

# Building iOS 17 Apps with Xcode Storyboards





# **Building iOS 17 Apps with Xcode Storyboards**

---

Building iOS 17 Apps with Xcode Storyboards

ISBN-13: 978-1-951442-84-2

© 2024 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

Find more books at <https://www.payloadbooks.com>.



## Table of Contents

<b>1. Start Here.....</b>	<b>1</b>
1.1 Source Code Download.....	1
1.2 Feedback.....	1
1.3 Errata.....	1
1.4 Find more books.....	1
<b>2. Joining the Apple Developer Program.....</b>	<b>3</b>
2.1 Downloading Xcode 15 and the iOS 17 SDK .....	3
2.2 Apple Developer Program.....	3
2.3 When to Enroll in the Apple Developer Program?.....	3
2.4 Enrolling in the Apple Developer Program .....	4
2.5 Summary .....	5
<b>3. Installing Xcode 15 and the iOS 17 SDK .....</b>	<b>7</b>
3.1 Identifying Your macOS Version .....	7
3.2 Installing Xcode 15 and the iOS 17 SDK.....	7
3.3 Starting Xcode .....	8
3.4 Adding Your Apple ID to the Xcode Preferences.....	8
3.5 Developer and Distribution Signing Identities .....	9
3.6 Summary .....	9
<b>4. A Guided Tour of Xcode 15.....</b>	<b>11</b>
4.1 Starting Xcode 15 .....	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 App in Xcode.....	22
4.6 Running the App on a Physical iOS Device.....	22
4.7 Managing Devices and Simulators.....	23
4.8 Enabling Network Testing.....	24
4.9 Dealing with Build Errors .....	24
4.10 Monitoring Application Performance .....	24
4.11 Exploring the User Interface Layout Hierarchy .....	25
4.12 Summary .....	27
<b>5. An Introduction to Xcode 15 Playgrounds.....</b>	<b>29</b>
5.1 What is a Playground? .....	29
5.2 Creating a New Playground .....	29
5.3 A Swift Playground Example .....	30
5.4 Viewing Results .....	32
5.5 Adding Rich Text Comments .....	34
5.6 Working with Playground Pages .....	35
5.7 Working with UIKit in Playgrounds.....	35
5.8 Adding Resources to a Playground .....	36
5.9 Working with Enhanced Live Views.....	38

## Table of Contents

5.10 When to Use Playgrounds.....	40
5.11 Summary .....	40
<b>6. Swift Data Types, Constants and Variables .....</b>	<b>41</b>
6.1 Using a Swift Playground .....	41
6.2 Swift Data Types.....	41
6.2.1 Integer Data Types .....	42
6.2.2 Floating Point Data Types.....	42
6.2.3 Bool Data Type.....	42
6.2.4 Character Data Type.....	42
6.2.5 String Data Type.....	43
6.2.6 Special Characters/Escape Sequences .....	44
6.3 Swift Variables.....	44
6.4 Swift Constants .....	45
6.5 Declaring Constants and Variables.....	45
6.6 Type Annotations and Type Inference .....	45
6.7 The Swift Tuple .....	46
6.8 The Swift Optional Type.....	47
6.9 Type Casting and Type Checking.....	50
6.10 Summary .....	52
<b>7. Swift Operators and Expressions .....</b>	<b>53</b>
7.1 Expression Syntax in Swift .....	53
7.2 The Basic Assignment Operator.....	53
7.3 Swift Arithmetic Operators.....	53
7.4 Compound Assignment Operators.....	54
7.5 Comparison Operators.....	54
7.6 Boolean Logical Operators.....	55
7.7 Range Operators.....	55
7.8 The Ternary Operator .....	56
7.9 Nil Coalescing Operator.....	56
7.10 Bitwise Operators.....	57
7.10.1 Bitwise NOT .....	57
7.10.2 Bitwise AND .....	57
7.10.3 Bitwise OR.....	58
7.10.4 Bitwise XOR.....	58
7.10.5 Bitwise Left Shift.....	58
7.10.6 Bitwise Right Shift.....	59
7.11 Compound Bitwise Operators.....	59
7.12 Summary .....	60
<b>8. Swift Control Flow.....</b>	<b>61</b>
8.1 Looping Control Flow .....	61
8.2 The Swift for-in Statement.....	61
8.2.1 The while Loop .....	62
8.3 The repeat ... while loop .....	62
8.4 Breaking from Loops .....	63
8.5 The continue Statement .....	63
8.6 Conditional Control Flow.....	64

8.7 Using the if Statement .....	64
8.8 Using if ... else ... Statements .....	64
8.9 Using if ... else if ... Statements .....	65
8.10 The guard Statement .....	65
8.11 Summary .....	66
<b>9. The Swift Switch Statement .....</b>	<b>67</b>
9.1 Why Use a switch Statement? .....	67
9.2 Using the switch Statement Syntax .....	67
9.3 A Swift switch Statement Example .....	67
9.4 Combining case Statements .....	68
9.5 Range Matching in a switch Statement .....	69
9.6 Using the where statement .....	69
9.7 Fallthrough .....	70
9.8 Summary .....	70
<b>10. Swift Functions, Methods and Closures .....</b>	<b>71</b>
10.1 What is a Function? .....	71
10.2 What is a Method? .....	71
10.3 How to Declare a Swift Function .....	71
10.4 Implicit Returns from Single Expressions .....	72
10.5 Calling a Swift Function .....	72
10.6 Handling Return Values .....	72
10.7 Local and External Parameter Names .....	73
10.8 Declaring Default Function Parameters .....	73
10.9 Returning Multiple Results from a Function .....	74
10.10 Variable Numbers of Function Parameters .....	74
10.11 Parameters as Variables .....	75
10.12 Working with In-Out Parameters .....	75
10.13 Functions as Parameters .....	76
10.14 Closure Expressions .....	78
10.15 Shorthand Argument Names .....	79
10.16 Closures in Swift .....	79
10.17 Summary .....	80
<b>11. The Basics of Swift Object-Oriented Programming .....</b>	<b>81</b>
11.1 What is an Instance? .....	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 De-initializing a Class Instance .....	83
11.8 Calling Methods and Accessing Properties .....	84
11.9 Stored and Computed Properties .....	85
11.10 Lazy Stored Properties .....	86
11.11 Using self in Swift .....	87
11.12 Understanding Swift Protocols .....	88
11.13 Opaque Return Types .....	89

## Table of Contents

11.14 Summary .....	90
<b>12. An Introduction to Swift Subclassing and Extensions .....</b>	<b>91</b>
12.1 Inheritance, Classes and Subclasses .....	91
12.2 A Swift Inheritance Example .....	91
12.3 Extending the Functionality of a Subclass .....	92
12.4 Overriding Inherited Methods .....	92
12.5 Initializing the Subclass .....	93
12.6 Using the SavingsAccount Class .....	94
12.7 Swift Class Extensions .....	94
12.8 Summary .....	95
<b>13. An Introduction to Swift Structures and Enumerations .....</b>	<b>97</b>
13.1 An Overview of Swift Structures .....	97
13.2 Value Types vs. Reference Types .....	98
13.3 When to Use Structures or Classes .....	100
13.4 An Overview of Enumerations .....	100
13.5 Summary .....	101
<b>14. Working with Array and Dictionary Collections in Swift .....</b>	<b>103</b>
14.1 Mutable and Immutable Collections .....	103
14.2 Swift Array Initialization .....	103
14.3 Working with Arrays in Swift .....	104
14.3.1 Array Item Count .....	104
14.3.2 Accessing Array Items .....	104
14.3.3 Random Items and Shuffling .....	104
14.3.4 Appending Items to an Array .....	105
14.3.5 Inserting and Deleting Array Items .....	105
14.3.6 Array Iteration .....	105
14.4 Creating Mixed Type Arrays .....	106
14.5 Swift Dictionary Collections .....	106
14.6 Swift Dictionary Initialization .....	106
14.7 Sequence-based Dictionary Initialization .....	107
14.8 Dictionary Item Count .....	108
14.9 Accessing and Updating Dictionary Items .....	108
14.10 Adding and Removing Dictionary Entries .....	108
14.11 Dictionary Iteration .....	108
14.12 Summary .....	109
<b>15. Understanding Error Handling in Swift 5 .....</b>	<b>111</b>
15.1 Understanding Error Handling .....	111
15.2 Declaring Error Types .....	111
15.3 Throwing an Error .....	112
15.4 Calling Throwing Methods and Functions .....	112
15.5 Accessing the Error Object .....	114
15.6 Disabling Error Catching .....	114
15.7 Using the defer Statement .....	114
15.8 Summary .....	115
<b>16. The iOS 17 App and Development Architecture .....</b>	<b>117</b>



16.1 An Overview of the iOS 17 Operating System Architecture.....	117
16.2 Model View Controller (MVC).....	118
16.3 The Target-Action pattern, IBOutlets, and IBActions.....	118
16.4 Subclassing.....	119
16.5 Delegation .....	119
16.6 Summary .....	119
<b>17. Creating an Interactive iOS 17 App .....</b>	<b>121</b>
17.1 Creating the New Project.....	121
17.2 Creating the User Interface.....	121
17.3 Building and Running the Sample App.....	123
17.4 Adding Actions and Outlets .....	124
17.5 Building and Running the Finished App .....	128
17.6 Hiding the Keyboard .....	128
17.7 Summary .....	129
<b>18. Understanding iOS 17 Views, Windows, and the View Hierarchy .....</b>	<b>131</b>
18.1 An Overview of Views and the UIKit Class Hierarchy.....	131
18.2 The UIWindow Class.....	131
18.3 The View Hierarchy .....	131
18.4 Viewing Hierarchy Ancestors in Interface Builder.....	133
18.5 View Types .....	133
18.5.1 The Window.....	134
18.5.2 Container Views.....	134
18.5.3 Controls.....	134
18.5.4 Display Views .....	134
18.5.5 Text and WebKit Views .....	134
18.5.6 Navigation Views and Tab Bars.....	134
18.5.7 Alert Views.....	134
18.6 Summary .....	134
<b>19. An Introduction to Auto Layout in iOS 17 .....</b>	<b>135</b>
19.1 An Overview of Auto Layout.....	135
19.2 Alignment Rects .....	136
19.3 Intrinsic Content Size .....	136
19.4 Content Hugging and Compression Resistance Priorities .....	136
19.5 Safe Area Layout Guide.....	136
19.6 Three Ways to Create Constraints.....	137
19.7 Constraints in More Detail .....	137
19.8 Summary .....	138
<b>20. Working with iOS 17 Auto Layout Constraints in Interface Builder.....</b>	<b>139</b>
20.1 An Example of Auto Layout in Action .....	139
20.2 Working with Constraints.....	139
20.3 The Auto Layout Features of Interface Builder .....	142
20.3.1 Suggested Constraints .....	142
20.3.2 Visual Cues.....	143
20.3.3 Highlighting Constraint Problems .....	144
20.3.4 Viewing, Editing, and Deleting Constraints.....	147
20.4 Creating New Constraints in Interface Builder.....	149

## Table of Contents

20.5 Adding Aspect Ratio Constraints .....	149
20.6 Resolving Auto Layout Problems .....	149
20.7 Summary .....	151
<b>21. Implementing iOS 17 Auto Layout Constraints in Code .....</b>	<b>153</b>
21.1 Creating Constraints Using NSLayoutConstraint .....	153
21.2 Adding a Constraint to a View .....	154
21.3 Turning off Auto Resizing Translation .....	155
21.4 Creating Constraints Using NSLayoutConstraint .....	155
21.5 An Example App .....	156
21.6 Creating the Views .....	156
21.7 Creating and Adding the Constraints .....	157
21.8 Using Layout Anchors .....	159
21.9 Removing Constraints .....	159
21.10 Summary .....	159
<b>22. Implementing Cross-Hierarchy Auto Layout Constraints in iOS 17.....</b>	<b>161</b>
22.1 The Example App .....	161
22.2 Establishing Outlets .....	162
22.3 Writing the Code to Remove the Old Constraint .....	163
22.4 Adding the Cross Hierarchy Constraint .....	163
22.5 Testing the App.....	164
22.6 Summary .....	164
<b>23. Understanding the iOS 17 Auto Layout Visual Format Language.....</b>	<b>165</b>
23.1 Introducing the Visual Format Language .....	165
23.2 Visual Format Language Examples.....	165
23.3 Using the constraints(withVisualFormat:) Method.....	166
23.4 Summary .....	167
<b>24. Using Trait Variations to Design Adaptive iOS 17 User Interfaces.....</b>	<b>169</b>
24.1 Understanding Traits and Size Classes.....	169
24.2 Size Classes in Interface Builder.....	169
24.3 Enabling Trait Variations .....	170
24.4 Setting “Any” Defaults .....	170
24.5 Working with Trait Variations in Interface Builder.....	170
24.6 Attributes Inspector Trait Variations .....	171
24.7 Using Constraint Variations .....	173
24.8 An Adaptive User Interface Tutorial.....	174
24.9 Designing the Initial Layout .....	174
24.10 Adding Universal Image Assets.....	176
24.11 Increasing Font Size for iPad Devices .....	177
24.12 Adding Width Constraint Variations .....	177
24.13 Testing the Adaptivity.....	180
24.14 Summary .....	180
<b>25. Using Storyboards in Xcode 15 .....</b>	<b>181</b>
25.1 Creating the Storyboard Example Project .....	181
25.2 Accessing the Storyboard.....	181
25.3 Adding Scenes to the Storyboard.....	183

25.4 Configuring Storyboard Segues .....	184
25.5 Configuring Storyboard Transitions.....	184
25.6 Associating a View Controller with a Scene .....	185
25.7 Passing Data Between Scenes .....	185
25.8 Unwinding Storyboard Segues .....	186
25.9 Triggering a Storyboard Segue Programmatically.....	187
25.10 Summary .....	187
<b>26. Organizing Scenes over Multiple Storyboard Files .....</b>	<b>189</b>
26.1 Organizing Scenes into Multiple Storyboards.....	189
26.2 Establishing a Connection between Different Storyboards.....	191
26.3 Summary .....	191
<b>27. Using Xcode 15 Storyboards to Create an iOS 17 Tab Bar App .....</b>	<b>193</b>
27.1 An Overview of the Tab Bar .....	193
27.2 Understanding View Controllers in a Multiview App .....	193
27.3 Setting up the Tab Bar Example App .....	193
27.4 Reviewing the Project Files.....	194
27.5 Adding the View Controllers for the Content Views .....	194
27.6 Adding the Tab Bar Controller to the Storyboard .....	194
27.7 Designing the View Controller User interfaces.....	196
27.8 Configuring the Tab Bar Items .....	197
27.9 Building and Running the App .....	198
27.10 Summary .....	198
<b>28. An Overview of iOS 17 Table Views and Xcode 15 Storyboards.....</b>	<b>199</b>
28.1 An Overview of the Table View.....	199
28.2 Static vs. Dynamic Table Views .....	199
28.3 The Table View Delegate and dataSource .....	199
28.4 Table View Styles .....	200
28.5 Self-Sizing Table Cells.....	201
28.6 Dynamic Type.....	201
28.7 Table View Cell Styles .....	202
28.8 Table View Cell Reuse.....	203
28.9 Table View Swipe Actions .....	204
28.10 Summary .....	205
<b>29. Using Xcode 15 Storyboards to Build Dynamic TableViews.....</b>	<b>207</b>
29.1 Creating the Example Project .....	207
29.2 Adding the TableView Controller to the Storyboard .....	207
29.3 Creating the UITableViewController and UITableViewCell Subclasses .....	208
29.4 Declaring the Cell Reuse Identifier.....	209
29.5 Designing a Storyboard UITableView Prototype Cell .....	210
29.6 Modifying the AttractionTableViewCell Class .....	210
29.7 Creating the Table View Datasource .....	211
29.8 Downloading and Adding the Image Files .....	213
29.9 Compiling and Running the App.....	214
29.10 Handling TableView Swipe Gestures.....	214
29.11 Summary .....	215

<b>30. Implementing iOS 17 TableView Navigation using Storyboards.....</b>	<b>217</b>
30.1 Understanding the Navigation Controller .....	217
30.2 Adding the New Scene to the Storyboard .....	217
30.3 Adding a Navigation Controller .....	218
30.4 Establishing the Storyboard Segue.....	218
30.5 Modifying the AttractionDetailViewController Class .....	219
30.6 Using prepare(for segue:) to Pass Data between Storyboard Scenes.....	221
30.7 Testing the App .....	221
30.8 Customizing the Navigation Title Size .....	222
30.9 Summary .....	223
<b>31. Integrating Search using the iOS UISearchController.....</b>	<b>225</b>
31.1 Introducing the UISearchController Class .....	225
31.2 Adding a Search Controller to the TableViewStory Project .....	226
31.3 Implementing the updateSearchResults Method .....	226
31.4 Reporting the Number of Table Rows .....	227
31.5 Modifying the cellForRowAt Method .....	227
31.6 Modifying the Trailing Swipe Delegate Method .....	228
31.7 Modifying the Detail Segue .....	229
31.8 Handling the Search Cancel Button.....	229
31.9 Testing the Search Controller .....	229
31.10 Summary .....	230
<b>32. Working with the iOS 17 Stack View Class.....</b>	<b>231</b>
32.1 Introducing the UIStackView Class.....	231
32.2 Understanding Subviews and Arranged Subviews .....	232
32.3 StackView Configuration Options .....	233
32.3.1 axis.....	233
32.3.2 distribution .....	233
32.3.3 spacing .....	234
32.3.4 alignment .....	235
32.3.5 baseLineRelativeArrangement .....	237
32.3.6 layoutMarginsRelativeArrangement.....	237
32.4 Creating a Stack View in Code .....	237
32.5 Adding Subviews to an Existing Stack View.....	238
32.6 Hiding and Removing Subviews .....	238
32.7 Summary .....	238
<b>33. An iOS 17 Stack View Tutorial.....</b>	<b>239</b>
33.1 About the Stack View Example App .....	239
33.2 Creating the First Stack View .....	239
33.3 Creating the Banner Stack View.....	241
33.4 Adding the Switch Stack Views .....	242
33.5 Creating the Top-Level Stack View.....	242
33.6 Adding the Button Stack View .....	243
33.7 Adding the Final Subviews to the Top Level Stack View .....	244
33.8 Dynamically Adding and Removing Subviews .....	246
33.9 Summary .....	247
<b>34. A Guide to iPad Multitasking .....</b>	<b>249</b>

34.1 Using iPad Multitasking .....	249
34.2 Picture-In-Picture Multitasking .....	251
34.3 Multitasking and Size Classes .....	251
34.4 Handling Multitasking in Code .....	252
34.4.1 willTransition(to newcollection: with coordinator:) .....	253
34.4.2 viewWillTransition(to size: with coordinator:) .....	253
34.4.3 traitCollectionDidChange( _: ) .....	253
34.5 Lifecycle Method Calls .....	254
34.6 Opting Out of Multitasking .....	254
34.7 Summary .....	255
<b>35. An iPadOS Multitasking Example .....</b>	<b>257</b>
35.1 Creating the Multitasking Example Project .....	257
35.2 Adding the Image Files .....	257
35.3 Designing the Regular Width Size Class Layout .....	258
35.4 Designing the Compact Width Size Class .....	260
35.5 Testing the Project in a Multitasking Environment .....	261
35.6 Summary .....	262
<b>36. An Overview of Swift Structured Concurrency .....</b>	<b>263</b>
36.1 An Overview of Threads .....	263
36.2 The Application Main Thread .....	263
36.3 Completion Handlers .....	263
36.4 Structured Concurrency .....	264
36.5 Preparing the Project .....	264
36.6 Non-Concurrent Code .....	265
36.7 Introducing async/await Concurrency .....	266
36.8 Asynchronous Calls from Synchronous Functions .....	266
36.9 The await Keyword .....	267
36.10 Using async-let Bindings .....	268
36.11 Handling Errors .....	269
36.12 Understanding Tasks .....	270
36.13 Unstructured Concurrency .....	270
36.14 Detached Tasks .....	271
36.15 Task Management .....	271
36.16 Working with Task Groups .....	272
36.17 Avoiding Data Races .....	273
36.18 The for-await Loop .....	274
36.19 Asynchronous Properties .....	275
36.20 Summary .....	275
<b>37. Working with Directories in Swift on iOS 17 .....</b>	<b>277</b>
37.1 The Application Documents Directory .....	277
37.2 The FileManager, FileHandle, and Data Classes .....	277
37.3 Understanding Pathnames in Swift .....	278
37.4 Obtaining a Reference to the Default FileManager Object .....	278
37.5 Identifying the Current Working Directory .....	278
37.6 Identifying the Documents Directory .....	278
37.7 Identifying the Temporary Directory .....	279

## Table of Contents

37.8 Changing Directory .....	279
37.9 Creating a New Directory .....	279
37.10 Deleting a Directory .....	280
37.11 Listing the Contents of a Directory .....	280
37.12 Getting the Attributes of a File or Directory .....	281
37.13 Summary .....	282
<b>38. Working with Files in Swift on iOS 17 .....</b>	<b>283</b>
38.1 Obtaining a FileManager Instance Reference.....	283
38.2 Checking for the Existence of a File .....	283
38.3 Comparing the Contents of Two Files.....	283
38.4 Checking if a File is Readable/Writable/Executable/Deletable.....	284
38.5 Moving/Renaming a File.....	284
38.6 Copying a File.....	284
38.7 Removing a File.....	284
38.8 Creating a Symbolic Link.....	285
38.9 Reading and Writing Files with FileManager.....	285
38.10 Working with Files using the FileHandle Class .....	285
38.11 Creating a FileHandle Object.....	285
38.12 FileHandle File Offsets and Seeking.....	286
38.13 Reading Data from a File .....	286
38.14 Writing Data to a File .....	287
38.15 Truncating a File .....	287
38.16 Summary.....	287
<b>39. iOS 17 Directory Handling and File I/O in Swift – A Worked Example.....</b>	<b>289</b>
39.1 The Example App .....	289
39.2 Setting up the App Project .....	289
39.3 Designing the User Interface .....	289
39.4 Checking the Data File on App Startup .....	290
39.5 Implementing the Action Method .....	291
39.6 Building and Running the Example.....	291
39.7 Summary .....	292
<b>40. Preparing an iOS 17 App to use iCloud Storage.....</b>	<b>293</b>
40.1 iCloud Data Storage Services.....	293
40.2 Preparing an App to Use iCloud Storage .....	293
40.3 Enabling iCloud Support for an iOS 17 App .....	294
40.4 Reviewing the iCloud Entitlements File.....	295
40.5 Accessing Multiple Ubiquity Containers .....	295
40.6 Ubiquity Container URLs.....	296
40.7 Summary .....	296
<b>41. Managing Files using the iOS 17 UIDocument Class.....</b>	<b>297</b>
41.1 An Overview of the UIDocument Class.....	297
41.2 Subclassing the UIDocument Class.....	297
41.3 Conflict Resolution and Document States.....	297
41.4 The UIDocument Example App.....	298
41.5 Creating a UIDocument Subclass .....	298
41.6 Designing the User Interface .....	298

41.7 Implementing the App Data Structure .....	299
41.8 Implementing the contents(forType:) Method .....	300
41.9 Implementing the load(fromContents:) Method.....	300
41.10 Loading the Document at App Launch.....	300
41.11 Saving Content to the Document .....	303
41.12 Testing the App.....	304
41.13 Summary .....	304
<b>42. Using iCloud Storage in an iOS 17 App .....</b>	<b>305</b>
42.1 iCloud Usage Guidelines .....	305
42.2 Preparing the iCloudStore App for iCloud Access .....	305
42.3 Enabling iCloud Capabilities and Services .....	306
42.4 Configuring the View Controller.....	308
42.5 Implementing the loadFile Method.....	308
42.6 Implementing the metadataQueryDidFinishGathering Method .....	310
42.7 Implementing the saveDocument Method.....	312
42.8 Enabling iCloud Document and Data Storage .....	313
42.9 Running the iCloud App .....	313
42.10 Making a Local File Ubiquitous.....	314
42.11 Summary .....	314
<b>43. Using iCloud Drive Storage in an iOS 17 App .....</b>	<b>315</b>
43.1 Preparing an App to use iCloud Drive Storage .....	315
43.2 Making Changes to the NSUbiquitousContainers Key.....	316
43.3 Creating the iCloud Drive Example Project .....	316
43.4 Modifying the Info.plist File .....	316
43.5 Designing the User Interface .....	317
43.6 Accessing the Ubiquitous Container .....	318
43.7 Saving the File to iCloud Drive .....	319
43.8 Testing the App.....	319
43.9 Summary .....	320
<b>44. An Overview of the iOS 17 Document Browser View Controller.....</b>	<b>321</b>
44.1 An Overview of the Document Browser View Controller .....	321
44.2 The Anatomy of a Document-Based App .....	322
44.3 Document Browser Project Settings.....	322
44.4 The Document Browser Delegate Methods.....	323
44.4.1 didRequestDocumentCreationWithHandler .....	323
44.4.2 didImportDocumentAt.....	324
44.4.3 didPickDocumentURLs .....	324
44.4.4 failedToImportDocumentAt.....	324
44.5 Customizing the Document Browser .....	324
44.6 Adding Browser Actions .....	325
44.7 Summary .....	326
<b>45. An iOS 17 Document Browser Tutorial .....</b>	<b>327</b>
45.1 Creating the DocumentBrowser Project.....	327
45.2 Declaring the Supported File Types.....	327
45.3 Completing the didRequestDocumentCreationWithHandler Method.....	329
45.4 Finishing the UIDocument Subclass .....	331

## Table of Contents

45.5 Modifying the Document View Controller .....	331
45.6 Testing the Document Browser App .....	333
45.7 Summary .....	333
<b>46. Synchronizing iOS 17 Key-Value Data using iCloud .....</b>	<b>335</b>
46.1 An Overview of iCloud Key-Value Data Storage .....	335
46.2 Sharing Data Between Apps .....	335
46.3 Data Storage Restrictions .....	336
46.4 Conflict Resolution .....	336
46.5 Receiving Notification of Key-Value Changes .....	336
46.6 An iCloud Key-Value Data Storage Example .....	336
46.7 Enabling the App for iCloud Key-Value Data Storage .....	336
46.8 Designing the User Interface .....	337
46.9 Implementing the View Controller .....	337
46.10 Modifying the viewDidLoad Method .....	338
46.11 Implementing the Notification Method .....	338
46.12 Implementing the saveData Method .....	339
46.13 Testing the App .....	339
46.14 Summary .....	339
<b>47. iOS 17 Database Implementation using SQLite .....</b>	<b>341</b>
47.1 What is SQLite? .....	341
47.2 Structured Query Language (SQL) .....	341
47.3 Trying SQLite on macOS .....	341
47.4 Preparing an iOS App Project for SQLite Integration .....	343
47.5 SQLite, Swift, and Wrappers .....	343
47.6 Key FMDB Classes .....	343
47.7 Creating and Opening a Database .....	343
47.8 Creating a Database Table .....	344
47.9 Extracting Data from a Database Table .....	344
47.10 Closing an SQLite Database .....	344
47.11 Summary .....	345
<b>48. An Example SQLite-based iOS 17 App using Swift and FMDB .....</b>	<b>347</b>
48.1 About the Example SQLite App .....	347
48.2 Creating and Preparing the SQLite App Project .....	347
48.3 Checking Out the FMDB Source Code .....	347
48.4 Designing the User Interface .....	349
48.5 Creating the Database and Table .....	350
48.6 Implementing the Code to Save Data to the SQLite Database .....	351
48.7 Implementing Code to Extract Data from the SQLite Database .....	352
48.8 Building and Running the App .....	353
48.9 Summary .....	354
<b>49. Working with iOS 17 Databases using Core Data .....</b>	<b>355</b>
49.1 The Core Data Stack .....	355
49.2 Persistent Container .....	356
49.3 Managed Objects .....	356
49.4 Managed Object Context .....	356
49.5 Managed Object Model .....	356



49.6 Persistent Store Coordinator.....	356
49.7 Persistent Object Store.....	357
49.8 Defining an Entity Description .....	357
49.9 Initializing the Persistent Container.....	357
49.10 Obtaining the Managed Object Context.....	358
49.11 Getting an Entity Description .....	358
49.12 Setting the Attributes of a Managed Object.....	358
49.13 Saving a Managed Object.....	358
49.14 Fetching Managed Objects.....	358
49.15 Retrieving Managed Objects based on Criteria .....	359
49.16 Accessing the Data in a Retrieved Managed Object.....	359
49.17 Summary .....	359
<b>50. An iOS 17 Core Data Tutorial.....</b>	<b>361</b>
50.1 The Core Data Example App .....	361
50.2 Creating a Core Data-based App .....	361
50.3 Creating the Entity Description .....	361
50.4 Designing the User Interface .....	362
50.5 Initializing the Persistent Container.....	363
50.6 Saving Data to the Persistent Store using Core Data.....	363
50.7 Retrieving Data from the Persistent Store using Core Data .....	364
50.8 Building and Running the Example App .....	365
50.9 Summary .....	365
<b>51. An Introduction to CloudKit Data Storage on iOS 17.....</b>	<b>367</b>
51.1 An Overview of CloudKit .....	367
51.2 CloudKit Containers.....	367
51.3 CloudKit Public Database .....	367
51.4 CloudKit Private Databases .....	367
51.5 Data Storage and Transfer Quotas .....	368
51.6 CloudKit Records.....	368
51.7 CloudKit Record IDs .....	370
51.8 CloudKit References .....	370
51.9 CloudKit Assets.....	370
51.10 Record Zones.....	371
51.11 CloudKit Sharing .....	371
51.12 CloudKit Subscriptions .....	371
51.13 Obtaining iCloud User Information.....	372
51.14 CloudKit Console.....	372
51.15 Summary .....	373
<b>52. An Introduction to CloudKit Sharing .....</b>	<b>375</b>
52.1 Understanding CloudKit Sharing .....	375
52.2 Preparing for CloudKit Sharing .....	375
52.3 The CKShare Class .....	375
52.4 The UICloudSharingController Class .....	376
52.5 Accepting a CloudKit Share.....	379
52.6 Fetching a Shared Record.....	380
52.7 Summary .....	381

<b>53. An iOS 17 CloudKit Example .....</b>	<b>383</b>
53.1 About the Example CloudKit Project.....	383
53.2 Creating the CloudKit Example Project.....	383
53.3 Designing the User Interface .....	384
53.4 Establishing Outlets and Actions .....	386
53.5 Implementing the notifyUser Method .....	387
53.6 Accessing the Private Database .....	387
53.7 Hiding the Keyboard .....	389
53.8 Implementing the selectPhoto method .....	389
53.9 Saving a Record to the Cloud Database .....	390
53.10 Testing the Record Saving Method .....	392
53.11 Reviewing the Saved Data in the CloudKit Console .....	392
53.12 Searching for Cloud Database Records.....	394
53.13 Updating Cloud Database Records.....	395
53.14 Deleting a Cloud Record.....	396
53.15 Testing the App.....	396
53.16 Summary .....	396
<b>54. An iOS 17 CloudKit Sharing Example.....</b>	<b>397</b>
54.1 Preparing the Project for CloudKit Sharing .....	397
54.2 Adding the Share Button.....	397
54.3 Creating the CloudKit Share.....	398
54.4 Accepting a CloudKit Share.....	399
54.5 Fetching the Shared Record.....	400
54.6 Testing the CloudKit Share Example.....	401
54.7 Summary .....	402
<b>55. An Overview of iOS 17 Multitouch, Taps, and Gestures .....</b>	<b>403</b>
55.1 The Responder Chain .....	403
55.2 Forwarding an Event to the Next Responder .....	403
55.3 Gestures .....	404
55.4 Taps .....	404
55.5 Touches.....	404
55.6 Touch Notification Methods.....	404
55.6.1 touchesBegan method .....	404
55.6.2 touchesMoved method .....	404
55.6.3 touchesEnded method.....	404
55.6.4 touchesCancelled method.....	405
55.7 Touch Prediction .....	405
55.8 Touch Coalescing .....	405
55.9 Summary .....	405
<b>56. An Example iOS 17 Touch, Multitouch, and Tap App.....</b>	<b>407</b>
56.1 The Example iOS Tap and Touch App.....	407
56.2 Creating the Example iOS Touch Project .....	407
56.3 Designing the User Interface .....	407
56.4 Enabling Multitouch on the View.....	408
56.5 Implementing the touchesBegan Method.....	408
56.6 Implementing the touchesMoved Method .....	409

56.7 Implementing the touchesEnded Method .....	409
56.8 Getting the Coordinates of a Touch.....	409
56.9 Building and Running the Touch Example App .....	410
56.10 Checking for Touch Predictions .....	410
56.11 Accessing Coalesced Touches.....	411
56.12 Summary .....	411
<b>57. Detecting iOS 17 Touch Screen Gesture Motions.....</b>	<b>413</b>
57.1 The Example iOS 17 Gesture App.....	413
57.2 Creating the Example Project.....	413
57.3 Designing the App User Interface.....	413
57.4 Implementing the touchesBegan Method.....	414
57.5 Implementing the touchesMoved Method .....	414
57.6 Implementing the touchesEnded Method .....	415
57.7 Building and Running the Gesture Example.....	415
57.8 Summary .....	415
<b>58. Identifying Gestures using iOS 17 Gesture Recognizers.....</b>	<b>417</b>
58.1 The UIGestureRecognizer Class.....	417
58.2 Recognizer Action Messages .....	418
58.3 Discrete and Continuous Gestures .....	418
58.4 Obtaining Data from a Gesture.....	418
58.5 Recognizing Tap Gestures.....	418
58.6 Recognizing Pinch Gestures .....	418
58.7 Detecting Rotation Gestures.....	418
58.8 Recognizing Pan and Dragging Gestures.....	419
58.9 Recognizing Swipe Gestures.....	419
58.10 Recognizing Long Touch (Touch and Hold) Gestures.....	419
58.11 Summary .....	419
<b>59. An iOS 17 Gesture Recognition Tutorial .....</b>	<b>421</b>
59.1 Creating the Gesture Recognition Project .....	421
59.2 Designing the User Interface .....	421
59.3 Implementing the Action Methods .....	422
59.4 Testing the Gesture Recognition Application.....	423
59.5 Summary .....	423
<b>60. Implementing Touch ID and Face ID Authentication in iOS 17 Apps .....</b>	<b>425</b>
60.1 The Local Authentication Framework.....	425
60.2 Checking for Biometric Authentication Availability.....	425
60.3 Identifying Authentication Options .....	426
60.4 Evaluating Biometric Policy.....	426
60.5 A Biometric Authentication Example Project.....	427
60.6 Checking for Biometric Availability .....	428
60.7 Seeking Biometric Authentication .....	429
60.8 Adding the Face ID Privacy Statement .....	431
60.9 Testing the App.....	431
60.10 Summary .....	433
<b>61. Drawing iOS 17 2D Graphics with Core Graphics .....</b>	<b>435</b>

## Table of Contents

61.1	Introducing Core Graphics and Quartz 2D.....	435
61.2	The draw Method .....	435
61.3	Points, Coordinates, and Pixels .....	435
61.4	The Graphics Context .....	436
61.5	Working with Colors in Quartz 2D .....	436
61.6	Summary .....	437
<b>62.</b>	<b>Interface Builder Live Views and iOS 17 Embedded Frameworks .....</b>	<b>439</b>
62.1	Embedded Frameworks.....	439
62.2	Interface Builder Live Views.....	439
62.3	Creating the Example Project.....	440
62.4	Adding an Embedded Framework.....	441
62.5	Implementing the Drawing Code in the Framework.....	442
62.6	Making the View Designable.....	443
62.7	Making Variables Inspectable.....	444
62.8	Summary .....	444
<b>63.</b>	<b>An iOS 17 Graphics Tutorial using Core Graphics and Core Image .....</b>	<b>445</b>
63.1	The iOS Drawing Example App .....	445
63.2	Creating the New Project .....	445
63.3	Creating the UIView Subclass .....	445
63.4	Locating the draw Method in the UIView Subclass .....	446
63.5	Drawing a Line .....	446
63.6	Drawing Paths.....	447
63.7	Drawing a Rectangle.....	448
63.8	Drawing an Ellipse or Circle.....	449
63.9	Filling a Path with a Color .....	449
63.10	Drawing an Arc .....	451
63.11	Drawing a Cubic Bézier Curve.....	451
63.12	Drawing a Quadratic Bézier Curve .....	452
63.13	Dashed Line Drawing.....	453
63.14	Drawing Shadows.....	454
63.15	Drawing Gradients.....	454
63.16	Drawing an Image into a Graphics Context.....	458
63.17	Image Filtering with the Core Image Framework.....	460
63.18	Summary.....	461
<b>64.</b>	<b>iOS 17 Animation using UIViewPropertyAnimator.....</b>	<b>463</b>
64.1	The Basics of UIKit Animation .....	463
64.2	Understanding Animation Curves .....	464
64.3	Performing Affine Transformations .....	464
64.4	Combining Transformations .....	465
64.5	Creating the Animation Example App.....	465
64.6	Implementing the Variables.....	465
64.7	Drawing in the UIView .....	466
64.8	Detecting Screen Touches and Performing the Animation.....	466
64.9	Building and Running the Animation App .....	468
64.10	Implementing Spring Timing.....	468
64.11	Summary.....	469

<b>65. iOS 17 UIKit Dynamics – An Overview .....</b>	<b>471</b>
65.1 Understanding UIKit Dynamics .....	471
65.2 The UIKit Dynamics Architecture .....	471
65.2.1 Dynamic Items .....	471
65.2.2 Dynamic Behaviors.....	472
65.2.3 The Reference View.....	472
65.2.4 The Dynamic Animator .....	472
65.3 Implementing UIKit Dynamics in an iOS App.....	473
65.4 Dynamic Animator Initialization .....	473
65.5 Configuring Gravity Behavior.....	473
65.6 Configuring Collision Behavior .....	474
65.7 Configuring Attachment Behavior .....	475
65.8 Configuring Snap Behavior.....	476
65.9 Configuring Push Behavior .....	476
65.10 The UIDynamicItemBehavior Class.....	477
65.11 Combining Behaviors to Create a Custom Behavior .....	478
65.12 Summary .....	478
<b>66. An iOS 17 UIKit Dynamics Tutorial.....</b>	<b>479</b>
66.1 Creating the UIKit Dynamics Example Project.....	479
66.2 Adding the Dynamic Items.....	479
66.3 Creating the Dynamic Animator Instance.....	480
66.4 Adding Gravity to the Views .....	481
66.5 Implementing Collision Behavior.....	482
66.6 Attaching a View to an Anchor Point.....	483
66.7 Implementing a Spring Attachment Between two Views .....	485
66.8 Summary .....	486
<b>67. Integrating Maps into iOS 17 Apps using MKMapItem.....</b>	<b>487</b>
67.1 MKMapItem and MKPlacemark Classes.....	487
67.2 An Introduction to Forward and Reverse Geocoding .....	487
67.3 Creating MKPlacemark Instances.....	489
67.4 Working with MKMapItem .....	489
67.5 MKMapItem Options and Configuring Directions.....	490
67.6 Adding Item Details to an MKMapItem .....	491
67.7 Summary .....	492
<b>68. An Example iOS 17 MKMapItem App.....</b>	<b>493</b>
68.1 Creating the MapItem Project .....	493
68.2 Designing the User Interface .....	493
68.3 Converting the Destination using Forward Geocoding .....	494
68.4 Launching the Map .....	495
68.5 Building and Running the App .....	496
68.6 Summary .....	497
<b>69. Getting Location Information using the iOS 17 Core Location Framework.....</b>	<b>499</b>
69.1 The Core Location Manager .....	499
69.2 Requesting Location Access Authorization .....	499
69.3 Configuring the Desired Location Accuracy.....	500

## Table of Contents

69.4 Configuring the Distance Filter.....	500
69.5 Continuous Background Location Updates .....	501
69.6 The Location Manager Delegate.....	501
69.7 Starting and Stopping Location Updates .....	502
69.8 Obtaining Location Information from CLLocation Objects.....	502
69.8.1 Longitude and Latitude .....	503
69.8.2 Accuracy.....	503
69.8.3 Altitude.....	503
69.9 Getting the Current Location .....	503
69.10 Calculating Distances .....	503
69.11 Summary.....	503
<b>70. An Example iOS 17 Location App.....</b>	<b>505</b>
70.1 Creating the Example iOS 17 Location Project.....	505
70.2 Designing the User Interface .....	505
70.3 Configuring the CLLocationManager Object .....	506
70.4 Setting up the Usage Description Keys .....	507
70.5 Implementing the startWhenInUse Method .....	507
70.6 Implementing the startAlways Method.....	507
70.7 Implementing the resetDistance Method .....	508
70.8 Implementing the App Delegate Methods.....	508
70.9 Building and Running the Location App.....	509
70.10 Adding Continuous Background Location Updates .....	510
70.11 Summary.....	512
<b>71. Working with Maps on iOS 17 with MapKit and the MKMapView Class .....</b>	<b>513</b>
71.1 About the MapKit Framework .....	513
71.2 Understanding Map Regions .....	513
71.3 Getting Transit ETA Information .....	514
71.4 About the MKMapView Tutorial .....	514
71.5 Creating the Map Project .....	514
71.6 Adding the Navigation Controller .....	515
71.7 Creating the MKMapView Instance and Toolbar .....	515
71.8 Obtaining Location Information Permission.....	518
71.9 Setting up the Usage Description Keys .....	518
71.10 Configuring the Map View .....	518
71.11 Changing the MapView Region .....	519
71.12 Changing the Map Type.....	519
71.13 Testing the MapView App.....	519
71.14 Updating the Map View based on User Movement.....	520
71.15 Summary.....	521
<b>72. Working with MapKit Local Search in iOS 17.....</b>	<b>523</b>
72.1 An Overview of iOS Local Search.....	523
72.2 Adding Local Search to the MapSample App.....	524
72.3 Adding the Local Search Text Field .....	524
72.4 Performing the Local Search .....	526
72.5 Testing the App.....	527
72.6 Customized Annotation Markers .....	528

72.7 Annotation Marker Clustering.....	532
72.8 Summary .....	533
<b>73. Using MKDirections to get iOS 17 Map Directions and Routes .....</b>	<b>535</b>
73.1 An Overview of MKDirections .....	535
73.2 Adding Directions and Routes to the MapSample App .....	536
73.3 Adding the New Classes to the Project .....	537
73.4 Configuring the Results Table View .....	537
73.5 Implementing the Result Table View Segue .....	539
73.6 Adding the Route Scene .....	540
73.7 Identifying the User's Current Location.....	541
73.8 Getting the Route and Directions .....	542
73.9 Establishing the Route Segue .....	544
73.10 Testing the App.....	544
73.11 Summary .....	545
<b>74. Accessing the iOS 17 Camera and Photo Library .....</b>	<b>547</b>
74.1 The UIImagePickerController Class.....	547
74.2 Creating and Configuring a UIImagePickerController Instance .....	547
74.3 Configuring the UIImagePickerController Delegate .....	548
74.4 Detecting Device Capabilities .....	549
74.5 Saving Movies and Images .....	549
74.6 Summary .....	550
<b>75. An Example iOS 17 Camera App .....</b>	<b>551</b>
75.1 An Overview of the App .....	551
75.2 Creating the Camera Project .....	551
75.3 Designing the User Interface .....	551
75.4 Implementing the Action Methods .....	552
75.5 Writing the Delegate Methods.....	553
75.6 Seeking Camera and Photo Library Access .....	555
75.7 Building and Running the App .....	555
75.8 Summary .....	556
<b>76. iOS 17 Video Playback using AVPlayer and AVPlayerViewController .....</b>	<b>557</b>
76.1 The AVPlayer and AVPlayerViewController Classes .....	557
76.2 The iOS Movie Player Example App.....	557
76.3 Designing the User Interface .....	557
76.4 Initializing Video Playback.....	557
76.5 Build and Run the App .....	558
76.6 Creating an AVPlayerViewController Instance from Code .....	559
76.7 Summary .....	559
<b>77. An iOS 17 Multitasking Picture-in-Picture Tutorial.....</b>	<b>561</b>
77.1 An Overview of Picture-in-Picture Multitasking .....	561
77.2 Adding Picture-in-Picture Support to the AVPlayerDemo App .....	562
77.3 Adding the Navigation Controller .....	563
77.4 Setting the Audio Session Category.....	563
77.5 Implementing the Delegate.....	564
77.6 Opting Out of Picture-in-Picture Support.....	566

## Table of Contents

77.7 Additional Delegate Methods .....	566
77.8 Summary .....	566
<b>78. An Introduction to Extensions in iOS 17.....</b>	<b>567</b>
78.1 iOS Extensions – An Overview .....	567
78.2 Extension Types.....	567
78.2.1 Share Extension .....	567
78.2.2 Action Extension.....	568
78.2.3 Photo Editing Extension .....	568
78.2.4 Document Provider Extension.....	569
78.2.5 Custom Keyboard Extension.....	569
78.2.6 Audio Unit Extension.....	569
78.2.7 Shared Links Extension.....	569
78.2.8 Content Blocking Extension.....	569
78.2.9 Sticker Pack Extension .....	569
78.2.10 iMessage Extension.....	569
78.2.11 Intents Extension.....	569
78.3 Creating Extensions .....	570
78.4 Summary .....	570
<b>79. Creating an iOS 17 Photo Editing Extension.....</b>	<b>571</b>
79.1 Creating a Photo Editing Extension .....	571
79.2 Accessing the Photo Editing Extension.....	572
79.3 Configuring the Info.plist File .....	573
79.4 Designing the User Interface .....	574
79.5 The PHContentEditingController Protocol.....	575
79.6 Photo Extensions and Adjustment Data .....	575
79.7 Receiving the Content .....	576
79.8 Implementing the Filter Actions.....	577
79.9 Returning the Image to the Photos App.....	579
79.10 Testing the App.....	581
79.11 Summary.....	582
<b>80. Creating an iOS 17 Action Extension .....</b>	<b>583</b>
80.1 An Overview of Action Extensions .....	583
80.2 About the Action Extension Example .....	584
80.3 Creating the Action Extension Project.....	584
80.4 Adding the Action Extension Target .....	584
80.5 Changing the Extension Display Name .....	585
80.6 Designing the Action Extension User Interface.....	585
80.7 Receiving the Content .....	586
80.8 Returning the Modified Data to the Host App.....	588
80.9 Testing the Extension.....	589
80.10 Summary.....	591
<b>81. Receiving Data from an iOS 17 Action Extension.....</b>	<b>593</b>
81.1 Creating the Example Project.....	593
81.2 Designing the User Interface .....	593
81.3 Importing the Mobile Core Services Framework .....	594
81.4 Adding an Action Button to the App.....	594



81.5 Receiving Data from an Extension .....	595
81.6 Testing the App.....	596
81.7 Summary .....	596
<b>82. An Introduction to Building iOS 17 Message Apps .....</b>	<b>597</b>
82.1 Introducing Message Apps.....	597
82.2 Types of Message Apps .....	598
82.3 The Key Messages Framework Classes .....	599
82.3.1 MSMessagesAppViewController .....	599
82.3.2 MSConversation.....	599
82.3.3 MSMessage.....	600
82.3.4 MSMessageTemplateLayout.....	600
82.4 Sending Simple Messages.....	601
82.5 Creating an MSMessage Message.....	602
82.6 Receiving a Message.....	602
82.7 Supported Message App Platforms .....	603
82.8 Summary .....	603
<b>83. An iOS 17 Interactive Message App Tutorial.....</b>	<b>605</b>
83.1 About the Example Message App Project .....	605
83.2 Creating the MessageApp Project .....	605
83.3 Designing the MessageApp User Interface .....	607
83.4 Creating the Outlet Collection .....	611
83.5 Creating the Game Model.....	612
83.6 Responding to Button Selections .....	612
83.7 Preparing the Message URL.....	613
83.8 Preparing and Inserting the Message .....	614
83.9 Message Receipt Handling .....	615
83.10 Setting the Message Image .....	616
83.11 Summary .....	617
<b>84. An Introduction to Machine Learning on iOS .....</b>	<b>619</b>
84.1 Datasets and Machine Learning Models.....	619
84.2 Machine Learning in Xcode and iOS .....	619
84.3 iOS Machine Learning Frameworks.....	620
84.4 Summary .....	620
<b>85. Using Create ML to Build an Image Classification Model .....</b>	<b>621</b>
85.1 About the Dataset.....	621
85.2 Creating the Machine Learning Model .....	621
85.3 Importing the Training and Testing Data .....	623
85.4 Training and Testing the Model .....	623
85.5 Summary .....	625
<b>86. An iOS Vision and Core ML Image Classification Tutorial .....</b>	<b>627</b>
86.1 Preparing the Project.....	627
86.2 Adding the Model .....	627
86.3 Modifying the User Interface.....	627
86.4 Initializing the Core ML Request.....	628
86.5 Handling the Results of the Core ML Request.....	629

## Table of Contents

86.6 Making the Classification Request.....	630
86.7 Testing the App.....	631
86.8 Summary .....	632
<b>87. An iOS 17 Local Notification Tutorial.....</b>	<b>633</b>
87.1 Creating the Local Notification App Project .....	633
87.2 Requesting Notification Authorization .....	633
87.3 Designing the User Interface .....	634
87.4 Creating the Message Content.....	635
87.5 Specifying a Notification Trigger .....	635
87.6 Creating the Notification Request.....	635
87.7 Adding the Request.....	636
87.8 Testing the Notification .....	636
87.9 Receiving Notifications in the Foreground.....	637
87.10 Adding Notification Actions.....	638
87.11 Handling Notification Actions .....	639
87.12 Hidden Notification Content.....	640
87.13 Managing Notifications.....	642
87.14 Summary.....	643
<b>88. Playing Audio on iOS 17 using AVAudioPlayer .....</b>	<b>645</b>
88.1 Supported Audio Formats.....	645
88.2 Receiving Playback Notifications.....	645
88.3 Controlling and Monitoring Playback .....	646
88.4 Creating the Audio Example App .....	646
88.5 Adding an Audio File to the Project Resources .....	646
88.6 Designing the User Interface .....	646
88.7 Implementing the Action Methods .....	648
88.8 Creating and Initializing the AVAudioPlayer Object .....	648
88.9 Implementing the AVAudioPlayerDelegate Protocol Methods .....	649
88.10 Building and Running the App .....	649
88.11 Summary.....	649
<b>89. Recording Audio on iOS 17 with AVAudioRecorder.....</b>	<b>651</b>
89.1 An Overview of the AVAudioRecorder Tutorial.....	651
89.2 Creating the Recorder Project .....	651
89.3 Configuring the Microphone Usage Description .....	651
89.4 Designing the User Interface .....	652
89.5 Creating the AVAudioRecorder Instance.....	653
89.6 Implementing the Action Methods .....	654
89.7 Implementing the Delegate Methods .....	655
89.8 Testing the App.....	655
89.9 Summary .....	655
<b>90. An iOS 17 Speech Recognition Tutorial .....</b>	<b>657</b>
90.1 An Overview of Speech Recognition in iOS.....	657
90.2 Speech Recognition Authorization .....	657
90.3 Transcribing Recorded Audio .....	658
90.4 Transcribing Live Audio.....	658
90.5 An Audio File Speech Recognition Tutorial .....	658

90.6 Modifying the User Interface .....	658
90.7 Adding the Speech Recognition Permission .....	659
90.8 Seeking Speech Recognition Authorization .....	660
90.9 Performing the Transcription .....	661
90.10 Testing the App .....	661
90.11 Summary .....	661
<b>91. An iOS 17 Real-Time Speech Recognition Tutorial .....</b>	<b>663</b>
91.1 Creating the Project .....	663
91.2 Designing the User Interface .....	663
91.3 Adding the Speech Recognition Permission .....	664
91.4 Requesting Speech Recognition Authorization .....	664
91.5 Declaring and Initializing the Speech and Audio Objects .....	665
91.6 Starting the Transcription .....	665
91.7 Implementing the stopTranscribing Method .....	669
91.8 Testing the App .....	669
91.9 Summary .....	669
<b>92. An Introduction to iOS 17 Sprite Kit Programming .....</b>	<b>671</b>
92.1 What is Sprite Kit? .....	671
92.2 The Key Components of a Sprite Kit Game .....	671
92.2.1 Sprite Kit View .....	671
92.2.2 Scenes .....	671
92.2.3 Nodes .....	672
92.2.4 Physics Bodies .....	672
92.2.5 Physics World .....	673
92.2.6 Actions .....	673
92.2.7 Transitions .....	673
92.2.8 Texture Atlas .....	673
92.2.9 Constraints .....	673
92.3 An Example Sprite Kit Game Hierarchy .....	673
92.4 The Sprite Kit Game Rendering Loop .....	674
92.5 The Sprite Kit Level Editor .....	675
92.6 Summary .....	675
<b>93. An iOS 17 Sprite Kit Level Editor Game Tutorial .....</b>	<b>677</b>
93.1 About the Sprite Kit Demo Game .....	677
93.2 Creating the SpriteKitDemo Project .....	678
93.3 Reviewing the SpriteKit Game Template Project .....	678
93.4 Restricting Interface Orientation .....	679
93.5 Modifying the GameScene SpriteKit Scene File .....	679
93.6 Creating the Archery Scene .....	682
93.7 Transitioning to the Archery Scene .....	682
93.8 Adding the Texture Atlas .....	683
93.9 Designing the Archery Scene .....	684
93.10 Preparing the Archery Scene .....	686
93.11 Preparing the Animation Texture Atlas .....	686
93.12 Creating the Named Action Reference .....	688
93.13 Triggering the Named Action from the Code .....	688

## Table of Contents

93.14 Creating the Arrow Sprite Node .....	688
93.15 Shooting the Arrow .....	689
93.16 Adding the Ball Sprite Node .....	690
93.17 Summary .....	691
<b>94. An iOS 17 Sprite Kit Collision Handling Tutorial.....</b>	<b>693</b>
94.1 Defining the Category Bit Masks .....	693
94.2 Assigning the Category Masks to the Sprite Nodes.....	693
94.3 Configuring the Collision and Contact Masks.....	694
94.4 Implementing the Contact Delegate .....	695
94.5 Game Over .....	696
94.6 Summary .....	697
<b>95. An iOS 17 Sprite Kit Particle Emitter Tutorial .....</b>	<b>699</b>
95.1 What is the Particle Emitter? .....	699
95.2 The Particle Emitter Editor .....	699
95.3 The SKEmitterNode Class.....	699
95.4 Using the Particle Emitter Editor .....	700
95.5 Particle Emitter Node Properties .....	701
95.5.1 Background .....	701
95.5.2 Particle Texture.....	701
95.5.3 Particle Birthrate .....	702
95.5.4 Particle Life Cycle.....	702
95.5.5 Particle Position Range.....	702
95.5.6 Angle .....	702
95.5.7 Particle Speed.....	702
95.5.8 Particle Acceleration .....	702
95.5.9 Particle Scale .....	702
95.5.10 Particle Rotation .....	702
95.5.11 Particle Color .....	702
95.5.12 Particle Blend Mode .....	703
95.6 Experimenting with the Particle Emitter Editor .....	703
95.7 Bursting a Ball using Particle Emitter Effects.....	704
95.8 Adding the Burst Particle Emitter Effect .....	705
95.9 Adding an Audio Action .....	706
95.10 Summary .....	707
<b>96. Preparing and Submitting an iOS 17 Application to the App Store .....</b>	<b>709</b>
96.1 Verifying the iOS Distribution Certificate .....	709
96.2 Adding App Icons .....	711
96.3 Assign the Project to a Team .....	712
96.4 Archiving the Application for Distribution .....	712
96.5 Configuring the Application in App Store Connect.....	713
96.6 Validating and Submitting the Application .....	714
96.7 Configuring and Submitting the App for Review .....	715
<b>Index .....</b>	<b>717</b>

## 1. Start Here

This book aims to teach the skills necessary to create iOS apps using the iOS 17 SDK, UIKit, Xcode 15 Storyboards, and the Swift programming language.

Beginning with the basics, this book outlines the steps necessary to set up an iOS development environment. Next, an introduction to the architecture of iOS 17 and programming in Swift is provided, followed by an in-depth look at the design of iOS apps 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 include Auto Layout, local map search, user interface animation using UIKit dynamics, iMessage app development, 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, CloudKit data storage, and the document browser.

Other features of iOS 17 and Xcode 15 are also covered in detail, including iOS machine learning features.

The aim of this book, therefore, is to teach you the skills necessary to build your own apps for iOS 17. Assuming you are ready to download the iOS 17 SDK and Xcode 15, have a Mac, and some ideas for some apps to develop, you are ready to get started.

### 1.1 Source Code Download

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

<https://www.payloadbooks.com/product/ios17xcode/>

### 1.2 Feedback

We want you to be satisfied with your purchase of this book. Therefore, if you find any errors in the book or have any comments, questions, or concerns, please contact us at [info@payloadbooks.com](mailto:info@payloadbooks.com).

### 1.3 Errata

While we make every effort to ensure the accuracy of the content of this book, inevitably, 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.payloadbooks.com/ios-17-xcode-errata/>

If you find an error not listed in the errata, please email our technical support team at [info@payloadbooks.com](mailto:info@payloadbooks.com).

### 1.4 Find more books

Visit our website to view our complete book catalog at <https://www.payloadbooks.com>.



## 2. Joining the Apple Developer Program

The first step in the process of learning to develop iOS 17 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 15 and the iOS 17 SDK

The latest versions of both the iOS SDK and Xcode can be downloaded free of charge from the macOS 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.

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 Siri integration, 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). Membership also includes access to the Apple Developer forums; an invaluable resource both 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 Xcode, macOS 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 your apps. As to whether 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 apps or have yet to come up with a compelling idea for an app to develop then much of what you need is provided without program membership. As your skill level increases and your ideas for apps to develop take shape you can, after all, always enroll in the developer program later.

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 Siri support, iCloud storage, 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 apps 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 in to 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 in to the Member Center again and note that access is now available to a wide range of options and resources, as illustrated in Figure 2-1:

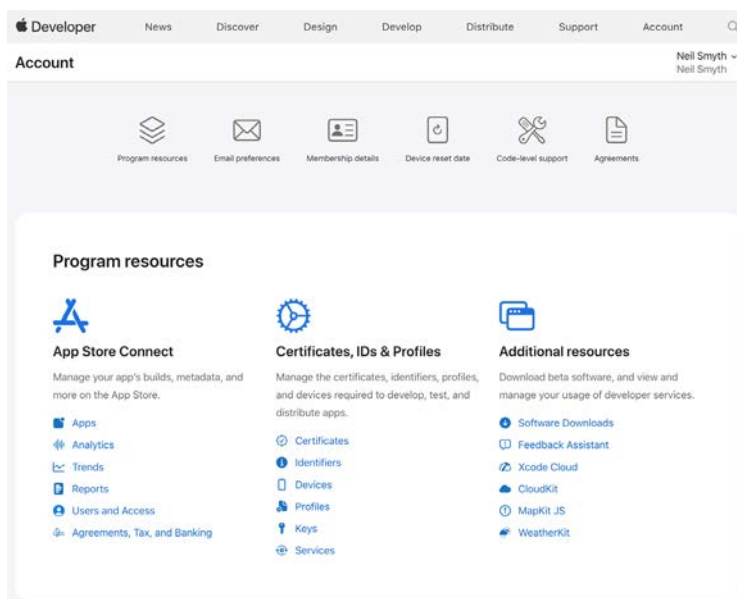


Figure 2-1



## 2.5 Summary

An important early step in the iOS 17 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 17 SDK and Xcode 15 development environment.



## 3. Installing Xcode 15 and the iOS 17 SDK

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

All of the examples in this book are based on Xcode version 15 and use features unavailable in earlier Xcode versions. This chapter will cover the steps involved in installing Xcode 15 and the iOS 17 SDK on macOS.

### 3.1 Identifying Your macOS Version

When developing with iOS apps, the Xcode 15 environment requires a system running macOS Ventura 13.5 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 *macOS* line:



Figure 3-1

If the “About This Mac” dialog does not indicate that macOS 13.5 or later is running, click on the *Software Update...* button to download and install the appropriate operating system upgrades.

### 3.2 Installing Xcode 15 and the iOS 17 SDK

The best way to obtain the latest Xcode and iOS SDK versions 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. This will install both Xcode and the iOS SDK.

### 3.3 Starting Xcode

Having successfully installed the SDK and Xcode, the next step is to launch it so we are ready to start development work. To start up Xcode, open the macOS Finder and search for *Xcode*. Since you will be frequently using this tool, take this opportunity to drag and drop it onto 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.

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

Whether or not you 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* -> *Settings...* menu option followed by 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 password before clicking on the *Sign In* button to add the account to the preferences.

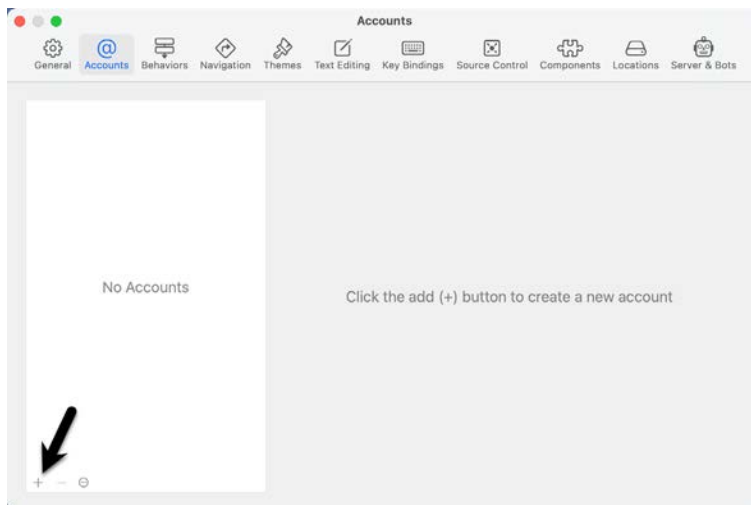


Figure 3-3

### 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 to display a list of available signing identity types. To create a signing identity, simply click on the + button highlighted in Figure 3-4 and make the appropriate selection from the menu:

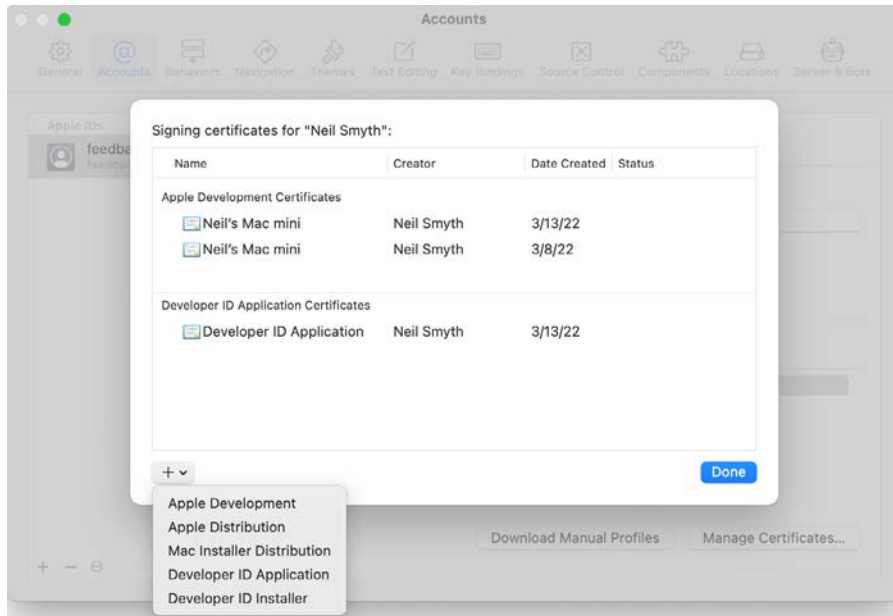


Figure 3-4

If the Apple ID has been used to enroll in the Apple Developer program, the option to create an *Apple Distribution* certificate will appear in the menu which will, when clicked, generate the signing identity required to submit the app to the Apple App Store. You will also need to create a *Developer ID Application* certificate if you plan to integrate features such as iCloud and Siri into your app projects. If you have not yet signed up for the Apple Developer program, select the *Apple Development* option to allow apps to be tested during development.

### 3.6 Summary

This book was written using Xcode 15 and the iOS 17 SDK running on macOS 14.0 (Sonoma). Before beginning iOS development, the first step is to install Xcode and configure it with your Apple ID via the accounts section of the Preferences screen. Once these steps have been performed, a development certificate must be generated which will be used to sign apps developed within Xcode. This will allow you to build and test your apps on physical iOS-based devices.

When you are ready to upload your finished app to the App Store, you will also need to generate a distribution certificate, a process requiring membership in the Apple Developer Program as outlined in the previous chapter.

Having installed the iOS SDK and successfully launched Xcode 15, we can now look at Xcode in more detail, starting with a guided tour.



## 4. A Guided Tour of Xcode 15

Just about every activity related to developing and testing iOS apps 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 key areas that comprise the Xcode development environment. In the course of providing this overview, the chapter will also work through creating a straightforward iOS app project to display a label that 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 app.

### 4.1 Starting Xcode 15

As with all iOS examples in this book, the development of our example will take place within the Xcode 15 development environment. Therefore, if you have not already installed this tool with the latest iOS SDK, refer first to the chapter of this book. Then, assuming that the installation is complete, launch Xcode either by clicking on the icon on the dock (assuming you created one) or using the macOS Finder to locate Xcode in the Applications folder of your system.

When launched for the first time the screen illustrated in Figure 4-1 will appear by default:

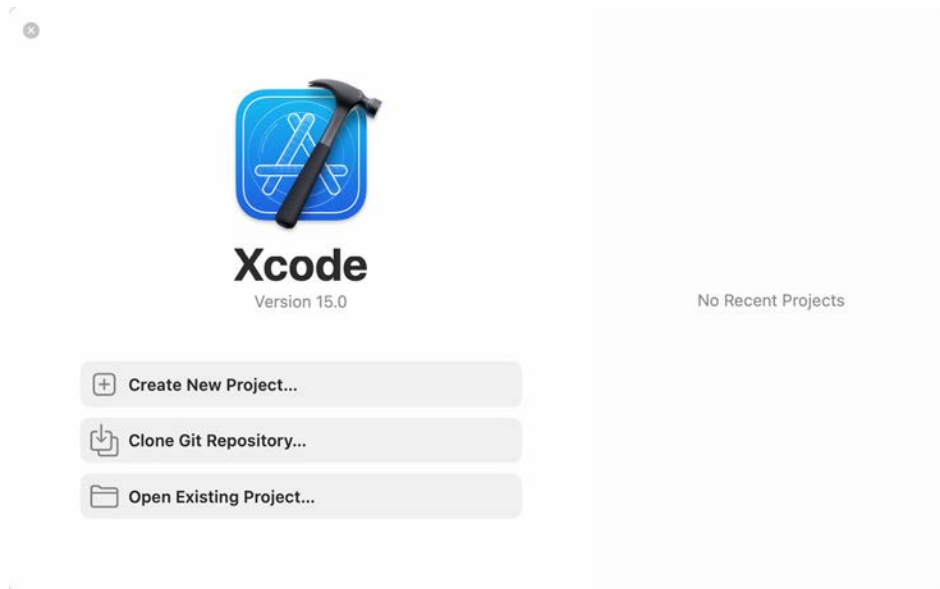


Figure 4-1

If you do not see this window, select the *Window -> Welcome to Xcode* menu option to display it. Within this window, click on the option to *Create a New Project*. This selection will display the main Xcode project window together with the *project template* panel, where we can select a template matching the type of project we want to develop. Within this window, select the *iOS* tab so that the template panel appears as follows:

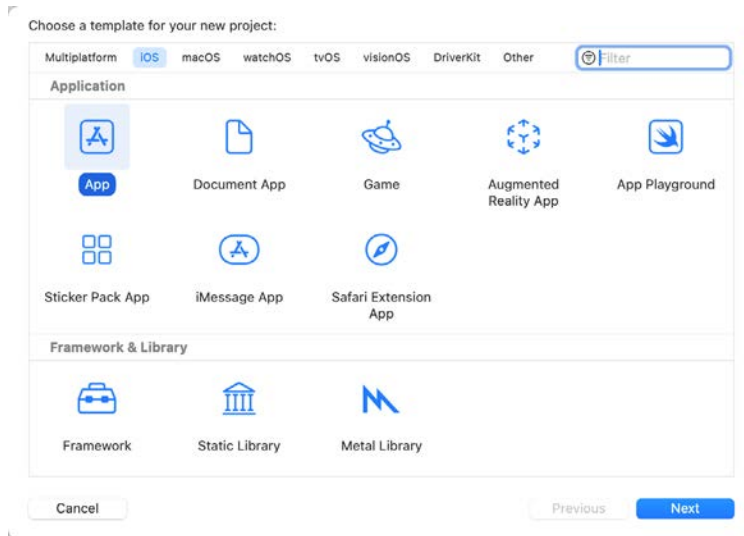


Figure 4-2

The toolbar on the window's top edge allows for selecting the target platform, providing options to develop an app for iOS, watchOS, visionOS, tvOS, or macOS. An option is also available for creating multiplatform apps using SwiftUI.

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

- **App** – This creates a basic template for an app containing a single view and corresponding view controller.
- **Document App** – Creates a project intended to use the iOS document browser. The document browser provides a visual environment where the user can navigate and manage local and cloud-based files from within an iOS app.
- **Game** – Creates a project configured to take advantage of Sprite Kit, Scene Kit, OpenGL ES, and Metal for developing 2D and 3D games.
- **Augmented Reality App** – Creates a template project pre-configured to use ARKit to integrate augmented reality support into an iOS app.
- **Sticker Pack App** – Allows a sticker pack app 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.
- **iMessage App** – 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 available through the Message App Store.
- **Safari Extension App** – This option creates a project to be used as the basis for developing an extension for the Safari web browser.

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



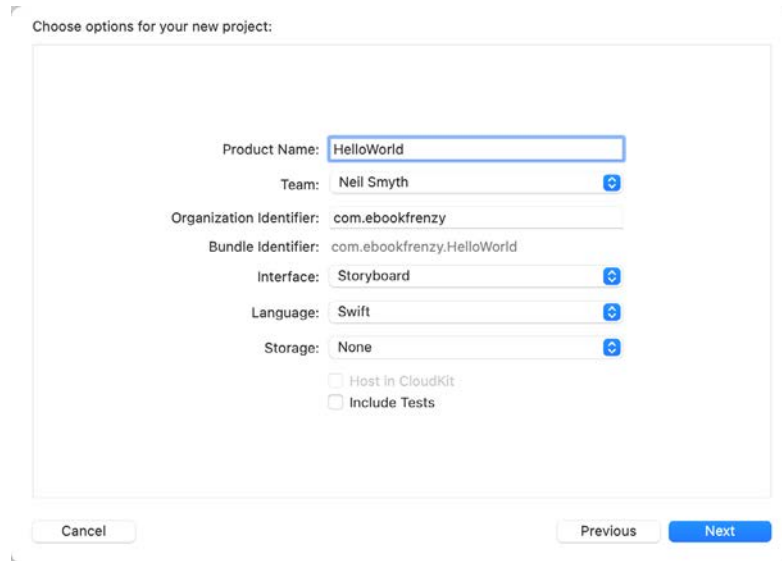


Figure 4-3

On this screen, enter a Product name for the app that will be created, in this case, “HelloWorld”. Next, choose your account from the Team menu if you have already signed up for the Apple developer program. Otherwise, leave the option set to None.

The text entered into the Organization Name field will be placed within the copyright comments of all the source files that make up the project.

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

When developing an app in Xcode, the user interface can be designed using either Storyboards or SwiftUI. For this book, we will be using Storyboards, so make sure that the Interface menu is set to Storyboard. SwiftUI development is covered in our *iOS 17 App Development Essentials* book:

<https://www.payloadbooks.com/index.php/product/ios-17-app-development-essentials-ebook/>

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. Therefore, 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 allows placing the project under Git source code control. Source code control systems such as Git allow different project revisions to be managed and restored, and for changes made over the project’s development lifecycle to be tracked. Since this is typically used for larger projects, or those involving more than one developer, this option can be turned off for this and the other projects created in the book.

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

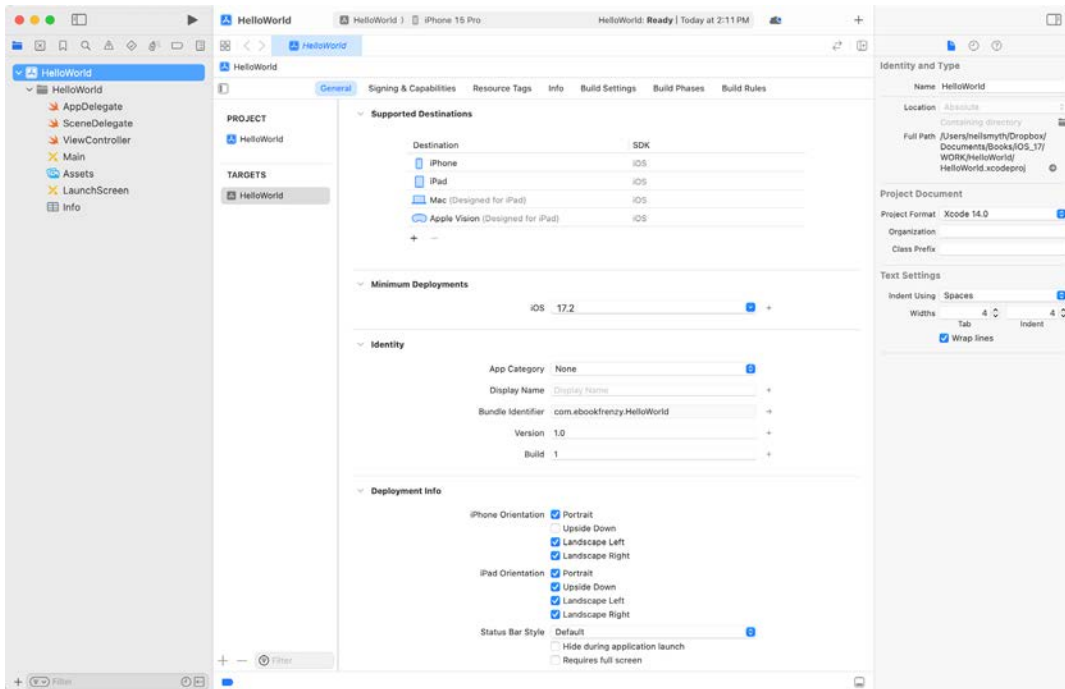


Figure 4-4

Before proceeding, we should take some time to look at what Xcode has done for us. First, it has created a group of files we will need to complete our app. Some of these are Swift source code files, where we will enter the code to make our app 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* will also have been added to the project. This file contains the user interface design for the screen that appears on the device while the app is loading.

Also present will be one or more *Property List* files that 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 several 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. In addition, 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 app project. This summary includes the identifier specified during the project creation process and the target devices. In addition, options are also provided to configure the orientations of the device that are to be supported by the app, together with opportunities to upload icons (the small images the user selects on the device screen to launch the app) and launch screen images (displayed to the user while the app loads) for the app.

The Signing section allows selecting an Apple identity when building the app. This identity 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 outlined in the previous chapter, select that identity now using the Team menu. Testing apps on physical devices will not be possible if no team is selected, though the simulator environment may still be

used.

The Supported Destinations and Minimum Deployment sections of the screen also include settings to specify the device types and iOS versions on which the completed app is intended to run, as shown in Figure 4-5:

▼ Supported Destinations

Destination	SDK
 iPhone	iOS
 iPad	iOS
 Mac (Designed for iPad)	iOS
 Apple Vision (Designed for iPad)	iOS

+ -

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 it 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 Pro 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. As will be shown in later chapters, Xcode 15 and iOS 17 include several features designed specifically to make the goal of *universal* app projects easy to achieve. With this in mind, ensure that the destination list at least includes the iPhone and iPad.

In addition to the General screen, tabs are provided to view and modify additional settings consisting of Signing & Capabilities, Resource Tags, 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 time, ensure the *Project Navigator* is selected in the left-hand panel and select the top item (the app 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.

## 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. Therefore, a vital component of any app involves a user interface through which the user will interact with the app 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 several 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, select the *Main* item in the list in the left-hand panel. Interface Builder will subsequently appear in the center panel, as shown in Figure 4-6:

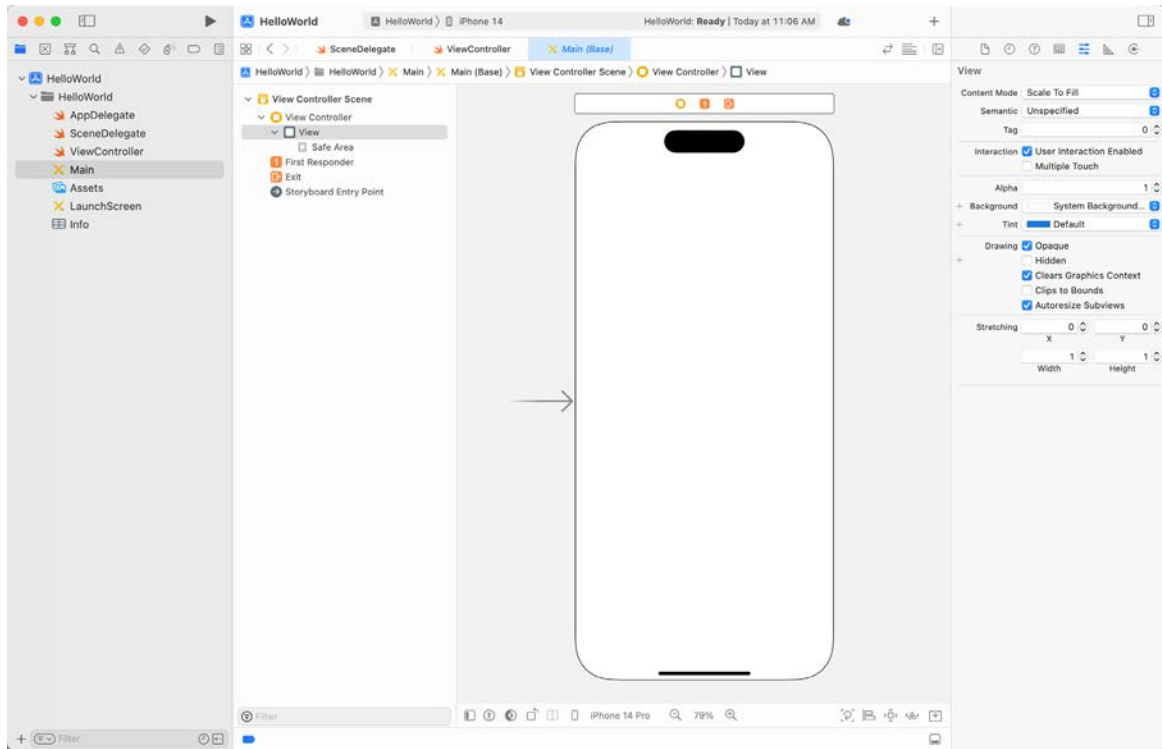


Figure 4-6

In the center panel, a visual representation of the app's user interface 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 App template 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 Library button in the Xcode toolbar, as indicated in Figure 4-7:

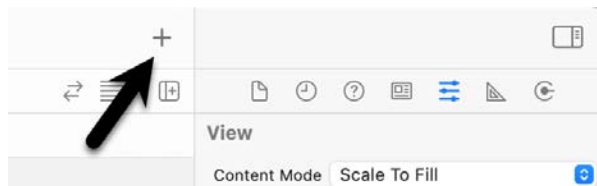


Figure 4-7

This button will display the UI components 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.

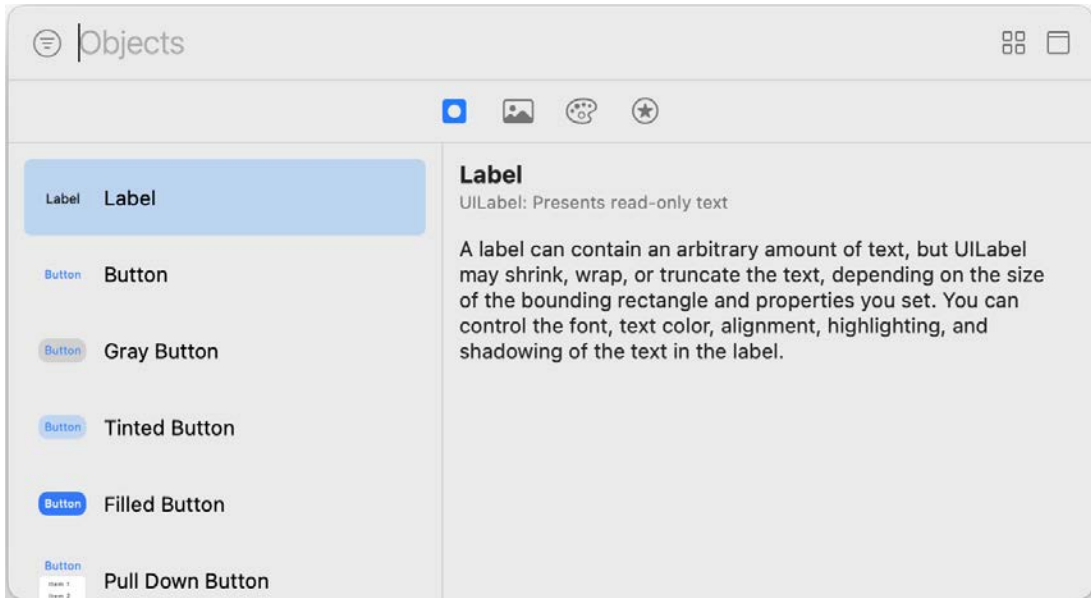


Figure 4-8

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. Hold the Option key while clicking on the required Library item to keep the panel visible in this mode. 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.

To edit property settings, we need to display the Xcode right-hand panel (if it is not already shown). This panel is referred to as the *Utilities panel* and can be displayed and hidden by clicking the right-hand button in the Xcode toolbar:

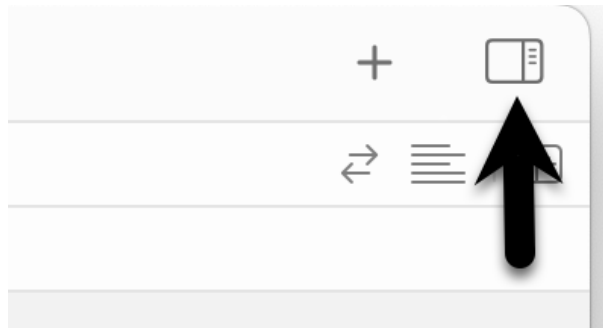


Figure 4-9

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

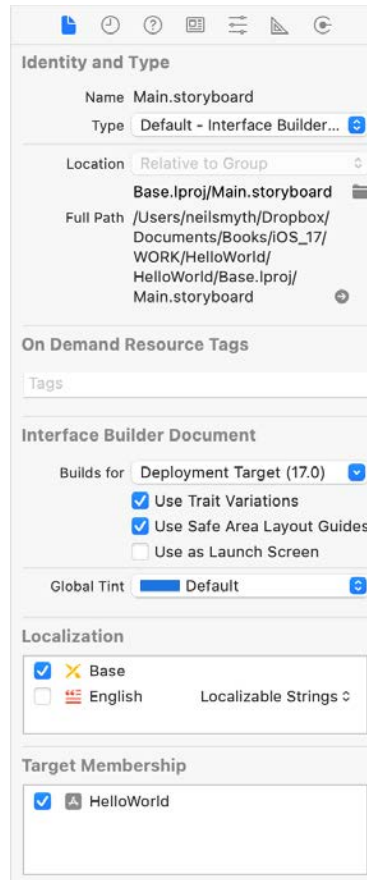


Figure 4-10

Along the top edge of the panel is a row of buttons that change the settings displayed in the upper half of the panel. By default, the *File Inspector* is typically shown. Options are also provided to display quick help, the *Identity Inspector*, *History Inspector*, *Attributes Inspector*, *Size Inspector*, and *Connections Inspector*. Take some time to review each of these selections to familiarize yourself 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 ensuring the View is selected and that the *Attributes Inspector* (*View -> Inspectors -> Attributes*) is displayed in the Utilities panel. Next, 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. Finally, choose a visually pleasing color using the color selection tool 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:

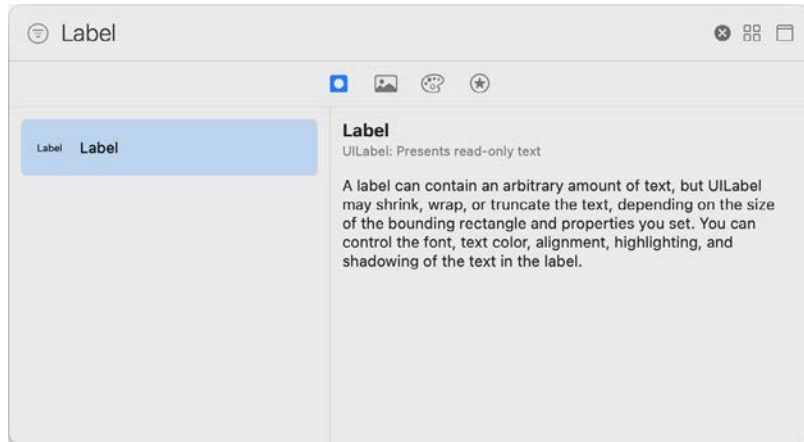


Figure 4-11

After locating 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. Next, 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 -> Inspectors -> Attributes*) to center the text in the middle of the label view:

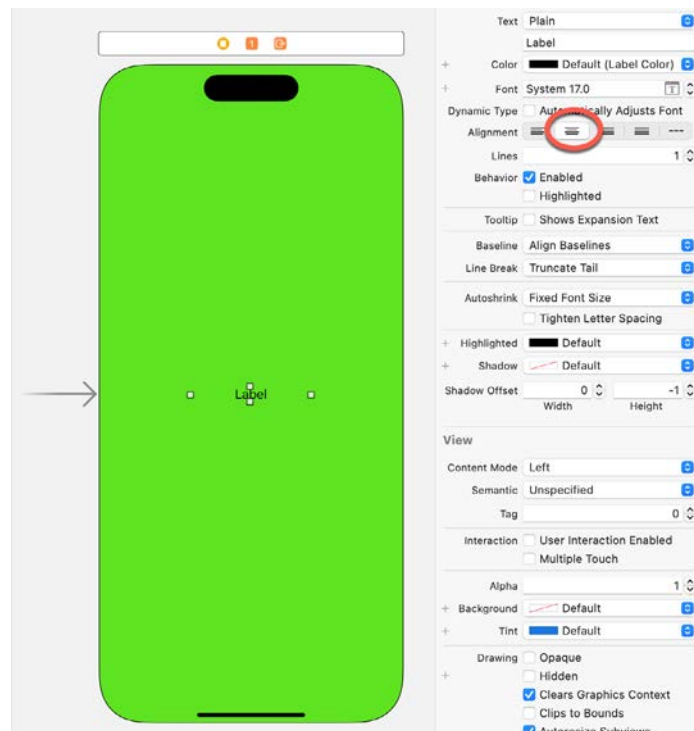


Figure 4-12

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 the “T” button to display the font selection menu

next to the font name. 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:

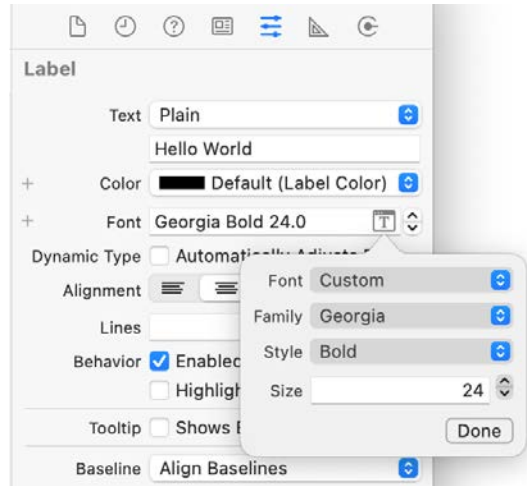


Figure 4-13

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 the screen on which the app ultimately runs. This involves using the Auto Layout capabilities of iOS, a topic that will be covered extensively in later chapters. For this example, select the Label object, display the Align menu as shown in Figure 4-14, and enable both the *Horizontally in Container* and *Vertically in Container* options with offsets of 0 before clicking on the *Add 2 Constraints* button.

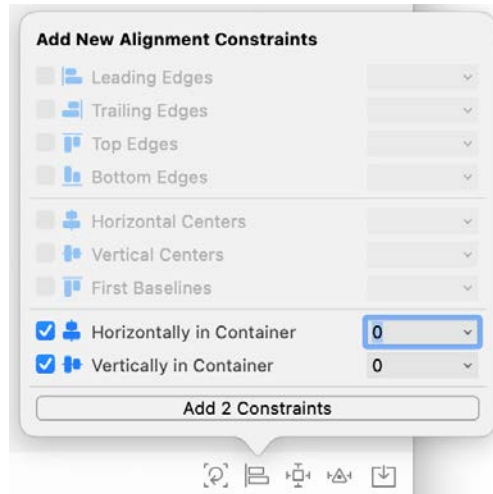


Figure 4-14

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).



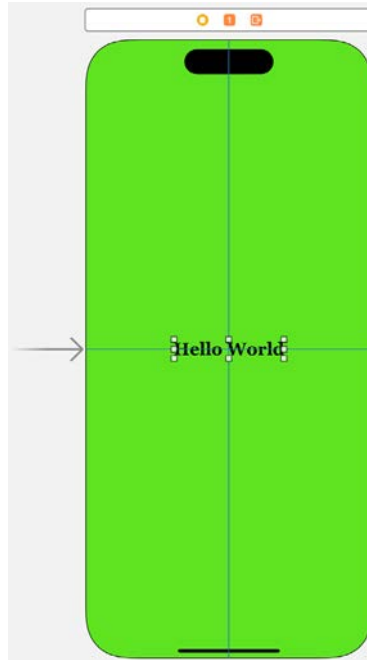


Figure 4-15

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. It 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.

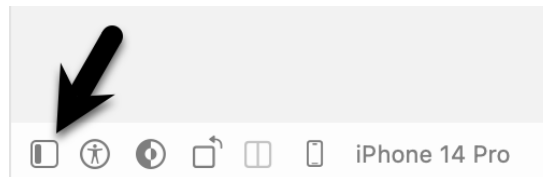


Figure 4-16

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

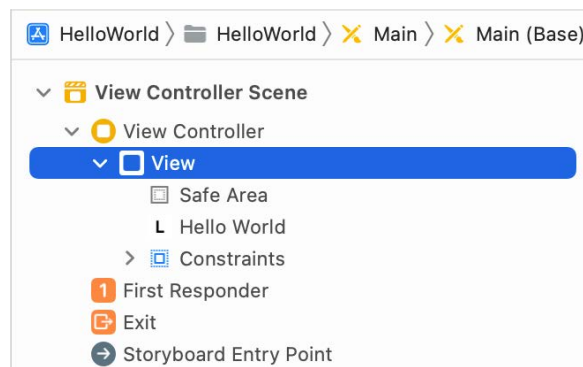


Figure 4-17

## 4.5 Building and Running an iOS App in Xcode

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 or iPad device. For 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 15* simulator option selected:

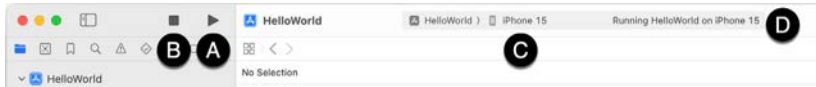


Figure 4-18

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 15 to iPad Air (5th Generation), and rerun the app. Once again, the label will appear centered on the screen even with the larger screen size. Finally, verify that the layout is correct in landscape orientation by using the *Device -> 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 valuable way to test an app on various 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 app ready for testing, refer to the device menu 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:

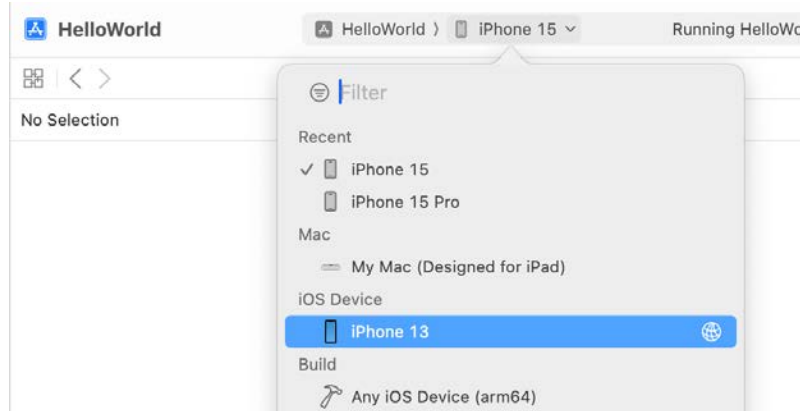


Figure 4-20

If the menu indicates that developer mode is disabled on the device, navigate to the Privacy & Security screen in the device's Settings app, locate the Developer Mode setting, and enable it. You will then need to restart the device. After the device restarts, a dialog will appear in which you will need to turn on developer mode. After entering your security code, the device will be ready for use with Xcode.

With the target device selected, ensure the device is unlocked and click on the run button, at which point Xcode will install and launch the app. As will be discussed later in this chapter, a physical device may also be configured for network testing, whereby apps are installed and tested via a network connection without needing 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, 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:

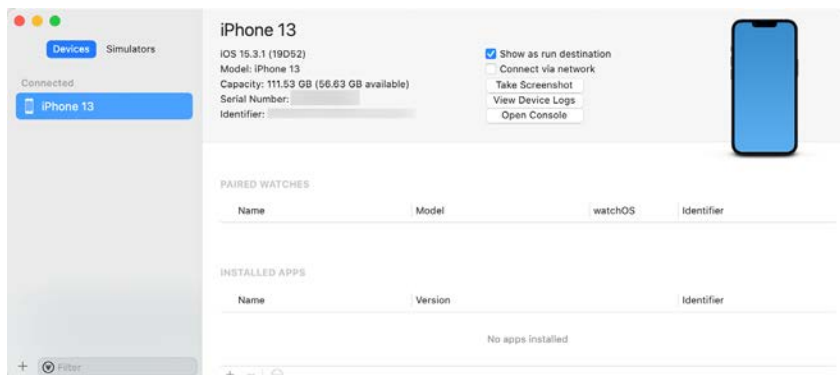


Figure 4-21

A wide range of simulator configurations are set up within Xcode by default and can be viewed by selecting the *Simulators* button at the top of the left-hand panel. Other simulator configurations can be added by clicking on the + button in the window's bottom left-hand corner. Once selected, a dialog will appear, allowing the simulator to be configured in terms of the device model, iOS version, and name.

### 4.8 Enabling Network Testing

In addition to testing an app on a physical device connected to the development system via a USB cable, Xcode 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 is that the device and development computer be connected to the same WiFi network. Assuming this requirement has been met, clicking 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

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, either on a device or simulator or within the Live Preview canvas. 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 various of other options. The seventh option from the left displays the debug navigator when selected, as illustrated in Figure 4-23. When displayed, this panel shows 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.

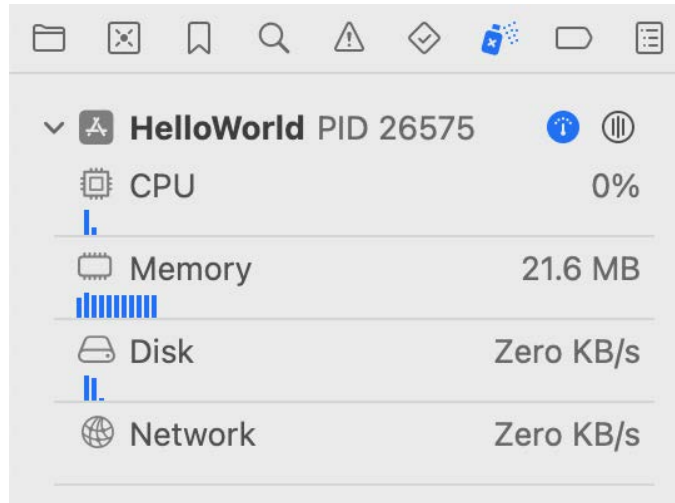


Figure 4-23

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:

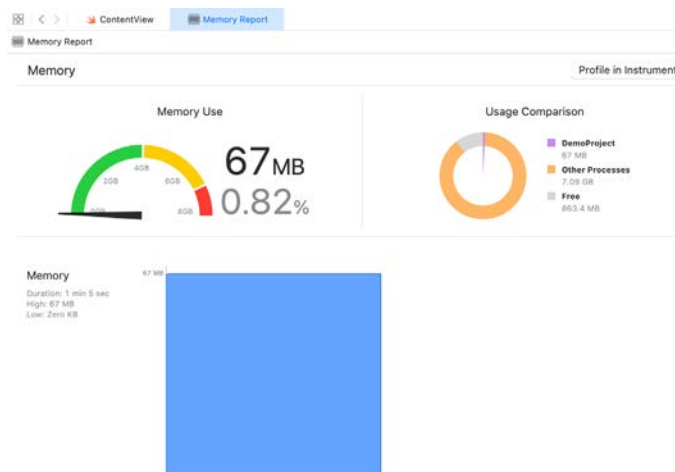


Figure 4-24

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 Exploring 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 instance is obscured by another appearing on top of it or a layout is not appearing as intended. This is also useful for learning how iOS works behind the scenes to construct a layout if only to appreciate how much work iOS is saving us from having to do.

To access the view hierarchy in this mode, the app needs to be running on a device or simulator. Once the app is running, click on the *Debug View Hierarchy* button indicated in Figure 4-25:



Figure 4-25

Once activated, a 3D “exploded” view of the layout will appear. Clicking and dragging within the view will rotate the hierarchy allowing the layers of views that make up the user interface to be inspected:

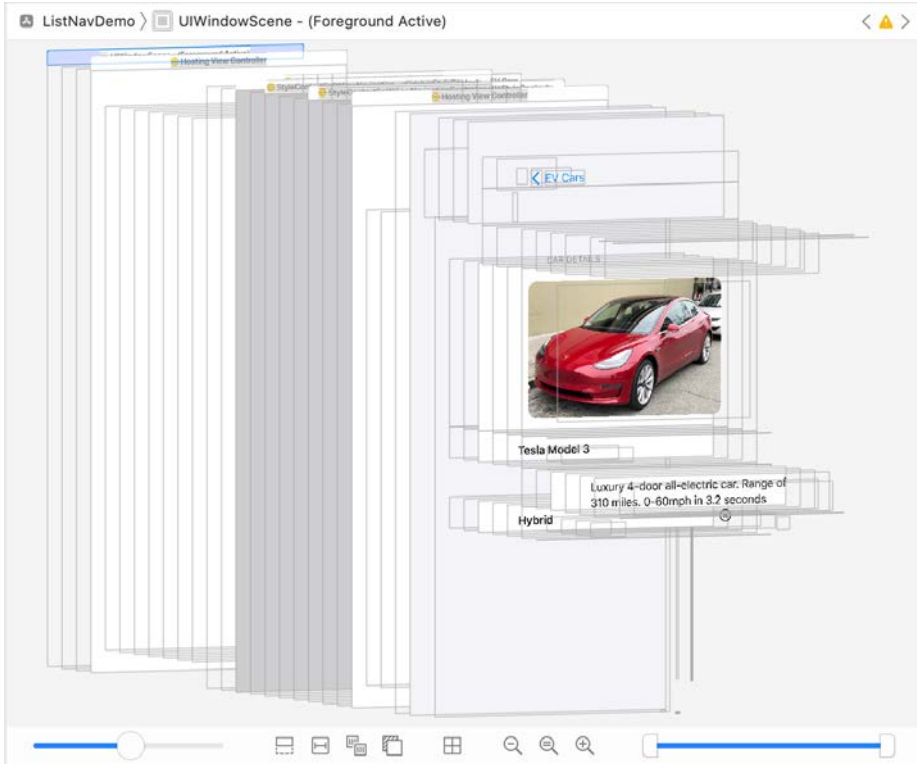


Figure 4-26

Moving the slider in the bottom left-hand corner of the panel will adjust the spacing between the different views in the hierarchy. The two markers in the right-hand slider (Figure 4-27) may also be used to narrow the range of views visible in the rendering. This can be useful, for example, to focus on a subset of views located in the middle of the hierarchy tree:

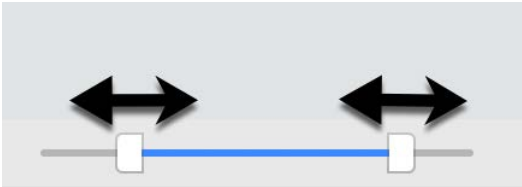


Figure 4-27

While the hierarchy is being debugged, the left-hand panel will display the entire view hierarchy tree for the

layout as shown in Figure 4-28 below:

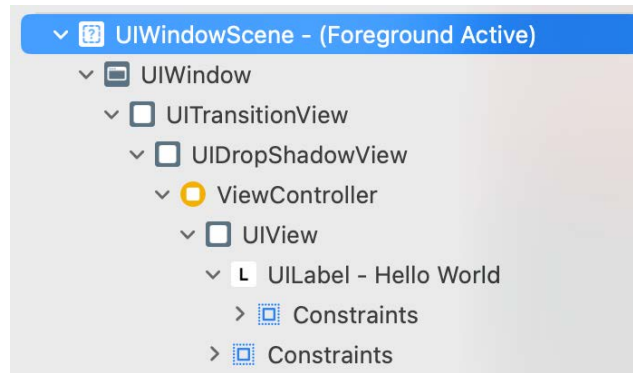


Figure 4-28

Selecting an object in the hierarchy tree will highlight the corresponding item in the 3D rendering and vice versa. The far right-hand panel will also display the Object Inspector populated with information about the currently selected object. Figure 4-29, for example, shows part of the Object Inspector panel while a Label view is selected within the view hierarchy.

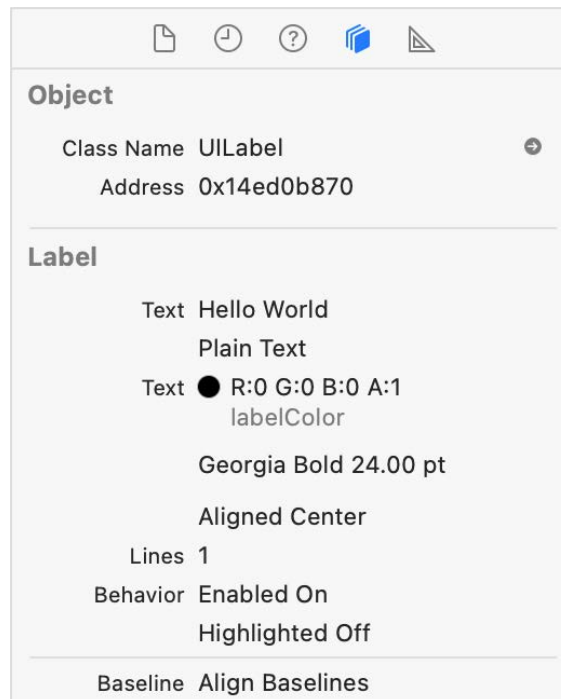


Figure 4-29

## 4.12 Summary

Apps are primarily created within the Xcode development environment. This chapter has provided a basic overview of the Xcode environment and worked through creating a straightforward example app. Finally, a brief overview was provided of some of the performance monitoring features in Xcode 15. In subsequent chapters of the book, many more features and capabilities of Xcode and Interface Builder will be covered.





# 7. Swift Operators and Expressions

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

## 7.1 Expression Syntax in Swift

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

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

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

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

## 7.2 The Basic Assignment Operator

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

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

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

## 7.3 Swift Arithmetic Operators

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

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

The following table lists the primary Swift arithmetic operators:

Operator	Description
-(unary)	Negates the value of a variable or expression

*	Multiplication
/	Division
+	Addition
-	Subtraction
%	Remainder/Modulo

Table 7-1

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

For example:

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

### 7.4 Compound Assignment Operators

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

```
x = x + y
```

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

```
x += y
```

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

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

Operator	Description
x += y	Add x to y and place result in x
x -= y	Subtract y from x and place result in x
x *= y	Multiply x by y and place result in x
x /= y	Divide x by y and place result in x
x %= y	Perform Modulo on x and y and place result in x

Table 7-2

### 7.5 Comparison Operators

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

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

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

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

a *true* value being stored in the variable `result`:

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

```
result = x < y
```

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

Operator	Description
<code>x == y</code>	Returns true if x is equal to y
<code>x &gt; y</code>	Returns true if x is greater than y
<code>x &gt;= y</code>	Returns true if x is greater than or equal to y
<code>x &lt; y</code>	Returns true if x is less than y
<code>x &lt;= y</code>	Returns true if x is less than or equal to y
<code>x != y</code>	Returns true if x is not equal to y

Table 7-3

## 7.6 Boolean Logical Operators

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

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

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

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

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

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

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

## 7.7 Range Operators

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

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

`x...y`

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

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

`x..<y`

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

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

`x...`

or

`...y`

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

`2...`

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

`...6`

## 7.8 The Ternary Operator

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

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

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

```
let x = 10
```

```
let y = 20
```

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

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

```
Largest number is 20
```

## 7.9 Nil Coalescing Operator

The *nil coalescing operator* (`??`) allows a default value to be used in the event that an optional has a nil value. The following example will output text which reads "Welcome back, customer" because the *customerName* optional is set to nil:

```
let customerName: String? = nil
print("Welcome back, \(customerName ?? "customer")")
```

If, on the other hand, *customerName* is not nil, the optional will be unwrapped and the assigned value displayed:

```
let customerName: String? = "John"
print("Welcome back, \(customerName ?? "customer")")
```

On execution, the print statement output will now read “Welcome back, John”.

## 7.10 Bitwise Operators

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

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

For the purposes of this exercise we will be working with the binary representation of two numbers (for the sake of brevity we will be using 8-bit values in the following examples). First, the decimal number 171 is represented in binary as:

```
10101011
```

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

```
00000011
```

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

### 7.10.1 Bitwise NOT

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

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

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

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

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

### 7.10.2 Bitwise AND

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

```
10101011 AND
```

## Swift Operators and Expressions

```
00000011
=====
00000011
```

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

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

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

### 7.10.3 Bitwise OR

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

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

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

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

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

### 7.10.4 Bitwise XOR

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

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

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

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

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

### 7.10.5 Bitwise Left Shift

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

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

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

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

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

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

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

### 7.10.6 Bitwise Right Shift

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

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

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

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

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

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

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

## 7.11 Compound Bitwise Operators

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

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

Table 7-4

## 7.12 Summary

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



## 17. Creating an Interactive iOS 17 App

The previous chapter looked at the design patterns we need to learn and use regularly while developing iOS-based apps. In this chapter, we will work through a detailed example intended to demonstrate the View-Controller relationship together with the implementation of the Target-Action pattern to create an example interactive iOS app.

### 17.1 Creating the New Project

The purpose of the app we are going to create is to perform unit conversions from Fahrenheit to Centigrade. The first step is creating a new Xcode project to contain our app. Start Xcode and, on the Welcome screen, select *Create a new Xcode project*. Make sure iOS is selected in the toolbar on the template screen before choosing the *App* template. Click *Next*, set the product name to *UnitConverter*, enter your company identifier, and select your development team if you have one. Before clicking *Next*, change the *Language* to Swift and the *Interface* to Storyboard. On the final screen, choose a location to store the project files and click *Create* to proceed to the main Xcode project window.

### 17.2 Creating the User Interface

Before we begin developing the logic for our interactive app, we will start by designing the user interface. When we created the new project, Xcode generated a storyboard file for us and named it *Main.storyboard*. Within this file, we will create our user interface, so select the *Main* item from the project navigator in the left-hand panel to load it into Interface Builder.

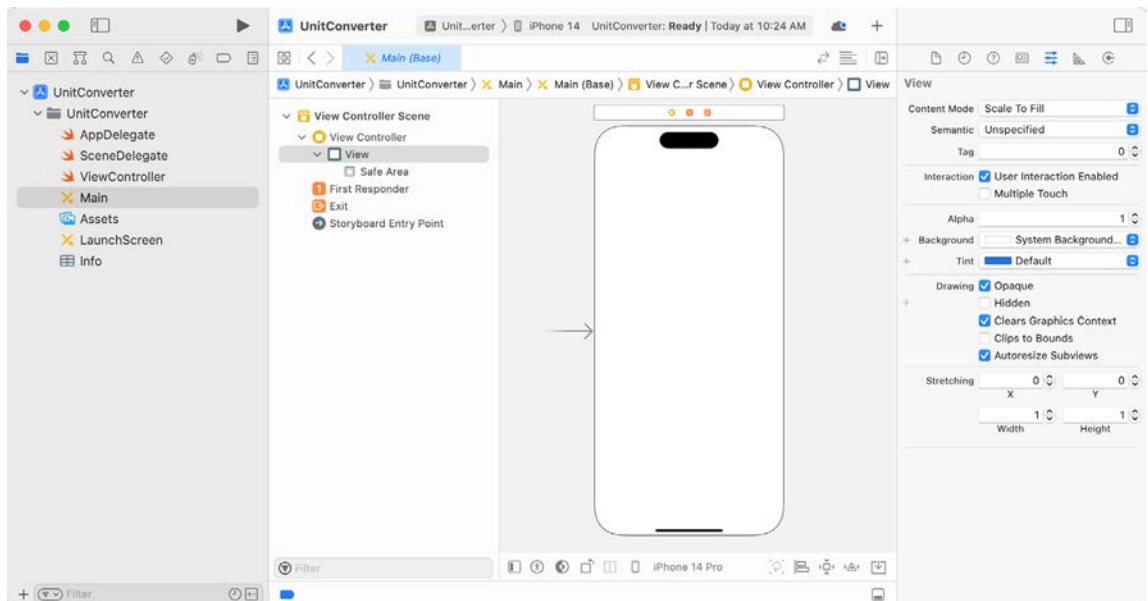


Figure 17-1

## Creating an Interactive iOS 17 App

Display the Library panel by clicking on the toolbar button shown in Figure 17-2 while holding down the Option key and dragging a Text Field object from the library onto the View design area:

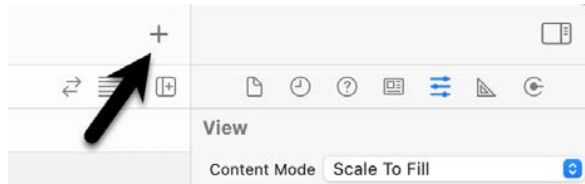


Figure 17-2

Resize the object and position it so it appears as outlined in Figure 17-3:

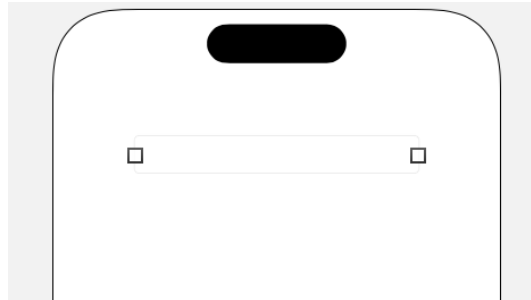


Figure 17-3

Within the Attributes Inspector panel (*View -> Inspectors -> Attributes*), type the words *Enter temperature* into the *Placeholder* text field. This text will then appear in light gray in the text field as a visual cue to the user. Since only numbers and decimal points will be required to be input for the temperature, locate the *Keyboard Type* property in the Attributes Inspector panel and change the setting to *Numbers and Punctuation*.

Now that we have created the text field into which the user will enter a temperature value, the next step is adding a *Button* object that may be pressed to initiate the conversion. To achieve this, drag and drop a *Button* object from the Library to the View. Next, double-click the button object to change to text edit mode and type the word *Convert* onto the button. Finally, select the button and drag it beneath the text field until the blue dotted line indicates it is centered horizontally within the containing view before releasing the mouse button.

The last user interface object we need to add is the label where the result of the conversion will be displayed. Add this by dragging a *Label* object from the Library panel to the View and positioning it beneath the button. Stretch the width of the label so that it is approximately two-thirds of the overall width of the view, and reposition it using the blue guidelines to ensure it is centered relative to the containing view. Finally, modify the *Alignment* attribute for the label object so that the text is centered.

Double-click on the label to highlight the text and press the backspace key to clear it (we will set the text from within a method of our View Controller class when the conversion calculation has been performed). Though the label is no longer visible when it is not selected, it is still present in the view. If you click where it is located, it will be highlighted with the resize dots visible. It is also possible to view the layout outlines of all the scenes' views, including the label, by selecting the *Editor -> Canvas -> Bounds Rectangles* menu option.

For the user interface design layout to adapt to the many different device orientations and iPad and iPhone screen sizes, it will be necessary to add some Auto Layout constraints to the views in the storyboard. Auto Layout will be covered in detail in subsequent chapters, but for this example, we will request that Interface Builder add what it considers to be the appropriate constraints for this layout. In the lower right-hand corner of the Interface Builder panel is a toolbar. Click on the background view of the current scene followed by the

*Resolve Auto Layout Issues* button as highlighted in Figure 17-4:



Figure 17-4

From the menu, select the *Reset to Suggested Constraints* option listed under *All Views in View Controller*:

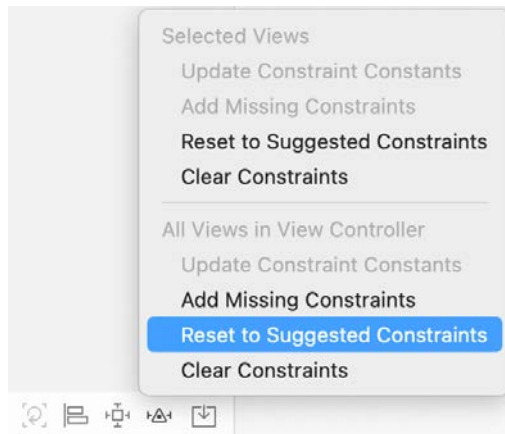


Figure 17-5

At this point, our project's user interface design phase is complete, and the view should appear as illustrated in Figure 17-6. We are now ready to try out a test build and run.

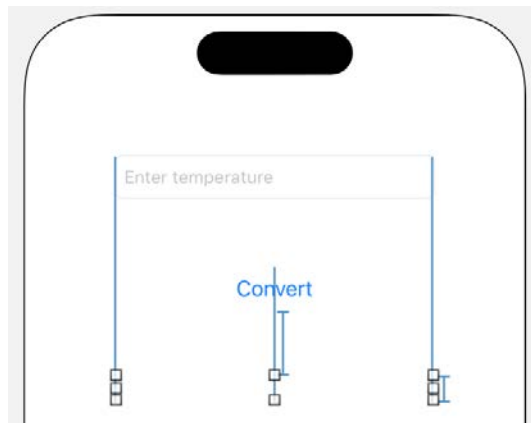


Figure 17-6

## 17.3 Building and Running the Sample App

Before we implement the view controller code for our app and then connect it to the user interface we have designed, we should perform a test build and run of the app. Click on the run button in the toolbar (the triangular “play” button) to compile the app and run it in the simulator or a connected iOS device. If you are unhappy with

how your interface looks, feel free to reload it into Interface Builder and make improvements. Assuming the user interface appears to your satisfaction, we are ready to start writing Swift code to add logic to our controller.

### 17.4 Adding Actions and Outlets

When the user enters a temperature value into the text field and touches the convert button, we need to trigger an action to calculate the temperature. The calculation result will then be presented to the user via the label object. The *Action* will be a method we will declare and implement in our View Controller class. Access to the text field and label objects from the view controller method will be implemented through *Outlets*.

Before we begin, now is a good time to highlight an example of subclassing as previously described in the chapter titled “*The iOS 17 App and Development Architecture*”. The UIKit Framework contains a class called `UIViewController` which provides the basic foundation for adding view controllers to an app. To create a functional app, however, we inevitably need to add functionality specific to our app to this generic view controller class. This is achieved by subclassing the `UIViewController` class and extending it with the additional functionality we need.

When we created our new project, Xcode anticipated our needs, automatically created a subclass of `UIViewController`, and named it `ViewController`. In so doing, Xcode also created a source code file named *ViewController.swift*.

Selecting the *ViewController.swift* file in the Xcode project navigator panel will display the contents of the file in the editing pane:

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        // Do any additional setup after loading the view.
    }
}
```

As we can see from the above code, a new class called `ViewController` has been created that is a subclass of the `UIViewController` class belonging to the UIKit framework.

The next step is to extend the subclass to include the two outlets and our action method. This could be achieved by manually declaring the outlets and actions within the *ViewController.swift* file. However, a much more straightforward approach is to use the Xcode Assistant Editor to do this for us.

With the *Main.storyboard* file selected, display the Assistant Editor by selecting the *Editor -> Assistant* menu option. Alternatively, it may also be displayed by selecting the *Adjust Editor Options* button in the row of Editor toolbar buttons in the top right-hand corner of the main Xcode window and selecting the Assistant menu option, as illustrated in the following figure:

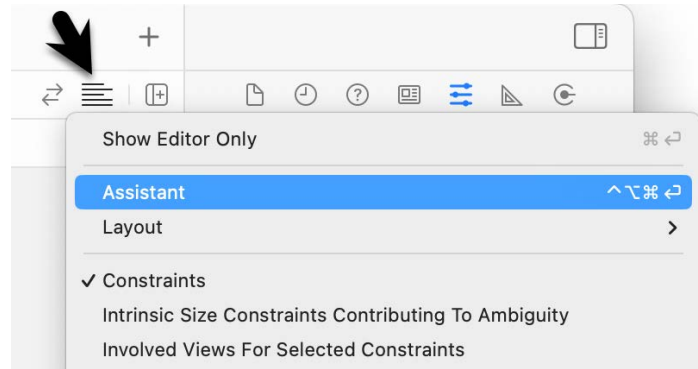


Figure 17-7

The editor panel will, by default, appear to the right of the main editing panel in the Xcode window. For example, in Figure 17-8, the panel (marked A) to the immediate right of the Interface Builder panel is the Assistant Editor:

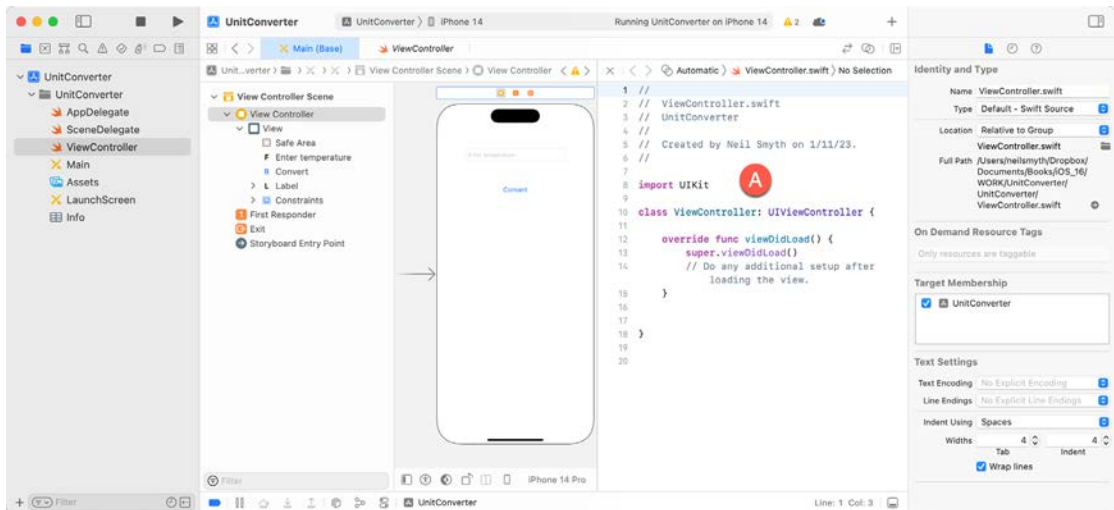


Figure 17-8

By default, the Assistant Editor will be in *Automatic* mode, whereby it automatically attempts to display the correct source file based on the currently selected item in Interface Builder. If the correct file is not displayed, use the toolbar at the top of the editor panel to select the correct file. The button displaying interlocking circles in this toolbar can be used to switch to *Manual* mode allowing the file to be selected from a pull-right menu containing all the source files in the project.

Make sure that the *ViewController.swift* file is displayed in the Assistant Editor and establish an outlet for the Text Field object by right-clicking on the Text Field object in the view. Drag the resulting line to the area immediately beneath the class declaration line in the Assistant Editor panel, as illustrated in Figure 17-9:



Figure 17-9

Upon releasing the line, the configuration panel illustrated in Figure 17-10 will appear, requesting details about the outlet to be defined.

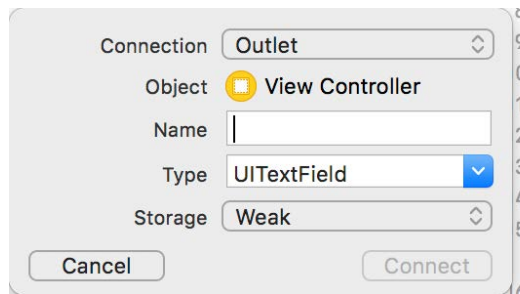


Figure 17-10

Since this is an outlet, the *Connection* menu should be left as *Outlet*. The type and storage values are also correct for this type of outlet. The only task that remains is to enter a name for the outlet, so in the *Name* field, enter *tempText* before clicking on the *Connect* button.

Once the connection has been established, select the *ViewController.swift* file and note that the outlet property has been declared for us by the assistant:

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var tempText: UITextField!

    .
    .
}
```

Repeat the above steps to establish an outlet for the Label object named *resultLabel*.

Next, we need to establish the action that will be called when the user touches the Convert button in our user interface. The steps to declare an action using the Assistant Editor are the same as those for an outlet. Once again, select the *Main.storyboard* file, but this time right-click on the button object. Drag the resulting line to the area beneath the existing *viewDidLoad* method in the Assistant Editor panel before releasing it. The connection box will once again appear. Since we are creating an action rather than an outlet, change the *Connection* menu to *Action*. Name the action *convertTemp* and make sure the *Event type* is set to *Touch Up Inside*:

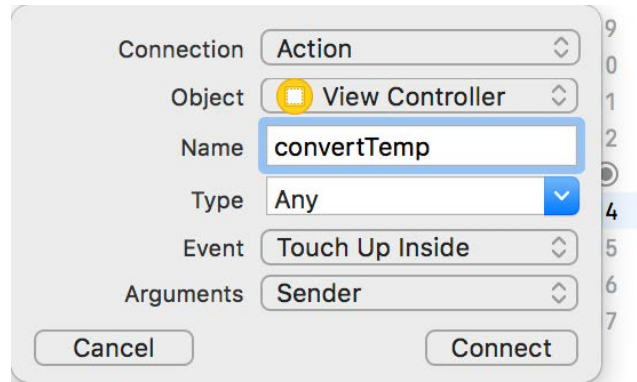


Figure 17-11

Click on the *Connect* button to create the action.

Close the Assistant Editor panel, select the *ViewController.swift* file, and note that a stub method for the action has now been declared for us by the assistant:

```
@IBAction func convertTemp(_ sender: Any) {
}
```

All that remains is to write the Swift code in the action method to perform the conversion:

```
@IBAction func convertTemp(_ sender: Any) {
    guard let tempString = tempText.text else { return }

    if let fahrenheit = Double(tempString) {
        let celsius = (fahrenheit - 32)/1.8
        let resultText = "Celsius \(celsius)"
        resultLabel.text = resultText
    }
}
```

Before proceeding, it is probably a good idea to pause and explain what is happening in the above code. However, those already familiar with Swift may skip the following few paragraphs.

In this file, we are implementing the *convertTemp* method, a template for which was created for us by the Assistant Editor. This method takes as a single argument a reference to the *sender*. The sender is the object that triggered the call to the method (in this case, our Button object). The sender is declared as being of type *Any* (different type options are available using the *Type* menu in the connection dialog shown in Figure 17-11 above). This special type can be used to represent any type of class. While we won't be using this object in the current example, this can be used to create a general-purpose method in which the method's behavior changes depending on how (i.e., via which object) it was called. We could, for example, create two buttons labeled *Convert to Fahrenheit* and *Convert to Celsius*, respectively, each of which calls the same *convertTemp* method. The method would then access the *sender* object to identify which button triggered the event and perform the corresponding type of unit conversion.

Within the method's body, we use a guard statement to verify that the *tempText* view contains some text. If it does not, the method simply returns.

Next, dot notation is used to access the *text* property (which holds the text displayed in the text field) of the *UITextField* object to access the text in the field. This property is itself an object of type *String*. This string is

## Creating an Interactive iOS 17 App

converted to be of type Double and assigned to a new constant named *fahrenheit*. Since it is possible that the user has not entered a valid number into the field, optional binding is employed to prevent an attempt to perform the conversion on invalid data.

Having extracted the text entered by the user and converted it to a number, we then perform the conversion to Celsius and store the result in another constant named *celsius*. Next, we create a new string object and initialize it with text comprising the word Celsius and the result of our conversion. In doing so, we declare a constant named *resultText*.

Finally, we use dot notation to assign the new string to the text property of our UILabel object to display it to the user.

## 17.5 Building and Running the Finished App

From within the Xcode project window, click on the run button in the Xcode toolbar (the triangular “play” style button) to compile the app and run it in the simulator or a connected iOS device. Once the app is running, click inside the text field and enter a Fahrenheit temperature. Next, click the Convert button to display the equivalent temperature in Celsius. Assuming all went to plan, your app should appear as outlined in the following figure:

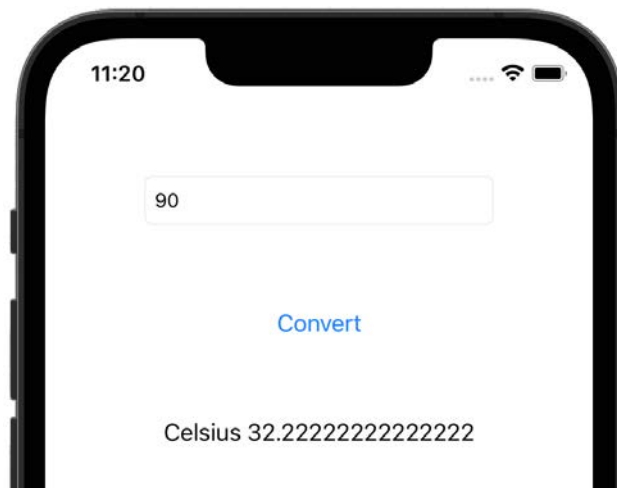


Figure 17-12

## 17.6 Hiding the Keyboard

The final step in the app implementation is to add a mechanism for hiding the keyboard. Ideally, the keyboard should withdraw from view when the user touches the background view or taps the return key on the keyboard (note when testing on the simulator that the keyboard may not appear unless the *I/O -> Keyboard -> Toggle Software Keyboard* menu option is selected).

To achieve this, we will begin by implementing the *touchesBegan* event handler method on the view controller in the *ViewController.swift* file as follows:

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {  
    tempText.endEditing(true)  
}
```

The keyboard will now be hidden when the user touches the background view.

The next step is to hide the keyboard when the return key is tapped. To do this, display the Assistant Editor and right-click and drag from the Text Field to a position beneath the *viewDidLoad* method within the *ViewController*.



*swift* file. On releasing the line, change the settings in the connection dialog to establish an Action connection named *textFieldReturn* for the *Did End on Exit* event with the Type menu set to *UITextField* as shown in Figure 17-13 and click on the *Connect* button to establish the connection.

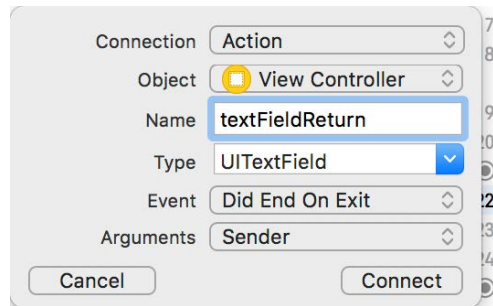


Figure 17-13

Select the *ViewController.swift* file in the project navigator, locate and edit the *textFieldReturn* stub method so that it now reads as follows:

```
@IBAction func textFieldReturn(_ sender: UITextField) {
    _ = sender.resignFirstResponder()
}
```

In the above method, we call the *resignFirstResponder* method of the object that triggered the event. The *first responder* is the object with which the user is currently interacting (in this instance, the virtual keyboard displayed on the device screen). Note that the result of the method call is assigned to a value represented by the underscore character (*\_*). The *resignFirstResponder()* method returns a Boolean value indicating whether or not the resign request was successful. Assigning the result this way indicates to the Swift compiler that we are intentionally ignoring this value.

Save the code and then build and run the app. When the app starts, select the text field so the keyboard appears. Touching any area of the background or tapping the return key should cause the keyboard to disappear.

## 17.7 Summary

In this chapter, we have demonstrated some of the theories covered in previous chapters, in particular, separating the view from the controller, subclassing, and implementing the Target-Action pattern through actions and outlets.

This chapter also provided steps to hide the keyboard when the user touches either the keyboard Return key or the background view.



## 19. An Introduction to Auto Layout in iOS 17

Arguably one of the most important parts of designing the user interface for an app involves getting the layout correct. In an ideal world, designing a layout would consist of dragging view objects to the desired location on the screen and fixing them at these positions using absolute X and Y screen coordinates. However, in reality, the world of iOS devices is more complex than that, and a layout must be able to adapt to variables such as the device rotating between portrait and landscape modes, dynamic changes to content, and differences in screen resolution and size.

Before the release of iOS 6, layout handling involved using a concept referred to as *autosizing*. Autosizing involves using a series of “springs” and “struts” to define, on a view-by-view basis, how a subview will be resized and positioned relative to the superview in which it is contained. Limitations of autosizing, however, typically meant that considerable amounts of coding were required to augment the autosizing in response to orientation or other changes.

One of the most significant features in iOS 6 was the introduction of Auto Layout, which has continued to evolve with the release of subsequent iOS versions. Auto Layout is an extensive subject area allowing layouts of just about any level of flexibility and complexity to be created once the necessary skills have been learned.

The goal of this and subsequent chapters will be to introduce the basic concepts of Auto Layout, work through some demonstrative examples and provide a basis to continue learning about Auto Layout as your app design needs evolve. Auto Layout introduces a lot of new concepts and can, initially, seem a little overwhelming. By the end of this sequence of chapters, however, it should be more apparent how the pieces fit together to provide a powerful and flexible layout management system for iOS-based user interfaces.

### 19.1 An Overview of Auto Layout

The purpose of Auto Layout is to allow the developer to describe the behavior required from the views in a layout independent of the device screen size and orientation. This behavior is implemented by creating *constraints* on the views that comprise a user interface screen. A button view, for example, might have a constraint that tells the system that it is to be positioned in the horizontal center of its superview. A second constraint might also declare that the bottom edge of the button should be positioned a fixed distance from the bottom edge of the superview. Having set these constraints, no matter what happens to the superview, the button will always be centered horizontally and a fixed distance from the bottom edge.

Unlike autosizing, Auto Layout allows constraints to be declared between a subview and superview and between subviews. Auto Layout, for example, would allow a constraint to be configured such that two button views are always positioned a specific distance apart from each other regardless of changes in size and orientation of the superview. Constraints can also be configured to cross superview boundaries to allow, for example, two views with different supervIEWS (though on the same screen) to be aligned. This is a concept referred to as *cross-view hierarchy constraints*.

Constraints can also be explicit or variable (otherwise referred to in Auto Layout terminology as *equal* or *unequal*). Take, for example, a width constraint on a label object. An explicit constraint could be declared to fix the width of the label at 70 points. This might be represented as a constraint equation that reads as follows:

```
myLabel.width = 70
```

However, this explicit width setting might become problematic if the label is required to display dynamic content. For example, an attempt to display text on the label that requires a greater width will result in the content being clipped.

Constraints can, however, be declared using less than, equal to, greater than, *or equal to* controls. For example, the width of a label could be constrained to any width as long as it is less than or equal to 800:

```
myLabel.width <= 800
```

The label is now permitted to grow in width up to the specified limit, allowing longer content to be displayed without clipping.

Auto Layout constraints are by nature interdependent. As such, situations can arise where a constraint on one view competes with a constraint on another view to which it is connected. In such situations, it may be necessary to make one constraint *stronger* and the other *weaker* to provide the system with a way of arriving at a layout solution. This is achieved by assigning *priorities* to constraints.

Priorities are assigned on a scale of 0 to 1000, with 1000 representing a *required constraint* and lower numbers equating to *optional constraints*. When faced with a decision between the needs of a required constraint and an optional constraint, the system will meet the needs of the required constraint exactly while attempting to get as close as possible to those of the optional constraint. In the case of two optional constraints, the needs of the constraint with the higher priority will be addressed before those of the lower.

## 19.2 Alignment Rects

When working with constraints, it is important to be aware that constraints operate on the content of a view, not the frame in which a view is displayed. This content is referred to as the *alignment rect* of the view. Alignment constraints, such as those that cause the center of one view to align with that of another, will do so based on the alignment rects of the views, disregarding any padding that may have been configured for the frame of the view.

## 19.3 Intrinsic Content Size

Some views also have what is known as an *intrinsic content size*. This is the preferred size that a view believes it needs to be to display its content to the user. A Button view, for example, will have an intrinsic content size in terms of height and width based primarily on the text or image it is required to display and internal rules on the margins that should be placed around that content. When a view has an intrinsic content size, Auto Layout will automatically assign two constraints for each dimension for which the view has indicated an intrinsic content size preference (i.e., height and/or width). One constraint is intended to prevent the view's size from becoming larger than the size of the content (otherwise known as the *content hugging* constraint). The other constraint is intended to prevent the view from being sized smaller than the content (referred to as the *compression resistance* constraint).

## 19.4 Content Hugging and Compression Resistance Priorities

The resizing behavior of a view with an intrinsic content size can be controlled by specifying compression resistance and content hugging priorities. For example, a view with high compression resistance and low content hugging priority will be allowed to grow but will resist shrinking in the corresponding dimension. Similarly, a high compression resistance and a high content hugging priority will cause the view to resist any resizing, keeping the view as close as possible to its intrinsic content size.

## 19.5 Safe Area Layout Guide

In addition to the views that comprise the layout, a screen may also contain navigation and tab bars at the top and bottom of the screen. If the layout is designed to use the full screen height, there is a risk that some views

will be obscured by navigation and tab bars. To avoid this problem, UIView provides a *safe area layout guide* for constrained views. Constraining views to the safe area instead of the outer edges of the parent UIView ensures that the views are not obscured by title and tab bars. For example, the screen in Figure 19-1 includes both navigation and tab bars. The dotted line represents the safe area layout guide to which the top edge of the Button and bottom edge of the Label has been constrained:

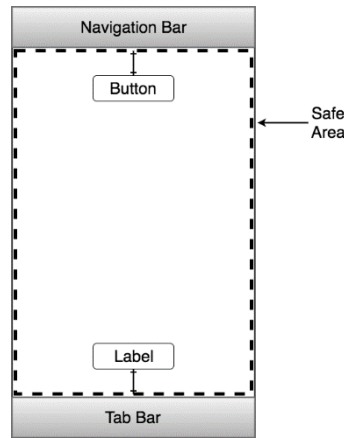


Figure 19-1

## 19.6 Three Ways to Create Constraints

There are three ways in which constraints in a user interface layout can be created:

- **Interface Builder** – Interface Builder has been modified extensively to support the visual implementation of Auto Layout constraints in user interface designs. Examples of using this approach are covered in the “*Working with iOS 17 Auto Layout Constraints in Interface Builder*” and “*Implementing Cross-Hierarchy Auto Layout Constraints in iOS 17*” chapters of this book.
- **Visual Format Language** – The visual format language defines a syntax that allows constraints to be declared using a sequence of ASCII characters that visually approximate the nature of the constraint being created to make constraints in code both easier to write and understand. Use of the visual format language is documented in the chapter entitled “*Understanding the iOS 17 Auto Layout Visual Format Language*”.
- **Writing API code** – This approach involves directly writing code to create constraints using the standard programming API calls. This topic is covered in “*Implementing iOS 17 Auto Layout Constraints in Code*”.

Wherever possible, Interface Builder is the recommended approach to creating constraints. When creating constraints in code, the visual format language is generally recommended over the API-based approach.

## 19.7 Constraints in More Detail

A constraint is created as an instance of the `NSLayoutConstraint` class, which, having been created, is then added to a view. The rules for a constraint can generally be represented as an equation, the most complex form of which can be described as follows:

```
view1.attribute = multiplier * view2.attribute2 + constant
```

The above equation establishes a constraint relationship between `view1` and `view2`, respectively. In each case, an attribute is targeted by the constraint. Attributes are represented by `NSLayoutConstraint.Attribute.<name>` constants where `<name>` is one of several options, including `left`, `right`, `top`, `bottom`, `leading`, `trailing`, `width`, `height`, `centerX`, `centerY`, and `baseline` (i.e., `NSLayoutConstraint.Attribute.width`). The multiplier and constant elements are floating point values that modify the constraint.

A simple constraint that dictates that view1 and view2 should, for example, be the same width would be represented using the following equation:

```
view1.width = view2.width
```

Similarly, the equation for a constraint to align the horizontal center of view1 with the horizontal center of view2 would read as follows:

```
view1.centerX = view2.centerX
```

A slightly more complex constraint to position view1 so that its bottom edge is positioned a distance of 20 points above the bottom edge of view2 would be expressed as follows:

```
view1.bottom = view2.bottom - 20
```

The following constraint equation specifies that view1 is to be twice the width of view2 minus a width of 30 points:

```
view1.width = view2.width * 2 - 30
```

So far, the examples have focused on equality. As previously discussed, constraints also support inequality through `<=` and `>=` operators. For example:

```
view1.width >= 100
```

A constraint based on the above equation would limit the width of view1 to any value greater than or equal to 100.

The reason for representing constraints in equations is less apparent when working with constraints within Interface Builder. Still, it will become invaluable when using the API or the visual format language to set constraints in code.

## 19.8 Summary

Auto Layout uses constraints to descriptively express a user interface's geometric properties, behavior, and view relationships.

Constraints can be created using Interface Builder or in code using either the visual format language or the standard SDK API calls of the `NSLayoutConstraint` class.

Constraints are typically expressed using a linear equation, an understanding of which will be particularly beneficial when working with constraints in code.

Having covered the basic concepts of Auto Layout, the next chapter will introduce the creation and management of constraints within Interface Builder.

## 25. Using Storyboards in Xcode 15

Storyboarding is a feature built into Xcode that allows the various screens that comprise an iOS app and the navigation path through those screens to be visually assembled. Using the Interface Builder component of Xcode, the developer drags and drops view and navigation controllers onto a canvas and designs the user interface of each view in the usual manner. The developer then drags lines to link individual trigger controls (such as a button) to the corresponding view controllers that are to be displayed when the user selects the control. Having designed both the screens (referred to in the context of storyboarding as *scenes*) and specified the transitions between scenes (referred to as *segues*), Xcode generates all the code necessary to implement the defined behavior in the completed app. The transition style for each segue (page fold, cross dissolve, etc.) may also be defined within Interface Builder. Further, segues may be triggered programmatically when behavior cannot be graphically defined using Interface Builder.

Xcode saves the finished design to a *storyboard file*. Typically, an app will have a single storyboard file, though there is no restriction preventing using multiple storyboard files within a single app.

The remainder of this chapter will work through creating a simple app using storyboarding to implement multiple scenes with segues defined to allow user navigation.

### 25.1 Creating the Storyboard Example Project

Begin by launching Xcode and creating a new project named *Storyboard* using the iOS *App* template with the language menu set to *Swift* and the *Storyboard* Interface option selected. Then, save the project to a suitable location by clicking the *Create* button.

### 25.2 Accessing the Storyboard

Upon creating the new project, Xcode will have created what appears to be the usual collection of files for a single-view app, including a storyboard named file *Main.storyboard*. Select this file in the project navigator panel to view the storyboard canvas as illustrated in Figure 25-1.

The view displayed on the canvas is the view for the *ViewController* class created for us by Xcode when we selected the *App* template. The arrow pointing inwards to the left side of the view indicates that this is the initial view controller and will be the first view displayed when the app launches. To change the initial view controller, drag this arrow to any other scene in the storyboard and drop it in place.

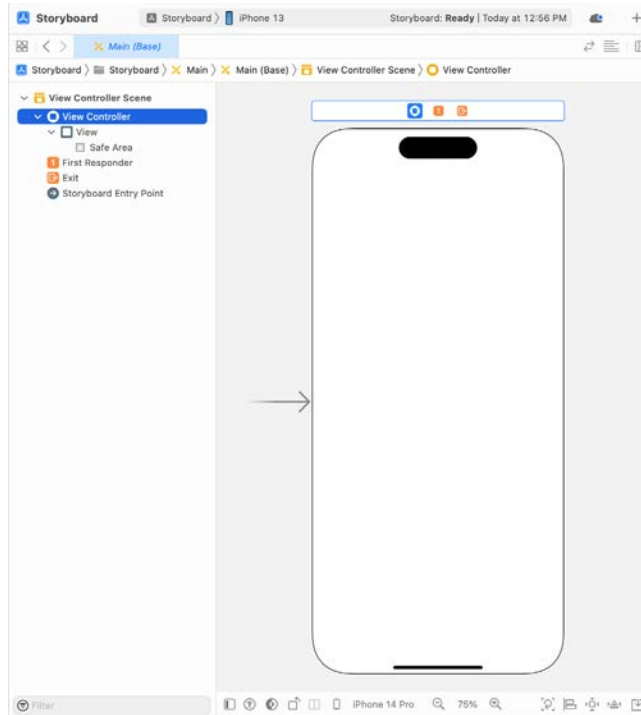


Figure 25-1

Objects may be added to the view in the usual manner by displaying the Library panel and dragging and dropping objects onto the view canvas. For this example, drag a label and a button onto the view canvas. Using the properties panel, change the label text to *Scene 1* and the button text to *Go to Scene 2*.

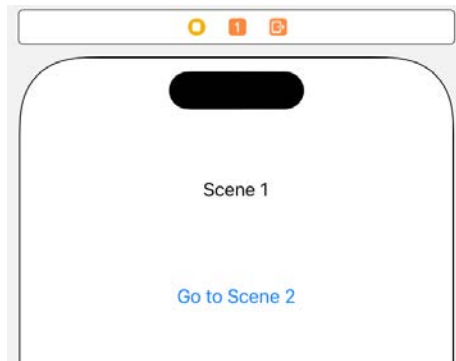


Figure 25-2

Using the *Resolve Auto Layout Issues* menu, select the *Reset to Suggested Constraints* option listed under *All Views in View Controller*.

It will be necessary first to establish an outlet to manipulate text displayed on the label object from within the app code. Select the label in the storyboard canvas and display the Assistant Editor (*Editor -> Assistant*). Check that the Assistant Editor is showing the content of the *ViewController.swift* file. Then, right-click on the label and drag the resulting line to just below the class declaration line in the Assistant Editor panel. In the resulting connection dialog, enter *scene1Label* as the outlet name and click on the *Connect* button. Upon completion of the connection, the top of the *ViewController.swift* file should read as follows:



```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var scene1Label: UILabel!
    .
    .
}
```

## 25.3 Adding Scenes to the Storyboard

To add a second scene to the storyboard, drag a View Controller object from the Library panel onto the canvas. Figure 25-3 shows a second scene added to a storyboard:



Figure 25-3

Drag and drop a label and a button into the second scene and configure the objects so that the view appears as shown in Figure 25-4. Then, repeat the steps performed for the first scene to configure Auto Layout constraints on the two views.

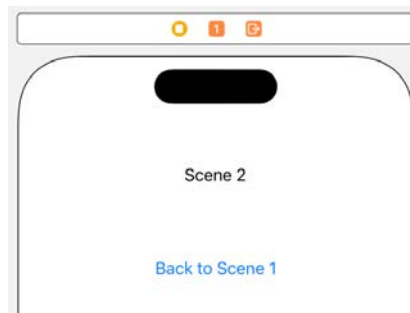


Figure 25-4

As many scenes as necessary may be added to the storyboard, but we will use just two scenes for this exercise.

Having implemented the scenes, the next step is to configure segues between the scenes.

## 25.4 Configuring Storyboard Segues

As previously discussed, a segue is a transition from one scene to another within a storyboard. Within the example app, touching the *Go To Scene 2* button will segue to scene 2. Conversely, the button on scene 2 is intended to return the user to scene 1. To establish a segue, hold down the Ctrl key on the keyboard, click over a control (in this case, the button on scene 1), and drag the resulting line to the scene 2 view. Upon releasing the mouse button, a menu will appear. Select the *Present Modally* menu option to establish the segue. Once the segue has been added, a connector will appear between the two scenes, as highlighted in Figure 25-5:

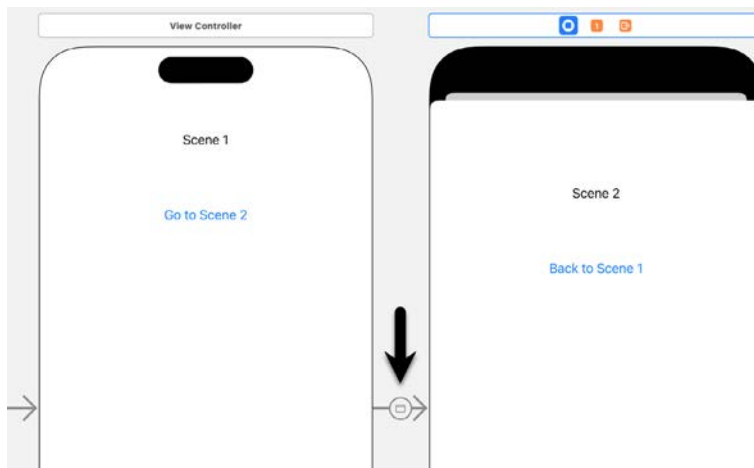


Figure 25-5

As more scenes are added to a storyboard, it becomes increasingly difficult to see more than a few scenes at one time on the canvas. To zoom out, double-click on the canvas. To zoom back in again, double-click once again on the canvas. The zoom level may also be changed using the plus and minus control buttons located in the status bar along the bottom edge of the storyboard canvas or by right-clicking on the storyboard canvas background to access a menu containing several zoom level options.

## 25.5 Configuring Storyboard Transitions

Xcode allows changing the visual appearance of the transition that occurs during a segue. To change the transition, select the corresponding segue connector, display the Attributes Inspector, and modify the *Transition* setting. For example, in Figure 25-6, the transition has been changed to *Cross Dissolve*:

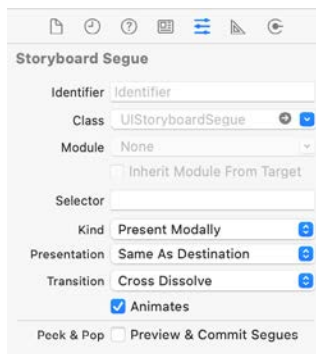


Figure 25-6

If animation is not required during the transition, turn off the *Animates* option. Run the app on a device or simulator and test that touching the “Go to Scene 2” button causes Scene 2 to appear.

## 25.6 Associating a View Controller with a Scene

At this point in the example, we have two scenes but only one view controller (the one created by Xcode when we selected the *iOS App* template). To add any functionality behind scene 2, it will also need a view controller. The first step is to add the class source file for a view controller to the project. Right-click on the *Storyboard* target at the top of the project navigator panel and select *New File...* from the resulting menu. In the new file panel, select *iOS* in the top bar, followed by *Cocoa Touch Class* in the main panel, and click *Next* to proceed. On the options screen, ensure that the *Subclass of* menu is set to *UIViewController* and that the *Also create XIB file* option is deselected (since the view already exists in the storyboard there is no need for a XIB user interface file), name the class *Scene2ViewController* and proceed through the screens to create the new class file.

Select the *Main.storyboard* file in the project navigator panel and click the View Controller button located in the panel above the Scene 2 view, as shown in Figure 25-7:

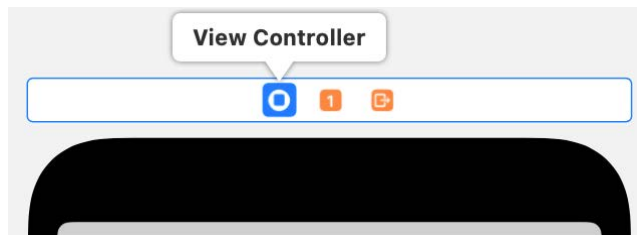


Figure 25-7

With the view controller for scene 2 selected within the storyboard canvas, display the Identity Inspector (*View -> Inspectors -> Identity*) and change the *Class* from *UIViewController* to *Scene2ViewController*:

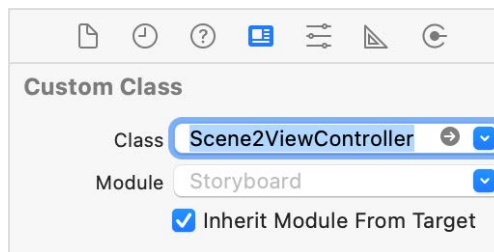


Figure 25-8

Scene 2 now has a view controller and corresponding Swift source file where code may be written to implement any required functionality.

Select the label object in scene 2 and display the Assistant Editor. Next, ensure that the *Scene2ViewController.swift* file is displayed in the editor, and then establish an outlet for the label named *scene2Label*.

## 25.7 Passing Data Between Scenes

One of the most common requirements when working with storyboards involves transferring data from one scene to another during a segue transition. Before the storyboard runtime environment performs a segue, a call is made to the *prepare(for segue:)* method of the current view controller. If any tasks need to be performed before the segue, implement this method in the current view controller and add code to perform any necessary tasks. Passed as an argument to this method is a segue object from which a reference to the destination view controller may be obtained and subsequently used to transfer data.

To see this in action, begin by selecting *Scene2ViewController.swift* and adding a new property variable:

```
import UIKit

class Scene2ViewController: UIViewController {

    @IBOutlet weak var scene2Label: UILabel!

    var labelText: String?
    .
    .
    .
}
```

This property will hold the text to be displayed on the label when the storyboard transitions to this scene. As such, some code needs to be added to the *viewDidLoad* method located in the *Scene2ViewController.swift* file:

```
override func viewDidLoad() {
    super.viewDidLoad()
    scene2Label.text = labelText
}
```

Finally, select the *ViewController.swift* file and implement the *prepare(for segue:)* method as follows:

```
override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
    let destination = segue.destination
                                as! Scene2ViewController
    destination.labelText = "Arrived from Scene 1"
}
```

This method obtains a reference to the destination view controller and then assigns a string to the *labelText* property of the object so that it appears on the label.

Rerun the app and note that the new label text appears when scene 2 is displayed. This is because we have, albeit using an elementary example, transferred data from one scene to the next.

## 25.8 Unwinding Storyboard Segues

The next step is configuring the button on scene 2 to return to scene 1. It might seem that the obvious choice is to implement a segue from the button in scene 2 to scene 1. Instead of returning to the original instance of scene 1, however, this would create an entirely new instance of the *ViewController* class. If a user were to perform this transition repeatedly, the app would continue using more memory and eventually be terminated by the operating system.

The app should instead make use of the Storyboard *unwind* feature. This involves implementing a method in the view controller of the scene to which the user is to be returned and then connecting a segue to that method from the source view controller. This enables an unwind action to be performed across multiple scene levels.

To implement this in our example app, begin by selecting the *ViewController.swift* file and implementing a method to be called by the unwind segue named *returned*:

```
@IBAction func returned(segue: UIStoryboardSegue) {
    scene1Label.text = "Returned from Scene 2"
}
```

All this method requires for this example is that it sets some new text on the label object of scene 1. Once the

method has been added, it is important to save the *ViewController.swift* file before continuing.

The next step is to establish the unwind segue. To achieve this, locate scene 2 within the storyboard canvas and right-click and drag from the button view to the Exit entry in the document outline panel, as shown in Figure 25-9. Release the line and select the *returnedWithSegue* method from the resulting menu:

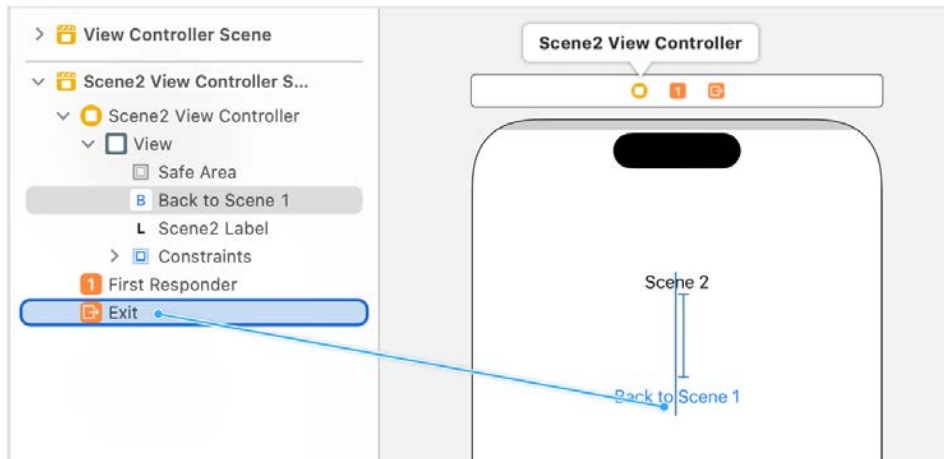


Figure 25-9

Once again, run the app and note that the button on scene 2 now returns to scene 1 and, in the process, calls the *returned* method resulting in the label on scene 1 changing.

## 25.9 Triggering a Storyboard Segue Programmatically

In addition to wiring up controls in scenes to trigger a segue, it is possible to initiate a preconfigured segue from within the app code. This can be achieved by assigning an identifier to the segue and then making a call to the *performSegue(withIdentifier:)* method of the view controller from which the segue is to be triggered.

To set the identifier of a segue, select it in the storyboard canvas, display the Attributes Inspector, and set the value in the *Identifier* field.

Assuming a segue with the identifier of *SegueToScene1*, this could be triggered from within code as follows:

```
self.performSegue(withIdentifier: "SegueToScene1", sender: self)
```

### 25.10 Summary

The Storyboard feature of Xcode allows for the navigational flow between the various views in an iOS app to be visually constructed without the need to write code. In this chapter, we have covered the basic concepts behind storyboarding, worked through creating an example iOS app using storyboards, and explored the storyboard unwind feature.



## 32. Working with the iOS 17 Stack View Class

With hindsight, it seems hard to believe, but until the introduction of iOS 9, there was no easy way to build stack-based user interface layouts that would adapt automatically to different screen sizes and changes in device orientation. While such results could eventually be achieved with careful use of size classes and Auto Layout, this was far from simple. That changed with the introduction of the `UIStackView` class in the iOS 9 SDK.

### 32.1 Introducing the `UIStackView` Class

The `UIStackView` class is a user interface element that allows subviews to be arranged linearly in a column or row orientation. The class extensively uses Auto Layout and automatically sets up many of the Auto Layout constraints needed to provide the required layout behavior. In addition, the class goes beyond simply stacking views, allowing additional Auto Layout constraints to be added to subviews, and providing a range of properties that enable the layout behavior of those subviews to be modified to meet different requirements.

The `UIStackView` object is available for inclusion within Storyboard scenes simply by dragging and dropping either the *Horizontal Stack View* or *Vertical Stack View* from the Library panel onto the scene canvas. Once added to a scene, subviews are added simply by dragging and dropping the required views onto the stack view.

Existing views in a storyboard scene may be wrapped in a stack view simply by Shift-clicking on the views so that they are all selected before clicking on the Embed In button located at the bottom of the Interface Builder panel, as highlighted in Figure 32-1 and selecting the Stack View option. Interface Builder will decide whether to encapsulate the selected views into a horizontal or vertical stack depending on the layout positions of the views:

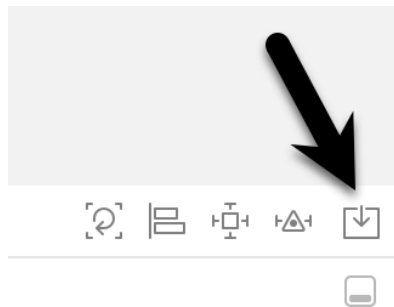


Figure 32-1

By default, the stack view will resize to accommodate the subviews as they are added. However, as with any other view type, Auto Layout constraints may be used to constrain and influence the resize behavior of the stack view in relation to the containing view and any other views in the scene layout.

Once added to a storyboard scene, a range of properties is available within the Attributes Inspector to customize the layout behavior of the object.

Stack views may be used to create simple column or row-based layouts or nested within each other to create more complex layouts. Figure 32-2, for example, shows an example layout consisting of a vertical stack view

containing three horizontal stack views, each containing a variety of subviews:

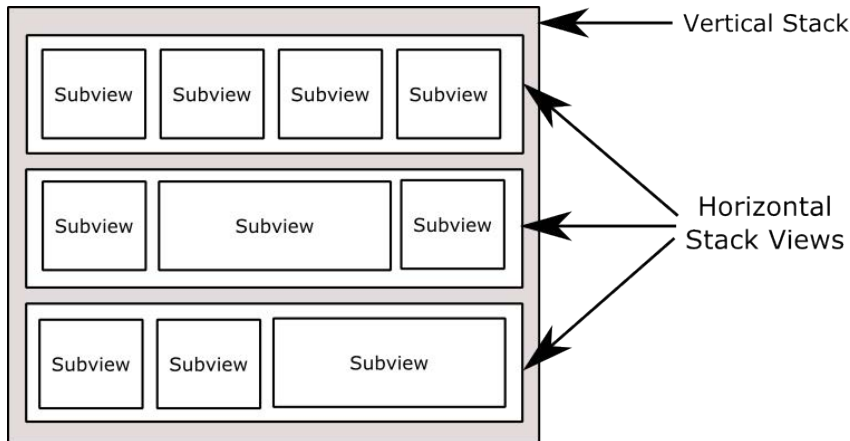


Figure 32-2

`UIStackView` class instances may also be created and managed from within the code of an iOS app. Stack view instances can be created in code and initialized with an array of subviews. Views may also be inserted and removed dynamically from within code, and the attributes of the stack view changed via a range of properties. The subviews of a stack view object are held in an array that can be accessed via the *arrangedSubviews* property of the stack view instance.

## 32.2 Understanding Subviews and Arranged Subviews

The `UIStackView` class contains a property named *subviews*. This is an array containing each of the child views of the stack view object. Figure 32-3, for example, shows the view hierarchy for a stack view with four subviews:

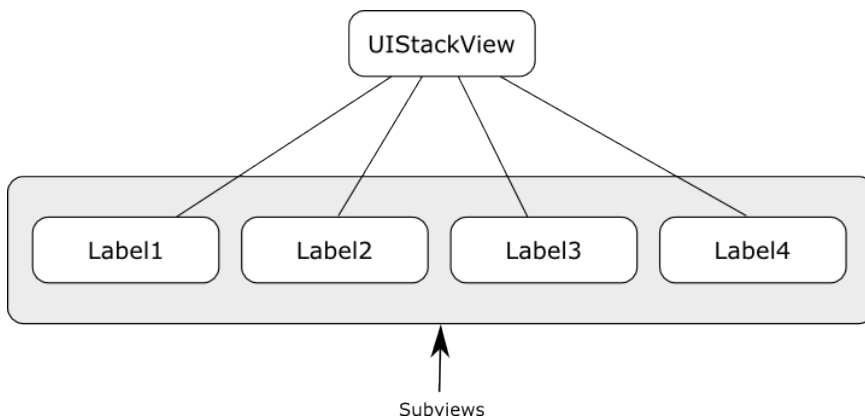


Figure 32-3

At any particular time, however, the stack view will not necessarily be responsible for arranging the layout and positions of all the subviews it contains. The stack view might, for example, only be configured to arrange the `Label3` and `Label4` views in the above hierarchy. This means that `Label1` and `Label2` may still be visible within the user interface but will not be positioned within the stack view. Subviews being arranged by the stack view are contained within a second array accessible via the *arrangedSubviews* property. Figure 32-4 shows both the subviews and the subset of the subviews which are currently being arranged by the stack view.



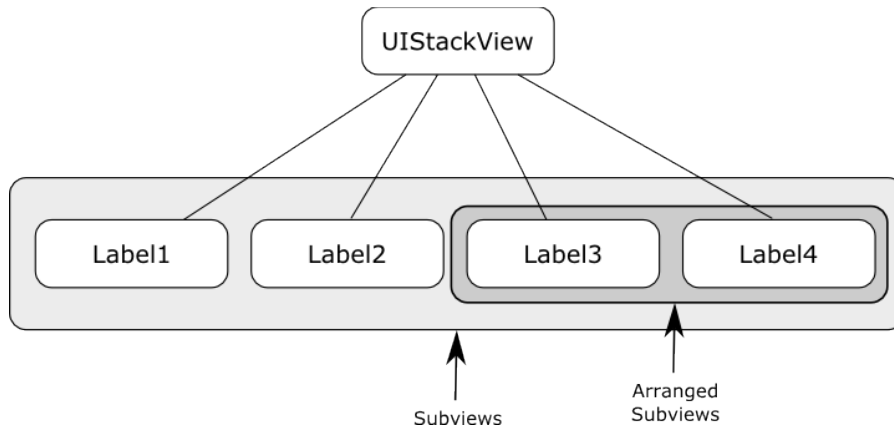


Figure 32-4

As will be outlined later in this chapter, the distinction between subview and arranged subviews is particularly important when removing arranged subviews from a stack view.

### 32.3 StackView Configuration Options

A range of options is available to customize how the stack view arranges its subviews. These properties are available both from within the Interface Builder Attributes Inspector panel at design time and also to be set dynamically from within the code of the app:

#### 32.3.1 axis

The `axis` property controls the orientation of the stack in terms of whether the subviews are arranged in a vertical column layout or a horizontal row. When setting this property in code, the axis should be set to `UILayoutConstraintAxis.vertical` or `UILayoutConstraintAxis.horizontal`.

#### 32.3.2 distribution

The `distribution` property dictates how the subviews of the stack view are sized. Options available are as follows:

- **Fill** – The subviews are resized to fill the entire space available along the stack view’s axis. In other words, the height of the subviews will be modified to fill the full height of the stack view in a vertical orientation, while the widths will be changed for a stack view in a horizontal orientation. The amount by which each subview is resized relative to the other views can be controlled via the compression resistance and hugging priorities of the views (details of which were covered in the chapter entitled “*An Introduction to Auto Layout in iOS 17*”) and the position of the views in the stack view’s `arrangedSubviews` array.

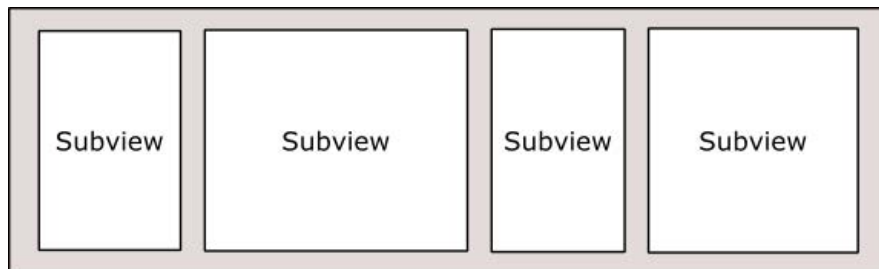


Figure 32-5

- **FillEqually** – The subviews are resized equally to fill the stack view along the view’s axis. Therefore, all the subviews in a vertical stack will be equal in height, while the subviews in a horizontal axis orientation will be

equal in width.

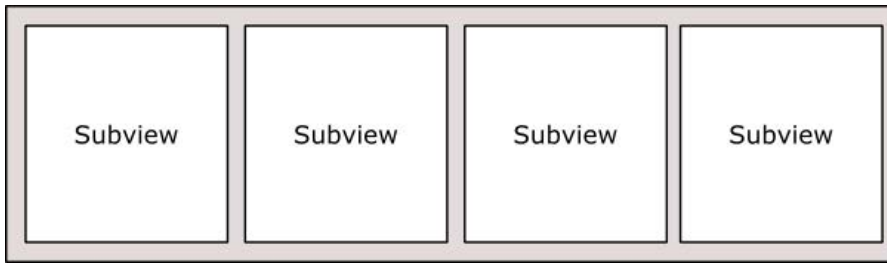


Figure 32-6

- **FillProportionally** – In this mode, the subviews are resized proportionally to their intrinsic content size along the axis of the stack view to fill the width or height of the view.
- **EqualSpacing** – Padding is used to space the subviews equally to fill the stack view along the axis. The size of the subviews will be reduced if necessary to fit within the available space based on the compression resistance priority setting and the position within the arrangedSubviews array.

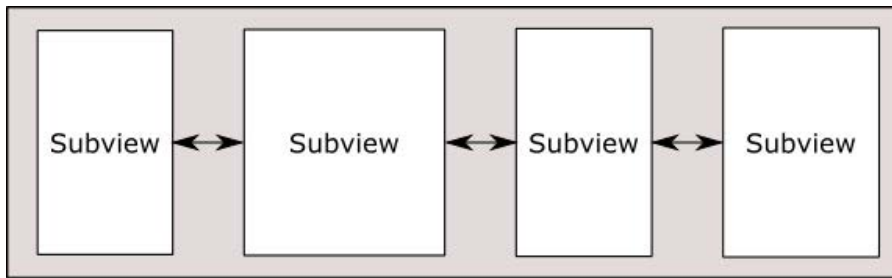


Figure 32-7

- **EqualCentering** – This mode positions the subviews along the stack view's axis with equal center-to-center spacing. The spacing in this mode is influenced by the spacing property (outlined below). Where possible, the stack view will honor the prevailing spacing property value but will reduce this value if necessary. If the views still do not fit, the size of the subviews will be reduced if necessary to fit within the available space based on the compression resistance priority setting and the position within the arrangedSubviews array.

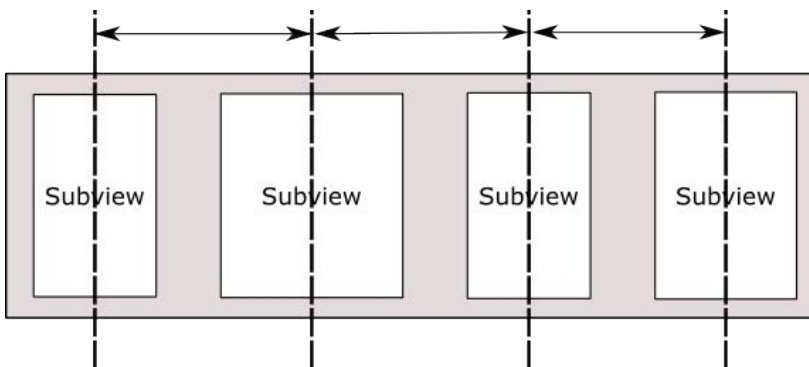


Figure 32-8

### 32.3.3 spacing

The spacing property specifies the distance (in points) between the edges of adjacent subviews within a stack view. When the stack view distribution property is set to *FillProportionally*, the spacing value dictates the spacing

between the subviews. In *EqualSpacing* and *EqualCentering* modes, the spacing value indicates the minimum allowed spacing between the adjacent edges of the subviews. A negative spacing value causes subviews to overlap.

### 32.3.4 alignment

The alignment property controls the positioning of the subviews perpendicularly to the stack view's axis. Available alignment options are as follows:

- **Fill** – In fill mode, the subviews are resized to fill the space perpendicularly to the stack view's axis. In other words, the widths of the subviews in a vertical stack view are resized to fill the entire width of the stack view.

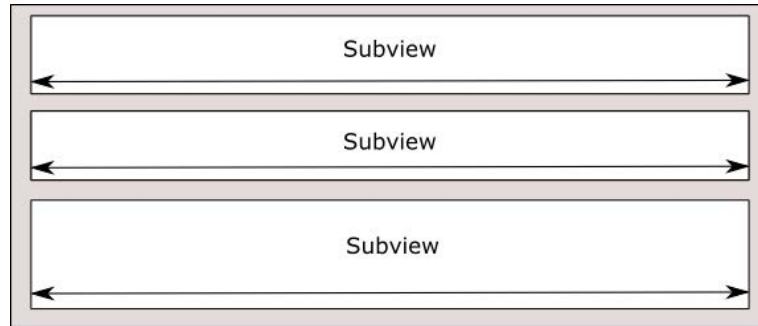


Figure 32-9

- **Leading** – In a vertically oriented stack view, the leading edges of the subviews are aligned with the leading edge of the stack view.

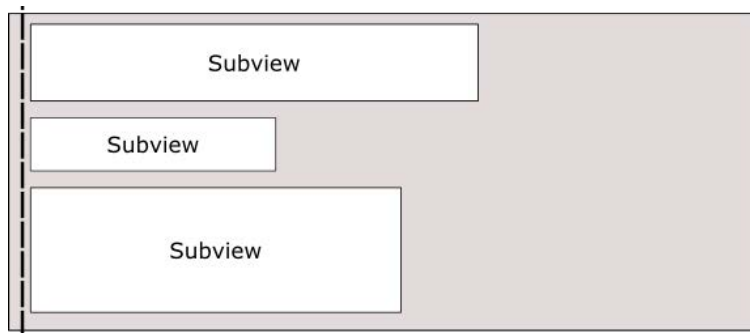


Figure 32-10

- **Trailing** – In a vertically oriented stack view, the trailing edges of the subviews are aligned with the trailing edge of the stack view.

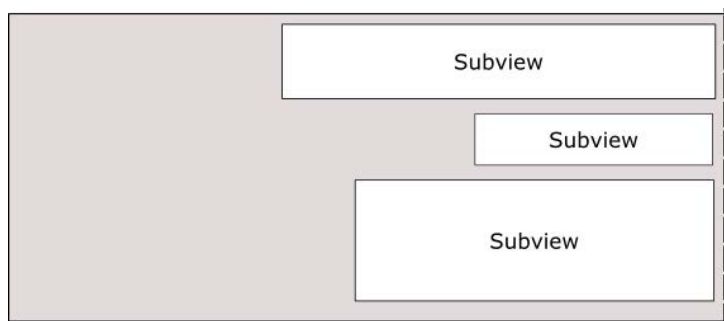


Figure 32-11

- **Top** – In a horizontally oriented stack view, the top edges of the subviews are aligned with the top edge of the stack view.

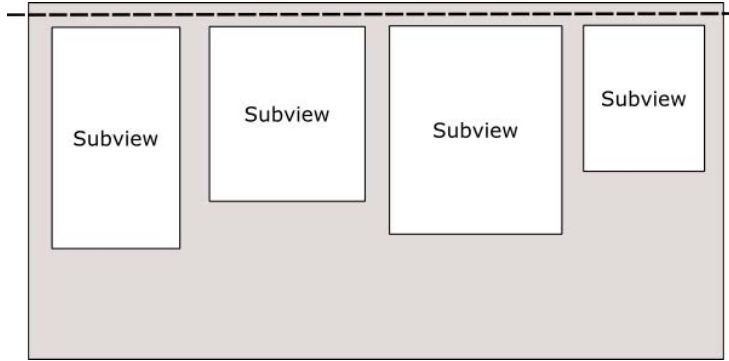


Figure 32-12

- **Bottom** – In a horizontally oriented stack view, the bottom edges of the subviews are aligned with the bottom edge of the stack view.

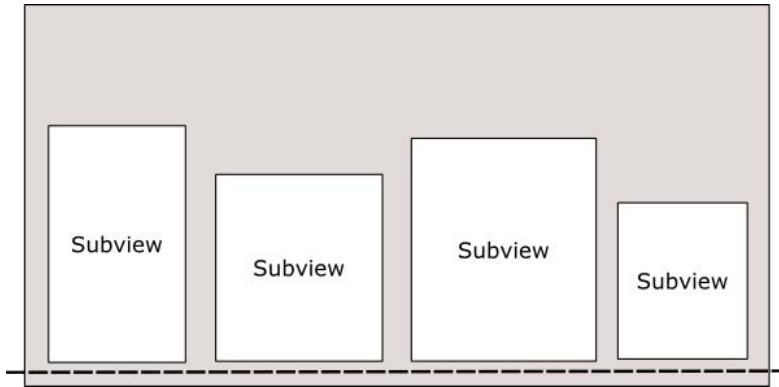


Figure 32-13

- **Center** – The centers of the subviews are aligned with the center axis of the stack view.

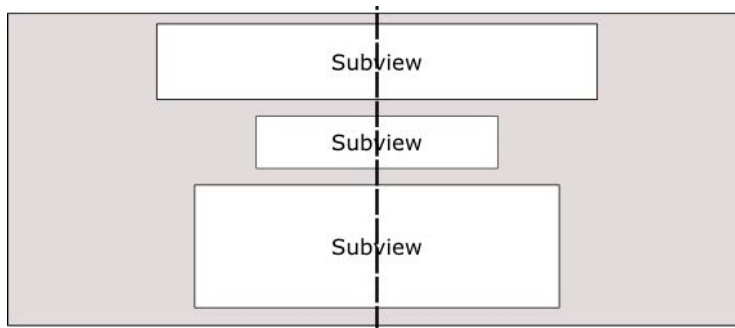


Figure 32-14

- **FirstBaseline** – Used only with horizontal stack views, this mode aligns all subviews with their first baseline. For example, an array of subviews displaying text content would all be aligned based on the vertical position of the first line of text.

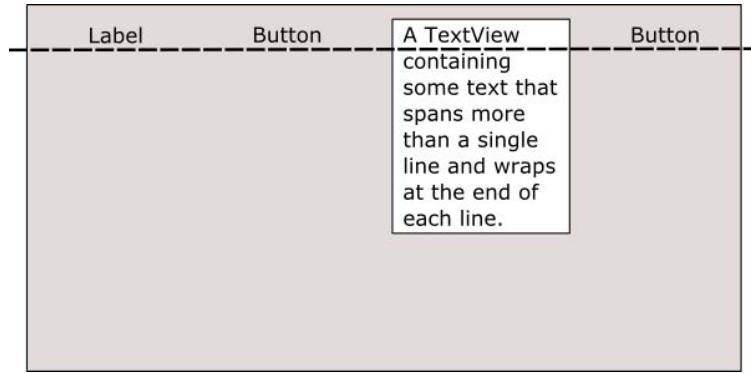


Figure 32-15

- **LastBaseline** – Similar to FirstBaseline, this mode aligns all subviews with their last baseline. For example, an array of subviews displaying text content would all be aligned based on the vertical position of the last line of text.

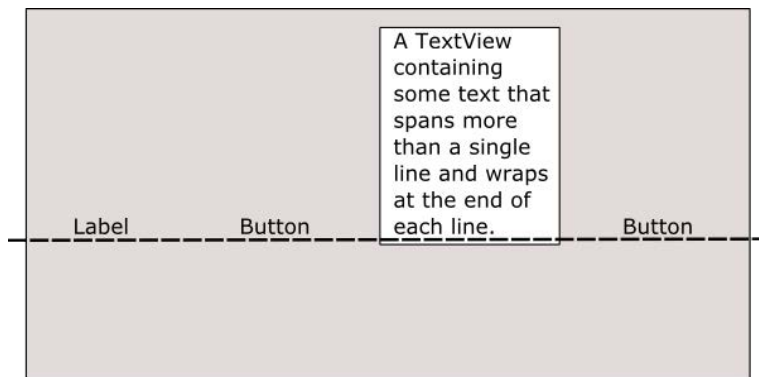


Figure 32-16

### 32.3.5 baseLineRelativeArrangement

Used only for vertical stack views, this property is a Boolean value that controls whether or not the vertical spacing between subviews is arranged relative to the baseline of the text contained within the views.

### 32.3.6 layoutMarginsRelativeArrangement

A Boolean value which, if set to true, causes subviews to be arranged relative to the layout margins of the containing stack view. If set to false, the subviews are arranged relative to the edges of the stack view.

## 32.4 Creating a Stack View in Code

UIStackView instances can be created in code by passing through an array object containing the subviews to be arranged by the stack. Once created, all the previously outlined properties may also be set dynamically from within the code. The following Swift code, for example, creates a new stack view object, configures it for horizontal axis orientation with *FillEqually* distribution, and assigns two Label objects as subviews:

```
let labelOne = UILabel(frame: CGRect(x: 0, y: 0, width: 200, height: 21))
labelOne.text = "Hello"
labelOne.backgroundColor = UIColor.red
```

```
let labelTwo = UILabel(frame: CGRect(x: 0, y: 0, width: 200, height: 21))
```

## Working with the iOS 17 Stack View Class

```
labelTwo.text = "There"
labelTwo.backgroundColor = UIColor.blue

let myStack = UIStackView(arrangedSubviews: [labelOne, labelTwo])

myStack.distribution = .fillEqually
myStack.axis = .horizontal
```

### 32.5 Adding Subviews to an Existing Stack View

Additional subviews may be appended to the end of a stack view's `arrangedSubviews` array using the `addArrangedSubview` method as follows:

```
myStack.addArrangedSubview(labelThree)
```

Alternatively, a subview may be inserted into a specific index position within the array of arranged subviews via a call to the `insertArrangedSubview:atIndex` method. The following line of code, for example, inserts an additional label at index position 0 within the `arrangedSubviews` array of a stack view:

```
myStack.insertArrangedSubview(labelZero, atIndex: 0)
```

### 32.6 Hiding and Removing Subviews

To remove an arranged subview from a stack view, call the `removeArrangedSubview` method of the stack view object, passing through the view object to be removed:

```
myStack.removeArrangedSubview(labelOne)
```

It is essential to be aware that the `removeArrangedSubview` method only removes the specified view from the `arrangedSubviews` array of the stack view. The view still exists in the `subviews` array and will probably still be visible within the user interface layout after removal (typically in the top left-hand corner of the stack view).

An alternative to removing the subview is to simply hide it. This has the advantage of making it easy to display the subview later within the app code. A helpful way to hide a subview is to obtain a reference to the subview to be hidden from within the `arrangedSubviews` array. For example, the following code identifies and then hides the subview located at index position 1 in the array of arranged subviews:

```
let subview = myStack.arrangedSubviews[1]
subview.hidden = true
```

If the subview is not needed again, however, it can be removed entirely by calling the `removeFromSuperview` method of the subview after it has been removed from the `arrangedSubviews` array as follows:

```
myStack.removeArrangedSubview(labelOne)
labelOne.removeFromSuperview()
```

This approach will remove the view entirely from the view hierarchy.

### 32.7 Summary

The `UIStackView` class allows user interface views to be arranged in rows or columns. A wide range of configuration options combined with the ability to dynamically create and manage stack views from within code make this a powerful and flexible user interface layout solution.

With the basics of the `UIStackView` class covered in this chapter, the next chapter will create an example iOS app that uses this class.

## 34. A Guide to iPad Multitasking

Since the introduction of iOS 9, users can display and interact with two apps side by side on the iPad screen, a concept referred to as *multitasking*. Although the inclusion of support for multitasking within an iPadOS app is optional, enabling support where appropriate is recommended to provide the user with the best possible experience when using the app.

This chapter will introduce multitasking in terms of what it means to the user and the steps that can be taken to effectively adopt and support multitasking within an iOS app running on an iPad device. Once these areas have been covered, the next chapter (*“An iPadOS Multitasking Example”*) will create an example project designed to support multitasking.

Before reading this chapter, it is important to understand that multitasking support makes extensive use of both the Size Classes and Auto Layout features of iOS, topics which were covered in the *“An Introduction to Auto Layout in iOS 17”* and *“Using Trait Variations to Design Adaptive iOS 17 User Interfaces”* chapters of this book.

### 34.1 Using iPad Multitasking

Before implementing multitasking support for an iPad app, it is first essential to understand multitasking from the user’s perspective. Traditionally, when an app was launched from the iPad screen, it would fill the entire display and continue to do so until placed into the background by the user. However, since the introduction of iOS 9, two apps can now share the iPad display.

Multitasking mode is initiated by tapping the three gray dots at the top of the screen to display the menu shown in Figure 34-1:

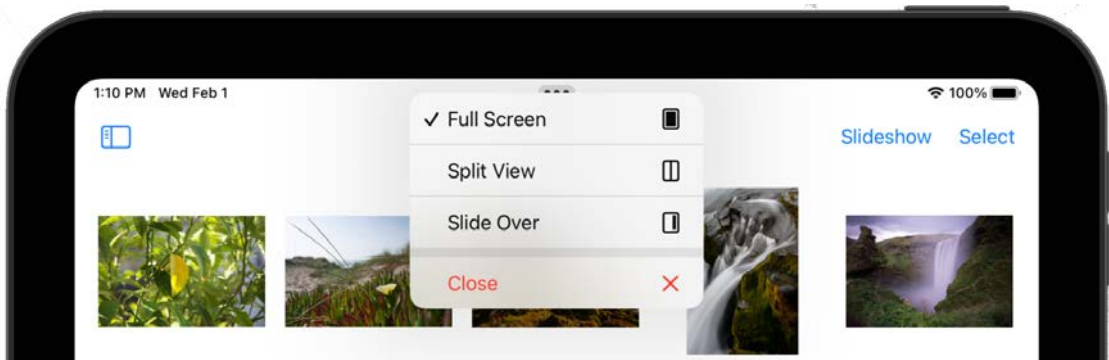


Figure 34-1

The Full Screen option, as the name suggests, will make the app currently in the foreground occupy the full device screen. On the other hand, the Split View option will split the screen between the current app and any other app of your choice. When selected, the current app will slide to the size allowing you to choose a second app from the launch screen:



Figure 34-2

Once a second app has been selected, both apps will appear in adjacent panels. Once the display is in Split View mode, touching and dragging the narrow white button located in the divider between the two panels (indicated in Figure 34-3) allows the position of the division between the primary and secondary apps to be adjusted:

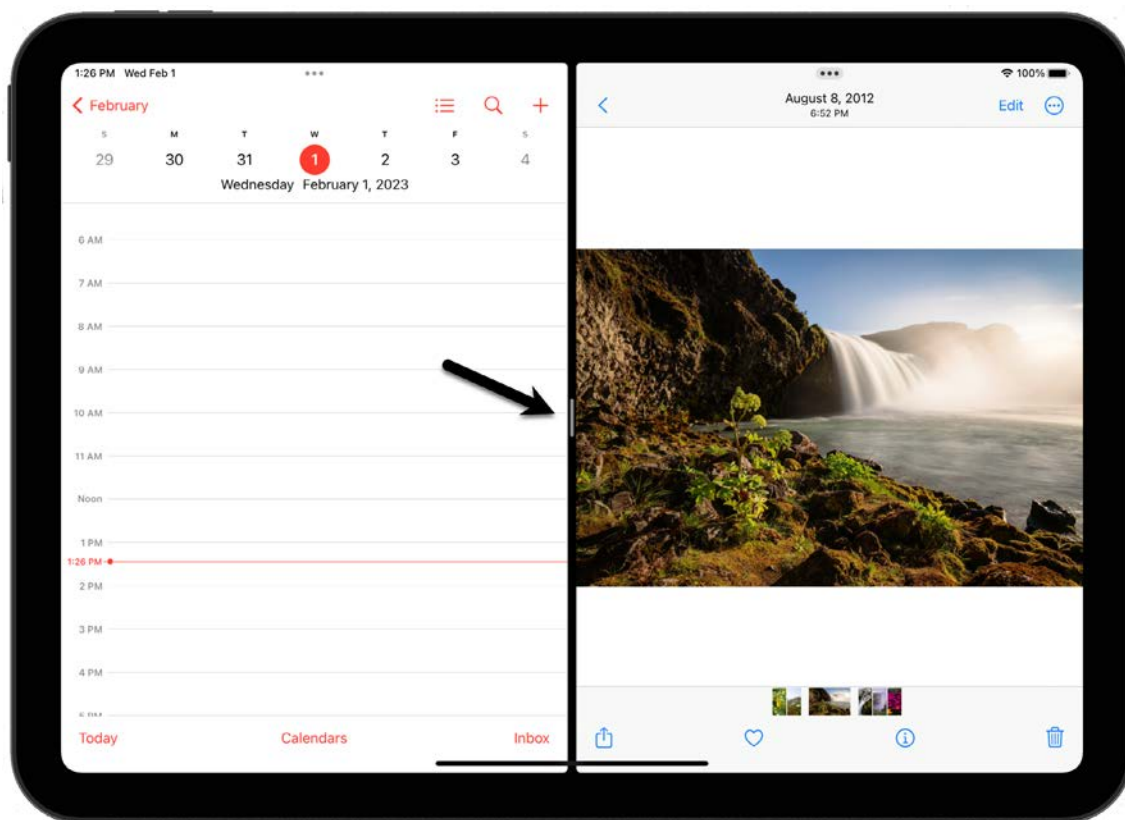


Figure 34-3

In Slide Over Mode, the app will appear in a floating window, as is the case in Figure 34-4 below:



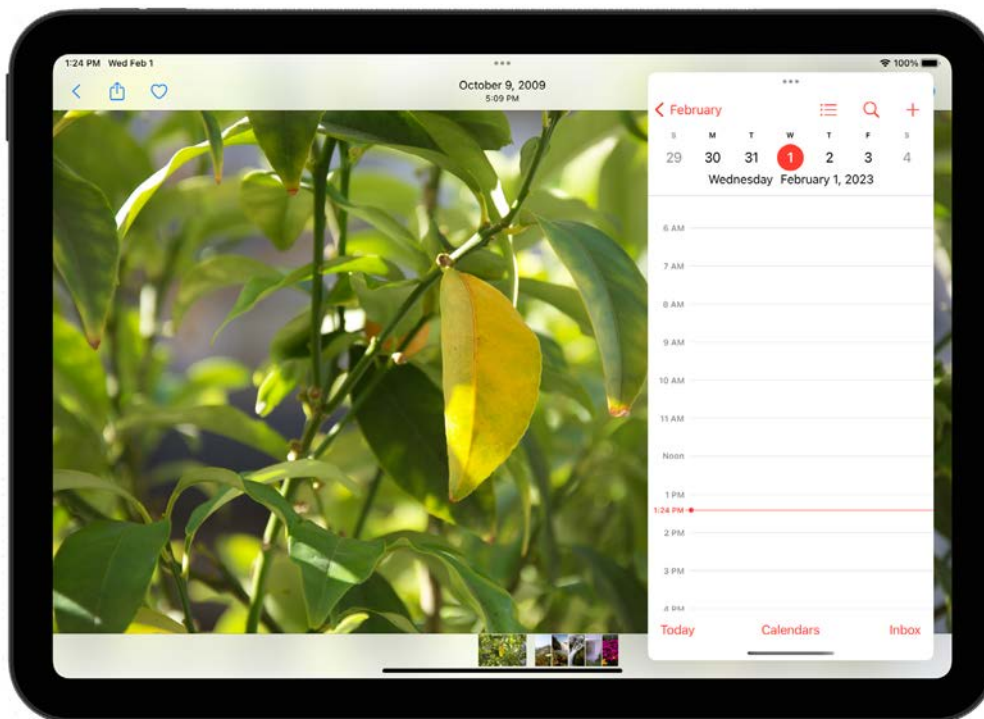


Figure 34-4

The three dots at the top of the secondary app can move the slide-over window to the left or right side of the screen by touching and dragging.

To hide a Slide Over view app, drag it sideways off the right side of the screen. Then, swipe left from the right-hand screen edge to restore the app.

## 34.2 Picture-In-Picture Multitasking

In addition to Split View and Slide Over modes, multitasking also supports the presentation of a movable and resizable video playback window over the top of the primary app window. This topic will be covered in the chapter entitled “*An iPadOS Multitasking Example*”.

## 34.3 Multitasking and Size Classes

A key part of supporting multitasking involves ensuring that the storyboard scenes within an app can adapt to various window sizes. Each of the different window sizes an app will likely encounter in a multitasking mode corresponds to one of the existing size classes outlined in the chapter entitled “*Using Trait Variations to Design Adaptive iOS 17 User Interfaces*”. As outlined in that chapter, the height and width available to an app are classified as *compact* or *regular*. So, for example, an app running in portrait mode on an iPhone 14 would use the compact width and regular height size class, while the same app running in portrait orientation would use the regular width and compact height size class.

When running in a multitasking environment, the primary and secondary apps will pass through a range of compact and regular widths depending on the prevailing multitasking configuration. The diagrams in Figure 34-4 illustrate how the different multitasking modes translate to equivalent regular and compact-size classes.

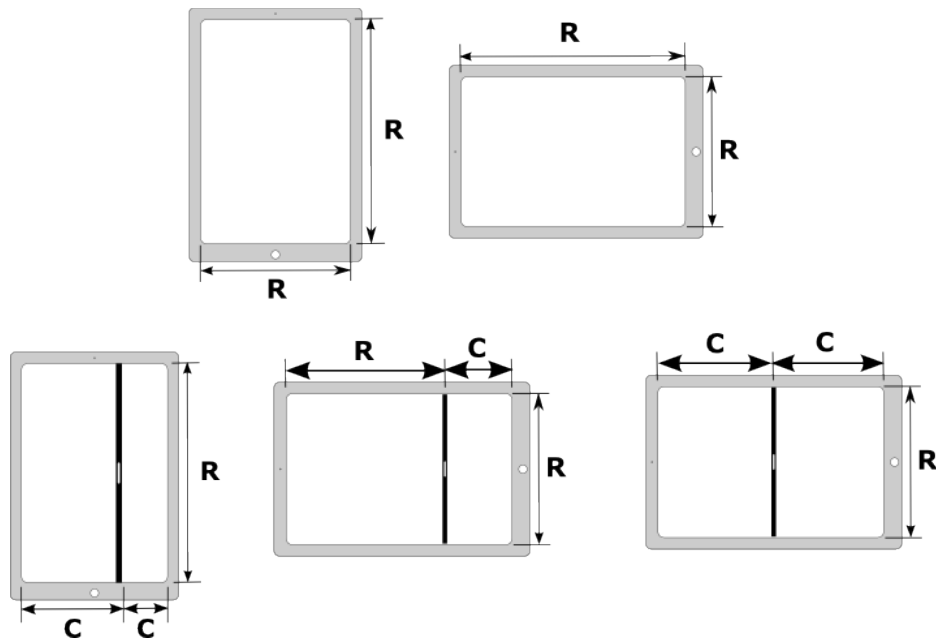


Figure 34-5

The above rules change slightly when the app runs in Split View mode on an iPad Pro. Due to the larger screen size of the iPad Pro, both apps are presented in Split View mode using the regular width, as illustrated in Figure 34-6:

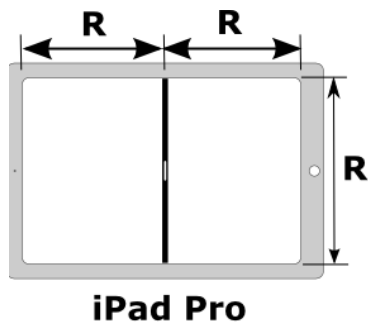


Figure 34-6

Implementing multitasking support within an iOS app involves designing layouts that adapt appropriately to the different size classes outlined in the above diagram.

## 34.4 Handling Multitasking in Code

Much can be achieved using Auto Layout and Size Classes to adapt to the size changes associated with multitasking. There will, however, inevitably be instances where some code needs to be executed when a scene transitions from one size class to another (for example, when an app transitions from Slide Over to Split View). Fortunately, UIKit will call three delegate methods on the container instance (typically a view controller) of the current scene during the transition to notify it of the transition where code can be added to perform app-specific tasks at different points in the transition. These delegate methods are outlined below in the order in which they are called during the transition:

### 34.4.1 willTransition(to newcollection: with coordinator:)

This method is called immediately before the traits for the currently displayed view controller view are changed. For example, the `UITraitCollection` class represents a trait, which contains a collection of values consisting of size class settings, the display density of the screen, and the user interface idiom (which is simply a value indicating whether the device is on which the app is running is an iPhone or iPad).

When called, this method is passed a `UITraitCollection` object containing the new trait collection from which information can be accessed and used to decide how to respond to the transition. The following code, for example, checks that the app is running on an iPad before identifying whether the horizontal size class is transitioning to a regular or compact size class:

```
override func willTransition(to newCollection: UITraitCollection, with
    coordinator: UIViewControllerTransitionCoordinator) {

    super.willTransition(to: newCollection,
        with: coordinator)

    if newCollection.userInterfaceIdiom == .pad
        if newCollection.horizontalSizeClass == .regular {
            // Transitioning to Regular Width Size Class
        } else if newCollection.horizontalSizeClass == .compact {
            // Transitioning to Compact Width Size Class
        }
    }
}
```

The second argument passed through to the method is a `UIViewControllerTransitionCoordinator` object. This is the coordinator object that is handling the transition and can be used to add additional animations to those being performed by UIKit during the transition.

### 34.4.2 viewWillTransition(to size: with coordinator:)

This method is also called before the size change is implemented and is passed a `CGSize` value and a coordinator object. The size object can be used to obtain the new height and width to which the view is transitioning. The following sample code outputs the new height and width values to the console:

```
override func viewWillTransition(to size: CGSize,
    with coordinator:
    UIViewControllerTransitionCoordinator) {

    super.viewWillTransition(to: size,
        with: coordinator)

    print("Height = \(size.height), Width = \(size.width)")
}
```

### 34.4.3 traitCollectionDidChange(\_:)

This method is called once the transition from one trait collection to another is complete and is passed the `UITraitCollection` object for the previous trait. In the following example implementation, the method checks to find out whether the previous horizontal size class was regular or compact:

```
override func traitCollectionDidChange(_ previousTraitCollection:
    UITraitCollection?) {

    super.traitCollectionDidChange(previousTraitCollection)

    if previousTraitCollection?.horizontalSizeClass == .regular {
        // The previous horizontal size class was regular
    }

    if previousTraitCollection?.horizontalSizeClass == .compact {
        // The previous horizontal size class was compact
    }
}
```

### 34.5 Lifecycle Method Calls

In addition to the transition delegate methods outlined above, several lifecycle methods are called on the app's application delegate during a multitasking transition. For example, when the user moves the divider, the *applicationWillResignActive* method is called at the point that the divider position changes. Likewise, when the user slides the divider to the edge of the screen so that the app is no longer visible, the *applicationDidEnterBackground* delegate method is called.

These method calls are particularly important when considering what happens behind the scenes when the user moves the divider. As the divider moves, the system repeatedly resizes the app off-screen and takes snapshots of various sizes as the slider moves. These snapshots make the sliding transition appear to take place smoothly. The *applicationWillResignActive* method may need to be used to preserve the state of the user interface so that when the user releases the divider, the same data and navigation position within the user interface is presented as before the slider change.

### 34.6 Opting Out of Multitasking

To disable multitasking support for an app, add the *UIRequiresFullScreen* key to the project's *Info.plist* file with the value set to true. This can be set manually within the *Info.plist* file itself or the *Deployment Info* section of the *General* settings panel for the project target:



Figure 34-7

## 34.7 Summary

Multitasking allows the user to display and interact with two apps concurrently when running on recent models of iPad devices. Multitasking apps are categorized as primary and secondary and can be displayed in either Slide Over or Split View configurations. Multitasking also supports a “Picture-in-Picture” option whereby video playback is displayed in a floating, resizable window over the top of the existing app.

Supporting multitasking within an iOS app primarily involves designing the user interface to support both regular and compact-size classes. A range of delegate methods also allows view controllers to receive size change notifications and respond accordingly.

Projects created in Xcode 15 are configured to support multitasking by default. However, it is also possible to opt out of multitasking with a change to the project *Info.plist* file.



## 49. Working with iOS 17 Databases using Core Data

The preceding chapters covered the concepts of database storage using the SQLite database. In these chapters, the assumption was made that the iOS app code would directly manipulate the database using SQLite API calls to construct and execute SQL statements. While this is a good approach for working with SQLite in many cases, it does require knowledge of SQL and can lead to some complexity in terms of writing code and maintaining the database structure. The non-object-oriented nature of the SQLite API functions further compounds this complexity. In recognition of these shortcomings, Apple introduced the Core Data Framework. Core Data is essentially a framework that places a wrapper around the SQLite database (and other storage environments), enabling the developer to work with data in terms of Swift objects without requiring any knowledge of the underlying database technology.

We will begin this chapter by defining some concepts that comprise the Core Data model before providing an overview of the steps involved in working with this framework. Once these topics have been covered, the next chapter will work through *“An iOS 17 Core Data Tutorial”*.

### 49.1 The Core Data Stack

Core Data consists of several framework objects that integrate to provide the data storage functionality. This stack can be visually represented as illustrated in Figure 49-1.

As shown in Figure 49-1, the iOS-based app sits on top of the stack and interacts with the managed data objects handled by the managed object context. Of particular significance in this diagram is that although the lower levels in the stack perform a considerable amount of the work involved in providing Core Data functionality, the app code does not interact with them directly.

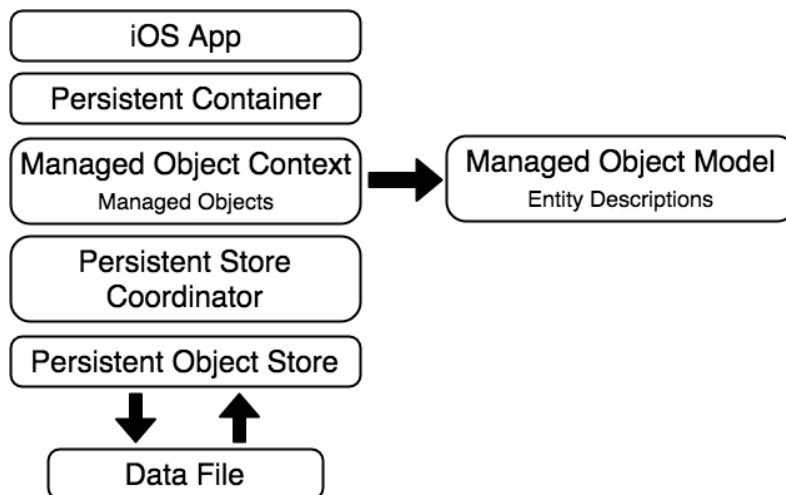


Figure 49-1

Before moving on to the more practical areas of working with Core Data, it is essential to explain the elements

that comprise the Core Data stack in a little more detail.

## 49.2 Persistent Container

The *persistent container* handles the creation of the Core Data stack and is designed to be easily subclassed to add additional app-specific methods to the base Core Data functionality. Once initialized, the persistent container instance provides access to the *managed object context*.

## 49.3 Managed Objects

*Managed objects* are the objects that are created by your app code to store data. For example, a managed object may be considered a row or a record in a relational database table. For each new record to be added, a new managed object must be created to store the data. Similarly, retrieved data will be returned as managed objects, one for each record matching the defined retrieval criteria. Managed objects are actually instances of the *NSManagedObject* class or a subclass thereof. These objects are contained and maintained by the *managed object context*.

## 49.4 Managed Object Context

Core Data based apps never interact directly with the persistent store. Instead, the app code interacts with the managed objects contained in the managed object context layer of the Core Data stack. The context maintains the status of the objects in relation to the underlying data store and manages the relationships between managed objects defined by the *managed object model*. All interactions with the underlying database are held temporarily within the context until the context is instructed to save the changes. At this point, the changes are passed down through the Core Data stack and written to the persistent store.

## 49.5 Managed Object Model

So far, we have focused on managing data objects but have not yet looked at how the data models are defined. This is the task of the *Managed Object Model*, which defines a concept referred to as *entities*.

Much as a class description defines a blueprint for an object instance, entities define the data model for managed objects. Essentially, an entity is analogous to the schema that defines a table in a relational database. As such, each entity has a set of attributes associated with it that define the data to be stored in managed objects derived from that entity. For example, a *Contacts* entity might contain *name*, *address*, and *phone number* attributes.

In addition to attributes, entities can also contain *relationships*, *fetch properties*, and *fetch requests*:

- **Relationships** – In the context of Core Data, relationships are the same as those in other relational database systems in that they refer to how one data object relates to another. Core Data relationships can be one-to-one, one-to-many, or many-to-many.
- **Fetch property** – This provides an alternative to defining relationships. Fetch properties allow properties of one data object to be accessed from another as though a relationship had been defined between those entities. Fetch properties lack the flexibility of relationships and are referred to by Apple's Core Data documentation as "weak, one-way relationships" best suited to "loosely coupled relationships."
- **Fetch request** – A predefined query that can be referenced to retrieve data objects based on defined predicates. For example, a fetch request can be configured into an entity to retrieve all contact objects where the name field matches "John Smith."

## 49.6 Persistent Store Coordinator

The *persistent store coordinator* coordinates access to multiple *persistent object stores*. As an iOS developer, you will never directly interact with the persistence store coordinator. In fact, you will very rarely need to develop an app that requires more than one persistent object store. When multiple stores are required, the coordinator



presents these stores to the upper layers of the Core Data stack as a single store.

## 49.7 Persistent Object Store

The term *persistent object store* refers to the underlying storage environment in which data are stored when using Core Data. Core Data supports three disk-based and one memory-based persistent store. Disk-based options consist of SQLite, XML, and binary. By default, the iOS SDK will use SQLite as the persistent store. In practice, the type of store used is transparent to you as the developer. Regardless of your choice of persistent store, your code will make the same calls to the same Core Data APIs to manage the data objects required by your app.

## 49.8 Defining an Entity Description

Entity descriptions may be defined from within the Xcode environment. For example, when a new project is created with the option to include Core Data, a template file will be created named `<projectname>.xcdatamodeld`. Selecting this file in the Xcode project navigator panel will load the model into the entity editing environment, as illustrated in Figure 49-2:

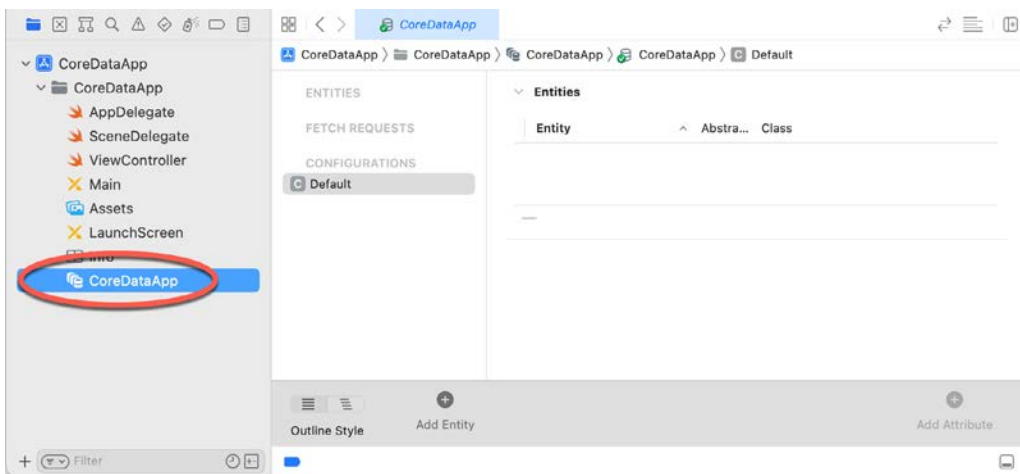


Figure 49-2

Create a new entity by clicking on the *Add Entity* button located in the bottom panel. The new entity will appear as a text box in the *Entities* list. By default, this will be named *Entity*. Double-click on this name to change it.

To add attributes to the entity, click the *Add Attribute* button in the bottom panel or use the + button beneath the *Attributes* section. Then, in the *Attributes* panel, name the attribute and specify the type and any other required options.

Repeat the above steps to add more attributes and additional entities.

The Xcode entity environment also allows relationships to be established between entities. Assume, for example, two entities named *Contacts* and *Sales*. First, select the *Contacts* entity and click on the + button beneath the *Relationships* panel to establish a relationship between the two tables. Then, in the detail panel, name the relationship, specify the destination as the *Sales* entity, and any other options required for the relationship.

As demonstrated, Xcode makes the process of entity description creation reasonably straightforward. While a detailed overview of the process is beyond this book's scope, many other resources are dedicated to the subject.

## 49.9 Initializing the Persistent Container

The persistent container is initialized by creating a new `NSPersistentContainer` instance, passing through the name of the model to be used, and then making a call to the `loadPersistentStores` method of that object as follows:

```
let container = NSPersistentContainer(name: "CoreDataDemo")
container.loadPersistentStores(completionHandler: {
    (description, error) in
        if let error = error {
            fatalError("Unable to load persistent stores: \(error)")
        }
    })
```

### 49.10 Obtaining the Managed Object Context

Since many Core Data methods require the managed object context as an argument, the next step after defining entity descriptions often involves obtaining a reference to the context. This can be achieved by accessing the `viewContext` property of the persistent container instance:

```
let managedObjectContext = persistentContainer.viewContext
```

### 49.11 Getting an Entity Description

Before managed objects can be created and manipulated in code, the corresponding entity description must first be loaded. This is achieved by calling the `entity(forName:in:)` method of the `NSEntityDescription` class, passing through the name of the required entity and the context as arguments. For example, the following code fragment obtains the description for an entity with the name *Contacts*:

```
let entity = NSEntityDescription.entity(
    forName: "Contacts", in: context)
```

### 49.12 Setting the Attributes of a Managed Object

As previously discussed, entities and the managed objects from which they are instantiated contain data in the form of attributes. Once a managed object instance has been created, as outlined above, those attribute values can store the data before the object is saved. For example, assuming a managed object named *contact* with attributes named *name*, *address*, and *phone*, respectively, the values of these attributes may be set as follows before the object is saved to storage:

```
contact.name = "John Smith"
contact.address = "1 Infinite Loop"
contact.phone = "555-564-0980"
```

### 49.13 Saving a Managed Object

Once a managed object instance has been created and configured with the data to be stored, it can be saved to storage using the `save` method of the managed object context as follows:

```
do {
    try context.save()
} catch let error {
    // Handle error
}
```

### 49.14 Fetching Managed Objects

Once managed objects are saved into the persistent object store, those objects and the data they contain will likely need to be retrieved. Objects are retrieved by executing a fetch request and are returned in an array. The following code assumes that both the context and entity description have been obtained before making the fetch request:

```
let request: NSFetchedRequest<Contacts> = Contacts.fetchRequest()
request.entity = entity
```

```
do {
    let results = try context.fetch(request as!
                                   NSFetchRequest<NSFetchRequestResult>)
} catch let error {
    // Handle error
}
```

Upon execution, the *results* array will contain all the managed objects retrieved by the request.

## 49.15 Retrieving Managed Objects based on Criteria

The preceding example retrieved all managed objects from the persistent object store for a specified entity. More often than not, only managed objects that match specified criteria are required during a retrieval operation. This is performed by defining a *predicate* that dictates criteria a managed object must meet to be eligible for retrieval. For example, the following code implements a predicate to extract only those managed objects where the *name* attribute matches “John Smith”:

```
let request: NSFetchRequest<Contacts> = Contacts.fetchRequest()
request.entity = entity

let pred = NSPredicate(format: "(name = %@)", "John Smith")
request.predicate = pred

do {
    let results = try context.fetch(request as!
                                   NSFetchRequest<NSFetchRequestResult>)
} catch let error {
    // Handle error
}
```

## 49.16 Accessing the Data in a Retrieved Managed Object

Once results have been returned from a fetch request, the data within the returned objects may be accessed using *keys* to reference the stored values. The following code, for example, accesses the first result from a fetch operation results array and extracts the values for the *name*, *address*, and *phone* keys from that managed object:

```
let match = results[0] as! NSManagedObject

let nameString = match.value(forKey: "name") as! String
let addressString = match.value(forKey: "address") as! String
let phoneString = match.value(forKey: "phone") as! String
```

## 49.17 Summary

The Core Data Framework stack provides a flexible alternative to directly managing data using SQLite or other data storage mechanisms. Providing an object-oriented abstraction layer on top of the data makes managing data storage significantly easier for the iOS app developer. Now that the basics of Core Data have been covered, the next chapter, entitled “An iOS 17 Core Data Tutorial”, will work through creating an example app.



## 65. iOS 17 UIKit Dynamics – An Overview

UIKit Dynamics provides a powerful and flexible mechanism for combining user interaction and animation into iOS user interfaces. What distinguishes UIKit Dynamics from other approaches to animation is the ability to declare animation behavior in terms of real-world physics.

Before moving on to a detailed tutorial in the next chapter, this chapter will provide an overview of the concepts and methodology behind UIKit Dynamics in iOS.

### 65.1 Understanding UIKit Dynamics

UIKit Dynamics allows for the animation of user interface elements (typically view items) to be implemented within a user interface, often in response to user interaction. To fully understand the concepts behind UIKit Dynamics, it helps to visualize how real-world objects behave.

Holding an object in the air and then releasing it, for example, will cause it to fall to the ground. This behavior is, of course, the result of gravity. However, whether or not, and by how much, an object bounces upon impact with a solid surface is dependent upon that object's elasticity and its velocity at the point of impact.

Similarly, pushing an object positioned on a flat surface will cause that object to travel a certain distance depending on the magnitude and angle of the pushing force combined with the level of friction at the point of contact between the two surfaces.

An object tethered to a moving point will react in various ways, such as following the anchor point, swinging in a pendulum motion, or even bouncing and spinning on the tether in response to more aggressive motions. However, an object similarly attached using a spring will behave entirely differently in response to the movement of the point of attachment.

Considering how objects behave in the real world, imagine the ability to selectively apply these same physics-related behaviors to view objects in a user interface, and you will begin understanding the basic concepts behind UIKit Dynamics. Not only does UIKit Dynamics allow user interface interaction and animation to be declared using concepts we are already familiar with, but in most cases, it allows this to be achieved with just a few simple lines of code.

### 65.2 The UIKit Dynamics Architecture

Before looking at how UIKit Dynamics are implemented in app code, it helps to understand the different elements that comprise the dynamics architecture.

The UIKit Dynamics implementation comprises four key elements: a *dynamic animator*, a set of one or more *dynamic behaviors*, one or more *dynamic items*, and a *reference view*.

#### 65.2.1 Dynamic Items

The dynamic items are the view elements within the user interface to be animated in response to specified dynamic behaviors. A dynamic item is any view object that implements the *UIDynamicItem* protocol, which includes the *UIView* and *UICollectionView* classes and any subclasses thereof (such as *UIButton* and *UILabel*).

Any custom view item can work with UIKit Dynamics by conforming to the `UIDynamicItem` protocol.

### 65.2.2 Dynamic Behaviors

Dynamic behaviors are used to configure the behavior to be applied to one or more dynamic items. A range of predefined dynamic behavior classes is available, including `UIAttachmentBehavior`, `UICollisionBehavior`, `UIGravityBehavior`, `UIDynamicItemBehavior`, `UIPushBehavior`, and `UISnapBehavior`. Each is a subclass of the `UIDynamicBehavior` class, which will be covered in detail later in this chapter.

In general, an instance of the class corresponding to the desired behavior (`UIGravityBehavior` for gravity, for example) will be created, and the dynamic items for which the behavior is to be applied will be added to that instance. Dynamic items can be assigned to multiple dynamic behavior instances simultaneously and may be added to or removed from a dynamic behavior instance during runtime.

Once created and configured, behavior objects are added to the *dynamic animator* instance. Once added to a dynamic animator, the behavior may be removed at any time.

### 65.2.3 The Reference View

The reference view dictates the area of the screen within which the UIKit Dynamics animation and interaction are to take place. This is typically the parent superclass view or collection view, of which the dynamic item views are children.

### 65.2.4 The Dynamic Animator

The dynamic animator coordinates the dynamic behaviors and items and works with the underlying physics engine to perform the animation. The dynamic animator is represented by an instance of the `UIDynamicAnimator` class and is initialized with the corresponding reference view at creation time. Once created, suitably configured dynamic behavior instances can be added and removed as required to implement the desired user interface behavior.

The overall architecture for a UIKit Dynamics implementation can be represented visually using the diagram outlined in Figure 65-1:

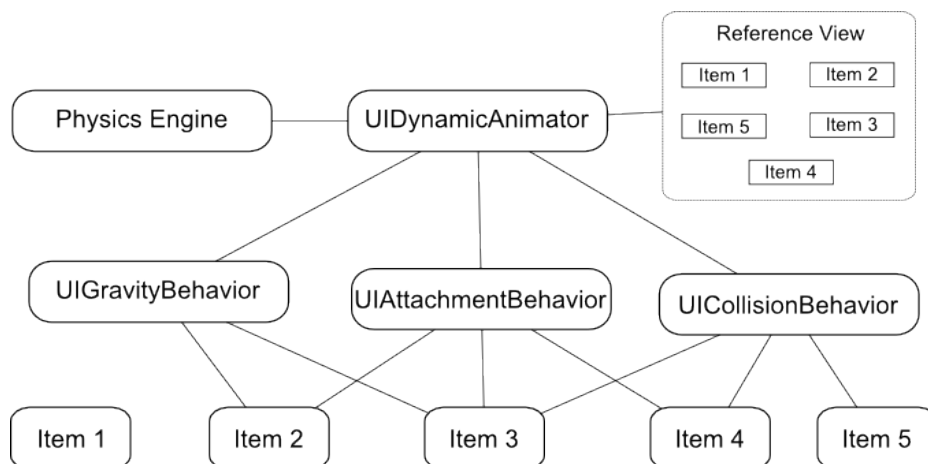


Figure 65-1

The above example has added three dynamic behaviors to the dynamic animator instance. The reference view contains five dynamic items, all but one of which have been added to at least one dynamic behavior instance.

## 65.3 Implementing UIKit Dynamics in an iOS App

The implementation of UIKit Dynamics in an app requires three very simple steps:

1. Create an instance of the *UIDynamicAnimator* class to act as the dynamic animator and initialize it with reference to the reference view.
2. Create and configure a dynamic behavior instance and assign to it the dynamic items on which the specified behavior is to be imposed.
3. Add the dynamic behavior instance to the dynamic animator.
4. Repeat from step 2 to create and add additional behaviors.

## 65.4 Dynamic Animator Initialization

The first step in implementing UIKit Dynamics is to create and initialize an instance of the *UIDynamicAnimator* class. The first step is to declare an instance variable for the reference:

```
var animator: UIDynamicAnimator?
```

Next, the dynamic animator instance can be created. The following code, for example, creates and initializes the animator instance within the *viewDidLoad* method of a view controller, using the view controller's parent view as the reference view:

```
override func viewDidLoad() {
    super.viewDidLoad()
    animator = UIDynamicAnimator(referenceView: self.view)
}
```

With the dynamic animator created and initialized, the next step is to configure behaviors, the details for which differ slightly depending on the nature of the behavior.

## 65.5 Configuring Gravity Behavior

Gravity behavior is implemented using the *UIGravityBehavior* class, the purpose of which is to cause view items to want to “fall” within the reference view as though influenced by gravity. UIKit Dynamics gravity is slightly different from real-world gravity in that it is possible to define a vector for the direction of the gravitational force using x and y components (x, y) contained within a *CGVector* instance. The default vector for this class is (0.0, 1.0), corresponding to downward acceleration at a speed of 1000 points per second<sup>2</sup>. A negative x or y value will reverse the direction of gravity.

A *UIGravityBehavior* instance can be initialized as follows, passing through an array of dynamic items on which the behavior is to be imposed (in this case, two views named *view1* and *view2*):

```
let gravity = UIGravityBehavior(items: [view1, view2])
```

Once created, the default vector can be changed if required at any time:

```
let vector = CGVectorMake(0.0, 0.5)
gravity.gravityDirection = vector
```

Finally, the behavior needs to be added to the dynamic animator instance:

```
animator?.addBehavior(gravity)
```

At any point during the app lifecycle, dynamic items may be added to, or removed from, the behavior:

```
gravity.addItem(view3)
gravity.removeItem(view)
```

Similarly, the entire behavior may be removed from the dynamic animator:

```
animator?.removeBehavior(gravity)
```

When gravity behavior is applied to a view, and in the absence of opposing behaviors, the view will immediately move in the direction of the specified gravity vector. In fact, as currently defined, the view will fall out of the bounds of the reference view and disappear. This can be prevented by setting up a collision behavior.

## 65.6 Configuring Collision Behavior

UIKit Dynamics is all about making items move on the device display. When an item moves, there is a high chance it will collide either with another item or the boundaries of the encapsulating reference view. As previously discussed, in the absence of any form of collision behavior, a moving item can move out of the visible area of the reference view. Such a configuration will also cause a moving item to simply pass over the top of any other items that happen to be in its path. Collision behavior (defined using the `UICollisionBehavior` class) allows such collisions to behave in ways more representative of the real world.

Collision behavior can be implemented between dynamic items (such that certain items can collide with others) or within boundaries (allowing collisions to occur when an item reaches a designated boundary). Boundaries can be defined such that they correspond to the boundaries of the reference view, or entirely new boundaries can be defined using lines and Bezier paths.

As with gravity behavior, a collision is generally created and initialized with an array object containing the items to which the behavior is to be applied. For example:

```
let collision = UICollisionBehavior(items: [view1, view2])
animator?.addBehavior(collision)
```

As configured, `view1` and `view2` will now collide when coming into contact. The physics engine will decide what happens depending on the items' elasticity and the collision's angle and speed. In other words, the engine will animate the items to behave as if they were physical objects subject to the laws of physics.

By default, an item under the influence of a collision behavior will collide with other items in the same collision behavior set and any boundaries set up. To declare the reference view as a boundary, set the *translatesReferenceBoundsIntoBoundary* property of the behavior instance to *true*:

```
collision.translatesReferenceBoundsIntoBoundary = true
```

A boundary inset from the edges of the reference view may be defined using the *setsTranslateReferenceBoundsIntoBoundaryWithInsets* method, passing through the required insets as an argument in the form of a *UIEdgeInsets* object.

The *collisionMode* property may be used to change default collision behavior by assigning one of the following constants:

- **`UICollisionBehaviorMode.items`** – Specifies that collisions only occur between items added to the collision behavior instance. Boundary collisions are ignored.
- **`UICollisionBehaviorMode.boundaries`** – Configures the behavior to ignore item collisions, recognizing only collisions with boundaries.
- **`UICollisionBehaviorMode.everything`** – Specifies that collisions occur between items added to the behavior and all boundaries. This is the default behavior.

The following code, for example, enables collisions only for items:

```
collision.collisionMode = UICollisionBehaviorMode.items
```



If an app needs to react to a collision, declare a class instance that conforms to the `UICollisionBehaviorDelegate` class by implementing the following methods and assign it as the delegate for the `UICollisionBehavior` object instance.

- `collisionBehavior(_:beganContactForItem:withBoundaryIdentifier:atPoint:)`
- `collisionBehavior(_:beganContactForItem:withItem:atPoint:)`
- `collisionBehavior(_:endedContactForItem:withBoundaryIdentifier:)`
- `collisionBehavior(_:endedContactForItem:withItem:)`

When implemented, the app will be notified when collisions begin and end. In most cases, the delegate methods will be passed information about the collision, such as the location and the items or boundaries involved.

In addition, aspects of the collision behavior, such as friction and the elasticity of the colliding items (such that they bounce on contact), may be configured using the `UIDynamicBehavior` class. This class will be covered in detail later in this chapter.

## 65.7 Configuring Attachment Behavior

As the name suggests, the `UIAttachmentBehavior` class allows dynamic items to be configured to behave as if attached. These attachments can take the form of two items attached or an item attached to an anchor point at specific coordinates within the reference view. In addition, the attachment can take the form of an imaginary piece of cord that does not stretch or a spring attachment with configurable damping and frequency properties that control how “bouncy” the attached item is in response to motion.

By default, the attachment point within the item itself is positioned at the center of the view. This can, however, be changed to a different position causing the real-world behavior outlined in Figure 65-2 to occur:

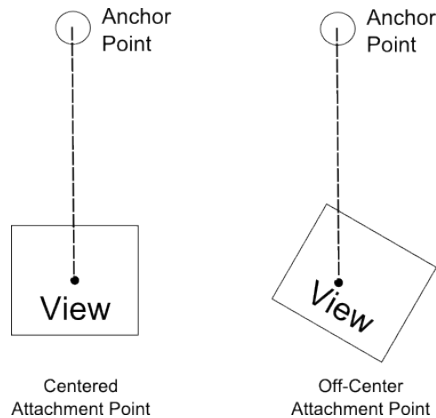


Figure 65-2

The physics engine will generally simulate animation to match what would typically happen in the real world. As illustrated above, the item will tilt when not attached in the center. If the anchor point moves, the attached view will also move. Depending on the motion, the item will swing in a pendulum motion and, assuming appropriate collision behavior configuration, bounce off any boundaries it collides with as it swings.

As with all UIKit Dynamics behavior, the physics engine performs all the work to achieve this. The only effort required by the developer is to write a few lines of code to set up the behavior before adding it to the dynamic animator instance. The following code, for example, sets up an attachment between two dynamic items:

```
let attachment = UIAttachmentBehavior(item: view1,
```

```

                                attachedToItem: view2)
animator?.addBehavior(attachment)

```

The following code, on the other hand, specifies an attachment between view1 and an anchor point with the frequency and damping values set to configure a spring effect:

```

let anchorpoint = CGPoint(x: 100, y: 100)
let attachment = UIAttachmentBehavior(item: view1,
                                attachedToAnchor: anchorPoint)
attachment.frequency = 4.0
attachment.damping = 0.0

```

The above examples attach to the center point of the view. The following code fragment sets the same attachment as above, but with an attachment point offset 20, 20 points relative to the center of the view:

```

let anchorpoint = CGPoint(x: 100, y: 100)
let offset = UIOffset(horizontal: 20, vertical: 20)

let attachment = UIAttachmentBehavior(item: view1,
                                offsetFromCenter: offset,
                                attachedToAnchor: anchorPoint)

```

## 65.8 Configuring Snap Behavior

The `UISnapBehavior` class allows a dynamic item to be “snapped” to a specific location within the reference view. When implemented, the item will move toward the snap location as though pulled by a spring and, depending on the damping property specified, oscillate several times before finally snapping into place. Until the behavior is removed from the dynamic animator, the item will continue to snap to the location when subsequently moved to another position.

The damping property can be set to any value between 0.0 and 1.0, with 1.0 specifying maximum oscillation. The default value for damping is 0.5.

The following code configures snap behavior for dynamic item view1 with damping set to 1.0:

```

let point = CGPoint(x: 100, y: 100)
let snap = UISnapBehavior(item: view1, snapToPoint: point)
snap.damping = 1.0

animator?.addBehavior(snap)

```

## 65.9 Configuring Push Behavior

Push behavior, defined using the `UIPushBehavior` class, simulates the effect of pushing one or more dynamic items in a specific direction with a specified force. The force can be specified as continuous or instantaneous. In the case of a continuous push, the force is continually applied, causing the item to accelerate over time. The instantaneous push is more like a “shove” than a push in that the force is applied for a short pulse causing the item to gain velocity quickly but gradually lose momentum and eventually stop. Once an instantaneous push event has been completed, the behavior is disabled (though it can be re-enabled).

The direction of the push can be defined in radians or using x and y components. By default, the pushing force is applied to the center of the dynamic item, though, as with attachments, this can be changed to an offset relative to the center of the view.

A force of magnitude 1.0 is defined as being a force of one UIKit Newton, which equates to a view sized at 100

x 100 points with a density of value 1.0 accelerating at a rate of 100 points per second<sup>2</sup>. As explained in the next section, the density of a view can be configured using the `UIDynamicItemBehavior` class.

The following code pushes an item with instantaneous force at a magnitude of 0.2 applied on both the x and y axes, causing the view to move diagonally down and to the right:

```
let push = UIPushBehavior(items: [view1],
                           mode: UIPushBehaviorMode.instantaneous)

let vector = CGVector(dx: 0.2, dy: 0.2)
push.pushDirection = vector
```

Continuous push behavior can be achieved by changing the *mode* in the above code property to `UIPushBehaviorMode.continuous`.

To change the point where force is applied, configure the behavior using the `setTargetOffsetFromCenter(_:for:)` method of the behavior object, specifying an offset relative to the center of the view. For example:

```
let offset = UIOffset(horizontal: 20, vertical: 20)
push.setTargetOffsetFromCenter(offset, for:view1)
```

In most cases, an off-center target for the pushing force will cause the item to rotate as it moves, as indicated in Figure 65-3:

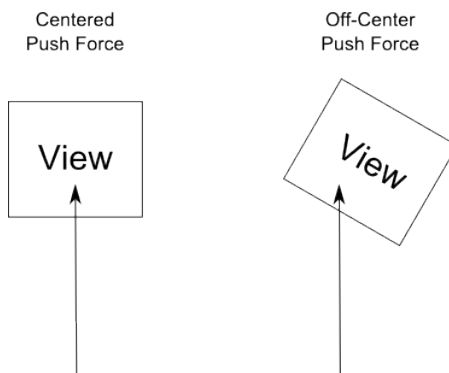


Figure 65-3

## 65.10 The `UIDynamicItemBehavior` Class

The `UIDynamicItemBehavior` class allows additional behavior characteristics to be defined that complement a number of the above primitive behaviors. This class can, for example, be used to define the density, resistance, and elasticity of dynamic items so that they do not move as far when subjected to an instantaneous push or bounce to a greater extent when involved in a collision. Dynamic items also can rotate by default. If rotation is not required for an item, this behavior can be turned off using a `UIDynamicItemBehavior` instance.

The behavioral properties of dynamic items that the `UIDynamicItemBehavior` class can govern are as follows:

- **allowsRotation** – Controls whether or not the item is permitted to rotate during animation.
- **angularResistance** – The amount by which the item resists rotation. The higher the value, the faster the item will stop rotating.
- **density** – The mass of the item.
- **elasticity** – The amount of elasticity an item will exhibit when involved in a collision. The greater the elasticity, the more the item will bounce.

- **friction** – The resistance exhibited by an item when it slides against another item.
- **resistance** – The overall resistance that the item exhibits in response to behavioral influences. The greater the value, the sooner the item will come to a complete stop during animation.

In addition, the class includes the following methods that may be used to increase or decrease the angular or linear velocity of a specified dynamic item:

- **angularVelocity(for:)** – Increases or decreases the angular velocity of the specified item. Velocity is specified in radians per second, where a negative value reduces the angular velocity.
- **linearVelocity(for:)** – Increases or decreases the linear velocity of the specified item. Velocity is specified in points per second, where a negative value reduces the velocity.

The following code example creates a new `UIDynamicItemBehavior` instance and uses it to set resistance and elasticity for two views before adding the behavior to the dynamic animator instance:

```
let behavior = UIDynamicItemBehavior(items: [view1, view2])
behavior.elasticity = 0.2
behavior.resistance = 0.5
animator?.addBehavior(behavior)
```

## 65.11 Combining Behaviors to Create a Custom Behavior

Multiple behaviors may be combined to create a single custom behavior using an instance of the `UIDynamicBehavior` class. The first step is to create and initialize each of the behavior objects. An instance of the `UIDynamicBehavior` class is then created, and each behavior is added to it via calls to the *addChildBehavior* method. Once created, only the `UIDynamicBehavior` instance needs to be added to the dynamic animator. For example:

```
// Create multiple behavior objects here

let customBehavior = UIDynamicBehavior()

customBehavior.addChildBehavior(behavior)
customBehavior.addChildBehavior(attachment)
customBehavior.addChildBehavior(gravity)
customBehavior.addChildBehavior(push)

animator?.addBehavior(customBehavior)
```

## 65.12 Summary

UIKit Dynamics provides a new way to bridge the gap between user interaction with an iOS device and corresponding animation within an app user interface. UIKit Dynamics takes a novel approach to animation by allowing view items to be configured such that they behave in much the same way as physical objects in the real world. This chapter has covered an overview of the basic concepts behind UIKit Dynamics and provided some details on how such behavior is implemented in terms of coding. The next chapter will work through a tutorial demonstrating many of these concepts.

## 78. An Introduction to Extensions in iOS 17

Extensions are a feature originally introduced as part of the iOS 8 release designed to allow certain capabilities of an app to be made available for use within other apps. For example, the developer of a photo editing app might have devised some unique image filtering capabilities and decided that those features would be particularly useful to users of the iOS Photos app. To achieve this, the developer would implement these features in a Photo Editing extension which would then appear as an option to users when editing an image within the Photos app.

Extensions fall into various categories, and several rules and guidelines must be followed in the implementation process. While subsequent chapters will cover the creation of extensions of various types in detail, this chapter is intended to serve as a general overview and introduction to the subject of extensions in iOS.

### 78.1 iOS Extensions – An Overview

The sole purpose of an extension is to make a specific feature of an existing app available for access within other apps. Extensions are separate executable binaries that run independently of the corresponding app. Although extensions are individual binaries, they must be supplied and installed as part of an app bundle. The app with which an extension is bundled is called the *containing app*. Except for Message App extensions, the containing app must provide useful functionality. An empty app must not be provided solely to deliver an extension to the user.

Once an extension has been installed, it will be accessible from other apps through various techniques depending on the extension type. The app from which an extension is launched and used is referred to as a *host app*.

For example, an app that translates text to a foreign language might include an extension that can be used to translate the text displayed by a host app. In such a scenario, the user would access the extension via the Share button in the host app's user interface, and the extension would display a view controller displaying the translated text. On dismissing the extension, the user is returned to the host app.

### 78.2 Extension Types

iOS supports several different extension types dictated by *extension points*. An extension point is an area of the iOS operating system that has been opened up to allow extensions to be implemented. When developing an extension, it is important to select the extension point most appropriate to the extension's features. The extension types supported by iOS are constantly evolving, though the key types can be summarized as follows:

#### 78.2.1 Share Extension

Share extensions provide a quick access mechanism for sharing content such as images, videos, text, and websites within a host app with social network sites or content-sharing services. It is important to understand that Apple does not expect developers to write Share extensions designed to post content to destinations such as Facebook or Twitter (such sharing options are already built into iOS) but rather as a mechanism to make sharing easier for developers hosting their own sharing and social sites. Share extensions appear within the activity view controller panel when the user taps the Share button from within a host app.

Share extensions can use the `SLComposeServiceViewController` class to implement the interface for posting

content. This class displays a view containing a preview of the information to be posted and provides the ability to modify the content before posting it. In addition, a custom user interface can be designed using Interface Builder for more complex requirements.

The actual mechanics of posting the content will depend on how the target platform works.

### 78.2.2 Action Extension

The Action extension point enables extensions to be created that fall into the Action category. Action extensions allow the content within a host app to be transformed or viewed differently. As with Share extensions, Action extensions are accessed from the activity view controller via the Share button. Figure 78-1, for example, shows an example Action extension named “Change It Up” in the activity view controller of the iOS Notes app.

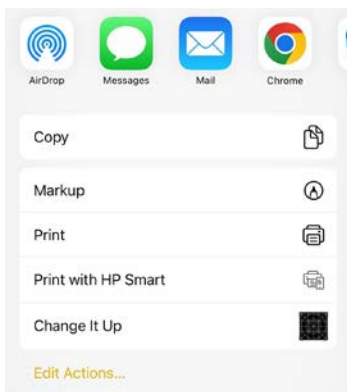


Figure 78-1

Action extensions are context-sensitive in that they only appear as an option when the type of content in the host app matches one of the content types for which the extension has declared support. For example, an Action extension that works with images will not appear in the activity view controller panel for a host app displaying text-based content.

Action extensions are covered in detail in the chapters entitled “*Creating an iOS 17 Action Extension*” and “*Receiving Data from an iOS 17 Action Extension*”.

### 78.2.3 Photo Editing Extension

The Photo Editing extension allows an app’s photo editing capabilities to be accessed from within the built-in iOS Photos app. Photo Editing extensions are displayed when the user selects an image in the Photos app, chooses the edit option, and taps on the button in the top left-hand corner of the Photo editing screen. For example, Figure 78-2 shows the Photos app displaying two Photo Editing extension options:

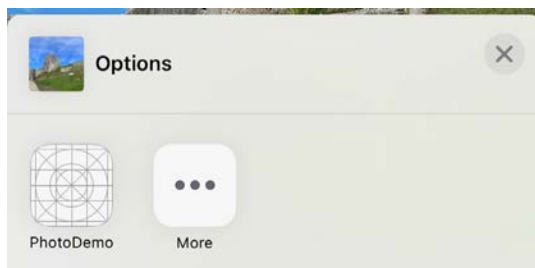


Figure 78-2

Photo Editing Extensions are covered in detail in the chapter entitled “*Creating an iOS 17 Photo Editing*”

*Extension”.*

#### 78.2.4 Document Provider Extension

The Document Provider extension allows a containing app to act as a document repository for other apps running on the device. Depending on the level of support implemented in the extension, host apps can import, export, open, and move documents to and from the storage repository the containing app provides. In most cases, the storage repository represented by the containing app will be a third-party cloud storage service providing an alternative to Apple’s iCloud service.

A Document Provider extension consists of a Document Picker View Controller extension and an optional File Provider extension. The Document Picker View Controller extension provides a user interface, allowing users to browse and select the documents available for the Document Provider extension.

The optional File Provider extension provides the host app with access to the documents outside the app’s sandbox and is necessary if the extension is to support move and open operations on the documents stored via the containing app.

#### 78.2.5 Custom Keyboard Extension

As the name suggests, the Custom Keyboard Extension allows creating and installing custom keyboards onto iOS devices. Keyboards developed using the Custom Keyboard extension point are available to be used by all apps on the device and, once installed, are selected from within the keyboard settings section of the Settings app on the device.

#### 78.2.6 Audio Unit Extension

Audio Unit Extensions allow sound effects, virtual instruments, and other sound-based capabilities to be available to other audio-oriented host apps such as GarageBand.

#### 78.2.7 Shared Links Extension

The Shared Links Extension provides a mechanism for an iOS app to store URL links in the Safari browser shared links list.

#### 78.2.8 Content Blocking Extension

Content Blocking allows extensions to be added to the Safari browser to block certain types of content from appearing when users browse the web. This feature is typically used to create ad-blocking solutions.

#### 78.2.9 Sticker Pack Extension

An extension to the built-in iOS Messages App that allows packs of additional images to be provided for inclusion in the message content.

#### 78.2.10 iMessage Extension

Allows interactive content to be integrated into the Messages app. This can range from custom user interfaces to interactive games. iMessage extensions are covered in the chapters entitled “*An Introduction to Building iOS 17 Message Apps*” and “*An iOS 17 Interactive Message App Tutorial*”.

#### 78.2.11 Intents Extension

When integrating an app with the SiriKit framework, these extensions define the actions to be performed in response to voice commands using Siri.

### 78.3 Creating Extensions

By far, the easiest approach to developing extensions is to use the extension templates provided by Xcode. Once the project for a containing app is loaded into Xcode, extensions can be added as new targets by selecting the

*File -> New -> Targets...* menu option. This will display the panel shown in Figure 78-3, listing a template for each of the extension types:

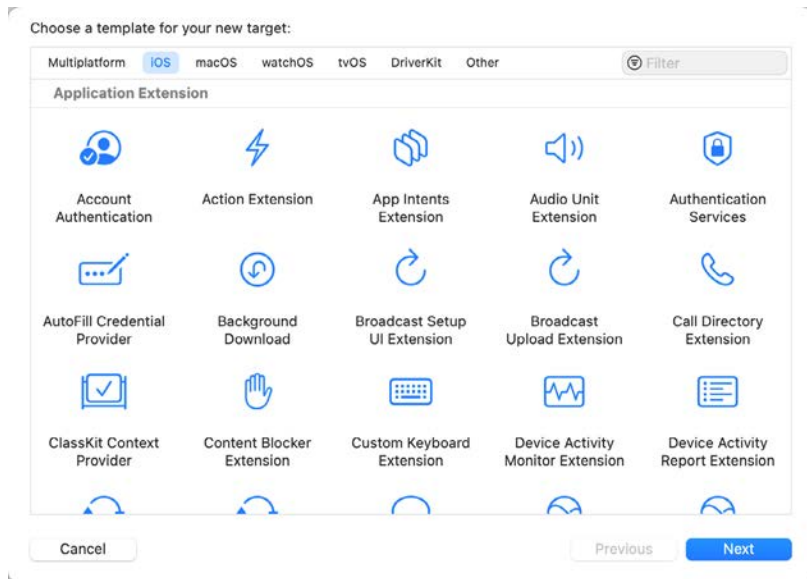


Figure 78-3

Once an extension template is selected, click on *Next* to name and create the template. Once the extension has been created from the template, the steps to implement the extension will differ depending on the type of extension selected. The next few chapters will detail the steps in implementing Photo Editing, Action, and Message app extensions.

## 78.4 Summary

Extensions in iOS provide a way for narrowly defined areas of functionality of one app to be made available from within other apps. iOS 17 currently supports a variety of extension types. When developing extensions, it is important to select the most appropriate extension point before beginning development work and to be aware that some app features may not be appropriate candidates for an extension.

Although extensions run as separate independent binaries, they can only be installed as part of an app bundle. The app with which an extension is bundled is called a *containing app*. An app from which an extension is launched is called a *host app*.

Having covered the basics of extensions in this chapter, subsequent chapters will focus in detail on the more commonly used extension types.



## 92. An Introduction to iOS 17 Sprite Kit Programming

Suppose you have ever had an idea for a game but didn't create it because you lacked the skills or time to write complex game code and logic; look no further than Sprite Kit. Introduced as part of the iOS 7 SDK, Sprite Kit allows 2D games to be developed relatively easily.

Sprite Kit provides almost everything needed to create 2D games for iOS, watchOS, tvOS, and macOS with minimum coding. Sprite Kit's features include animation, physics simulation, collision detection, and special effects. These features can be harnessed within a game with just a few method calls.

In this and the next three chapters, the topic of games development with Sprite Kit will be covered to bring the reader up to a level of competence to begin creating games while also providing a knowledge base on which to develop further Sprite Kit development skills.

### 92.1 What is Sprite Kit?

Sprite Kit is a programming framework that makes it easy for developers to implement 2D-based games that run on iOS, macOS, tvOS, and watchOS. It provides a range of classes that support the rendering and animation of graphical objects (otherwise known as *sprites*) that can be configured to behave in specific programmer-defined ways within a game. Through *actions*, various activities can be run on sprites, such as animating a character so that it appears to be walking, making a sprite follow a specific path within a game scene, or changing the color and texture of a sprite in real-time.

Sprite Kit also includes a physics engine allowing physics-related behavior to be imposed on sprites. For example, a sprite can, amongst other things, be made to move by subjecting it to a pushing force, configured to behave as though affected by gravity, or to bounce back from another sprite as the result of a collision.

In addition, the Sprite Kit particle emitter class provides a useful mechanism for creating special effects within a game, such as smoke, rain, fire, and explosions. A range of templates for existing special effects is provided with Sprite Kit and an editor built into Xcode for creating custom particle emitter-based special effects.

### 92.2 The Key Components of a Sprite Kit Game

A Sprite Kit game will typically consist of several different elements.

#### 92.2.1 Sprite Kit View

Every Sprite Kit game will have at least one SKView class. An SKView instance sits at the top of the component hierarchy of a game and is responsible for displaying the game content to the user. It is a subclass of the UIView class and, as such, has many of the traits of that class, including an associated view controller.

#### 92.2.2 Scenes

A game will also contain one or more scenes. One scene might, for example, display a menu when the game starts, while additional scenes may represent multiple levels within the game. Scenes are represented in a game by the SKScene class, a subclass of the SKNode class.

### 92.2.3 Nodes

Each scene within a Sprite Kit game will have several Sprite Kit node children. These nodes fall into several different categories, each of which has a dedicated Sprite Kit node class associated with it. These node classes are all subclasses of the `SKNode` class and can be summarized as follows:

- **SKSpriteNode** – Draws a sprite with a texture. These textures will typically be used to create image-based characters or objects in a game, such as a spaceship, animal, or monster.
- **SKLabelNode** – Used to display text within a game, such as menu options, the prevailing score, or a “game over” message.
- **SKShapeNode** – Allows nodes to be created containing shapes defined using Core Graphics paths. If a sprite is required to display a circle, for example, the `SKShapeNode` class could be used to draw the circle as an alternative to texturing an `SKSpriteNode` with an image of a circle.
- **SKEmitterNode** – The node responsible for managing and displaying particle emitter-based special effects.
- **SKVideoNode** – Allows video playback to be performed within a game node.
- **SKEffectNode** – Allows Core Image filter effects to be applied to child nodes. A sepia filter effect, for example, could be applied to all child nodes of an `SKEffectNode`.
- **SKCropNode** – Allows the pixels in a node to be cropped subject to a specified mask.
- **SKLightNode** – The lighting node is provided to add light sources to a SpriteKit scene, including casting shadows when the light falls on other nodes in the same scene.
- **SK3DNode** – The `SK3DNode` allows 3D assets created using the Scene Kit Framework to be embedded into 2D Sprite Kit games.
- **SKFieldNode** – Applies physics effects to other nodes within a specified area of a scene.
- **SKAudioNode** – Allows an audio source using 3D spacial audio effects to be included in a Sprite Kit scene.
- **SKCameraNode** – Provides the ability to control the position from which the scene is viewed. The camera node may also be adjusted dynamically to create panning, rotation, and scaling effects.

### 92.2.4 Physics Bodies

Each node within a scene can have associated with it a physics body. Physics bodies are represented by the `SKPhysicsBody` class. Assignment of a physics body to a node brings a wide range of possibilities in terms of the behavior associated with a node. When a node is assigned a physics body, it will, by default, behave as though subject to the prevailing forces of gravity within the scene. In addition, the node can be configured to behave as though having a physical boundary. This boundary can be defined as a circle, a rectangle, or a polygon of any shape.

Once a node has a boundary, collisions between other nodes can be detected, and the physics engine is used to apply real-world physics to the node, such as causing it to bounce when hitting other nodes. The use of contact bit masks can be employed to specify the types of nodes for which contact notification is required.

The physics body also allows forces to be applied to nodes, such as propelling a node in a particular direction across a scene using either a constant or one-time impulse force. Physical bodies can also be combined using various join types (sliding, fixed, hinged, and spring-based attachments).

The properties of a physics body (and, therefore, the associated node) may also be changed. Mass, density, velocity, and friction are just a few of the properties of a physics body available for modification by the game

developer.

### 92.2.5 Physics World

Each scene in a game has its own *physics world* object in the form of an instance of the `SKPhysicsWorld` class. A reference to this object, which is created automatically when the scene is initialized, may be obtained by accessing the *physicsWorld* property of the scene. The physics world object is responsible for managing and imposing the rules of physics on any nodes in the scene with which a physics body has been associated. Properties are available on the physics world instance to change the default gravity settings for the scene and also to adjust the speed at which the physics simulation runs.

### 92.2.6 Actions

An action is an activity performed by a node in a scene. Actions are the responsibility of `SKAction` class instances which are created and configured with the action to be performed. That action is then run on one or more nodes. An action might, for example, be configured to perform a rotation of 90 degrees. That action would then be run on a node to make it rotate within the scene. The `SKAction` class includes various action types, including fade in, fade out, rotation, movement, and scaling. Perhaps the most interesting action involves animating a sprite node through a series of texture frames.

Actions can be categorized as *sequence*, *group*, or *repeating* actions. An action sequence specifies a series of actions to be performed consecutively, while group actions specify a set of actions to be performed in parallel. Repeating actions are configured to restart after completion. An action may be configured to repeat several times or indefinitely.

### 92.2.7 Transitions

Transitions occur when a game changes from one scene to another. While it is possible to switch immediately from one scene to another, a more visually pleasing result might be achieved by animating the transition in some way. This can be implemented using the `SKTransition` class, which provides several different pre-defined transition animations, such as sliding the new scene down over the top of the old scene or presenting the effect of doors opening to reveal the new scene.

### 92.2.8 Texture Atlas

A large part of developing games involves handling images. Many of these images serve as textures for sprites. Although adding images to a project individually is possible, Sprite Kit also allows images to be grouped into a texture atlas. Not only does this make it easier to manage the images, but it also brings efficiencies in terms of image storage and handling. For example, the texture images for a particular sprite animation sequence would typically be stored in a single texture atlas. In contrast, another atlas might store the images for the background of a particular scene.

### 92.2.9 Constraints

Constraints allow restrictions to be imposed on nodes within a scene in terms of distance and orientation in relation to a point or another node. A constraint can, for example, be applied to a node such that its movement is restricted to within a certain distance of another node. Similarly, a node can be configured so that it is oriented to point toward either another node or a specified point within the scene. Constraints are represented by instances of the `SKConstraint` class and are grouped into an array and assigned to the *constraints* property of the node to which they are to be applied.

## 92.3 An Example Sprite Kit Game Hierarchy

To aid in visualizing how the various Sprite Kit components fit together, Figure 92-1 outlines the hierarchy for a simple game:

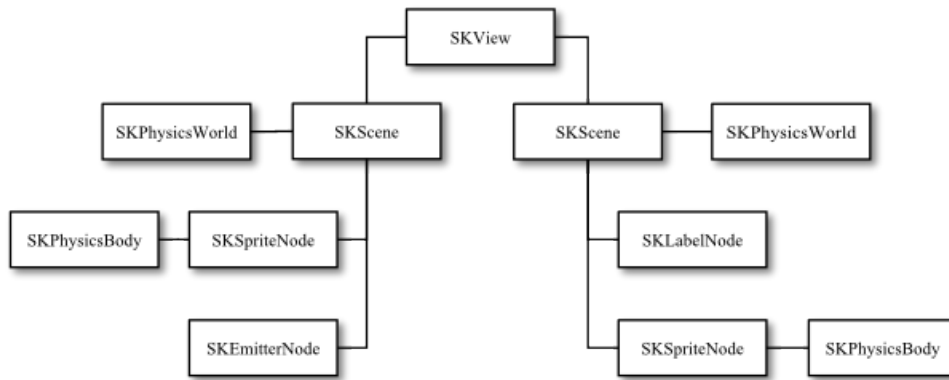


Figure 92-1

In this hypothetical game, a single SKView instance has two SKScene children, each with its own SKPhysicsWorld object. Each scene, in turn, has two node children. In the case of both scenes, the SKSpriteNode instances have been assigned SKPhysicsBody instances.

## 92.4 The Sprite Kit Game Rendering Loop

When working with Sprite Kit, it helps to understand how the animation and physics simulation process works. This process can best be described by looking at the Sprite Kit frame rendering loop.

Sprite Kit performs the work of rendering a game using a *game rendering loop*. Within this loop, Sprite Kit performs various tasks to render the visual and behavioral elements of the currently active scene, with an iteration of the loop performed for each successive frame displayed to the user.

Figure 92-2 provides a visual representation of the frame rendering sequence performed in the loop:

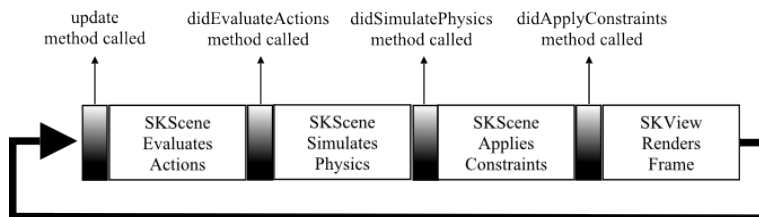


Figure 92-2

When a scene is displayed within a game, Sprite Kit enters the rendering loop and repeatedly performs the same sequence of steps as shown above. At several points in this sequence, the loop will make calls to your game, allowing the game logic to respond when necessary.

Before performing any other tasks, the loop begins by calling the *update* method of the corresponding SKScene instance. Within this method, the game should perform any tasks before the frame is updated, such as adding additional sprites or updating the current score.

The loop then evaluates and implements any pending actions on the scene, after which the game can perform more tasks via a call to the *didEvaluateActions* method.

Next, physics simulations are performed on the scene, followed by a call to the scene's *didSimulatePhysics* method, where the game logic may react where necessary to any changes resulting from the physics simulation.

The scene then applies any constraints configured on the nodes in the scene. Once this task has been completed,

a call is made to the scene's *didApplyConstraints* method if it has been implemented.

Finally, the SKView instance renders the new scene frame before the loop sequence repeats.

## 92.5 The Sprite Kit Level Editor

Integrated into Xcode, the Sprite Kit Level Editor allows scenes to be designed by dragging and dropping nodes onto a scene canvas and setting properties on those nodes using the SKNode Inspector. Though code writing is still required for anything but the most basic scene requirements, the Level Editor provides a useful alternative to writing code for some of the less complex aspects of SpriteKit game development. The editor environment also includes both live and action editors, allowing for designing and testing animation and action sequences within a Sprite Kit game.

## 92.6 Summary

Sprite Kit provides a platform for creating 2D games on iOS, tvOS, watchOS, and macOS. Games comprise an SKView instance with an SKScene object for each game scene. Scenes contain nodes representing the game's characters, objects, and items. Various node types are available, all of which are subclassed from the SKNode class. In addition, each node can have associated with it a physics body in the form of an SKPhysicsBody instance. A node with a physics body will be subject to physical forces such as gravity, and when given a physical boundary, collisions with other nodes may also be detected. Finally, actions are configured using the SKAction class, instances of which are then run by the nodes on which the action is to be performed.

The orientation and movement of a node can be restricted by implementing constraints using the SKConstraint class.

The rendering of a Sprite Kit game takes place within the *game loop*, with one loop performed for each game frame. At various points in this loop, the app can perform tasks to implement and manage the underlying game logic.

Having provided a high-level overview in this chapter, the next three chapters will take a more practical approach to exploring the capabilities of Sprite Kit by creating a simple game.



## 95. An iOS 17 Sprite Kit Particle Emitter Tutorial

In this, the last chapter dedicated to the Sprite Kit framework, the use of the Particle Emitter class and editor to add special effects to Sprite Kit-based games will be covered. Having provided an overview of the various elements that make up particle emitter special effects, the `SpriteKitDemo` app will be extended using particle emitter features to make the balls burst when an arrow hits. This will also involve the addition of an audio action.

### 95.1 What is the Particle Emitter?

The Sprite Kit particle emitter is designed to add special effects to games. It comprises the `SKEmitterNode` class and the Particle Emitter Editor bundled with Xcode. A particle emitter special effect begins with an image file representing the particle. The emitter generates multiple instances of the particle on the scene and animates each particle subject to a set of properties. These properties control aspects of the special effect, such as the rate of particle generation, the angle, and speed of motion of particles, whether or not particles rotate, and how the particles blend in with the background.

With some time and experimentation, a wide range of special effects, from smoke to explosions, can be created using particle emitters.

### 95.2 The Particle Emitter Editor

The Particle Emitter Editor is built into Xcode and provides a visual environment to design particle emitter effects. In addition to providing a platform for developing custom effects, the editor also offers a collection of pre-built particle-based effects, including rain, fire, magic, snow, and sparks. These template effects also provide an excellent starting point on which to base other special effects.

Within the editor environment, a canvas displays the current particle emitter configuration. A settings panel allows the various properties of the emitter node to be changed, with each modification reflected in the canvas in real time, thereby making creating and refining special effects much easier. Once the design of the special effect is complete, the effect is saved in a Sprite Kit particle file. This file actually contains an archived `SKEmitterNode` object configured to run the particle effects designed in the editor.

### 95.3 The `SKEmitterNode` Class

The `SKEmitterNode` displays and runs the particle emitter effect within a Sprite Kit game. As with other Sprite Node classes, the `SKEmitterNode` class has many properties and behaviors of other classes in the Sprite Kit family. Generally, an `SKEmitterNode` class is created and initialized with a Sprite Kit particle file created using the Particle Emitter editor. The following code fragment, for example, initializes an `SKEmitterNode` instance with a particle file, configures it to appear at a specific position within the current scene, and adds it to the scene so that it appears within the game:

```
if let burstNode = SKEmitterNode(fileName: "BurstParticle.sks") {
    burstNode.position = CGPoint(x: target_x, y: target_y)
    secondNode.removeFromParent()
    self.addChild(burstNode)
}
```

## An iOS 17 Sprite Kit Particle Emitter Tutorial

Once created, all of the emitter properties available within the Particle Emitter Editor are also controllable from within the code, allowing the effect's behavior to be changed in real time. The following code, for example, adjusts the number of particles the emitter is to emit before ending:

```
burstNode.numParticlesToEmit = 400
```

In addition, actions may be assigned to particles from within the app code to add additional behavior to a special effect. The particles can, for example, be made to display an animation sequence.

### 95.4 Using the Particle Emitter Editor

By far, the easiest and most productive approach to designing particle emitter-based special effects is to use the Particle Emitter Editor tool bundled with Xcode. To experience the editor in action, launch Xcode and create a new iOS Game-based project named *ParticleDemo* with the Language menu set to *Swift*.

Once the new project has been created, select the *File -> New -> File...* menu option. Then, in the resulting panel, choose the *SpriteKit Particle File* template option as outlined in Figure 95-1:

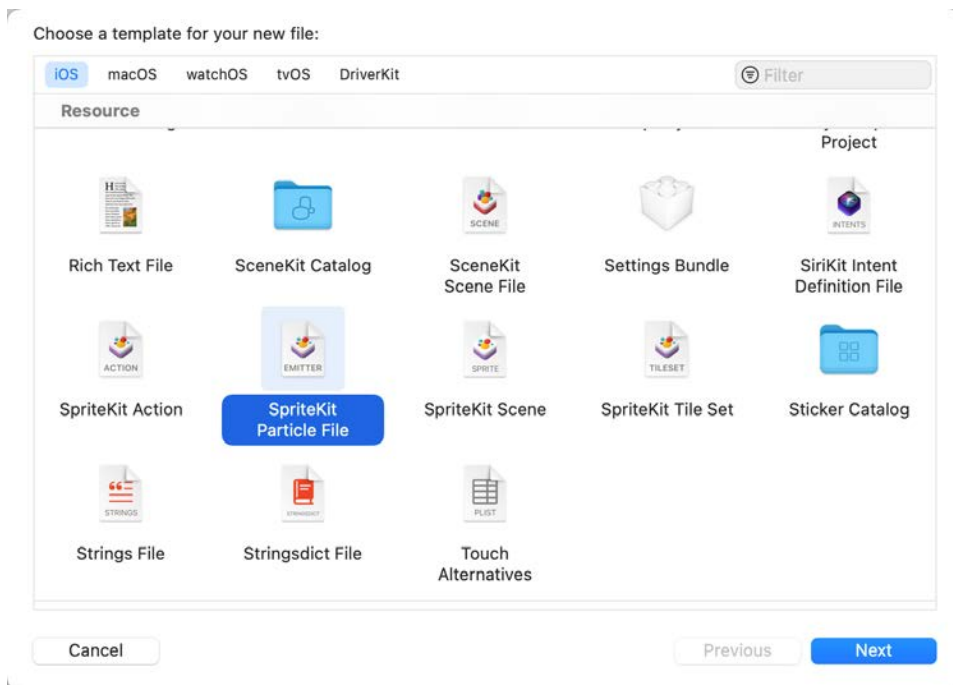


Figure 95-1

Click *Next* and choose a Particle template on which to base the special effect. For this example, we will use the *Fire* template. Click *Next* and name the file *RocketFlame* before clicking on *Create*.

At this point, Xcode will have added two files to the project. One is an image file named *spark.png* representing the particle, and the other is the *RocketFlame.sks* file containing the particle emitter configuration. In addition, Xcode should also have pre-loaded the Particle Emitter Editor panel with the fire effect playing in the canvas, as shown in Figure 95-2 (the editor can be accessed at any time by selecting the corresponding sks file in the project navigator panel).



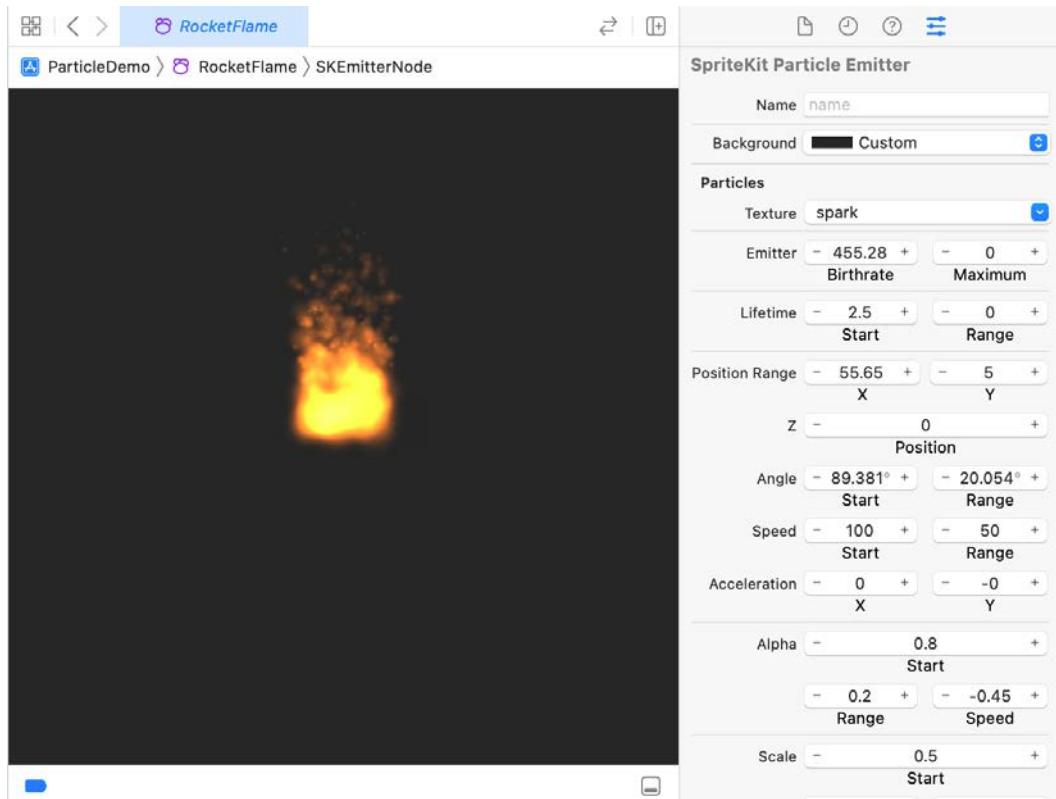


Figure 95-2

The right-hand panel of the editor provides access to and control of all of the properties associated with the emitter node. To access these property settings, click the right-hand toolbar button in the right-hand panel.

Much about particle emitter special effects can be learned through experimentation with the particle editor. However, before modifying the fire effects in this example, it will be helpful to provide an overview of what these properties do.

## 95.5 Particle Emitter Node Properties

A range of property settings controls the behavior of a particle emitter and associated particles. These properties can be summarized as follows:

### 95.5.1 Background

Though presented as an option within the editor, this is not actually a property of the emitter node. This option is provided so that the appearance of the effect can be tested against different backgrounds. This is particularly important when the particles are configured to blend with the background. Use this setting to test the particle effects against any background colors the effect is likely to appear with within the game.

### 95.5.2 Particle Texture

The image file containing the texture that will be used to represent the particles within the emitter.

### 95.5.3 Particle Birthrate

The birthrate defines the rate at which the node emits new particles. The greater the value, the faster new particles are generated. However, it is recommended that the minimum number of particles needed to achieve the desired effect be used to avoid performance degradation. Therefore, the total number of particles emitted may also be specified. A value of zero causes particles to be emitted indefinitely. If a limit is specified, the node will stop emitting particles when that value is reached.

### 95.5.4 Particle Life Cycle

The lifetime property controls the time in seconds a particle lives (and is therefore visible) before disappearing from view. The range property may be used to introduce variance in the lifetime from one particle to the next based on a random time value between 0 and the specified range value.

### 95.5.5 Particle Position Range

The position properties define the location from which particles are created. For example, the X and Y values can be used to declare an area around the center of the node location from which particles will be created randomly.

### 95.5.6 Angle

The angle at which a newly emitted particle will travel away from the creation point in counter-clockwise degrees, where a value of 0 degrees equates to rightward movement. Random variance in direction can be introduced via the range property.

### 95.5.7 Particle Speed

The speed property specifies the particles' initial speed at the creation time. The speed can be randomized by specifying a range value.

### 95.5.8 Particle Acceleration

The acceleration properties control the degree to which a particle accelerates or decelerates after emission in terms of both X and Y directions.

### 95.5.9 Particle Scale

The size of the particles can be configured to change using the scale properties. These settings cause the particles to grow or shrink throughout their lifetimes. Random resizing behavior can be implemented by specifying a range value. The speed setting controls the speed with which the size changes take place.

### 95.5.10 Particle Rotation

The rotation properties control the speed and amount of rotation applied to the particles after creation. Values are specified in degrees, with positive and negative values correlating to clockwise and counter-clockwise rotation. In addition, the speed of rotation may be specified in degrees per second.

### 95.5.11 Particle Color

The particles created by an emitter can be configured to transition through a range of colors during a lifetime. To add a new color in the lifecycle timeline, click on the color ramp at the location where the color is to change and select a new color. Change an existing color by double-clicking the marker to display the color selection dialog. Figure 95-3, for example, shows a color ramp with three color transitions specified:



Figure 95-3

To remove a color from the color ramp, click and drag it downward out of the editor panel.

The color blend settings control the amount by which the colors in the particle's texture blend with the prevailing color in the color ramp at any given time during the particle's life. The greater the Factor property, the greater the colors blend, with 0 indicating no blending. By adjusting the speed property, the blending factor can be randomized by specifying a range and the speed at which the blend is performed.

### 95.5.12 Particle Blend Mode

The Blend Mode property governs how particles blend with other images, colors, and graphics in Sprite Kit game scenes. Options available are as follows:

- **Alpha** – Blends transparent pixels in the particle with the background.
- **Add** – Adds the particle pixels to the corresponding background image pixels.
- **Subtract** – Subtracts the particle pixels from the corresponding background image pixels.
- **Multiply** – Multiplies the particle pixels by the corresponding background image pixels—resulting in a darker particle effect.
- **MultiplyX2** – This creates a darker particle effect than the standard Multiply mode.
- **Screen** – Inverts pixels, multiplies, and inverts a second time—resulting in lighter particle effects.
- **Replace** – No blending with the background. Only the particle's colors are used.

## 95.6 Experimenting with the Particle Emitter Editor

Creating compelling special effects with the particle emitter is largely a case of experimentation. As an example of adapting a template effect for another purpose, we will now modify the fire effect in the *RocketFlame.sks* file so that instead of resembling a campfire, it could be attached to the back of a sprite to represent the flame of a rocket launching into space.

Within Xcode, select the previously created *RocketFlame.sks* file so that it loads into the Particle Emitter Editor. The animation should appear and resemble a campfire, as illustrated in Figure 95-2.

1. The first step in modifying the effect is to change the angle of the flames so that they burn downwards. To achieve this, change the *Start* property of the *Angle* setting to 270 degrees. The fire should now be inverted.
2. Change the X value of the *Position Range* property to 5 so that the flames become narrower and more intense.
3. Increase the *Start* value of the *Speed* property to 450.
4. Change the *Lifetime* start property to 7.

The effect now resembles the flames a user might expect to see shooting out of the back of a rocket against a nighttime sky (Figure 95-4). Note also that the effects of the motion of the emitter node may be simulated by clicking and dragging the node around the canvas.

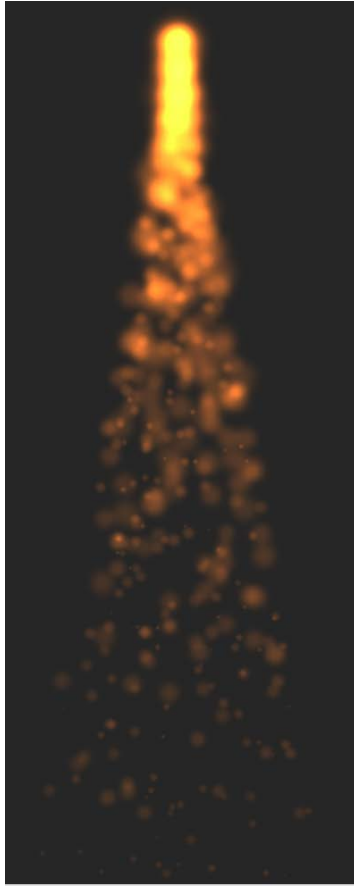


Figure 95-4

## 95.7 Bursting a Ball using Particle Emitter Effects

The final task is to update the `SpriteKitDemo` game so that the balls burst when they are hit by an arrow shot by the archer sprite.

The particles for the bursting ball will be represented by the *BallFragment.png* file located in the sample code download archive in the *sprite\_images* folder. Open the `SpriteKitDemo` project within Xcode, locate the *BallFragment.png* file in a Finder window, and drag and drop it onto the list of image sets in the *Assets* file.

Select the *File -> New -> File...* menu option and, in the resulting panel, select the *SpriteKit Particle File* template option. Click *Next*, and on the template screen, select the *Spark* template. Click *Next*, name the file *BurstParticle*, and click *Create*.

The Particle Emitter Editor will appear with the spark effect running. Since the scene on which the effect will run has a white background, click on the black swatch next to *Background* in the Attributes Inspector panel and change the color to white.

Click on the *Particles Texture* drop-down menu, select the *BallFragment* image, and change the *Blend Mode* menu to *Alpha*.

Many ball fragments should now be visible, blended with the yellow color specified in the ramp. Set the Emitter *Birthrate* property to 15 to reduce the number of particles emitted. Click on the yellow marker at the start of

the color ramp and change the color to *White* in the resulting color dialog. The particles should now look like fragments of the ball used in the game.

The fragments of a bursting ball would be expected to originate from any part of the ball. As such, the Position Range X and Y values need to match the dimensions of the ball. Set both of these values to 86 accordingly.

Finally, limit the number of particles by changing the Emitter *Maximum* property in the Particles section to 8.

The burst particle effect is now ready to be incorporated into the game logic.

## 95.8 Adding the Burst Particle Emitter Effect

When an arrow scores a hit on a ball node, the ball node will be removed from the scene and replaced with a *BurstParticle* SKEmitterNode instance. To implement this behavior, edit the *ArcheryScene.swift* file and modify the *didBegin(contact:)* method to add a new method call to extract the SKEmitterNode from the archive in the *BurstParticle* file, remove the ball node from the scene and replace it at the same position with the emitter:

```
func didBegin(_ contact: SKPhysicsContact) {
    let secondNode = contact.bodyB.node as! SKSpriteNode

    if (contact.bodyA.categoryBitMask == arrowCategory) &&
        (contact.bodyB.categoryBitMask == ballCategory) {

        let contactPoint = contact.contactPoint
        let contact_y = contactPoint.y
        let target_x = secondNode.position.x
        let target_y = secondNode.position.y
        let margin = secondNode.frame.size.height/2 - 25

        if (contact_y > (target_y - margin)) &&
            (contact_y < (target_y + margin)) {

            if let burstNode = SKEmitterNode(fileName: "BurstParticle.sks")
            {
                burstNode.position = CGPoint(x: target_x, y: target_y)
                secondNode.removeFromParent()
                self.addChild(burstNode)
            }

            score += 1
        }
    }
}
```

Compile and run the app. For example, when an arrow hits a ball, it should now be replaced by the particle emitter effect:

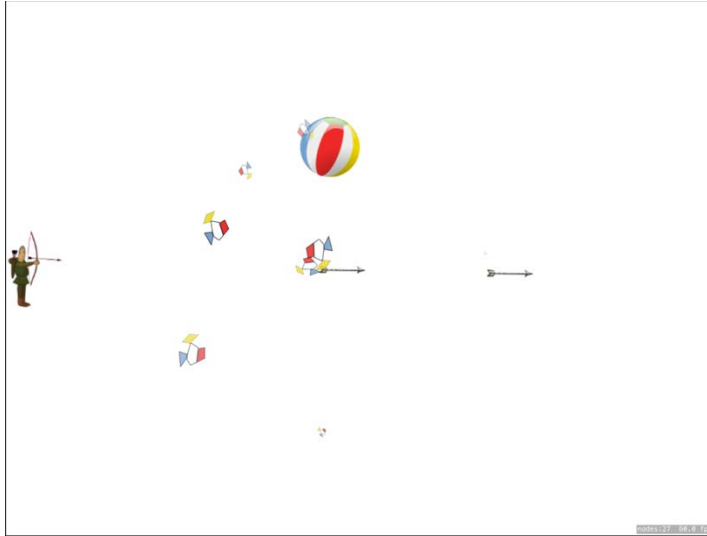


Figure 95-5

## 95.9 Adding an Audio Action

The final effect to add to the game is a bursting sound when an arrow hits the ball. We will again use the Xcode Action Editor to add this effect.

Begin by adding the sound file to the project. This file is named *burstsound.mp3* and is located in the *audiofiles* folder of the book code samples download. Locate this file in a Finder window and drag it onto the Project Navigator panel. In the resulting panel, enable the *Copy items if needed* option and click on *Finish*.

Within the Project Navigator panel, select the *ArcherScene.sks* file. Then, from the Library panel, locate the *Play-Sound-File-Named-Action* object and drag and drop it onto the timeline so that it is added to the *archerNode* object:

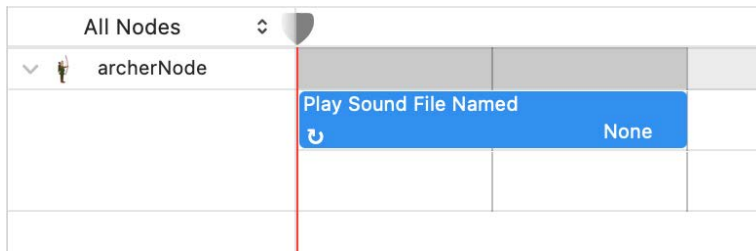


Figure 95-6

Select the new action object in the timeline and use the Attributes Inspector panel to set the *Filename* property to the *burstsound* file.

Right-click on the sound action and select the *Convert to Reference* menu option. Name the reference *audioAction* and click on the Create button. The action has now been saved to the *ArcherActions.sks* file. Next, select the object in the timeline, right-click, and select the *Delete* option to remove it from the scene file.

Finally, modify the *didBegin(contact:)* method to play the sound action when a ball bursts:

```
func didBegin(_ contact: SKPhysicsContact) {
    let secondNode = contact.bodyB.node as! SKSpriteNode
```

```

if (contact.bodyA.categoryBitMask == arrowCategory) &&
    (contact.bodyB.categoryBitMask == ballCategory) {

    let contactPoint = contact.contactPoint
    let contact_y = contactPoint.y
    let target_x = secondNode.position.x
    let target_y = secondNode.position.y
    let margin = secondNode.frame.size.height/2 - 25

    if (contact_y > (target_y - margin)) &&
        (contact_y < (target_y + margin)) {
        print("Hit")

        if let burstNode = SKEmitterNode(fileName: "BurstParticle.sks")
        {
            burstNode.position = CGPoint(x: target_x, y: target_y)
            secondNode.removeFromParent()
            self.addChild(burstNode)
            if let audioAction = SKAction(named: "audioAction") {
                burstNode.run(audioAction)
            }
        }
        score += 1
    }
}
}

```

Run the app and verify that the sound file plays when a hit is registered on a ball.

## 95.10 Summary

The particle emitter allows special effects to be added to Sprite Kit games. All that is required is an image file to represent the particles and some configuration of the particle emitter properties. This work can be simplified using the Particle Emitter Editor included with Xcode. The editor is supplied with a set of pre-configured special effects, such as smoke, fire, and rain, which can be used as supplied or modified to meet many special effects needs.





## Index

### Symbols

& 57  
 ^ 58  
 ^= 59  
 << 59  
 <=< 59  
 &= 59  
 >> 59  
 >= 59  
 | 58  
 |= 59  
 ~ 57  
 \$0 79  
 @IBDesignable 443  
 @IBInspectable 444  
 ?? operator 56

### A

Action Extension 568  
   add target 584  
   overview 583  
   receiving data from 593  
   tutorial 583  
 Adaptive User Interface  
   tutorial 174  
 addArc method 447, 451  
 addConstraint method 154  
 addCurve(to:) method 451  
 addEllipse(in:) method 449  
 addQuadCurve method 452  
 addRect method 448  
 addTask() function 272  
 Affine Transformations 464  
 Alert Views 134  
 Alignment Rects 136

alpha property 463  
 AND (&&) operator 55  
 AND operator 57  
 Animation 463  
   example 465  
 Animation Blocks 463  
 Animation Curves 464  
 AnyObject 106  
 App Icons 711  
 Apple Developer Program 3  
 applicationDidEnterBackground delegate method 254  
 Application Performance 24  
 applicationWillResignActive method 254  
 App project template 181  
 App Store  
   creating archive 712  
   submission 709  
 App Store Connect 713  
 Arranged Subviews 232  
 arrangedSubviews property 232  
 Array  
   forEach() 105  
   mixed type 106  
 Array Initialization 103  
 Array Item Count 104  
 Array Items  
   accessing 104  
   appending 105  
   inserting and deleting 105  
 Array Iteration 105  
 Arrays  
   immutable 103  
   mutable 103  
 as! keyword 51  
 Aspect Ratio Constraints 149  
 Assistant Editor 124, 125  
 async  
   suspend points 266  
 async/await 265

## Index

- Asynchronous Properties 275
- async keyword 266
- async-let bindings 268
- AsyncSequence protocol 274
- attributesOfItemAtPath method 281
- Audio 645
  - Audio Formats 645
  - Audio Session
    - category 563
  - Audio Unit Extension 569
- Augmented Reality App 12
- Auto Layout
  - addConstraint 154
  - Add New Constraints menu 141
  - Alignment Rects 136
  - Align menu 145
  - Auto Resizing Translation 155
  - Compression Resistance 136
  - constraintsWithVisualFormat 166
  - Content Hugging 136
  - Creating Constraints in Code 153
  - Cross Hierarchy Constraints 161
  - cross-view hierarchy constraints 135
  - Editing Constraints 147
  - Interface Builder example 139
  - Intrinsic Content Size 136
  - introduction 135
  - Removing Constraints 159
  - Suggested Constraints 142
  - Visual Format Language 137, 165
- Auto Layout Problems
  - resolving 149
- Auto Resizing Translation 155
- autosizing 135
- AVAudioPlayer 651
- AVAudioPlayerDelegate protocol
  - methods 645
- AVAudioRecorder 651
- AVAudioSession.Category.playback 563
- AVPlayerViewController 557, 559
- await keyword 266, 267

## B

- background colors
  - changing scene 196
- binary operators 53
- Biometric Authentication 425
- bit operators 57
- Bitwise AND 57
- Bitwise Left Shift 58
- bitwise OR 58
- bitwise right shift 59
- bitwise XOR 58
- Boolean Logical Operators 55
- bottomAnchor 156
- Bounds Rectangles 122
- break statement 63
- Build Errors 24
- Build Phases 15
- Build Rules 15
- Build Settings 15
- Bundle display name key 585

## C

- calculateETA(completionHandler:) method 514
- Camera
  - tutorial 551
- Camera and Photo Library 547
- cancelAll() function 273
- canHandle(adjustmentData:) method 575
- case Statement 68
- catch statement 113
  - multiple matches 113
- cellForRowAt indexPath method 203
- Cell Reuse Identifier 209
- centerXAnchor 155
- centerYAnchor 156
- CFBundleTypeName 323
- CFBundleTypeRole 323, 328
- CGAffineTransformMakeRotation() function 464
- CGColor property 436
- CGColorSpaceCreateDeviceCMYK() function 436
- CGColorSpaceCreateDeviceGray function 436
- CGContextRef 436

- CGCreateSetStrokeColorWithColor function 436
- CGGradient class 454
- CGImageRef 461
- CGPoint 435
- CGRect 435
- CGRect structure 435
- CGSize 435
- Character data type 42
- checkCancellation() method 271
- childNodes(withName:) method 683
- CIColor 460
- CIFilter 460
- CUIImage 460, 461
- CKAcceptSharesOperation 379, 380
- CKConfiguration 375
- CKContainer class 367
- CKDatabase 367, 368, 371, 396
- CKFetchRecordsOperation 380
- CKModifyRecordsOperation 369, 375
- CKRecord 368, 370, 371, 391, 396
- CKRecordID 370
- CKRecordTypeUserRecord 372
- CKRecordZone 371
- CKReference 370
- CKShare class 375
- CKShare.Metadata 380
- CKShareParticipant , 376
- CKSharingSupported key , 375
- Class Extensions 94
- CLGeocoder 487
- CLLocation 487, 499
- CLLocationManager 499
- CLLocationManagerDelegate protocol 501, 506
- CLLocationManager Object 506
- closed range operator 55
- Closure Expressions 78
  - shorthand argument names 79
- closures 71
- Closures 79
- CloudKit 367
  - add container 307, 384
  - Assets 370
  - Console 392
  - deleting a record 396
  - example 383
  - overview 367
  - Private Database 387
  - Private Databases 367
  - Public Database 367
  - quotas 368
  - Record IDs 370
  - Records 368
  - Record Zones 371
  - Saving a Record 390
  - searching 394
  - Subscriptions 371
  - tutorial 383
  - Updating records 395
- CloudKit Console 392
- CloudKit Containers 367
- CloudKit Data Storage 293
- CloudKit Share
  - accepting 379, 399
  - creating 398
- CloudKit Sharing 371, 375
  - creating a share object 375
  - example 397
  - fetching records 380
  - fetching shared records 400
- Info.plist 375
- overview 375
- preparing project for 397
- CLPlacemark 488
- coalesced touches 405
- Coalesced Touches
  - accessing 411
- coalescedTouchesForTouch method 405
- Cocoa Touch 117
- Color
  - working with 436
- colorspace 436
- compact size class 169
- company identifier 13
- Comparison Operators 54

## Index

Completion Handlers 263  
Component Properties 18  
Compound Bitwise Operators 59  
Compression Resistance 136  
computed properties 85  
concrete type 89  
Conditional Control Flow 64  
constants 45  
constraint() method 156  
Constraints 135  
    editing 147  
    Outlets 162  
    Removing 159  
constraintsWithVisualFormat 166  
constraintsWithVisualFormat method 167  
constraintWithItem method 153  
Container Views 134  
Content Blocking Extension 569  
Content Hugging 136  
continue Statement 63  
Controls 134  
Coordinates 435  
Core Animation 463  
Core Data 355  
    Entity Description 357  
    Fetched Property 356  
    Fetch request 356  
    Managed Object Context 356  
    Managed Object Model 356  
    Managed Objects 356  
    Persistent Object Store 357  
    Persistent Store Coordinator 356  
    relationships 356  
    stack 355  
    tutorial 361  
Core Graphics 435  
Core Image Framework 460  
Core Location  
    basics of 499  
CoreML  
    classification request 630  
    example 627

CoreML framework 620  
CouldKit  
    References 370  
CPU cores 263  
Create ML 619  
    building a model 621  
CreateMLUI 619  
cross-view hierarchy constraints 135  
Current Location  
    getting 503  
Current Working Directory 278  
Custom Keyboard Extension 569

## D

data encapsulation 82  
Data Races 273  
defaultContainer method 367  
Default Function Parameters 73  
defer statement 114  
Delegation 119  
dequeueReusableCell(withIdentifier:) method 203, 213  
design patterns 117  
Detached Tasks 271  
Developer Mode setting 23  
Developer Program 3  
Dictionary Collections 106  
Dictionary Entries  
    adding and removing 108  
Dictionary Initialization 106  
Dictionary Item Count 108  
Dictionary Items  
    accessing and updating 108  
Dictionary Iteration 108  
didBegin(contact:) method 695  
didChangeAuthorizationStatus delegate method 502  
didEnd(contact:) method 695  
Did End on Exit event 129  
didFinishLaunchingWithOptions 119  
didFinishPickingMediaWithInfo method 548  
didMove(to view:) method 678, 695  
didUpdateLocations delegate method 502  
Directories

- working with filesystem 277
- Directory
  - attributes of 281
  - changing 279
  - contents of 280
  - creating 279
  - deleting 280
- dispatch\_async 588
- display
  - dimension of 459
- Display Views 134
- do-catch statement 113
  - multiple matches 113
- Document App 12
- Document Based App 322
- Document Browser View Controller 321
  - adding actions 325
  - declaring file types 327
  - delegate methods 323
  - tutorial 327
- Document Provider Extension 569
- Documents Directory 277
  - locating 278
- Double 42
- downcasting 50
- Drawing
  - arc 451
  - Cubic Bézier Curve 451
  - dashed line 453
  - ellipse 449
  - filling with color 449
  - gradients 454
  - images 458
  - line 446
  - paths 447
  - Quadratic Bézier Curve 452
  - rectangle 448
  - shadows 454
- drawLinearGradient method 455
- drawRadialGradient method 457
- draw(rect:) method 435, 446
- Dynamic Animator 472

- Dynamic Type 201

## E

- Embedded Frameworks 439
  - creating 441
- enum 100, 111
  - associated values 101
- Enumeration 100
- Errata 1
- Error
  - throwing 112
- Error Catching
  - disabling 114
- Error Object
  - accessing 114
- ErrorType protocol 111
- Event forwarding 403
- exclusive OR 58
- Expression Syntax 53
- Extensions 567
  - creating 570
  - overview 567
- Extensions and Adjustment Data 575
- Extension Types 567
- external parameter names 73

## F

- Face ID
  - checking availability 425
  - example 427
  - policy evaluation 426
  - privacy statement 431
  - seeking authentication 429
- Face ID Authentication
  - Authentication 425
- fallthrough statement 70
- File
  - access permissions 284
  - comparing 283
  - copying 284
  - deleting 284
  - existence of 283

## Index

- offsets and seeking 286
- reading and writing 285
- reading data 286
- renaming and moving 284
- symbolic link 285
- truncation 287
- writing data 287
- File Inspector 18
- fillEllipse(in:) method 449
- fillPath method 449
- fill(rect:) method 449
- finishContentEditing(completionHandler:) method 579
- firstBaselineAnchor 156
- first responder 129
- Float 42
- flow control 61
- FMDatabase 343
- FMDatabaseQueue 343
- FMDB Classes 343
- FMDB Source Code 347
- FMResultSet 343
- font setting 20
- for-await 274
- forced unwrapping 47
- forEach() 105
- for loop 61
- forward-geocoding 488
- Forward Geocoding 494
- function
  - arguments 71
  - parameters 71
- Function Parameters
  - variable number of 74
- functions 71
  - as parameters 76
  - default function parameters 73
  - external parameter names 73
  - In-Out Parameters 75
  - parameters as variables 75
  - return multiple results 74

## G

- Game project template 12
- geocodeAddressString method 488
- Geocoding 487, 494
- Gesture
  - identification 417
- Gesture Recognition 421
- Gestures 404
  - continuous 418
  - discreet 418
- Graphics Context 436
- guard statement 65

## H

- half-closed range operator 56
- heightAnchor 156
- Horizontal Stack View 231

## I

- IBAction 119
- IBOutlet 119
- iCloud
  - application preparation 293
  - conflict resolution 297
  - document storage 293, 305
  - enabling on device 313
  - enabling support 294
  - entitlements 295
  - guidelines 305
  - key-value change notifications 336
  - key-value conflict resolution 336
  - key-value data storage 335
  - key-value storage 293
  - key-value storage restrictions 336
  - searching 308
  - storage services 293
  - UBUIQUITY\_CONTAINER\_URL 296
- iCloud Drive
  - enabling 315
  - overview 315
- iCloud User Information
  - obtaining 372
- if ... else if ... Statements 65

- if ... else ... Statements 64
- if-let 48
- if Statement 64
- Image Filtering 460
- imagePickerControllerDidCancel delegate 554
- iMessage App 12
- iMessage Extension 569
- implicitly unwrapped 50
- Inheritance, Classes and Subclasses 91
- Initial View Controller 190
- init method 83
- in keyword 78
- inout keyword 76
- In-Out Parameters 75
- Instance Properties 82
- Intents Extension 569
- Interface Builder 15
  - Live Views 439
- Intrinsic Content Size 136
- iOS 12
  - architecture 117
- iOS Distribution Certificate 709
- iOS SDK
  - installation 7
  - system requirements 7
- iPad Pro
  - multitasking 252
  - Split View 252
- isActive property 159
- isCancelled property 271
- isEmpty property 273
- is keyword 52
- isSourceTypeAvailable method 549

## K

- keyboard
  - change return key 525
- Keyboard Type property 122
- Key Messages Framework 599
- kUTTypeImage 547
- kUTTypeMovie 547

## L

- LAError.biometryNotAvailable 426
- LAError.biometryNotEnrolled 426
- LAError.passcodeNotSet 426
- lastBaselineAnchor 156
- Layout Anchors
  - constraint() method 156
  - isActive property 159
- Layout Hierarchy 25
- lazy
  - keyword 87
- Lazy properties 86
- leadingAnchor 155
- leftAnchor 155
- Left Shift Operator 58
- Library panel
  - displaying 16
- libsqlite3.tbd 347
- Live Views 439
- loadItem(forTypeIdentifier:)
  - 587
- Local Authentication Framework 425
- Local Notifications 633
- local parameter names 73
- Local Search
  - overview 523
- Location Access Authorization 499
- Location Accuracy 500
- Location Information 499
  - permission request 518
- Location Manager Delegate 501
- Long Touch Gestures 419
- Loops
  - breaking from 63
- LSHandlerRank 323, 328
- LSItemContentTypes 323

## M

- Machine learning
  - datasets 619
  - models 619
- Machine Learning

## Index

- example 627
  - iOS Frameworks 620
  - overview 619
  - Main.storyboard file 126
  - Main Thread 263
  - MapKit
    - Local Search 523
    - Transit ETA Information 514
  - MapKit Framework 513
  - Map Regions 513
  - Map Type
    - changing 519
  - mapView(didUpdate userLocation:) method 520
  - MapView Region
    - changing 519
  - mathematical expressions 53
  - mediaTypes property 547
  - Message App
    - preparing message URL 613
    - tutorial 605
    - types of 598
  - Message Apps 597
    - introduction 597
  - Message App Store 597
  - metadataQueryDidFinishGathering method 310
  - Methods
    - declaring 82
  - Mixed Type Arrays 106
  - MKDirections class 535
  - MKDirections.Request class 535
  - MKLocalSearch class 523
  - MKLocalSearchRequest 524
  - MKLocalSearchRequest class 523
  - MKLocalSearchResponse class 523
  - MKMapItem 487
    - example app 493
    - options 490
    - turn-by-turn directions 490
  - MKMapItem forCurrentLocation method 490
  - MKMapType.Hybrid 519
  - MKMapType.HybridFlyover 519
  - MKMapType.Satellite 519
  - MKMapType.SatelliteFlyover 519
  - MKMapType.Standard 519
  - MKMapView 513
    - tutorial 514
  - MKPlacemark 487
    - creating 489
  - MKPolylineRenderer class 536
  - MKRouteStep class 535
  - MKUserLocation 521
  - Model View Controller (MVC) 117
  - modifyRecordsResultBlock 376
  - MSConversation class 599
  - MSMessage
    - creating a message 602
  - MSMessage class 600, 601
  - MSMessagesAppViewController 599, 602
  - MSMessageTemplateLayout class 600
  - Multiple Storyboard Files 189
  - Multitasking 249
    - disabling 254
    - example 257
    - handling in code 252
    - Lifecycle Methods 254
    - Picture-in-Picture 251, 561
  - Multitouch
    - enabling 408
  - multiview application 193
  - MVC 118
- ## N
- NaturalLangauge framework 620
  - navigation controller 217
    - stack 217
  - Navigation Controller
    - adding to storyboard 218
    - overview 217
  - Network Testing 24
  - new line 44
  - nextResponder property 403
  - nil coalescing operator 56
  - Notification Actions
    - adding 638



- Notification Authorization
  - requesting 633
- Notification Request
  - creating 635
- Notifications
  - managing 642
- Notification Trigger
  - specifying 635
- NOT (!) operator 55
- NSData 277
- NSDocumentDirectory 278
- NSExtensionContext 586
- NSExtensionItem 586, 587, 589, 595
- NSFileHandle 277
  - creating 285
  - working with files 285
- NSFileManager 277, 283
  - creating 283
  - defaultManager 283
  - reading and writing files 285
- NSFileManager class
  - default manager 278
- NSItemProvider 586, 587, 588, 595
- NSLayoutAnchor 153, 155
  - constraint() method 156
- NSLayoutAttributeBaseline 144
- NSLayoutConstraint 137, 153
- NSLocationAlwaysUsageDescription 500
- NSLocationWhenInUseUsageDescription 500
- NSMetaDataQuery 308
- NSMicrophoneUsageDescription 657
- NSSearchPathForDirectoriesInDomains 278
- NSSecureCoding protocol 588
- NSSpeechRecognitionUsageDescription 658
- NSUbiquitousContainers
  - iCloud Drive 316
- numberOfSectionsInTableView
  - method 212

## O

- Objective-C 41
- offsetInFile method 286

- Opaque Return Types 89
- openInMaps(launchOptions:) method 489
- operands 53
- optional
  - implicitly unwrapped 50
- optional binding 48
- Optional Type 47
- OR (||) operator 55
- OR operator 58
- outlet collection 611

## P

- Pan and Dragging Gestures 419
- Parameter Names 73
  - external 73
  - local 73
- parent class 81
- Particle Emitter
  - node properties 701
  - overview 699
- Particle Emitter Editor 699
- Pathnames 278
- Performance
  - monitoring 24
- perRecordResultBlock 381
- PHAdjustmentData 581
- PHContentEditingController Protocol 575
- PHContentEditingInput 577
- PHContentEditingInput object 576
- PHContentEditingOutput class 580
- Photo Editing Extension 568
  - Info.plist configuration 573
  - tutorial 571
- PHSupportedMediaTypes key 573
- Picture-in-Picture 251, 561
  - opting out 566
- Pinch Gestures 418
- Pixels 435
- playground
  - working with UIKit 35
- Playground 29
  - adding resources 36

## Index

- creating a 29
- Enhanced Live Views 38
- pages 35
- rich text comments 34
- Rich Text Comments 34
- Playground editor 30
- PlaygroundSupport module 38
- Playground Timelines 32
- Points 435
- predictedTouchesForTouch method 405
- preferredFontForTextStyle property 202
- prepare(for segue:) method 221
- Profile in Instruments 25
- Project Navigator 14
- Protocols 88

## Q

- Quartz 2D API 435

## R

- Range Operators 55
- Recording Audio 651
- Refactor to Storyboard 189
- Referenced ID 191
- Reference Types 98
- registerClass method 203
- regular
  - size class 169
- removeArrangedSubview method 238
- removeConstraint method 159
- removeItemAtPath method 280
- repeat ... while loop 62
- resignFirstResponder 129
- responder chain 133, 403
- reverseGeocodeLocation method 488
- reverse-geocoding 488
- Reverse Geocoding 487
- RGBA components 436
- rightAnchor 155
- Right Shift Operator 59
- root controller 193
- Rotation

- restricting 679
- Rounded Rect Button 122

## S

- Safari Extension App 12
- Safe Area Layout 136
- scene delegate 399
- SceneDelegate.swift 379, 399
- screen
  - dimension of 459
- searchResultsUpdater property 225
- searchViewController property 225
- seekToEndOfFile method 286
- seekToFileOffset method 286
- Segue
  - unwind 186
- self 87
- setFillColor method 450
- setLineDash method 453
- setNeedsDisplayInRect method 435
- setNeedsDisplay method 435
- setShadow method 454
- setUbiquitous 314
- SFSpeechURLRecognitionRequest 658
- Share Button
  - adding 594
- shared cloud database 380
- sharedCloudDatabase 381
- Shared Links Extension 569
- Share Extension 567
- shorthand argument names 79, 105
- sign bit 59
- Signing Identities 9
- Size Classes 169
  - Defaults 170
  - in Interface Builder 169
- Size Inspector 18
- SK3DNode class 672
- SKAction class 673
- SKAudioNode class 672
- SKCameraNode class 672
- SKConstraint class 673

- SKCropNode class 672
- SKEffectNode class 672
- SKEmitterNode class 672, 699
- SKFieldNode class 672
- SKLabelNode class 672
- SKLightNode class 672
- SKPhysicsBody class 672
- SKPhysicsContactDelegate protocol 695
- SKPhysicsWorld class 673
- SKShapeNode class 672
- SKSpriteNode class 672
- SKTransition class 673
- SKVideoNode class 672
- sleep() method 265
- some
  - keyword 89
- Speech Recognition 657
  - real-time 663
  - seeking authorization 657, 660
  - Transcribing Live Audio 658
  - Transcribing Recorded Audio 658
  - tutorial 658, 663
- Sprite Kit
  - Actions 673
  - Category Bit Masks 693
  - components 671
  - Contact Delegate 695
  - Contact Masks 694
  - Nodes 672
  - overview 671
  - Physics Bodies 672
  - Physics World 673
  - Rendering Loop 674
  - Scenes 671
  - Texture Atlas 673
  - Transitions 673
- SpriteKit
  - Audio Action 706
  - Named Action Reference 688
- Sprite Kit Level Editor 675
- SpriteKit Live Editor 686
- Sprite Kit View 671
- SQLite 341
  - application preparation 343
  - closing database 344
  - data extraction 344
  - on Mac OS X 341
  - overview 341
  - swift wrappers 343
  - table creation 344
- StackView
  - adding subviews 238
  - alignment 235
  - axis 233
  - baseLineRelativeArrangement 237
  - Bottom 236
  - Center 236
  - configuration options 233
  - distribution 233
  - EqualCentering 234
  - EqualSpacing 234
  - Fill 233, 235
  - FillEqually 233
  - FillProportionally 234
  - FirstBaseLine 236
  - Hiding and Removing Subviews 238
  - LastBaseLine 237
  - layoutMarginsRelativeArrangement 237
  - leading 235
  - spacing 234
  - Top 236
  - trailing 235
  - tutorial 239
- Stack View Class 231
- startContentEditing method 576
- startContentEditingWithInput method 576
- Sticker Pack App 12
- Sticker Pack Extension 569
- Stored and Computed Properties 85
- stored properties 85
- Storyboard
  - add navigation controller 218
  - add table view controller 207
  - add view controller relationship 195

## Index

- design scene 196
- design table view cell prototype 210
- dynamic table view example 207
- file 181
- Insert Tab Bar Controller 194
- prepare(for: segue) method 221
- programming segues 187
- scenes 183
- segues 184
- static vs. dynamic table views 199
- Tab Bar 193
- Tab Bar example 193
- table view navigation 217
- table view overview 199
- table view segue 218
- unwind segue 186
- Storyboards
  - multiple 189
- Storyboard Transitions 184
- String
  - data type 43
- struct keyword 97
- Structured Concurrency 263, 264, 274
  - addTask() function 272
  - async/await 265
  - Asynchronous Properties 275
  - async keyword 266
  - async-let bindings 268
  - await keyword 266, 267
  - cancelAll() function 273
  - cancel() method 272
  - Data Races 273
  - detached tasks 271
  - error handling 269
  - for-await 274
  - isCancelled property 271
  - isEmpty property 273
  - priority 270
  - suspend point 268
  - suspend points 266
  - synchronous code 265
  - Task Groups 272
  - task hierarchy 270
  - Task object 266
  - Tasks 270
  - throw/do/try/catch 269
  - withTaskGroup() 272
  - withThrowingTaskGroup() 272
  - yield() method 272
- Structured Query Language 341
- Structures 97
- Subclassing 119
- subtraction operator 53
- subview 132
- superview 132
- suspend points 266, 268
- Swift
  - Arithmetic Operators 53
  - array iteration 105
  - arrays 103
  - Assignment Operator 53
  - async/await 265
  - async keyword 266
  - async-let bindings 268
  - await keyword 266, 267
  - base class 91
  - Binary Operators 54
  - Bitwise AND 57
  - Bitwise Left Shift 58
  - Bitwise NOT 57
  - Bitwise Operators 57
  - Bitwise OR 58
  - Bitwise Right Shift 59
  - Bitwise XOR 58
  - Bool 42
  - Boolean Logical Operators 55
  - break statement 63
  - calling a function 72
  - case statement 67
  - character data type 42
  - child class 91
  - class declaration 81
  - class deinitialization 83
  - class extensions 94

- class hierarchy 91
- class initialization 83
- Class Methods 82
- class properties 81
- closed range operator 55
- Closure Expressions 78
- Closures 79
- Comparison Operators 54
- Compound Bitwise Operators 59
- constant declaration 45
- constants 45
- continue statement 63
- control flow 61
- data types 41
- Dictionaries 106
- do ... while loop 62
- error handling 111
- Escape Sequences 44
- exclusive OR 58
- expressions 53
- floating point 42
- for Statement 61
- function declaration 71
- functions 71
- guard statement 65
- half-closed range operator 56
- if ... else ... Statements 64
- if Statement 64
- implicit returns 72
- Inheritance, Classes and Subclasses 91
- Instance Properties 82
- instance variables 82
- integers 42
- methods 81
- opaque return types 89
- operators 53
- optional binding 48
- optional type 47
- Overriding 92
- parent class 91
- protocols 88
- Range Operators 55

- Reference Types 98
- root class 91
- single expression functions 72
- single expression returns 72
- single inheritance 91
- Special Characters 44
- Stored and Computed Properties 85
- String data type 43
- structured concurrency 263
- structures 97
- subclass 91
- suspend points 266
- switch fallthrough 70
- switch statement 67
  - syntax 67
- Ternary Operator 56
- tuples 46
- type annotations 45
- type casting 50
- type checking 50
- type inference 45
- Value Types 98
- variable declaration 45
- variables 44
- while loop 62
- Swift Playground 29
- Swift Structures 97
- Swipe Gestures 419
- switch statement 67
  - example 67
- switch Statement 67
  - example 67
  - range matching 69
- synchronous code 265

## T

- Tab Bar Controller
  - adding to storyboard 194
- Tab Bar Items
  - configuring 197
- Table Cells
  - self-sizing 201

## Index

- Table View 199
  - cell styles 202
  - datasource 211
  - styles 200
- Table View Cell
  - reuse 203
- TableView Navigation 217
- Tap Gestures 418
- Taps 404
- Target-Action 118
- Task.detached() method 271
- Task Groups 272
  - addTask() function 272
  - cancelAll() function 273
  - isEmpty property 273
  - withTaskGroup() 272
  - withThrowingTaskGroup() 272
- Task Hierarchy 270
- Task object 266
- Tasks 270
  - cancel() 272
  - detached tasks 271
  - isCancelled property 271
  - overview 270
  - priority 270
- Temporary Directory 279
- ternary operator 56
- Texture Atlas 673
  - adding to project 683
  - example 686
- Threads
  - overview 263
- throw statement 112
- topAnchor 156
- Touch
  - coordinates of 409
- Touches 404
- touchesBegan 404
- touchesBegan event 128
- touchesCancelled 405
- touchesEnded 404
- touchesMoved 404
- Touch ID
  - checking availability 425
  - example 427
  - policy evaluation 426
  - seeking authentication 429
- Touch ID Authentication 425
- Touch Notification Methods 404
- Touch Prediction 405
- Touch Predictions
  - checking for 410
- touch scan rate 405
- Touch Up Inside event 118
- trailingAnchor 155
- traitCollectionDidChange method 253
- Traits 169
  - variations 173
- Trait Variations 169, 170
  - Attributes Inspector 171
  - in Interface Builder 170
- Transit ETA Information 514
- try statement 112
- try! statement 114
- Tuple 46
- Type Annotations 45
- type casting 50
- Type Checking 50
- Type Inference 45
- type safe programming 45

## U

- ubiquity-container-identifiers 295
- UIActivityViewController 595
- UIApplication 119
- UIAttachmentBehavior class 475
- UIButton 131
- UICloudSharingController
  - sample code 376
- UICloudSharingController Class 376
- UICollisionBehavior class 474
- UICollisionBehaviorMode 474
- UIColor class 436
- UIControl 134

- UIDocument 297
  - contents(forType:) 297
  - documentState 297
  - example 298
  - load(fromContents:) 297
  - overview 297
  - subclassing 298
- UIDocumentBrowserViewController 327
- UIDocumentState options 297
- UIDynamicAnimator class 472, 473
- UIDynamicItemBehavior class 477
- UIFontTextStyle
  - properties 202
- UIGestureRecognizer 417, 418
- UIGraphicsGetCurrentContext() function 436
- UIGravityBehavior class 473
- UIImagePickerController 547
  - delegate 548
  - source types 547
- UIImageWriteToSavedPhotosAlbum 549
- UIKit
  - in playgrounds 35
- UIKit Dynamics 471
  - architecture 471
  - Attachment Behavior 475
  - collision behavior 474
  - Dynamic Animator 472
  - dynamic behaviors 472
  - dynamic items 471
  - example 479
  - gravity behavior 473
  - overview of 471
  - push behavior 476
  - reference view 472
  - snap behavior 476
- UIKit Dynamics
  - dynamic items 471
- UIKit Newton 476
- UILabel 128
  - set color 35
- UILongPressGestureRecognizer 417
- UINavigationController 217
- UINavigationController 193
- UInt8 42
- UInt16 42
- UInt32 42
- UInt64 42
- UIPanGestureRecognizer 417
- UIPinchGestureRecognizer 417
- UIPushBehavior class 476
- UIRotationGestureRecognizer 417
- UISaveVideoAtPathToSavedPhotosAlbum 549
- UIScreen 459
- UIScreenEdgePanGestureRecognizer 417
- UIScrollView 134
- UISearchBarDelegate 225
- UISearchController 225
- UISearchControllerDelegate 225
- UISnapBehavior class 476
- UISpringTimingParameters class 468
- UIStackView class 231, 239
- UISwipeGestureRecognizer 417
- UITabBar 193
- UITabBarController 193
- UITableView 134, 207, 217
  - Prototype Cell 210
- UITableViewCell 199, 207
- UITableViewCell class 199
- UITableViewCellStyle
  - types 202
- UITableViewDataSource protocol 199
- UITableViewDelegate protocol 199
- UITapGestureRecognizer 417
- UITextField 131
- UITextView 134
- UIToolbar 134
- UIViewAnimationOptions 464
- UIViewController 119, 124
- UIViewPropertyAnimator 463
- UIWindow 131
- unary negative operator 53
- Unicode scalar 44

## Index

Universal Image Assets 176  
universal interface 169  
Universal User Interfaces 169  
UNMutableNotificationContent class 635  
UNNotificationRequest 635  
Unstructured Concurrency 270  
    cancel() method 272  
    detached tasks 271  
    isCancelled property 271  
    priority 270  
    yield() method 272  
UNUserNotificationCenter 633  
UNUserNotificationCenterDelegate 637  
upcasting 50  
updateSeatchResults 225  
userDidAcceptCloudKitShareWith method , 379, 379  
user location  
    updating 520  
userNotification  
    didReceive method 639  
UserNotifications framework 633  
Utilities panel 17

## V

Value Types 98  
variables 44  
variadic parameters 74  
Vertical Stack View 231  
Video Playback 557  
ViewController.swift file 124  
viewDidLoad method 128  
view hierarchies 131  
View Hierarchy 131  
views 131  
View Types 133  
viewWillTransitionToSize method 253  
Vision framework 620  
Vision Framework  
    example 627  
Visual Format Language 137, 165  
    constraintsWithVisualFormat 166  
    examples 165

VNCoreMLModel 628  
VNCoreMLRequest 628  
VNImageRequestHandler 628

## W

where clause 49  
where statement 69  
while Loop 62  
widthAnchor 156  
willTransitionToTraitCollection method 253  
windows 131  
withTaskGroup() 272  
withThrowingTaskGroup() 272  
WKWebView 134

## X

Xcode  
    installation 7  
    preferences 8  
    Utilities panel 17  
XCPShowView 38  
XOR operator 58

## Y

yield() method 272