviewing the SpriteKit Game Template Project.....	846
114.4 Restricting Interface Orientation	847
114.5 Modifying the GameScene SpriteKit Scene File.....	847
114.6 Creating the Archery Scene	849

114.7 Transitioning to the Archery Scene	850
114.8 Adding the Texture Atlas	851
114.9 Designing the Archery Scene	852
114.10 Preparing the Archery Scene	854
114.11 Preparing the Animation Texture Atlas	854
114.12 Creating the Named Action Reference	856
114.13 Testing Actions in an Action File	856
114.14 Triggering the Named Action from the Code.....	857
114.15 Creating the Arrow Sprite Node	857
114.16 Shooting the Arrow.....	858
114.17 Adding the Ball Sprite Node.....	859
114.18 Summary	860
115. An iOS 12 Sprite Kit Collision Handling Tutorial	861
115.1 Defining the Category Bit Masks.....	861
115.2 Assigning the Category Masks to the Sprite Nodes	861
115.3 Configuring the Collision and Contact Masks	862
115.4 Implementing the Contact Delegate.....	863
115.5 Game Over	864
115.6 Summary	866
116. An iOS 12 Sprite Kit Particle Emitter Tutorial	867
116.1 What is the Particle Emitter?	867
116.2 The Particle Emitter Editor.....	867
116.3 The SKEMitterNode Class.....	867
116.4 Using the Particle Emitter Editor	868
116.5 Particle Emitter Node Properties	869
116.5.1 <i>Background</i>	869
116.5.2 <i>Particle Texture</i>	869
116.5.3 <i>Particle Birthrate</i>	869
116.5.4 <i>Particle Life Cycle</i>	870
116.5.5 <i>Particle Position Range</i>	870
116.5.6 <i>Angle</i>	870
116.5.7 <i>Particle Speed</i>	870
116.5.8 <i>Particle Acceleration</i>	870
116.5.9 <i>Particle Scale</i>	870
116.5.10 <i>Particle Rotation</i>	870
116.5.11 <i>Particle Color</i>	870
116.5.12 <i>Particle Blend Mode</i>	871
116.6 Experimenting with the Particle Emitter Editor	871
116.7 Bursting a Ball using Particle Emitter Effects	872
116.8 Adding the Burst Particle Emitter Effect	873
116.9 Adding an Audio Action	874
116.10 Summary	875
117. Making Store Purchases with the SKStoreProductViewController Class	877
117.1 The SKStoreProductViewController Class	877
117.2 Creating the Example Project	878
117.3 Creating the User Interface.....	878

117.4 Displaying the Store Kit Product View Controller	878
117.5 Implementing the Delegate Method	880
117.6 Testing the Application	880
117.7 Summary.....	880
118. Building In-App Purchasing into iOS 12 Applications.....	881
118.1 In-App Purchase Options	881
118.2 Uploading App Store Hosted Content	881
118.3 Configuring In-App Purchase Items	882
118.4 Sending a Product Request.....	882
118.5 Accessing the Payment Queue	883
118.6 The Transaction Observer Object	883
118.7 Initiating the Purchase	883
118.8 The Transaction Process	883
118.9 Transaction Restoration Process	885
118.10 Testing In-App Purchases.....	885
118.11 Promoting In-App Purchases	886
118.12 Requesting App Reviews.....	886
118.13 Summary.....	887
119. Preparing an iOS 12 Application for In-App Purchases	889
119.1 About the Example Application	889
119.2 Creating the Xcode Project	889
119.3 Registering and Enabling the App ID for In App Purchasing	889
119.4 Configuring the Application in iTunes Connect	890
119.5 Creating an In-App Purchase Item.....	890
119.6 Summary.....	891
120. An iOS 12 In-App Purchase Tutorial	893
120.1 The Application User Interface	893
120.2 Designing the Storyboard	893
120.3 Configuring the View Controller Class	895
120.4 Initiating and Handling the Purchase.....	896
120.5 Testing the Application	898
120.6 Troubleshooting.....	899
120.7 Promoting the In-App Purchase	899
120.8 Summary.....	900
121. Configuring and Creating App Store Hosted Content for iOS 12 In-App Purchases	901
121.1 Configuring an Application for In-App Purchase Hosted Content	901
121.2 The Anatomy of an In-App Purchase Hosted Content Package.....	901
121.3 Creating an In-App Purchase Hosted Content Package	902
121.4 Archiving the Hosted Content Package	902
121.5 Validating the Hosted Content Package	903
121.6 Uploading the Hosted Content Package	903
121.7 Summary.....	904
122. Preparing and Submitting an iOS 12 Application to the App Store	905
122.1 Verifying the iOS Distribution Certificate	905
122.2 Adding App Icons	907

122.3 Designing the Launch Screen	908
122.4 Assign the Project to a Team	908
122.5 Archiving the Application for Distribution	909
122.6 Configuring the Application in iTunes Connect.....	909
122.7 Validating and Submitting the Application	910
122.8 Configuring and Submitting the App for Review	913
Index.....	915

1. Start Here

The goal of this book is to teach the skills necessary to create iOS applications using the iOS 12 SDK, Xcode 10 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 11 and need to get up to speed on the features of iOS 12 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 12 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 in addition to drag and drop integration and the document browser.

The key new features of iOS 12 and Xcode 10 are also covered in detail, including Siri shortcuts and the new iOS machine learning features.

The aim of this book, therefore, is to teach you the skills necessary to build your own apps for iOS 12. Assuming you are ready to download the iOS 12 SDK and Xcode 10, have an Intel-based Mac and ideas for some apps to develop, you are ready to get started.

1.2 For iOS 11 Developers

If you have already read the iOS 11 edition of this book, or have experience with the iOS 11 SDK then you might prefer to go directly to the new chapters in this iOS 12 edition of the book.

All chapters have been updated to reflect the changes and features introduced as part of iOS 12, Swift 4.2 and Xcode 10. Chapters included in this edition that were not contained in the previous edition, or have been significantly rewritten for iOS 12 and Xcode 10 are as follows:

- *Using iCloud Drive Storage in an iOS 12 App*
- *An Overview of Siri Shortcut App Integration*
- *Building Siri Shortcut Support into an iOS App*
- *An Introduction to Machine Learning on iOS*
- *Using Create ML to Build an Image Classification Model*
- *An iOS Vision and CoreML Image Classification Tutorial*

1.3 Source Code Download

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

<https://www.ebookfrenzy.com/retail/ios12/>

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.

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:

<https://www.ebookfrenzy.com/errata/ios12.html>

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

2. Joining the Apple Developer Program

The first step in the process of learning to develop iOS 12 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 10 and the iOS 12 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

Joining the Apple Developer Program

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:

<https://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:

The screenshot shows the Apple Developer Member Center interface. At the top, there's a navigation bar with the Apple logo, the text 'Developer', and a dropdown for 'Neil Smyth'. The main area is titled 'Account' and displays the user's name, 'Neil Smyth', and 'Apple Developer Program'. On the left, a sidebar titled 'Program Resources' lists several options: 'Overview' (which is selected and highlighted in grey), 'Membership', 'Certificates, IDs & Profiles', 'iTunes Connect', 'CloudKit Dashboard', and 'Code-Level Support'. Below this, under 'Additional Resources', are 'Documentation', 'Downloads', 'Forums', and 'Bug Reporter'. The right side of the screen contains two main sections: 'Certificates, Identifiers & Profiles' (with a circular icon) and 'iTunes Connect' (with a circular icon). Both sections have brief descriptions: 'Certificates, Identifiers & Profiles' says 'Manage the certificates, identifiers, profiles, and devices you need to develop and distribute apps.', and 'iTunes Connect' says 'Publish and manage your apps on the App Store with iTunes Connect.'

Figure 2-1

2.5 Summary

An important early step in the iOS 12 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 12 SDK and Xcode 10 development environment.

3. Installing Xcode 10 and the iOS 12 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 12 SDK on macOS.

3.1 Identifying Your macOS Version

The Xcode 10 environment requires that the version of macOS running on the system be version 10.13.4 or later. If you are unsure of the version of macOS on 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 *Version* line

If the “About This Mac” dialog does not indicate that macOS 10.13.4 or later is running, click on the *Software Update...* button to download and install the appropriate operating system upgrades.



Figure 3-1

3.2 Installing Xcode 10 and the iOS 12 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 12 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:

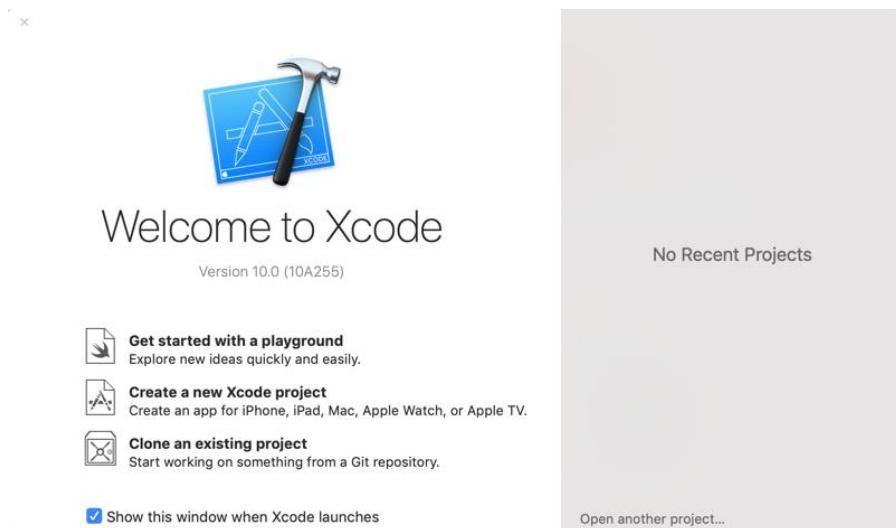


Figure 3-2

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.

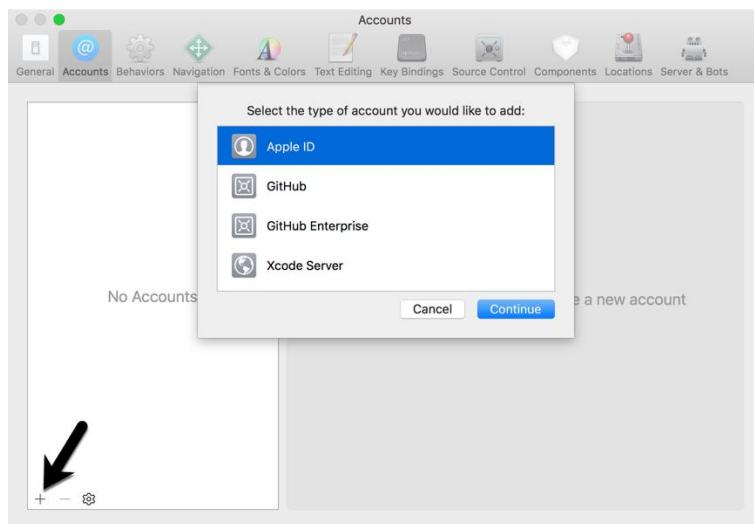


Figure 3-3

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:

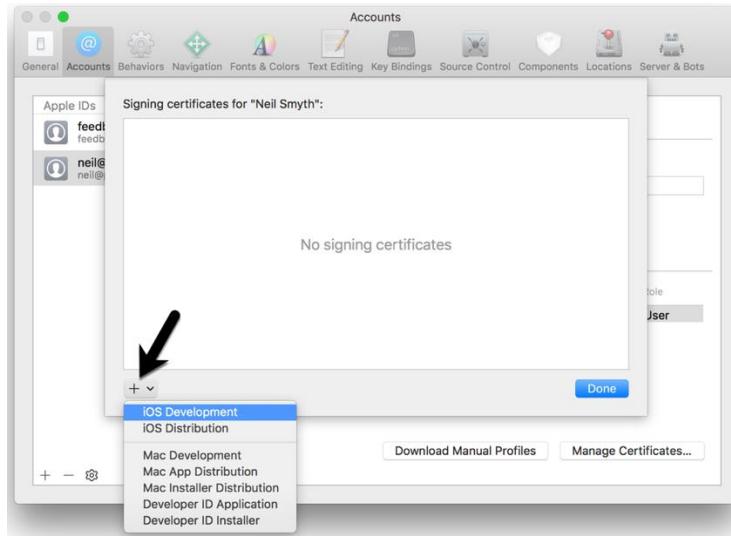


Figure 3-4

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 10 we can now look at Xcode in more detail.

4. A Guided Tour of Xcode 10

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 10

As with all iOS examples in this book, the development of our example will take place within the Xcode 10 development environment. If you have not already installed this tool together with the latest iOS SDK refer first to the *Installing Xcode 10 and the iOS 12 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:

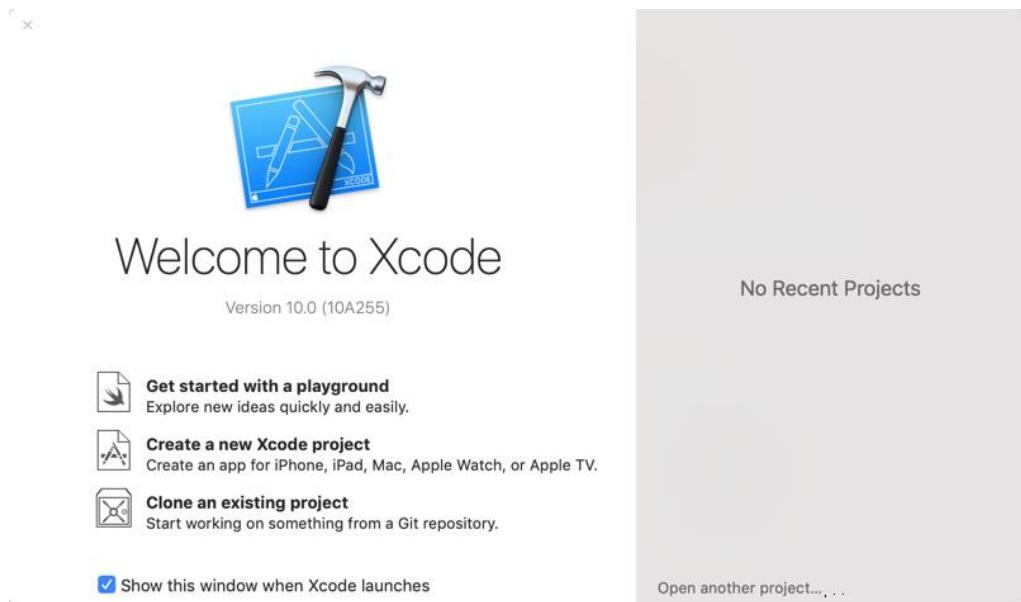


Figure 4-1

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:

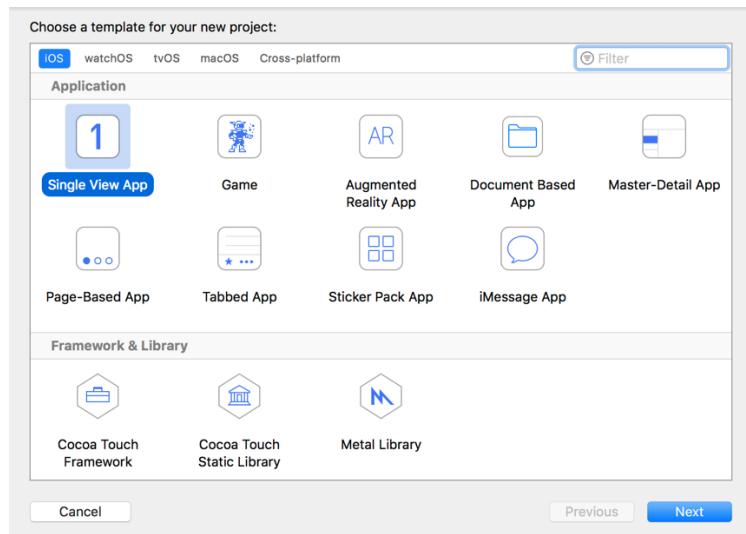


Figure 4-2

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:

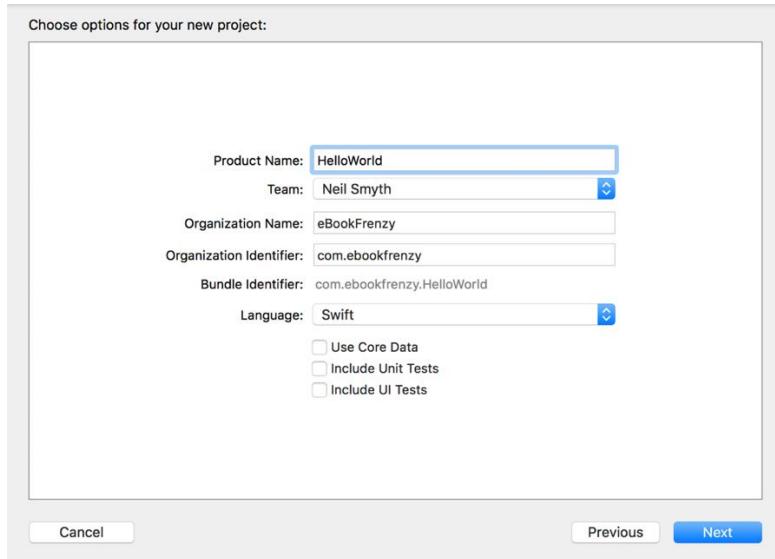


Figure 4-3

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:

A Guided Tour of Xcode 10

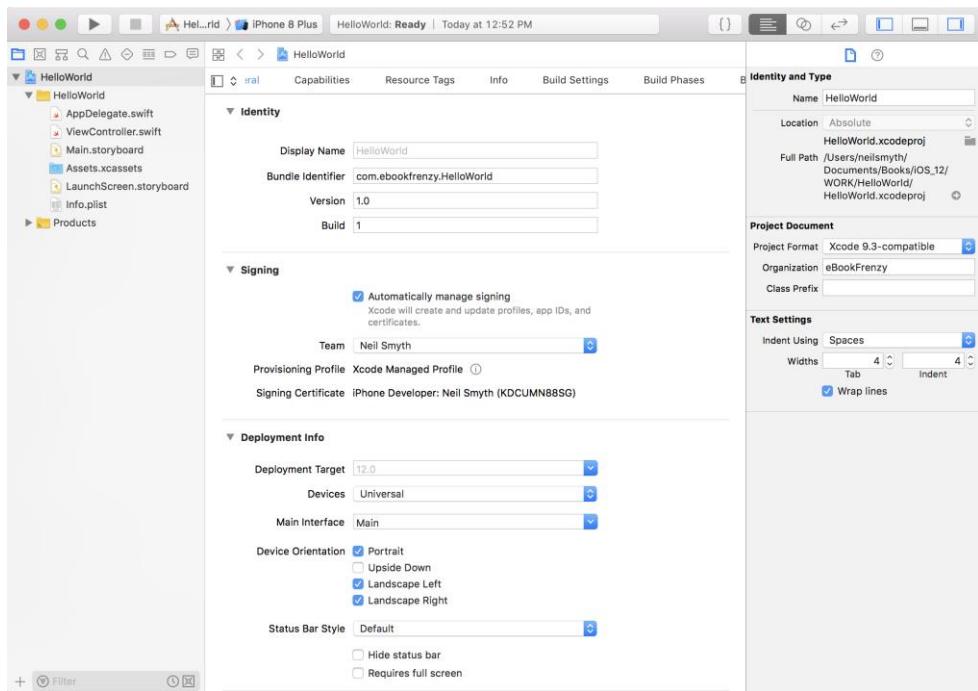


Figure 4-4

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/Max 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 10 and iOS 12 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:

A Guided Tour of Xcode 10

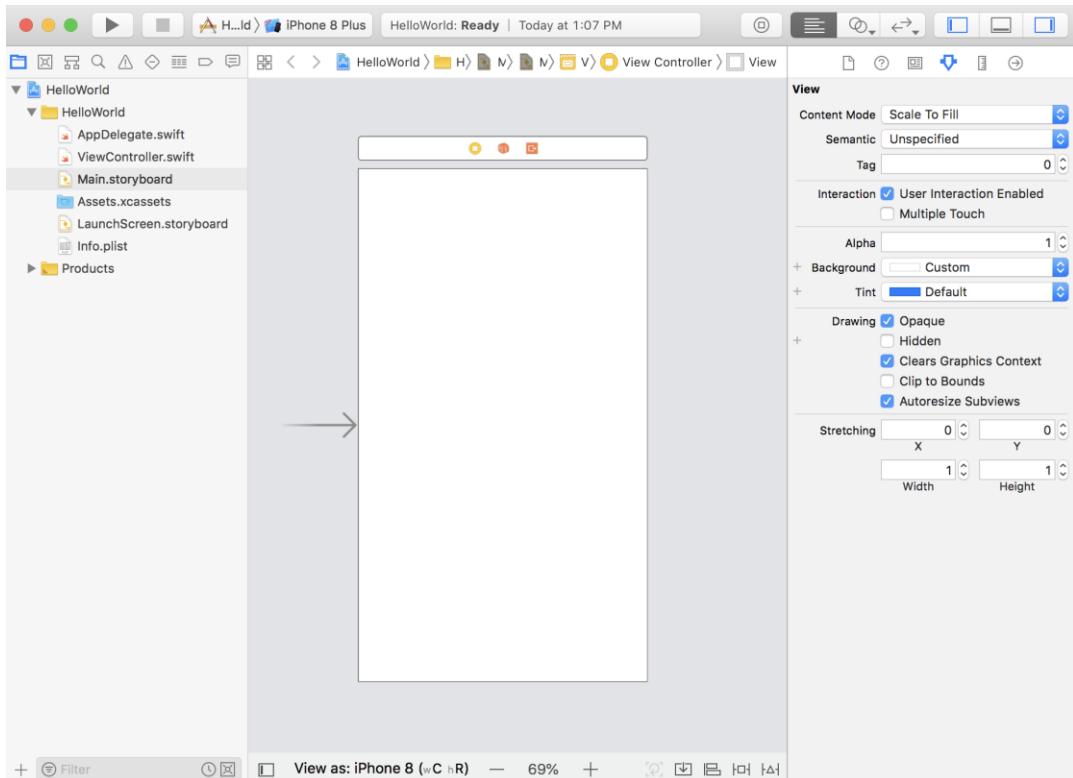


Figure 4-6

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. The user interface components are accessed from the Library panel which is displayed by clicking on the Show Library button in the Xcode toolbar as indicated in Figure 4-7:

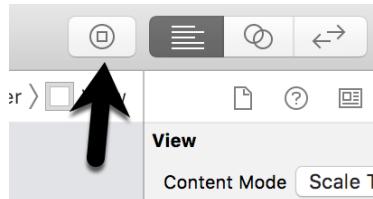
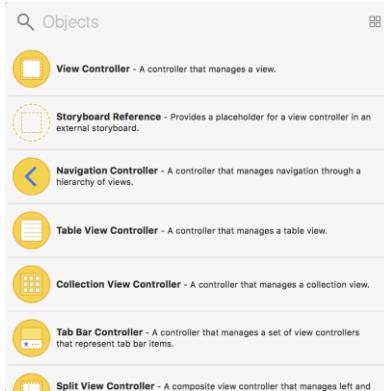


Figure 4-7

This button will display the UI components that can be used to construct our user interface. 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 top right-hand corner of the panel and to the right of the search box.

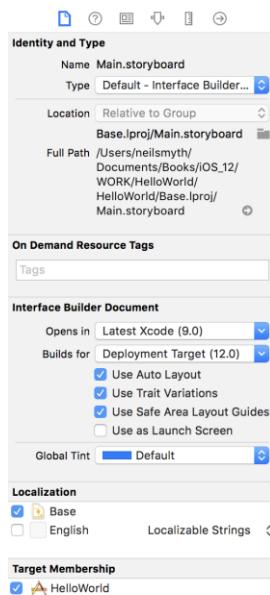
**Figure 4-8**

By default, the library panel will disappear either after an item has been dragged onto the layout or a click is performed outside of the panel. To keep the panel visible in this mode, hold down the Option key while clicking on the required Library item. Alternatively, displaying the Library panel by clicking on the toolbar button highlighted in Figure 4-7 while holding down the Option key will cause the panel to stay visible until it is manually closed.

In order to 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 section of the Xcode toolbar:

**Figure 4-9**

The Utilities panel, once displayed, will appear as illustrated in Figure 4-10:

**Figure 4-10**

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.

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 current property setting next to the *Background* setting and select the Custom option from the popup menu to display 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, display the Library panel as shown in Figure 4-7 above and either scroll down the list of objects in the Library panel to locate the Label object or, as illustrated in Figure 4-11, enter *Label* into the search box beneath the panel:

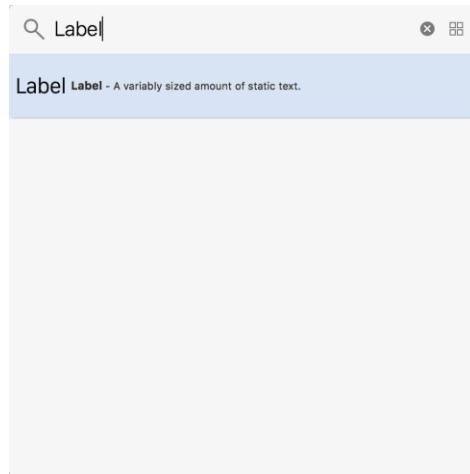


Figure 4-11

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 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.

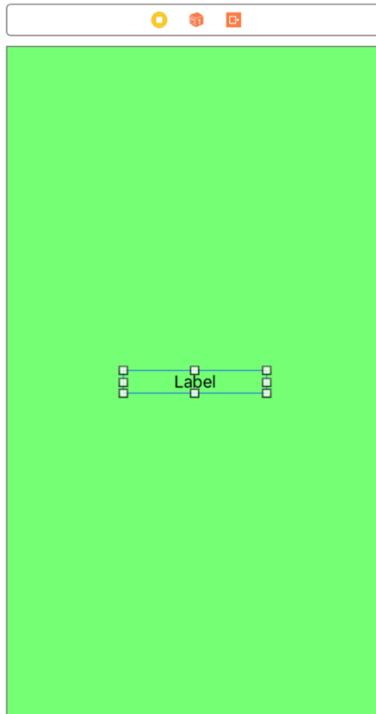


Figure 4-12

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 4-13:

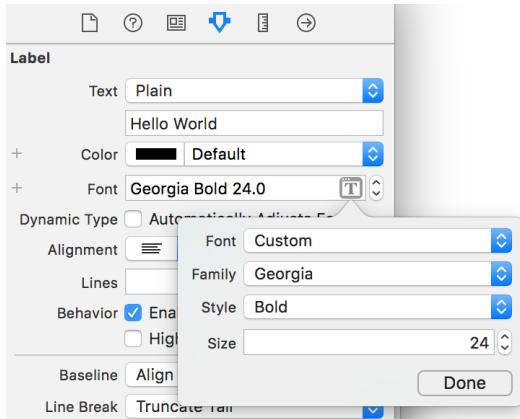


Figure 4-13

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 4-14 and enable both the

A Guided Tour of Xcode 10

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

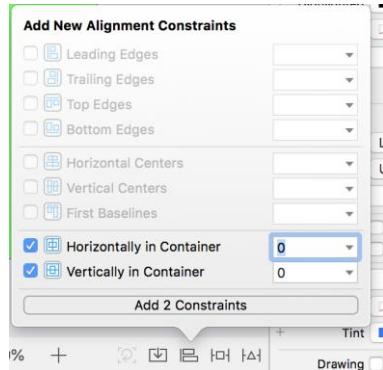


Figure 4-14

At this point, your View window will hopefully appear as outlined in Figure 4-15 (allowing, of course, for differences in your color and font choices).

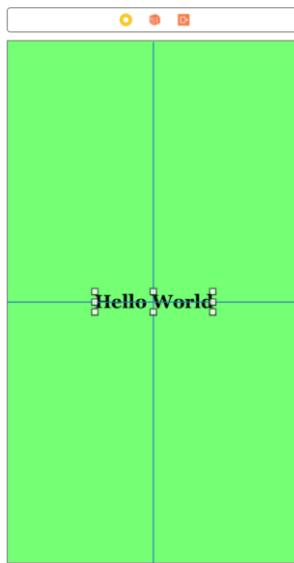


Figure 4-15

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 4-16) of the Interface Builder panel.



Figure 4-16

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.

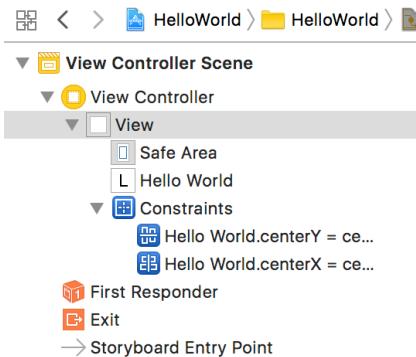


Figure 4-17

4.5 Building and Running an iOS 12 App in Xcode 10

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 4-18) has the *iPhone 8 Plus* simulator option selected:



Figure 4-18

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 4-19

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 8 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 4-20:

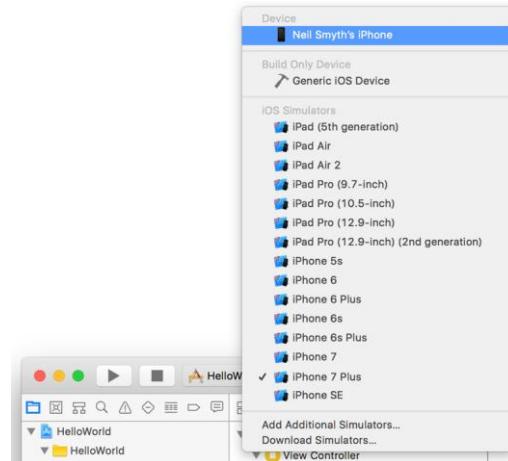


Figure 4-20

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 4-21, for example, shows a typical Device screen on a system where an iPhone has been detected:

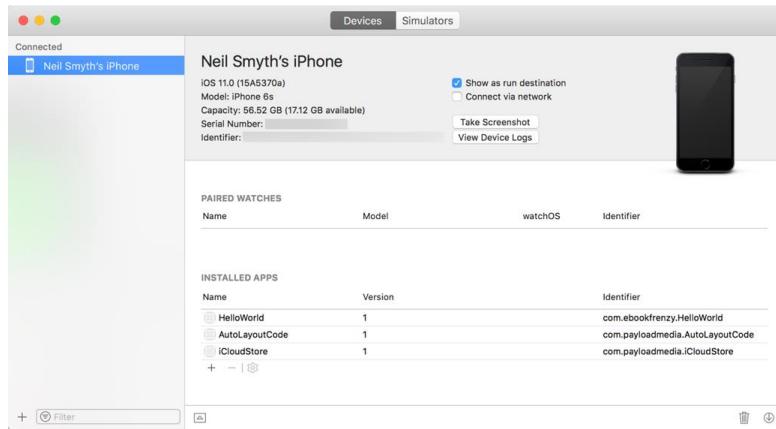


Figure 4-21

A wide range of simulator configurations are set up within Xcode by default and can be viewed 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 10 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 4-22:



Figure 4-22

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 4-23. 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.

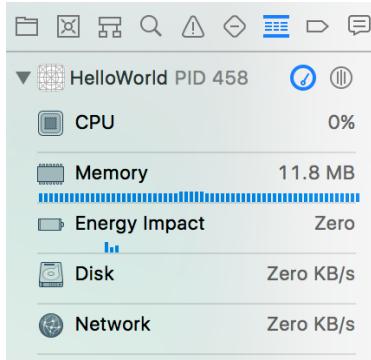


Figure 4-23

When one of these categories is selected, the main panel (Figure 4-24) updates to provide additional information about that particular aspect of the application’s performance:

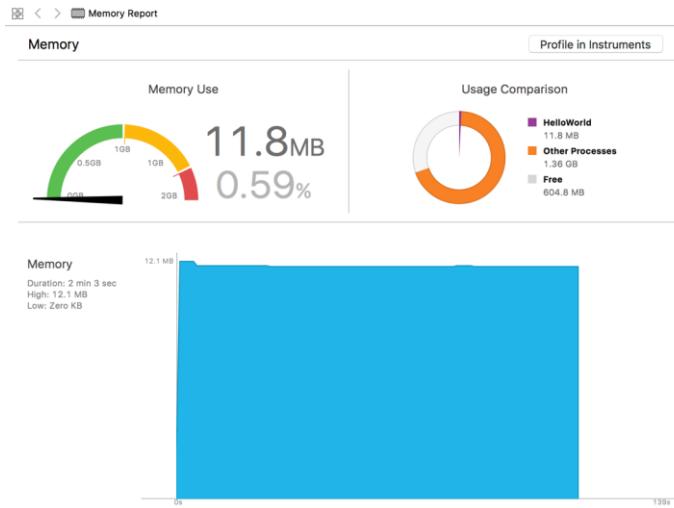


Figure 4-24

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 4-25:

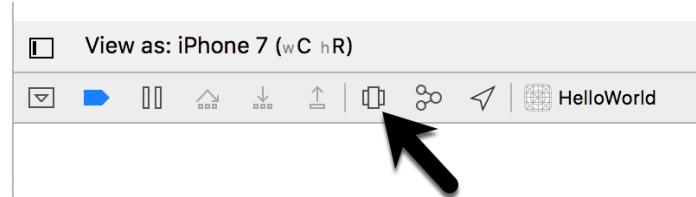


Figure 4-25

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 4-26 to switch to 3D mode:

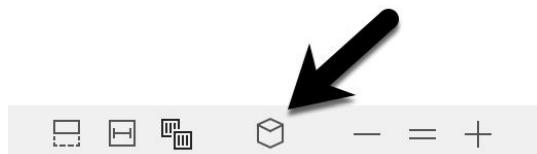


Figure 4-26

Figure 4-27 shows an example layout in this mode for a more complex user interface than that created in this chapter:

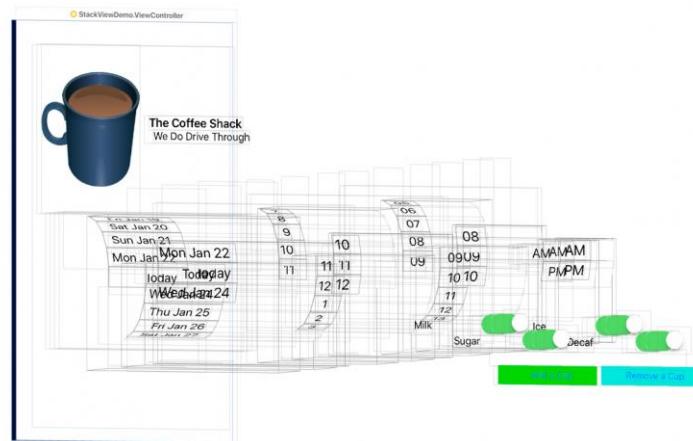


Figure 4-27

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

5. An Introduction to Xcode 10 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:

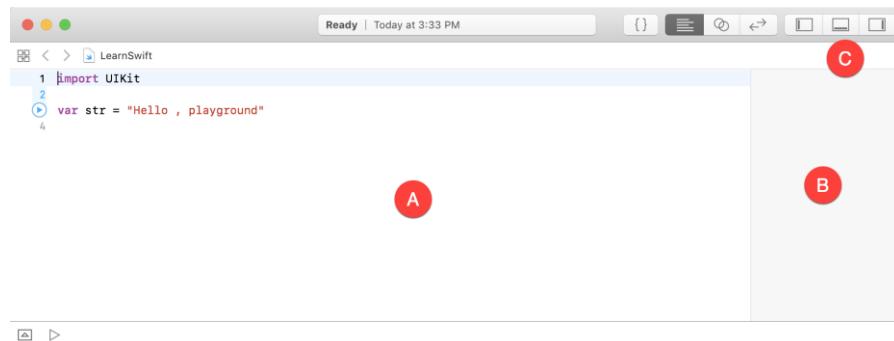


Figure 5-1

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.

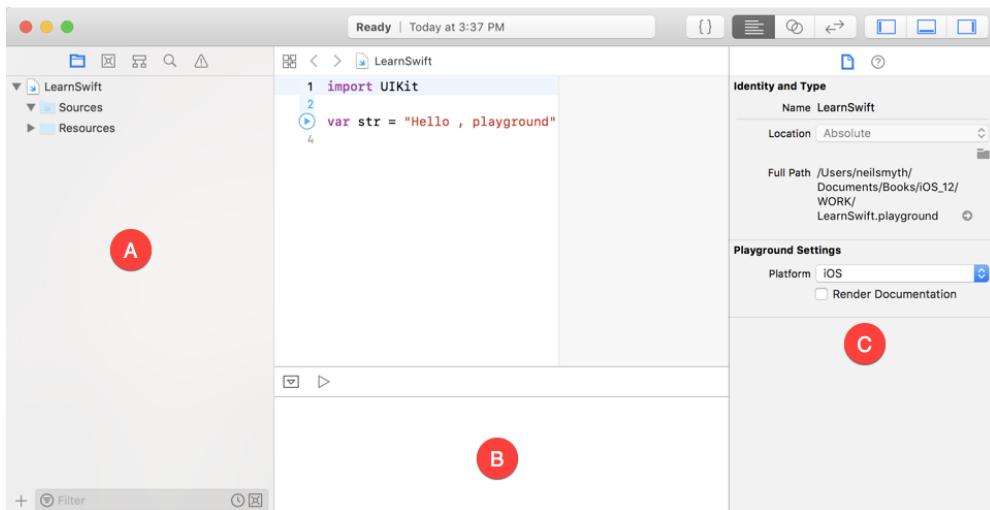


Figure 5-2

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 adding another line of Swift code so that it reads as follows:

```
import UIKit

var str = "Hello , playground"

print("Welcome to Swift")
```

All that the additional line of 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 although some extra code has been entered, nothing yet appears in the results panel. This is because the code has yet to be executed. One option to run the code is to click on the Execute Playground button located in the bottom left-hand corner of the main panel as indicated by the arrow in Figure 5-3:



Figure 5-3

When clicked, this button will execute all of the code in the current playground page from the first line of code to the last. Another option is to execute the code in stages using the run button located in the margin of the code editor as shown in Figure 5-4:

```

import UIKit
var str = "Hello , playground"
print("Welcome to Swift")

```

Figure 5-4

This button executes the line numbers with the shaded blue background including the line on which the button is currently positioned. In the above figure, for example, the button will execute lines 1 through 3 and then stop.

The position of the run button can be moved by hovering the mouse pointer over the line numbers in the editor. In Figure 5-5, for example, the run button is now positioned on line 5 and will execute lines 4 and 5 when clicked. Note that lines 1 to 3 are no longer highlighted in blue indicating that these have already been executed and are not eligible to be run this time:

```

import UIKit
var str = "Hello , playground"
print("Welcome to Swift")

```

Figure 5-5

This technique provides an easy way to execute the code in stages making it easier to understand how the code functions and to identify problems in code execution.

To reset the playground so that execution can be performed from the start of the code, simply click on the stop button as indicated in Figure 5-6.:



Figure 5-6

Using this incremental execution technique, execute lines 1 through 3 and note that output now appears in the results panel indicating that the variable has been initialized:

```
LearnSwift
1 import UIKit
2
3 var str = "Hello , playground"
4
5 print("Welcome to Swift")
6
```

The results panel shows the output: "Hello , playground".

Figure 5-7

Next, execute the remaining lines up to and including line 5 at which point the “Welcome to Swift” output should appear both in the results panel and Debug panel:

```
Ready to continue LearnSwift
LearnSwift
1 import UIKit
2
3 var str = "Hello , playground"
4
5 print("Welcome to Swift")
6
7
8
```

The results panel shows the output: "Hello , playground". The debug panel shows the output: "Welcome to Swift\n".

Figure 5-8

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, click on the run button positioned at line 13 to execute these new lines of code. 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 5-9:



The screenshot shows a Swift playground in Xcode. On the left, the code is displayed:

```

5 print("Welcome to Swift")
6
7 var x = 10
8
9 for index in 1...20 {
10     let y = index * x
11     x -= 1
12     print(y)
13 }

```

Line 13 contains a run button (a play icon). To the right, the results panel shows the output of the loop:

- "Welcome to Swift\n"
- 10
- (20 times)
- (20 times)
- (20 times)

Below these, three small square buttons with icons are visible, corresponding to the three additional buttons shown in Figure 5-9.

Figure 5-9

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 5-10:

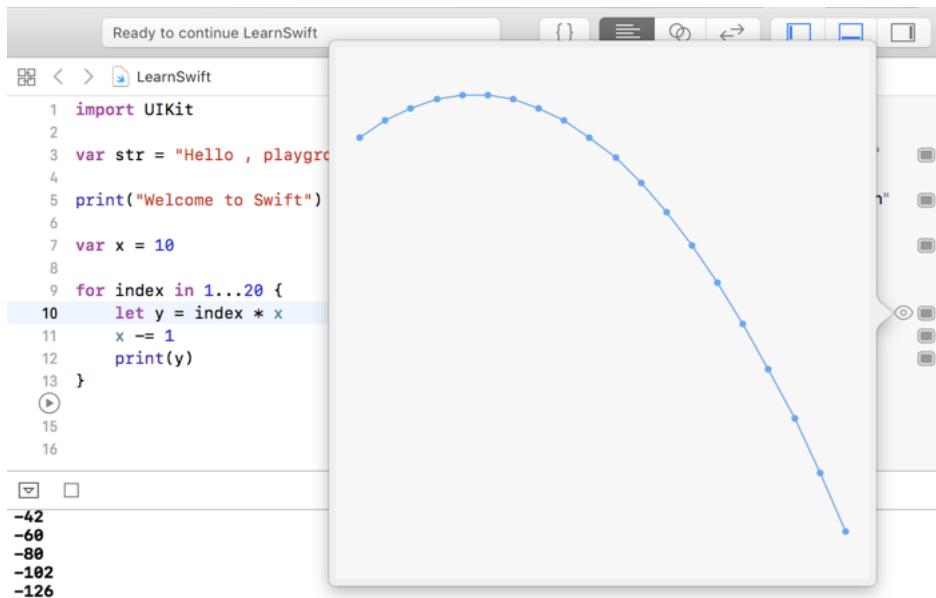


Figure 5-10

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

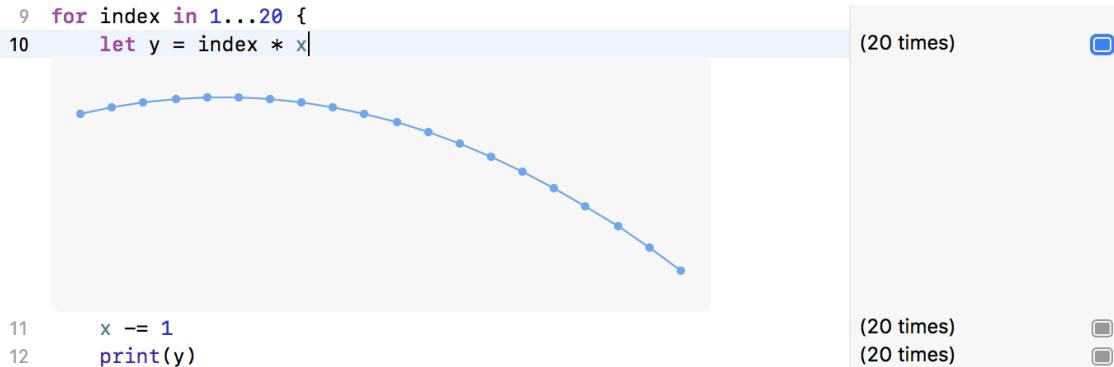


Figure 5-11

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 5-12:

```

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 5-12

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

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 LearnSwift entry at the top of the Navigator panel, right-clicking and selecting the *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 5-13:

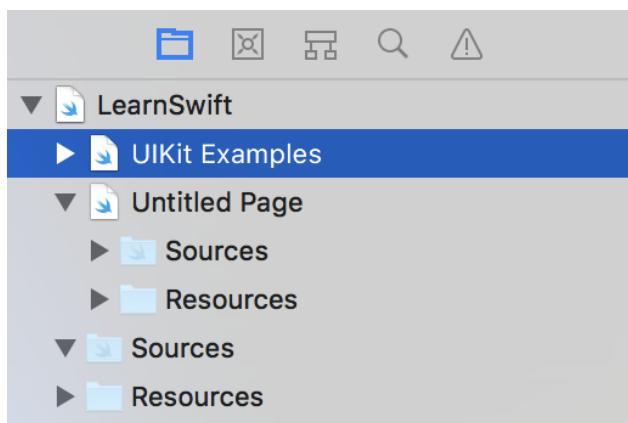


Figure 5-13

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 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 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 existing code can be removed) and run the code. This code provides 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 5-14

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 5-15

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

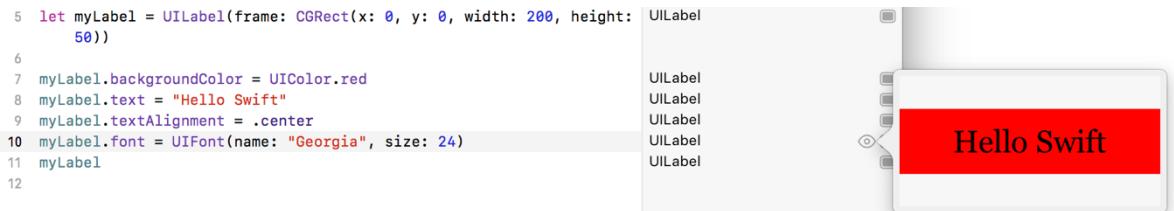


Figure 5-16

5.8 Adding Resources to a Playground

Another useful 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 5-17) and note the presence of a folder named *Resources*:

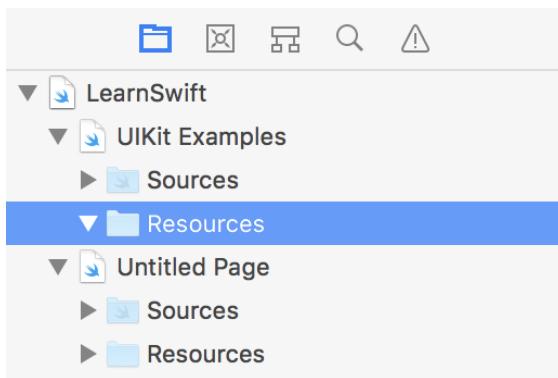


Figure 5-17

If you have not already done so, download and unpack the code samples archive from the following URL:
<https://www.ebookfrenzy.com/retail/ios12/>

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:

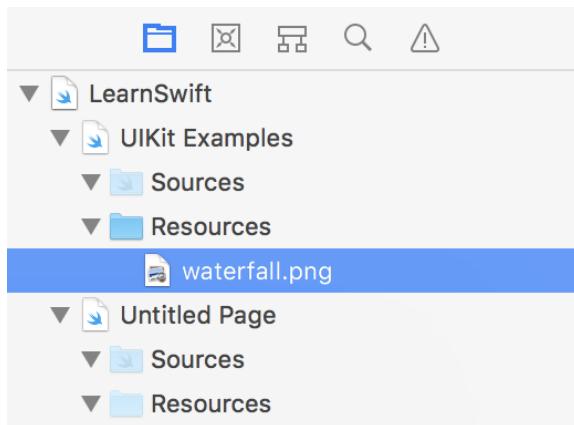


Figure 5-18

An Introduction to Xcode 10 Playgrounds

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, run the new statement and use either the Quick Look or inline option to view the results of the code:



Figure 5-19

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.

Once the code has been executed, 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 5-20:

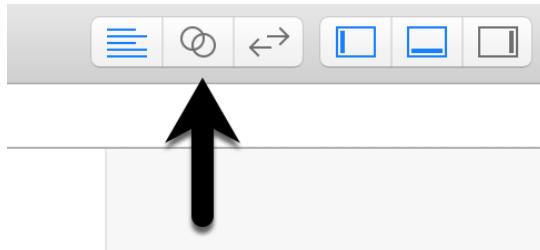


Figure 5-20

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, re-execute the code at which point the views should appear in the timeline (Figure 5-21). During the 5 second animation duration, the red square should rotate through 360 degrees while gradually changing color to blue:

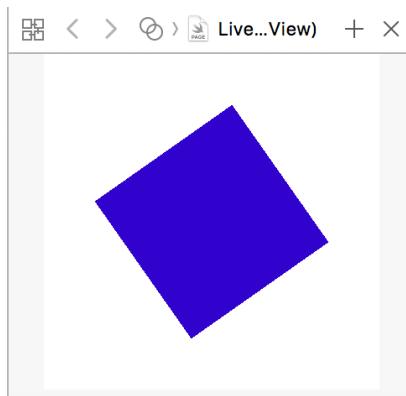


Figure 5-21

To repeat the execution of the code in the playground page, click on the stop button highlighted in Figure 5-6 to reset the playground and change the stop button into the run button (Figure 5-3). Click the run button to repeat the execution.

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 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 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) messages. 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 to keep the panic out of his voice.  
"""  
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 12 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 12 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:

Swift Data Types, Constants and Variables

```
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:

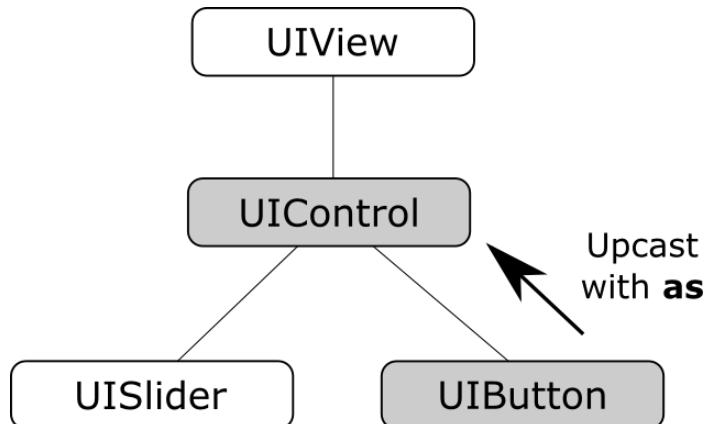


Figure 6-1

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:

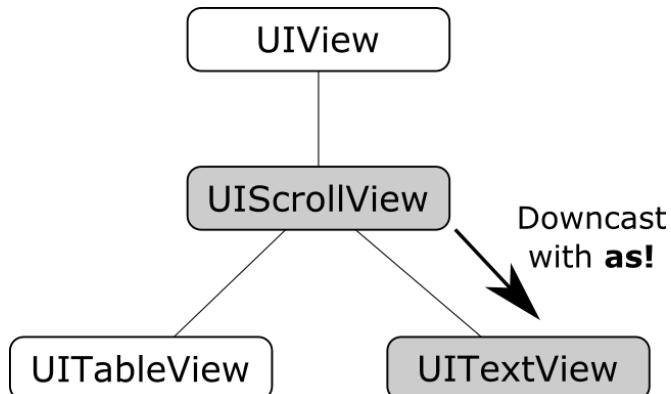


Figure 6-2

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.