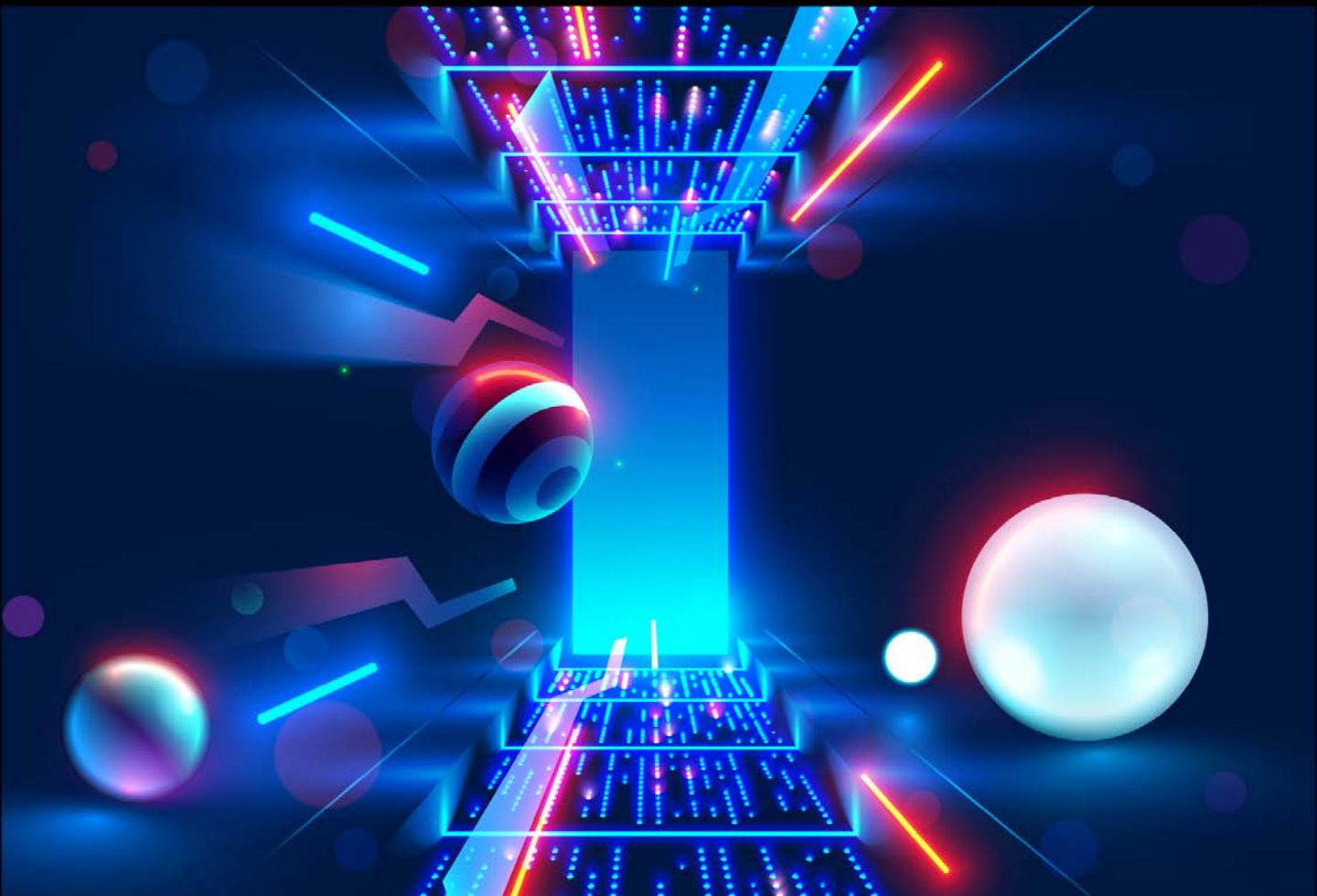


iOS 18 App Development Essentials



iOS 18 App Development Essentials

iOS 18 App Development Essentials

ISBN-13: 978-1-951442-99-6

© 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



<https://www.payloadbooks.com>

Table of Contents

1. Start Here.....	1
1.1 For Swift Programmers.....	1
1.2 For Non-Swift Programmers	1
1.3 Source Code Download.....	2
1.4 Feedback.....	2
1.5 Errata.....	2
1.6 Take the Knowledge Tests	2
2. Joining the Apple Developer Program.....	3
2.1 Downloading Xcode 16 and the iOS 18 SDK	3
2.2 Apple Developer Program.....	3
2.3 When to Enroll in the Apple Developer Program?.....	3
2.4 Enrolling in the Apple Developer Program	4
2.5 Summary	5
3. Installing Xcode 16 and the iOS 18 SDK	7
3.1 Identifying Your macOS Version	7
3.2 Installing Xcode 16 and the iOS 18 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
4. An Introduction to Xcode 16 Playgrounds.....	11
4.1 What is a Playground?	11
4.2 Creating a New Playground	11
4.3 A Swift Playground Example	12
4.4 Viewing Results	14
4.5 Adding Rich Text Comments	16
4.6 Working with Playground Pages	17
4.7 Working with SwiftUI and Live View in Playgrounds	17
4.8 Summary	20
5. Swift Data Types, Constants, and Variables	21
5.1 Using a Swift Playground	21
5.2 Swift Data Types	21
5.2.1 Integer Data Types	22
5.2.2 Floating Point Data Types	22
5.2.3 Bool Data Type	23
5.2.4 Character Data Type.....	23
5.2.5 String Data Type.....	23
5.2.6 Special Characters/Escape Sequences	24
5.3 Swift Variables.....	25
5.4 Swift Constants	25
5.5 Declaring Constants and Variables.....	25

Table of Contents

5.6 Type Annotations and Type Inference	25
5.7 The Swift Tuple	26
5.8 The Swift Optional Type.....	27
5.9 Type Casting and Type Checking.....	30
5.10 Take the Knowledge Test.....	32
5.11 Summary	32
6. Swift Operators and Expressions	33
6.1 Expression Syntax in Swift	33
6.2 The Basic Assignment Operator.....	33
6.3 Swift Arithmetic Operators.....	33
6.4 Compound Assignment Operators.....	34
6.5 Comparison Operators.....	34
6.6 Boolean Logical Operators.....	35
6.7 Range Operators.....	35
6.8 The Ternary Operator	36
6.9 Nil Coalescing Operator.....	36
6.10 Bitwise Operators	37
6.10.1 Bitwise NOT	37
6.10.2 Bitwise AND	37
6.10.3 Bitwise OR.....	38
6.10.4 Bitwise XOR.....	38
6.10.5 Bitwise Left Shift.....	38
6.10.6 Bitwise Right Shift.....	39
6.11 Compound Bitwise Operators.....	39
6.12 Take the Knowledge Test.....	40
6.13 Summary	40
7. Swift Control Flow.....	41
7.1 Looping Control Flow	41
7.2 The Swift for-in Statement.....	41
7.2.1 The while Loop	42
7.3 The repeat ... while loop	42
7.4 Breaking from Loops	43
7.5 The continue Statement	43
7.6 Conditional Control Flow.....	44
7.7 Using the if Statement	44
7.8 Using if ... else ... Statements	44
7.9 Using if ... else if ... Statements	45
7.10 The guard Statement.....	45
7.11 Take the Knowledge Test.....	46
7.12 Summary	46
8. The Swift Switch Statement	47
8.1 Why Use a switch Statement?	47
8.2 Using the switch Statement Syntax	47
8.3 A Swift switch Statement Example	47
8.4 Combining case Statements	48
8.5 Range Matching in a switch Statement.....	49

8.6 Using the where statement.....	49
8.7 Fallthrough.....	50
8.8 Take the Knowledge Test.....	50
8.9 Summary	50
9. Swift Functions, Methods, and Closures.....	53
9.1 What is a Function?	53
9.2 What is a Method?	53
9.3 How to Declare a Swift Function	53
9.4 Implicit Returns from Single Expressions.....	54
9.5 Calling a Swift Function	54
9.6 Handling Return Values	54
9.7 Local and External Parameter Names	55
9.8 Declaring Default Function Parameters.....	55
9.9 Returning Multiple Results from a Function.....	56
9.10 Variable Numbers of Function Parameters	56
9.11 Parameters as Variables	57
9.12 Working with In-Out Parameters	57
9.13 Functions as Parameters.....	58
9.14 Closure Expressions.....	60
9.15 Shorthand Argument Names.....	61
9.16 Closures in Swift.....	61
9.17 Take the Knowledge Test.....	62
9.18 Summary	62
10. The Basics of Swift Object-Oriented Programming.....	63
10.1 What is an Instance?	63
10.2 What is a Class?	63
10.3 Declaring a Swift Class	63
10.4 Adding Instance Properties to a Class.....	64
10.5 Defining Methods	64
10.6 Declaring and Initializing a Class Instance.....	65
10.7 Initializing and De-initializing a Class Instance	65
10.8 Calling Methods and Accessing Properties	66
10.9 Stored and Computed Properties.....	67
10.10 Lazy Stored Properties.....	68
10.11 Using self in Swift.....	69
10.12 Understanding Swift Protocols.....	70
10.13 Opaque Return Types.....	71
10.14 Take the Knowledge Test.....	72
10.15 Summary	72
11. An Introduction to Swift Subclassing and Extensions	73
11.1 Inheritance, Classes, and Subclasses.....	73
11.2 A Swift Inheritance Example	73
11.3 Extending the Functionality of a Subclass	74
11.4 Overriding Inherited Methods.....	74
11.5 Initializing the Subclass.....	75
11.6 Using the SavingsAccount Class	76

Table of Contents

11.7 Swift Class Extensions	76
11.8 Take the Knowledge Test.....	77
11.9 Summary	77
12. An Introduction to Swift Structures and Enumerations.....	79
12.1 An Overview of Swift Structures.....	79
12.2 Value Types vs. Reference Types	80
12.3 When to Use Structures or Classes	82
12.4 An Overview of Enumerations.....	82
12.5 Take the Knowledge Test.....	84
12.6 Summary	84
13. An Introduction to Swift Property Wrappers.....	85
13.1 Understanding Property Wrappers.....	85
13.2 A Simple Property Wrapper Example	85
13.3 Supporting Multiple Variables and Types	87
13.4 Take the Knowledge Test.....	90
13.5 Summary	90
14. Working with Array and Dictionary Collections in Swift.....	91
14.1 Mutable and Immutable Collections	91
14.2 Swift Array Initialization.....	91
14.3 Working with Arrays in Swift	92
14.3.1 Array Item Count	92
14.3.2 Accessing Array Items	92
14.3.3 Random Items and Shuffling	92
14.3.4 Appending Items to an Array	93
14.3.5 Inserting and Deleting Array Items	93
14.3.6 Array Iteration	93
14.4 Creating Mixed Type Arrays.....	94
14.5 Swift Dictionary Collections.....	94
14.6 Swift Dictionary Initialization	94
14.7 Sequence-based Dictionary Initialization.....	95
14.8 Dictionary Item Count.....	96
14.9 Accessing and Updating Dictionary Items	96
14.10 Adding and Removing Dictionary Entries	96
14.11 Dictionary Iteration	96
14.12 Take the Knowledge Test.....	97
14.13 Summary	97
15. Understanding Error Handling in Swift 5	99
15.1 Understanding Error Handling	99
15.2 Declaring Error Types	99
15.3 Throwing an Error	100
15.4 Calling Throwing Methods and Functions	100
15.5 Accessing the Error Object	102
15.6 Disabling Error Catching	102
15.7 Using the defer Statement	102
15.8 Take the Knowledge Test.....	103
15.9 Summary	103

16. An Overview of SwiftUI	105
16.1 UIKit and Interface Builder	105
16.2 SwiftUI Declarative Syntax	105
16.3 SwiftUI is Data Driven	106
16.4 SwiftUI vs. UIKit	106
16.5 Take the Knowledge Test.....	107
16.6 Summary	107
17. Using Xcode in SwiftUI Mode	109
17.1 Starting Xcode 16	109
17.2 Creating a SwiftUI Project	109
17.3 Xcode in SwiftUI Mode.....	111
17.4 The Preview Canvas	113
17.5 Preview Pinning	114
17.6 The Preview Toolbar	115
17.7 Modifying the Design.....	116
17.8 Editor Context Menu.....	118
17.9 Running the App on a Simulator	119
17.10 Running the App on a Physical iOS Device	119
17.11 Managing Devices and Simulators.....	120
17.12 Enabling Network Testing.....	121
17.13 Dealing with Build Errors.....	122
17.14 Monitoring Application Performance.....	122
17.15 Exploring the User Interface Layout Hierarchy.....	123
17.16 Take the Knowledge Test.....	125
17.17 Summary.....	125
18. SwiftUI Architecture	127
18.1 SwiftUI App Hierarchy.....	127
18.2 App.....	127
18.3 Scenes.....	127
18.4 Views.....	128
18.5 Take the Knowledge Test.....	128
18.6 Summary	128
19. The Anatomy of a Basic SwiftUI Project	129
19.1 Creating an Example Project	129
19.2 The DemoProjectApp.swift File	129
19.3 The ContentView.swift File.....	130
19.4 Assets.xcassets	130
19.5 DemoProject.entitlements	130
19.6 Preview Content.....	131
19.7 Take the Knowledge Test.....	131
19.8 Summary	131
20. Creating Custom Views with SwiftUI	133
20.1 SwiftUI Views	133
20.2 Creating a Basic View	133
20.3 Adding Views.....	134

Table of Contents

20.4 SwiftUI Hierarchies.....	135
20.5 Concatenating Text Views.....	136
20.6 Working with Subviews.....	136
20.7 Views as Properties	137
20.8 Modifying Views	137
20.9 Working with Text Styles.....	138
20.10 Modifier Ordering.....	139
20.11 Custom Modifiers	140
20.12 Basic Event Handling.....	141
20.13 Building Custom Container Views.....	141
20.14 Working with the Label View.....	143
20.15 Take the Knowledge Test.....	144
20.16 Summary.....	144
21. SwiftUI Stacks and Frames.....	145
21.1 SwiftUI Stacks.....	145
21.2 Spacers, Alignment and Padding.....	147
21.3 Grouping Views.....	149
21.4 Dynamic HStack and VStack Conversion	149
21.5 Text Line Limits and Layout Priority.....	150
21.6 Traditional vs. Lazy Stacks	151
21.7 SwiftUI Frames.....	152
21.8 Frames and the Geometry Reader	154
21.9 Take the Knowledge Test.....	154
21.10 Summary.....	154
22. SwiftUI State Properties, Observation, and Environment Objects.....	155
22.1 State Properties.....	155
22.2 State Binding.....	157
22.3 Observable Objects	157
22.4 Observation using Combine.....	158
22.5 Combine State Objects	159
22.6 Using the Observation Framework.....	160
22.7 Observation and @Bindable	161
22.8 Environment Objects.....	161
22.9 Take the Knowledge Test.....	163
22.10 Summary.....	163
23. A SwiftUI Example Tutorial.....	165
23.1 Creating the Example Project.....	165
23.2 Reviewing the Project.....	165
23.3 Modifying the Layout	167
23.4 Adding a Slider View to the Stack.....	168
23.5 Adding a State Property	168
23.6 Adding Modifiers to the Text View.....	169
23.7 Adding Rotation and Animation	170
23.8 Adding a TextField to the Stack.....	171
23.9 Adding a Color Picker	172
23.10 Tidying the Layout.....	173

23.11 Take the Knowledge Test.....	176
23.12 Summary.....	176
24. SwiftUI Stack Alignment and Alignment Guides.....	177
24.1 Container Alignment.....	177
24.2 Alignment Guides.....	179
24.3 Custom Alignment Types.....	182
24.4 Cross Stack Alignment.....	185
24.5 ZStack Custom Alignment.....	187
24.6 Take the Knowledge Test.....	191
24.7 Summary.....	191
25. Predictive Code Completion in Xcode.....	193
25.1 Enabling Predictive Code Completion.....	193
25.2 Creating the CodeCompletion Project.....	194
25.3 Working with Code Completion.....	194
25.4 Code Completion from Comments.....	196
25.5 Take the Knowledge Test.....	197
25.6 Summary.....	197
26. An Overview of Swift Structured Concurrency.....	199
26.1 An Overview of Threads.....	199
26.2 The Application Main Thread.....	199
26.3 Completion Handlers.....	199
26.4 Structured Concurrency.....	200
26.5 Preparing the Project.....	200
26.6 Non-Concurrent Code.....	200
26.7 Introducing async/await Concurrency.....	201
26.8 Asynchronous Calls from Synchronous Functions.....	202
26.9 The await Keyword.....	202
26.10 Using async-let Bindings.....	203
26.11 Handling Errors.....	204
26.12 Understanding Tasks.....	205
26.13 Unstructured Concurrency.....	205
26.14 Detached Tasks.....	206
26.15 Task Management.....	207
26.16 Working with Task Groups.....	207
26.17 Avoiding Data Races.....	208
26.18 The for-await Loop.....	209
26.19 Asynchronous Properties.....	210
26.20 Take the Knowledge Test.....	211
26.21 Summary.....	211
27. An Introduction to Swift Actors.....	213
27.1 An Overview of Actors.....	213
27.2 Declaring an Actor.....	213
27.3 Understanding Data Isolation.....	214
27.4 A Swift Actor Example.....	215
27.5 Introducing the MainActor.....	216
27.6 Take the Knowledge Test.....	218

Table of Contents

27.7 Summary	218
28. SwiftUI Concurrency and Lifecycle Event Modifiers	219
28.1 Creating the LifecycleDemo Project	219
28.2 Designing the App	219
28.3 The onAppear and onDisappear Modifiers	220
28.4 The onChange Modifier	221
28.5 ScenePhase and the onChange Modifier.....	221
28.6 Launching Concurrent Tasks.....	223
28.7 Take the Knowledge Test.....	224
28.8 Summary	224
29. SwiftUI Observable and Environment Objects – A Tutorial.....	225
29.1 About the ObservableDemo Project.....	225
29.2 Creating the Project	225
29.3 Adding the Observable Object.....	225
29.4 Designing the ContentView Layout.....	226
29.5 Adding the Second View.....	227
29.6 Adding Navigation	228
29.7 Using an Environment Object.....	229
29.8 Take the Knowledge Test.....	230
29.9 Summary	230
30. SwiftUI Data Persistence using AppStorage and SceneStorage.....	231
30.1 The @SceneStorage Property Wrapper.....	231
30.2 The @AppStorage Property Wrapper	231
30.3 Creating and Preparing the StorageDemo Project	232
30.4 Using Scene Storage	233
30.5 Using App Storage.....	234
30.6 Storing Custom Types.....	235
30.7 Take the Knowledge Test.....	237
30.8 Summary	237
31. SwiftUI Lists and Navigation	239
31.1 SwiftUI Lists.....	239
31.2 Modifying List Separators and Rows.....	240
31.3 SwiftUI Dynamic Lists.....	241
31.4 Creating a Refreshable List	243
31.5 SwiftUI UINavigationController and NavigationLink	244
31.6 Navigation by Value Type.....	246
31.7 Working with Navigation Paths	247
31.8 Navigation Bar Customization	247
31.9 Making the List Editable	248
31.10 Hierarchical Lists	250
31.11 Multicolumn Navigation.....	251
31.12 Take the Knowledge Test.....	251
31.13 Summary.....	251
32. A SwiftUI List and UINavigationController Tutorial	253
32.1 About the ListNavDemo Project.....	253

32.2 Creating the ListNavDemo Project.....	253
32.3 Preparing the Project.....	253
32.4 Adding the Car Structure.....	254
32.5 Adding the Data Store.....	254
32.6 Designing the Content View.....	255
32.7 Designing the Detail View.....	256
32.8 Adding Navigation to the List.....	258
32.9 Designing the Add Car View.....	259
32.10 Implementing Add and Edit Buttons.....	261
32.11 Adding a Navigation Path.....	263
32.12 Adding the Edit Button Methods.....	264
32.13 Summary.....	265
33. An Overview of Split View Navigation.....	267
33.1 Introducing NavigationSplitView.....	267
33.2 Using NavigationSplitView.....	267
33.3 Handling List Selection.....	268
33.4 NavigationSplitView Configuration.....	268
33.5 Controlling Column Visibility.....	269
33.6 Take the Knowledge Test.....	270
33.7 Summary.....	270
34. A NavigationSplitView Tutorial.....	271
34.1 About the Project.....	271
34.2 Creating the NavSplitDemo Project.....	271
34.3 Adding the Project Data.....	271
34.4 Creating the Navigation View.....	272
34.5 Building the Sidebar Column.....	272
34.6 Adding the Content Column List.....	273
34.7 Adding the Detail Column.....	274
34.8 Configuring the Split Navigation Experience.....	275
34.9 Summary.....	276
35. An Overview of List, OutlineGroup and DisclosureGroup.....	277
35.1 Hierarchical Data and Disclosures.....	277
35.2 Hierarchies and Disclosure in SwiftUI Lists.....	278
35.3 Using OutlineGroup.....	280
35.4 Using DisclosureGroup.....	281
35.5 Take the Knowledge Test.....	283
35.6 Summary.....	283
36. A SwiftUI List, OutlineGroup, and DisclosureGroup Tutorial.....	285
36.1 About the Example Project.....	285
36.2 Creating the OutlineGroupDemo Project.....	285
36.3 Adding the Data Structure.....	285
36.4 Adding the List View.....	287
36.5 Testing the Project.....	288
36.6 Using the Sidebar List Style.....	288
36.7 Using OutlineGroup.....	289
36.8 Working with DisclosureGroups.....	290

Table of Contents

36.9 Summary	294
37. Building SwiftUI Grids with LazyVGrid and LazyHGrid	295
37.1 SwiftUI Grids	295
37.2 GridItems	295
37.3 Creating the GridDemo Project	296
37.4 Working with Flexible GridItems	297
37.5 Adding Scrolling Support to a Grid.....	298
37.6 Working with Adaptive GridItems	300
37.7 Working with Fixed GridItems	301
37.8 Using the LazyHGrid View.....	303
37.9 Take the Knowledge Test.....	305
37.10 Summary.....	305
38. Building SwiftUI Grids with Grid and GridRow.....	307
38.1 Grid and GridRow Views.....	307
38.2 Creating the GridRowDemo Project	307
38.3 A Simple Grid Layout	308
38.4 Non-GridRow Children	309
38.5 Automatic Empty Grid Cells	310
38.6 Adding Empty Cells.....	311
38.7 Column Spanning.....	312
38.8 Grid Alignment and Spacing.....	312
38.9 Take the Knowledge Test.....	318
38.10 Summary.....	318
39. Building Custom Containers	319
39.1 Introducing custom containers	319
39.2 Working with ViewBuilder Closures.....	319
39.3 Supporting Section Headers	320
39.4 Take the Knowledge Test.....	322
39.5 Summary	322
40. A SwiftUI Custom Container Tutorial.....	323
40.1 About the Custom Container Project.....	323
40.2 Creating the CustomContainerDemo Project.....	323
40.3 Adding the Sample Data.....	324
40.4 Declaring the Item View.....	324
40.5 Designing the Container	325
40.6 Using the Custom Container	325
40.7 Completing the Item View.....	326
40.8 Adding Section Headers.....	328
40.9 Summary	330
41. Building Tabbed and Paged Views in SwiftUI.....	331
41.1 An Overview of SwiftUI TabView.....	331
41.2 Creating the TabViewDemo App.....	332
41.3 Adding the TabView Container	332
41.4 Adding the Content Views.....	332
41.5 Adding View Paging	332

41.6 Adding the Tab Items.....	333
41.7 Adding Tab Item Tags.....	333
41.8 Take the Knowledge Test.....	334
41.9 Summary	334
42. Building Context Menus in SwiftUI.....	335
42.1 Creating the ContextMenuDemo Project.....	335
42.2 Preparing the Content View	335
42.3 Adding the Context Menu	335
42.4 Testing the Context Menu.....	337
42.5 Take the Knowledge Test.....	337
42.6 Summary	337
43. Basic SwiftUI Graphics Drawing	339
43.1 Creating the DrawDemo Project.....	339
43.2 SwiftUI Shapes.....	339
43.3 Using Overlays.....	341
43.4 Drawing Custom Paths and Shapes	342
43.5 Color Mixing.....	344
43.6 Color Gradients and Shadows.....	345
43.7 Drawing Gradients.....	345
43.8 Mesh Gradients	348
43.9 Take the Knowledge Test.....	349
43.10 Summary.....	349
44. SwiftUI Animation and Transitions.....	351
44.1 Creating the AnimationDemo Example Project.....	351
44.2 Implicit Animation	351
44.3 Repeating an Animation	353
44.4 Explicit Animation.....	354
44.5 Animation and State Bindings.....	355
44.6 Automatically Starting an Animation	356
44.7 SwiftUI Transitions	358
44.8 Combining Transitions.....	359
44.9 Asymmetrical Transitions	360
44.10 Take the Knowledge Test.....	360
44.11 Summary.....	360
45. Working with Gesture Recognizers in SwiftUI.....	361
45.1 Creating the GestureDemo Example Project	361
45.2 Basic Gestures.....	361
45.3 The onChange Action Callback.....	362
45.4 The updating Callback Action.....	364
45.5 Composing Gestures.....	365
45.6 Take the Knowledge Test.....	367
45.7 Summary	367
46. Creating a Customized SwiftUI ProgressView	369
46.1 ProgressView Styles	369
46.2 Creating the ProgressViewDemo Project	370

Table of Contents

46.3 Adding a ProgressView	370
46.4 Using the Circular ProgressView Style.....	370
46.5 Declaring an Indeterminate ProgressView	371
46.6 ProgressView Customization.....	371
46.7 Take the Knowledge Test.....	374
46.8 Summary	374
47. Presenting Data with SwiftUI Charts	375
47.1 Introducing SwiftUI Charts	375
47.2 Passing Data to the Chart.....	376
47.3 Combining Mark Types.....	377
47.4 Filtering Data into Multiple Graphs	378
47.5 Changing the Chart Background	379
47.6 Changing the Interpolation Method	379
47.7 Take the Knowledge Test.....	380
47.8 Summary	380
48. A SwiftUI Charts Tutorial	381
48.1 Creating the ChartDemo Project	381
48.2 Adding the Project Data.....	381
48.3 Adding the Chart View.....	382
48.4 Creating Multiple Graphs.....	383
48.5 Summary	384
49. An Overview of SwiftUI DocumentGroup Scenes	385
49.1 Documents in Apps	385
49.2 Creating the DocDemo App.....	385
49.3 The DocumentGroup Scene	386
49.4 Declaring File Type Support	387
49.4.1 Document Content Type Identifier	387
49.4.2 Handler Rank.....	387
49.4.3 Type Identifiers.....	387
49.4.4 Filename Extensions	387
49.4.5 Custom Type Document Content Identifiers	387
49.4.6 Exported vs. Imported Type Identifiers	388
49.5 Configuring File Type Support in Xcode	388
49.6 The Document Structure.....	389
49.7 The Content View.....	391
49.8 Adding Navigation	391
49.9 Running the Example App.....	392
49.10 Customizing the Launch Screen	393
49.11 Take the Knowledge Test.....	395
49.12 Summary	395
50. A SwiftUI DocumentGroup Tutorial	397
50.1 Creating the ImageDocDemo Project	397
50.2 Modifying the Info.plist File	397
50.3 Adding an Image Asset.....	398
50.4 Modifying the ImageDocDemoDocument.swift File	398
50.5 Designing the Content View.....	399

50.6 Filtering the Image.....	401
50.7 Testing the App.....	402
50.8 Summary	402
51. An Introduction to Core Data and SwiftUI.....	403
51.1 The Core Data Stack.....	403
51.2 Persistent Container.....	404
51.3 Managed Objects.....	404
51.4 Managed Object Context	404
51.5 Managed Object Model.....	404
51.6 Persistent Store Coordinator.....	405
51.7 Persistent Object Store.....	405
51.8 Defining an Entity Description	405
51.9 Initializing the Persistent Container.....	406
51.10 Obtaining the Managed Object Context.....	406
51.11 Setting the Attributes of a Managed Object.....	406
51.12 Saving a Managed Object.....	406
51.13 Fetching Managed Objects.....	407
51.14 Retrieving Managed Objects based on Criteria	407
51.15 Take the Knowledge Test.....	408
51.16 Summary.....	408
52. A SwiftUI Core Data Tutorial.....	409
52.1 Creating the CoreDataDemo Project	409
52.2 Defining the Entity Description	409
52.3 Creating the Persistence Controller.....	411
52.4 Setting up the View Context.....	411
52.5 Preparing the ContentView for Core Data	412
52.6 Designing the User Interface	412
52.7 Saving Products	414
52.8 Testing the addProduct() Function	416
52.9 Deleting Products.....	416
52.10 Adding the Search Function	417
52.11 Testing the Completed App	420
52.12 Summary.....	420
53. An Overview of SwiftUI Core Data and CloudKit Storage	421
53.1 An Overview of CloudKit	421
53.2 CloudKit Containers.....	421
53.3 CloudKit Public Database.....	421
53.4 CloudKit Private Databases	422
53.5 Data Storage Quotas	422
53.6 CloudKit Records.....	422
53.7 CloudKit Record IDs	423
53.8 CloudKit References	423
53.9 Record Zones	423
53.10 CloudKit Console.....	423
53.11 CloudKit Sharing	424
53.12 CloudKit Subscriptions	424

Table of Contents

53.13 Take the Knowledge Test.....	424
53.14 Summary.....	424
54. A SwiftUI Core Data and CloudKit Tutorial.....	425
54.1 Enabling CloudKit Support.....	425
54.2 Enabling Background Notifications Support.....	427
54.3 Switching to the CloudKit Persistent Container.....	427
54.4 Testing the App.....	428
54.5 Reviewing the Saved Data in the CloudKit Console.....	428
54.6 Filtering and Sorting Queries.....	429
54.7 Editing and Deleting Records.....	430
54.8 Adding New Records.....	431
54.9 Summary.....	432
55. An Introduction to SwiftData.....	433
55.1 Introducing SwiftData.....	433
55.2 Model Classes.....	433
55.3 Model Container.....	434
55.4 Model Configuration.....	434
55.5 Model Context.....	434
55.6 Predicates and FetchDescriptors.....	435
55.7 The @Query Macro.....	435
55.8 Model Relationships.....	435
55.9 Model Attributes.....	436
55.10 Take the Knowledge Test.....	437
55.11 Summary.....	437
56. A SwiftData Tutorial.....	439
56.1 About the SwiftData Project.....	439
56.2 Creating the SwiftDataDemo Project.....	439
56.3 Adding the Data Models.....	439
56.4 Setting up the Model Container.....	440
56.5 Accessing the Model Context.....	440
56.6 Designing the Visitor List View.....	440
56.7 Establishing the Relationship.....	441
56.8 Creating the Visitor Detail View.....	442
56.9 Modifying the Content View.....	443
56.10 Testing the SwiftData Demo App.....	444
56.11 Adding the Search Predicate.....	444
56.12 Summary.....	447
57. Building Widgets with SwiftUI and WidgetKit.....	449
57.1 An Overview of Widgets.....	449
57.2 The Widget Extension.....	449
57.3 Widget Configuration Types.....	450
57.4 Widget Entry View.....	451
57.5 Widget Timeline Entries.....	451
57.6 Widget Timeline.....	451
57.7 Widget Provider.....	452
57.8 Reload Policy.....	452

57.9 Relevance.....	453
57.10 Forcing a Timeline Reload.....	453
57.11 Widget Sizes.....	454
57.12 Widget Placeholder.....	454
57.13 Take the Knowledge Test.....	455
57.14 Summary.....	455
58. A SwiftUI WidgetKit Tutorial.....	457
58.1 About the WidgetDemo Project.....	457
58.2 Creating the WidgetDemo Project.....	457
58.3 Building the App.....	457
58.4 Adding the Widget Extension.....	460
58.5 Adding the Widget Data.....	461
58.6 Creating Sample Timelines.....	462
58.7 Adding Image and Color Assets.....	463
58.8 Designing the Widget View.....	465
58.9 Modifying the Widget Provider.....	467
58.10 Configuring the Placeholder View.....	468
58.11 Previewing the Widget.....	468
58.12 Summary.....	470
59. Supporting WidgetKit Size Families.....	471
59.1 Supporting Multiple Size Families.....	471
59.2 Adding Size Support to the Widget View.....	472
59.3 Take the Knowledge Test.....	475
59.4 Summary.....	475
60. A SwiftUI WidgetKit Deep Link Tutorial.....	477
60.1 Adding Deep Link Support to the Widget.....	477
60.2 Adding Deep Link Support to the App.....	480
60.3 Testing the Widget.....	481
60.4 Summary.....	481
61. Adding Configuration Options to a WidgetKit Widget.....	483
61.1 Reviewing the Project Code.....	483
61.2 Adding an App Entity.....	484
61.3 Adding Entity Query.....	485
61.4 Modifying the App Intent.....	485
61.5 Modifying the Timeline Code.....	486
61.6 Testing Widget Configuration.....	487
61.7 Customizing the Configuration Intent UI.....	488
61.8 Take the Knowledge Test.....	489
61.9 Summary.....	489
62. An Overview of Live Activities in SwiftUI.....	491
62.1 Introducing Live Activities.....	491
62.2 Creating a Live Activity.....	491
62.3 Live Activity Attributes.....	491
62.4 Designing the Live Activity Presentations.....	492
62.4.1 Lock Screen/Banner.....	493

Table of Contents

62.4.2 Dynamic Island Expanded Regions.....	493
62.4.3 Dynamic Island Compact Regions	494
62.4.4 Dynamic Island Minimal	495
62.5 Starting a Live Activity	495
62.6 Updating a Live Activity.....	496
62.7 Activity Alert Configurations	497
62.8 Stopping a Live Activity.....	497
62.9 Take the Knowledge Test.....	498
62.10 Summary	498
63. A SwiftUI Live Activity Tutorial.....	499
63.1 About the LiveActivityDemo Project	499
63.2 Creating the Project	499
63.3 Building the View Model	499
63.4 Designing the Content View.....	500
63.5 Adding the Live Activity Extension.....	502
63.6 Enabling Live Activities Support.....	504
63.7 Enabling the Background Fetch Capability	504
63.8 Defining the Activity Widget Attributes	505
63.9 Adding the Percentage and Lock Screen Views	506
63.10 Designing the Widget Layouts	508
63.11 Launching the Live Activity.....	510
63.12 Updating the Live Activity.....	511
63.13 Stopping the Live Activity.....	511
63.14 Testing the App.....	512
63.15 Adding an Alert Notification.....	513
63.16 Understanding Background Updates	514
63.17 Summary	515
64. Adding a Refresh Button to a Live Activity.....	517
64.1 Adding Interactivity to Live Activities	517
64.2 Adding the App Intent.....	517
64.3 Setting a Stale Date.....	518
64.4 Detecting Stale Data.....	519
64.5 Testing the Live Activity Intent	520
64.6 Take the Knowledge Test.....	520
64.7 Summary	520
65. A Live Activity Push Notifications Tutorial.....	521
65.1 An Overview of Push Notifications.....	521
65.2 Registering an APNs Key	522
65.3 Enabling Push Notifications for the App	523
65.4 Enabling Frequent Updates.....	524
65.5 Requesting User Permission	524
65.6 Changing the Push Type	526
65.7 Obtaining a Push Token	527
65.8 Removing the Refresh Button	528
65.9 Take the Knowledge Test.....	529
65.10 Summary	529

66. Testing Live Activity Push Notifications.....	531
66.1 Using the Push Notifications Console.....	531
66.2 Configuring the Notification	532
66.3 Defining the Payload	533
66.4 Sending the Notification	534
66.5 Sending Push Notifications from the Command Line.....	534
66.6 Summary	536
67. Troubleshooting Live Activity Push Notifications	537
67.1 Push Notification Problems	537
67.2 Push Notification Delivery	537
67.3 Check the Payload Structure	538
67.4 Validating the Push and Authentication Tokens.....	538
67.5 Checking the Device Log.....	539
67.6 Take the Knowledge Test.....	539
67.7 Summary	539
68. Integrating UIViews with SwiftUI	541
68.1 SwiftUI and UIKit Integration.....	541
68.2 Integrating UIViews into SwiftUI	541
68.3 Adding a Coordinator	543
68.4 Handling UIKit Delegation and Data Sources	544
68.5 An Example Project	545
68.6 Wrapping the UIScrollView	545
68.7 Implementing the Coordinator	546
68.8 Using MyScrollView	547
68.9 Take the Knowledge Test.....	547
68.10 Summary	547
69. Integrating UIViewControllers with SwiftUI.....	549
69.1 UIViewControllers and SwiftUI.....	549
69.2 Creating the ViewControllerDemo project	549
69.3 Wrapping the UIImagePickerController	549
69.4 Designing the Content View.....	551
69.5 Completing MyImagePicker.....	552
69.6 Completing the Content View.....	554
69.7 Testing the App.....	554
69.8 Take the Knowledge Test.....	555
69.9 Summary	555
70. Integrating SwiftUI with UIKit.....	557
70.1 An Overview of the Hosting Controller	557
70.2 A UIHostingController Example Project.....	557
70.3 Adding the SwiftUI Content View	558
70.4 Preparing the Storyboard.....	559
70.5 Adding a Hosting Controller	560
70.6 Configuring the Segue Action	561
70.7 Embedding a Container View	564
70.8 Embedding SwiftUI in Code	566

Table of Contents

70.9 Take the Knowledge Test.....	567
70.10 Summary.....	567
71. Preparing and Submitting an iOS 18 Application to the App Store	569
71.1 Verifying the iOS Distribution Certificate.....	569
71.2 Adding App Icons	571
71.3 Assign the Project to a Team	572
71.4 Archiving the Application for Distribution.....	572
71.5 Configuring the Application in App Store Connect.....	573
71.6 Validating and Submitting the Application	574
71.7 Configuring and Submitting the App for Review	575
Index	577

3. Installing Xcode 16 and the iOS 18 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 16 and the iOS 18 SDK on macOS.

3.1 Identifying Your macOS Version

When developing with Xcode 16, a system running macOS Sonoma 14.5 or later or later is required. 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 Sonoma 14.5 or later is running, click on the *More Info...* button to open the Settings app and check for available operating system updates.

3.2 Installing Xcode 16 and the iOS 18 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.

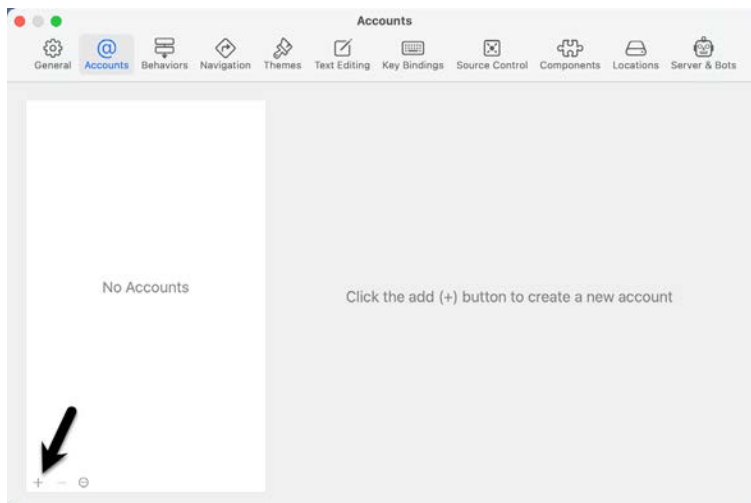


Figure 3-3

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:

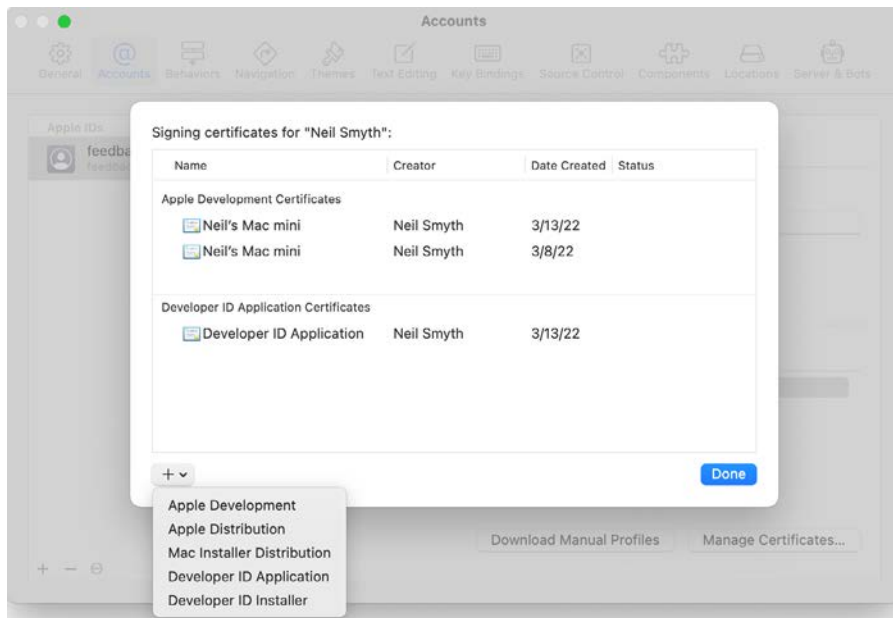


Figure 3-4

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 16 and the iOS 18 SDK running on macOS 13.5.2 (Ventura). Before beginning SwiftUI 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 16, we can now look at Xcode in more detail, starting with Playgrounds.

4. An Introduction to Xcode 16 Playgrounds

Before introducing the Swift programming language in the following chapters, it is first worth learning about a feature of Xcode known as *Playgrounds*. This is a feature of Xcode designed to make learning Swift and experimenting with the iOS SDK much easier. The concepts covered in this chapter can be put to use when experimenting with many of the introductory Swift code examples contained in the chapters that follow.

4.1 What is a Playground?

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

4.2 Creating a New Playground

To create a new Playground, start Xcode and select the *File -> New -> Playground...* menu option. Choose the iOS option on the resulting panel and select the Blank template.

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

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

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

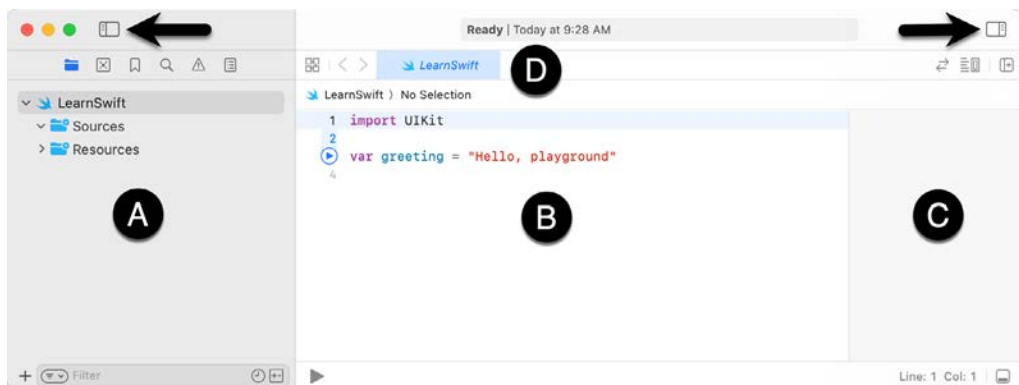


Figure 4-1

The panel on the left-hand side of the window (marked A in Figure 4-1) is the Navigator panel which provides access to the folders and files that make up the playground. To hide and show this panel, click on the button

indicated by the left-most arrow. The center panel (B) is the *playground editor* where the lines of Swift code are entered. The right-hand panel (C) is referred to as the *results panel* and is where the results of each Swift expression entered into the playground editor panel are displayed. The tab bar (D) will contain a tab for each file currently open within the playground editor. To switch to a different file, simply select the corresponding tab. To close an open file, hover the mouse pointer over the tab and click on the “X” button when it appears to the left of the file name.

The button marked by the right-most arrow in the above figure is used to hide and show the Inspectors panel (marked A in Figure 4-2 below) where a variety of properties relating to the playground may be configured. Clicking and dragging the bar (B) upward will display the Debug Area (C) where diagnostic output relating to the playground will appear when code is executed:

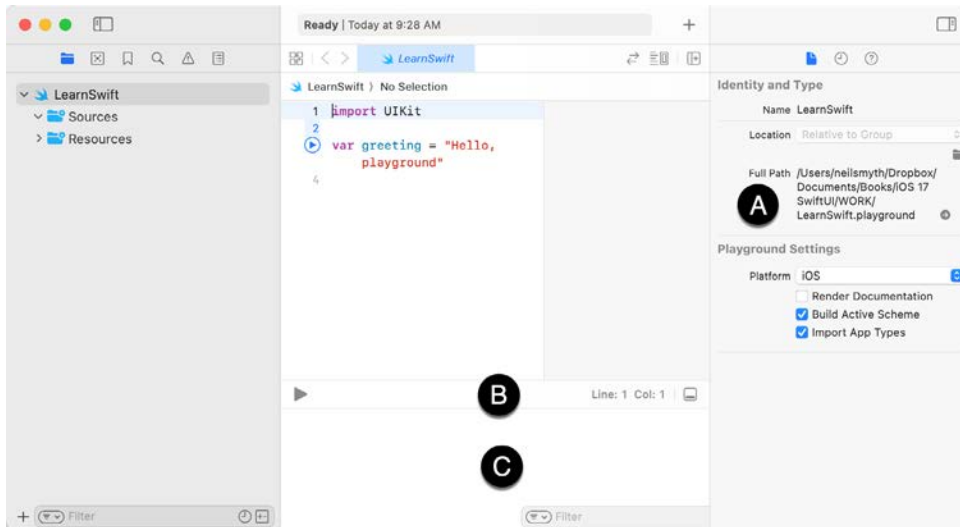


Figure 4-2

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

4.3 A Swift Playground Example

Perhaps the simplest of examples in any programming language (that at least does something tangible) is to write some code to output a single line of text. Swift is no exception to this rule so, within the playground window, begin adding another line of Swift code so that it reads as follows:

```
import UIKit
```

```
var greeting = "Hello, playground"
```

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

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

Note that although some extra code has been entered, nothing yet appears in the results panel. This is because the code has yet to be executed. One option to run the code is to click on the Execute Playground button located

in the bottom left-hand corner of the main panel as indicated by the arrow in Figure 4-3:

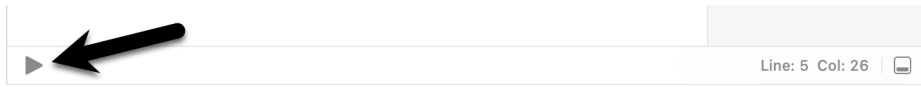


Figure 4-3

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

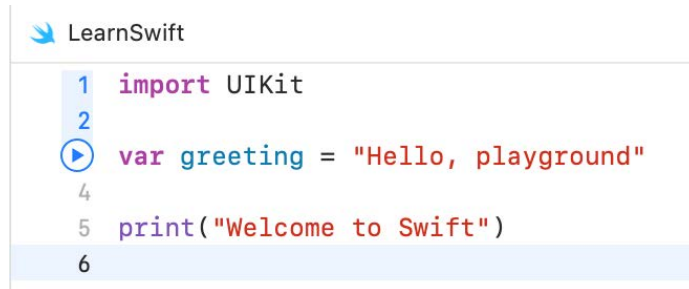


Figure 4-4

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

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

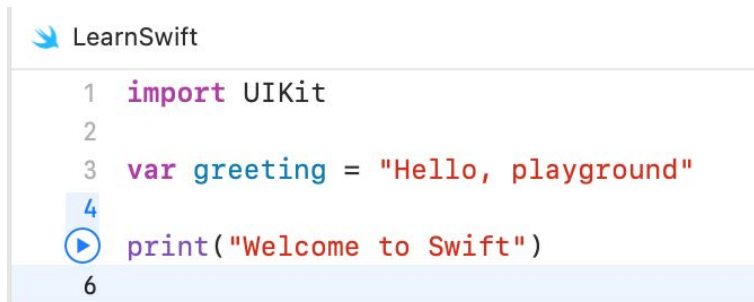


Figure 4-5

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

To reset the playground so that execution can be performed from the start of the code, simply click on the stop button as indicated in Figure 4-6:



Figure 4-6

Using this incremental execution technique, execute lines 1 through 3 and note that output now appears in the

results panel indicating that the variable has been initialized:

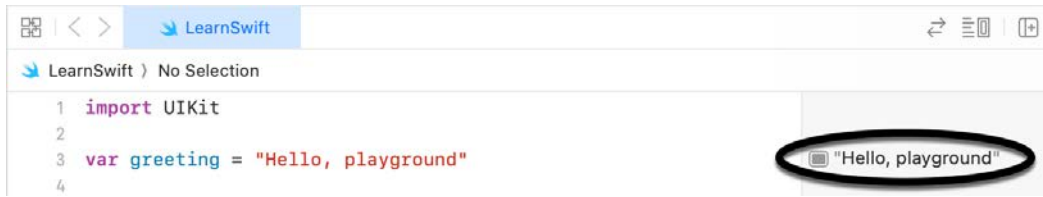


Figure 4-7

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

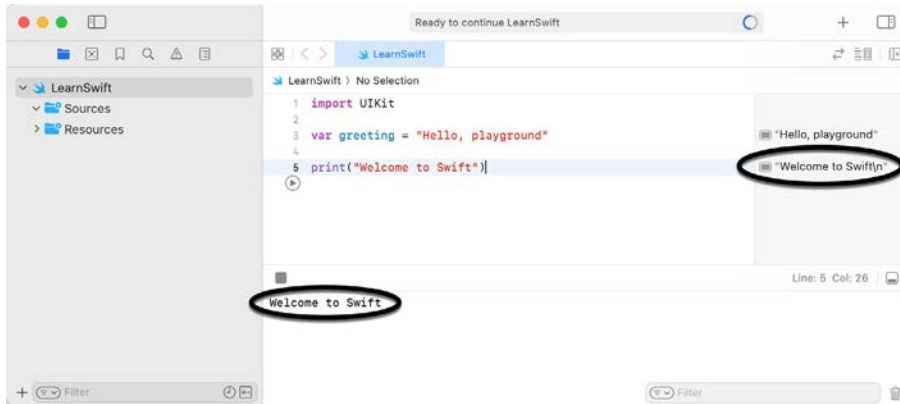


Figure 4-8

4.4 Viewing Results

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

```
var x = 10

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

This expression repeats a loop 20 times, performing arithmetic expressions on each iteration of the loop. Once the code has been entered into the editor, click on the run button positioned at line 13 to execute these new lines of code. The playground will execute the loop and display in the results panel the final value for each variable. More interesting information, however, may be obtained by hovering the mouse pointer over the results line so that an additional button appears, as shown in Figure 4-9:



Figure 4-9

Hovering over the output will display the *Quick Look* button on the far right which, when selected, will show a popup panel displaying the results, as shown in Figure 4-10:



Figure 4-10

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



Figure 4-11

4.5 Adding Rich Text Comments

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

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

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

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

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

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

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

As the comment content is added it is said to be displayed in *raw markup* format. To display in *rendered markup* format, either select the *Editor* -> *Show Rendered Markup* menu option, or enable the *Render Documentation* option located under *Playground Settings* in the Inspector panel (marked A in Figure 4-2). If the Inspector panel is not currently visible, click on the button indicated by the right-most arrow in Figure 4-1 to display it. Once rendered, the above rich text should appear, as illustrated in Figure 4-12:

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

Welcome to Playgrounds

This is your *first* playground which is intended to demonstrate:

- The use of **Quick Look**
- Placing results **in-line** with the code

Figure 4-12

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

https://developer.apple.com/library/content/documentation/Xcode/Reference/xcode_markup_formatting_ref/index.html

4.6 Working with Playground Pages

A playground can consist of multiple pages, with each page containing its own code, resources and, rich text comments. So far, the playground used in this chapter contains a single page. Add a page to the playground now by selecting the LearnSwift entry at the top of the Navigator panel, right-clicking, and selecting the *New Playground Page* menu option. If the Navigator panel is not currently visible, click the button indicated by the left-most arrow in Figure 4-1 above to display it. Note that two pages are now listed in the Navigator named “Untitled Page” and “Untitled Page 2”. Select and then click a second time on the “Untitled Page 2” entry so that the name becomes editable and change the name to *SwiftUI Example* as outlined in Figure 4-13:

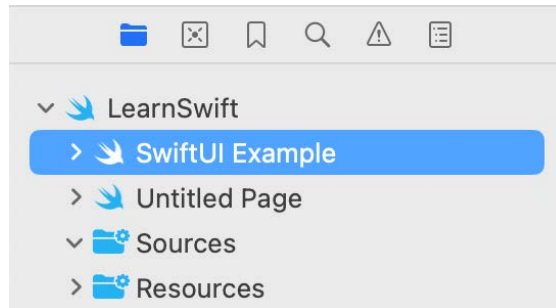


Figure 4-13

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

4.7 Working with SwiftUI and Live View in Playgrounds

In addition to allowing you to experiment with the Swift programming language, playgrounds may also be used to work with SwiftUI. Not only does this allow SwiftUI views to be prototyped, but when combined with the playground live view feature, it is also possible to run and interact with those views.

To try out SwiftUI and live view, begin by selecting the newly added SwiftUI Example page, deleting the current code lines, and modifying it to import both the SwiftUI and PlaygroundSupport frameworks:

```
import SwiftUI
import PlaygroundSupport
```

The PlaygroundSupport module provides several useful features for playgrounds including the ability to present a live view within the playground timeline.

Beneath the import statements, add the following code (rest assured, all of the techniques used in this example will be thoroughly explained in later chapters):

```
struct ExampleView: View {

    var body: some View {

        VStack {
            Rectangle()
                .fill(Color.blue)
                .frame(width: 200, height: 200)
            Button(action: {
                }) {
                Text("Rotate")
            }
        }
    }
}
```

```
        }  
    }  
    .padding(10)  
}  
}
```

This declaration creates a custom SwiftUI view named *ExampleView* consisting of a blue Rectangle view and a Button, both contained within a vertical stack (VStack).

The PlaygroundSupport module includes a class named PlaygroundPage which allows playground code to interact with the pages that make up a playground. This is achieved through a range of methods and properties of the class, one of which is the *current* property. This property, in turn, provides access to the current playground page. To execute the code within the playground, the *liveView* property of the current page needs to be set to our new container. To display the Live View panel, enable the Xcode Editor -> Live View menu option, as shown in Figure 4-14:

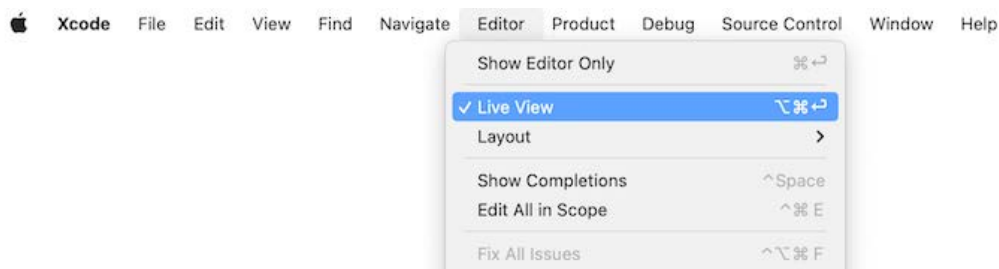


Figure 4-14

Once the live view panel is visible, add the code to assign the container to the live view of the current page as follows:

```
.  
.br/>  
VStack {  
    Rectangle()  
        .fill(Color.blue)  
        .frame(width: 200, height: 200)  
    Button(action: {  
        }) {  
        Text("Rotate")  
    }  
}  
    .padding(10)  
}
```

```
PlaygroundPage.current.setLiveView(ExampleView())  
    .padding(100)
```

With the changes made, click on the run button to start the live view. After a short delay, the view should appear, as shown in Figure 4-15 below:

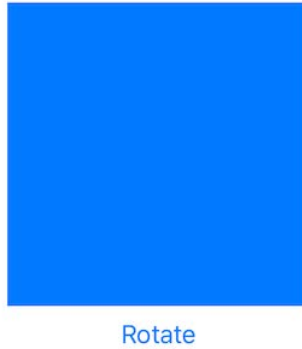


Figure 4-15

Since the button is not yet configured to do anything when clicked, it is difficult to see that the view is live. To see the live view in action, click on the stop button and modify the view declaration to rotate the blue square by 60° each time the button is clicked:

```
import SwiftUI
import PlaygroundSupport

struct ExampleView: View {

    @State private var rotation: Double = 0

    var body: some View {

        VStack {
            Rectangle()
                .fill(Color.blue)
                .frame(width: 200, height: 200)
                .rotationEffect(.degrees(rotation))
                .animation(.linear(duration: 2), value: rotation)
            Button(action: {
                rotation = (rotation < 360 ? rotation + 60 : 0)
            }) {
                Text("Rotate")
            }
        }
        .padding(10)
    }
}

PlaygroundPage.current.setLiveView(ExampleView()
    .padding(100))
```

Click the run button to launch the view in the live view and note that the square rotates each time the button is clicked.

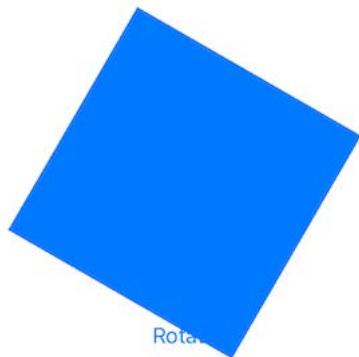


Figure 4-16

4.8 Summary

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

9. Swift Functions, Methods, and Closures

Swift functions, methods and closures are a vital part of writing well-structured and efficient code and provide a way to organize programs while avoiding code repetition. This chapter will look at how functions, methods, and closures are declared and used within Swift.

9.1 What is a Function?

A function is a named block of code that can be called upon to perform a specific task. It can be provided data on which to perform the task and is capable of returning results to the code that called it. For example, if a particular arithmetic calculation needs to be performed in a Swift program, the code to perform the arithmetic can be placed in a function. The function can be programmed to accept the values on which the arithmetic is to be performed (referred to as *parameters*) and to return the result of the calculation. At any point in the program code where the calculation is required the function is simply called, parameter values passed through as *arguments* and the result returned.

The terms *parameter* and *argument* are often used interchangeably when discussing functions. There is, however, a subtle difference. The values that a function is able to accept when it is called are referred to as *parameters*. At the point that the function is actually called and passed those values, however, they are referred to as *arguments*.

9.2 What is a Method?

A method is essentially a function that is associated with a particular class, structure, or enumeration. If, for example, you declare a function within a Swift class (a topic covered in detail in the chapter entitled “”), it is considered to be a method. Although the remainder of this chapter refers to functions, the same rules and behavior apply equally to methods unless otherwise stated.

9.3 How to Declare a Swift Function

A Swift function is declared using the following syntax:

```
func <function name> (<para name>: <para type>,  
                    <para name>: <para type>, ... ) -> <return type> {  
    // Function code  
}
```

This combination of function name, parameters and return type are referred to as the *function signature*. Explanations of the various fields of the function declaration are as follows:

- **func** – The prefix keyword used to notify the Swift compiler that this is a function.
- **<function name>** - The name assigned to the function. This is the name by which the function will be referenced when it is called from within the application code.
- **<para name>** - The name by which the parameter is to be referenced in the function code.
- **<para type>** - The type of the corresponding parameter.

- **<return type>** - The data type of the result returned by the function. If the function does not return a result then no return type is specified.
- **Function code** - The code of the function that does the work.

As an example, the following function takes no parameters, returns no result and simply displays a message:

```
func sayHello() {  
    print("Hello")  
}
```

The following sample function, on the other hand, takes an integer and a string as parameters and returns a string result:

```
func buildMessageFor(name: String, count: Int) -> String {  
    return("\(name), you are customer number \(count)")  
}
```

9.4 Implicit Returns from Single Expressions

In the previous example, the *return* statement was used to return the string value from within the *buildMessageFor()* function. It is worth noting that if a function contains a single expression (as was the case in this example), the return statement may be omitted. The *buildMessageFor()* method could, therefore, be rewritten as follows:

```
func buildMessageFor(name: String, count: Int) -> String {  
    "\(name), you are customer number \(count)"  
}
```

The return statement can only be omitted if the function contains a single expression. The following code, for example, will fail to compile since the function contains two expressions requiring the use of the return statement:

```
func buildMessageFor(name: String, count: Int) -> String {  
    let uppername = name.uppercased()  
    "\(uppername), you are customer number \(count)" // Invalid expression  
}
```

9.5 Calling a Swift Function

Once declared, functions are called using the following syntax:

```
<function name> (<arg1>, <arg2>, ... )
```

Each argument passed through to a function must match the parameters the function is configured to accept. For example, to call a function named *sayHello* that takes no parameters and returns no value, we would write the following code:

```
sayHello()
```

9.6 Handling Return Values

To call a function named *buildMessageFor* that takes two parameters and returns a result, on the other hand, we might write the following code:

```
let message = buildMessageFor(name: "John", count: 100)
```

In the above example, we have created a new variable called *message* and then used the assignment operator (=) to store the result returned by the function.

When developing in Swift, situations may arise where the result returned by a method or function call is not

used. When this is the case, the return value may be discarded by assigning it to ‘_’. For example:

```
_ = buildMessageFor(name: "John", count: 100)
```

9.7 Local and External Parameter Names

When the preceding example functions were declared, they were configured with parameters that were assigned names which, in turn, could be referenced within the body of the function code. When declared in this way, these names are referred to as *local parameter names*.

In addition to local names, function parameters may also have *external parameter names*. These are the names by which the parameter is referenced when the function is called. By default, function parameters are assigned the same local and external parameter names. Consider, for example, the previous call to the *buildMessageFor* method:

```
let message = buildMessageFor(name: "John", count: 100)
```

As declared, the function uses “name” and “count” as both the local and external parameter names.

The default external parameter names assigned to parameters may be removed by preceding the local parameter names with an underscore (‘_’) character as follows:

```
func buildMessageFor(_ name: String, _ count: Int) -> String {
    return("\(name), you are customer number \(count)")
}
```

With this change implemented, the function may now be called as follows:

```
let message = buildMessageFor("John", 100)
```

Alternatively, external parameter names can be added simply by declaring the external parameter name before the local parameter name within the function declaration. In the following code, for example, the external names of the first and second parameters have been set to “username” and “usercount” respectively:

```
func buildMessageFor(username name: String, usercount count: Int)
    -> String {
    return("\(name), you are customer number \(count)")
}
```

When declared in this way, the external parameter name must be referenced when calling the function:

```
let message = buildMessageFor(username: "John", usercount: 100)
```

Regardless of the fact that the external names are used to pass the arguments through when calling the function, the local names are still used to reference the parameters within the body of the function. It is important to also note that when calling a function using external parameter names for the arguments, those arguments must still be placed in the same order as that used when the function was declared.

9.8 Declaring Default Function Parameters

Swift provides the ability to designate a default parameter value to be used in the event that the value is not provided as an argument when the function is called. This simply involves assigning the default value to the parameter when the function is declared. Swift also provides a default external name based on the local parameter name for defaulted parameters (unless one is already provided) which must then be used when calling the function.

To see default parameters in action the *buildMessageFor* function will be modified so that the string “Customer” is used as a default in the event that a customer name is not passed through as an argument:

```
func buildMessageFor(_ name: String = "Customer", count: Int) -> String
```

```
{
    return ("\(name), you are customer number \(count)")
}
```

The function can now be called without passing through a name argument:

```
let message = buildMessageFor(count: 100)
print(message)
```

When executed, the above function call will generate output to the console panel which reads:

```
Customer, you are customer number 100
```

9.9 Returning Multiple Results from a Function

A function can return multiple result values by wrapping those results in a tuple. The following function takes as a parameter a measurement value in inches. The function converts this value into yards, centimeters and meters, returning all three results within a single tuple instance:

```
func sizeConverter(_ length: Float) -> (yards: Float, centimeters: Float,
                                         meters: Float) {

    let yards = length * 0.0277778
    let centimeters = length * 2.54
    let meters = length * 0.0254

    return (yards, centimeters, meters)
}
```

The return type for the function indicates that the function returns a tuple containing three values named yards, centimeters and meters respectively, all of which are of type Float:

```
-> (yards: Float, centimeters: Float, meters: Float)
```

Having performed the conversion, the function simply constructs the tuple instance and returns it.

Usage of this function might read as follows:

```
let lengthTuple = sizeConverter(20)

print(lengthTuple.yards)
print(lengthTuple.centimeters)
print(lengthTuple.meters)
```

9.10 Variable Numbers of Function Parameters

It is not always possible to know in advance the number of parameters a function will need to accept when it is called within application code. Swift handles this possibility through the use of *variadic parameters*. Variadic parameters are declared using three periods (...) to indicate that the function accepts zero or more parameters of a specified data type. Within the body of the function, the parameters are made available in the form of an array object. The following function, for example, takes as parameters a variable number of String values and then outputs them to the console panel:

```
func displayStrings(_ strings: String...)
{
    for string in strings {
        print(string)
    }
}
```



```

    }
}

displayStrings("one", "two", "three", "four")

```

9.11 Parameters as Variables

All parameters accepted by a function are treated as constants by default. This prevents changes being made to those parameter values within the function code. If changes to parameters need to be made within the function body, therefore, *shadow copies* of those parameters must be created. The following function, for example, is passed length and width parameters in inches, creates shadow variables of the two values and converts those parameters to centimeters before calculating and returning the area value:

```

func calculateArea(length: Float, width: Float) -> Float {

    var length = length
    var width = width

    length = length * 2.54
    width = width * 2.54
    return length * width
}

print(calculateArea(length: 10, width: 20))

```

9.12 Working with In-Out Parameters

When a variable is passed through as a parameter to a function, we now know that the parameter is treated as a constant within the body of that function. We also know that if we want to make changes to a parameter value we have to create a shadow copy as outlined in the above section. Since this is a copy, any changes made to the variable are not, by default, reflected in the original variable. Consider, for example, the following code:

```

var myValue = 10

func doubleValue (_ value: Int) -> Int {
    var value = value
    value += value
    return(value)
}

print("Before function call myValue = \(myValue)")

print("doubleValue call returns \(doubleValue(myValue))")

print("After function call myValue = \(myValue)")

```

The code begins by declaring a variable named *myValue* initialized with a value of 10. A new function is then declared which accepts a single integer parameter. Within the body of the function, a shadow copy of the value is created, doubled and returned.

The remaining lines of code display the value of the *myValue* variable before and after the function call is made. When executed, the following output will appear in the console:

Swift Functions, Methods, and Closures

Before function call myValue = 10

doubleValue call returns 20

After function call myValue = 10

Clearly, the function has made no change to the original myValue variable. This is to be expected since the mathematical operation was performed on a copy of the variable, not the *myValue* variable itself.

In order to make any changes made to a parameter persist after the function has returned, the parameter must be declared as an *in-out parameter* within the function declaration. To see this in action, modify the *doubleValue* function to include the *inout* keyword, and remove the creation of the shadow copy as follows:

```
func doubleValue (_ value: inout Int) -> Int {  
    var value = value  
    value += value  
    return(value)  
}
```

Finally, when calling the function, the *inout* parameter must now be prefixed with an & modifier:

```
print("doubleValue call returned \(doubleValue(&myValue))")
```

Having made these changes, a test run of the code should now generate output clearly indicating that the function modified the value assigned to the original *myValue* variable:

Before function call myValue = 10

doubleValue call returns 20

After function call myValue = 20

9.13 Functions as Parameters

An interesting feature of functions within Swift is that they can be treated as data types. It is perfectly valid, for example, to assign a function to a constant or variable, as illustrated in the declaration below:

```
func inchesToFeet (_ inches: Float) -> Float {  
    return inches * 0.0833333  
}
```

```
let toFeet = inchesToFeet
```

The above code declares a new function named *inchesToFeet* and subsequently assigns that function to a constant named *toFeet*. Having made this assignment, a call to the function may be made using the constant name instead of the original function name:

```
let result = toFeet(10)
```

On the surface this does not seem to be a particularly compelling feature. Since we could already call the function without assigning it to a constant or variable data type it does not seem that much has been gained.

The possibilities that this feature offers become more apparent when we consider that a function assigned to a constant or variable now has the capabilities of many other data types. In particular, a function can now be passed through as an argument to another function, or even returned as a result from a function.

Before we look at what is, essentially, the ability to plug one function into another, it is first necessary to explore the concept of function data types. The data type of a function is dictated by a combination of the parameters it accepts and the type of result it returns. In the above example, since the function accepts a floating-point parameter and returns a floating-point result, the function's data type conforms to the following:

```
(Float) -> Float
```

A function that accepts an Int and a Double as parameters and returns a String result, on the other hand, would have the following data type:

```
(Int, Double) -> String
```

In order to accept a function as a parameter, the receiving function simply declares the data type of the function it is able to accept.

For the purposes of an example, we will begin by declaring two unit conversion functions and assigning them to constants:

```
func inchesToFeet (_ inches: Float) -> Float {

    return inches * 0.08333333
}

func inchesToYards (_ inches: Float) -> Float {

    return inches * 0.0277778
}

let toFeet = inchesToFeet
let toYards = inchesToYards
```

The example now needs an additional function, the purpose of which is to perform a unit conversion and print the result in the console panel. This function needs to be as general purpose as possible, capable of performing a variety of different measurement unit conversions. In order to demonstrate functions as parameters, this new function will take as a parameter a function type that matches both the inchesToFeet and inchesToYards function data type together with a value to be converted. Since the data type of these functions is equivalent to (Float) -> Float, our general-purpose function can be written as follows:

```
func outputConversion(_ converterFunc: (Float) -> Float, value: Float) {

    let result = converterFunc(value)

    print("Result of conversion is \(result)")
}
```

When the outputConversion function is called, it will need to be passed a function matching the declared data type. That function will be called to perform the conversion and the result displayed in the console panel. This means that the same function can be called to convert inches to both feet and yards, simply by “plugging in” the appropriate converter function as a parameter. For example:

```
outputConversion(toYards, value: 10) // Convert to Yards
outputConversion(toFeet, value: 10) // Convert to Feet
```

Functions can also be returned as a data type simply by declaring the type of the function as the return type. The following function is configured to return either our toFeet or toYards function type (in other words a function which accepts and returns a Float value) based on the value of a Boolean parameter:

```
func decideFunction(_ feet: Bool) -> (Float) -> Float
{
    if feet {
        return toFeet
    }
}
```

```

    } else {
        return toYards
    }
}

```

9.14 Closure Expressions

Having covered the basics of functions in Swift it is now time to look at the concept of *closures* and *closure expressions*. Although these terms are often used interchangeably there are some key differences.

Closure expressions are self-contained blocks of code. The following code, for example, declares a closure expression and assigns it to a constant named `sayHello` and then calls the function via the constant reference:

```

let sayHello = { print("Hello") }
sayHello()

```

Closure expressions may also be configured to accept parameters and return results. The syntax for this is as follows:

```

{(<para name>: <para type>, <para name> <para type>, ... ) ->
    <return type> in
    // Closure expression code here
}

```

The following closure expression, for example, accepts two integer parameters and returns an integer result:

```

let multiply = {(_ val1: Int, _ val2: Int) -> Int in
    return val1 * val2
}
let result = multiply(10, 20)

```

Note that the syntax is similar to that used for declaring Swift functions with the exception that the closure expression does not have a name, the parameters and return type are included in the braces and the *in* keyword is used to indicate the start of the closure expression code. Functions are, in fact, just named closure expressions.

Before the introduction of structured concurrency in Swift 5.5 (a topic covered in detail in the chapter entitled “*An Overview of Swift Structured Concurrency*”), closure expressions were often (and still are) used when declaring completion handlers for asynchronous method calls. In other words, when developing iOS applications, it will often be necessary to make calls to the operating system where the requested task is performed in the background allowing the application to continue with other tasks. Typically, in such a scenario, the system will notify the application of the completion of the task and return any results by calling the completion handler that was declared when the method was called. Frequently the code for the completion handler will be implemented in the form of a closure expression. Consider the following code example:

```

eventstore.requestAccess(to: .reminder, completion: {(granted: Bool,
    error: Error?) -> Void in
    if !granted {
        print(error!.localizedDescription)
    }
})

```

When the tasks performed by the `requestAccess(to:)` method call are complete it will execute the closure expression declared as the `completion:` parameter. The completion handler is required by the method to accept a Boolean value and an Error object as parameters and return no results, hence the following declaration:

```

{(granted: Bool, error: Error?) -> Void in

```

In actual fact, the Swift compiler already knows about the parameter and return value requirements for the completion handler for this method call and is able to infer this information without it being declared in the closure expression. This allows a simpler version of the closure expression declaration to be written:

```
eventstore.requestAccess(to: .reminder, completion: {(granted, error) in
    if !granted {
        print(error!.localizedDescription)
    }
})
```

9.15 Shorthand Argument Names

A useful technique for simplifying closures involves using *shorthand argument names*. This allows the parameter names and “in” keyword to be omitted from the declaration and the arguments to be referenced as \$0, \$1, \$2 etc.

Consider, for example, a closure expression designed to concatenate two strings:

```
let join = { (string1: String, string2: String) -> String in
    string1 + string2
}
```

Using shorthand argument names, this declaration can be simplified as follows:

```
let join: (String, String) -> String = {
    $0 + $1
}
```

Note that the type declaration $((String, String) \rightarrow String)$ has been moved to the left of the assignment operator since the closure expression no longer defines the argument or return types.

9.16 Closures in Swift

A *closure* in computer science terminology generally refers to the combination of a self-contained block of code (for example a function or closure expression) and one or more variables that exist in the context surrounding that code block. Consider, for example the following Swift function:

```
func functionA() -> () -> Int {

    var counter = 0

    func functionB() -> Int {
        return counter + 10
    }
    return functionB
}

let myClosure = functionA()
let result = myClosure()
```

In the above code, *functionA* returns a function named *functionB*. In actual fact *functionA* is returning a closure since *functionB* relies on the *counter* variable which is declared outside the *functionB*’s local scope. In other words, *functionB* is said to have *captured* or *closed over* (hence the term closure) the *counter* variable and, as such, is considered a closure in the traditional computer science definition of the word.

To a large extent, and particularly as it relates to Swift, the terms *closure* and *closure expression* have started to be

used interchangeably. The key point to remember, however, is that both are supported in Swift.

9.17 Take the Knowledge Test



Click the link below or scan the QR code to test your knowledge and understanding of the Swift functions, methods, and closures:

<https://www.answerstopia.com/uc2w>



9.18 Summary

Functions, closures and closure expressions are self-contained blocks of code that can be called upon to perform a specific task, and provide a mechanism for structuring code and promoting reuse. This chapter has introduced the concepts of functions and closures in terms of declaration and implementation.

18. SwiftUI Architecture

A completed SwiftUI app is constructed from multiple components that are assembled hierarchically. Before embarking on creating even the most basic SwiftUI projects, it is helpful to understand how SwiftUI apps are structured. With this goal in mind, this chapter will introduce the key elements of SwiftUI app architecture, emphasizing App, Scene, and View elements.

18.1 SwiftUI App Hierarchy

When considering the structure of a SwiftUI application, it helps to view a typical hierarchy visually. Figure 18-1, for example, illustrates the hierarchy of a simple SwiftUI app:

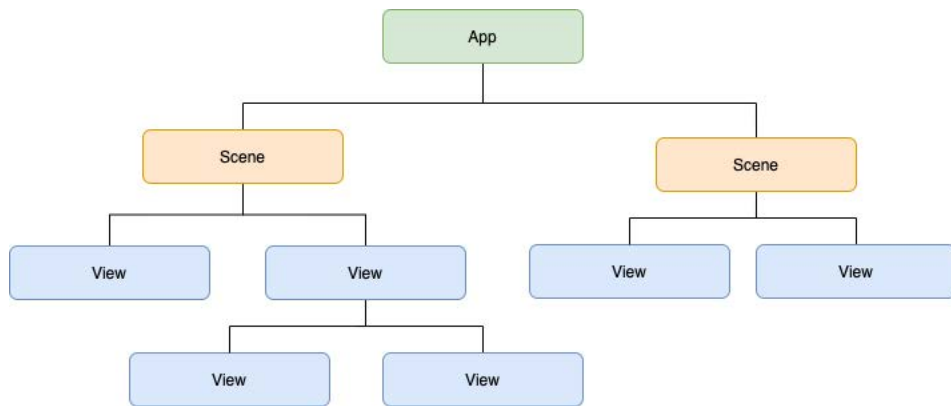


Figure 18-1

Before continuing, it is essential to distinguish the difference between the term “app” and the “App” element outlined in the above figure. The software applications that we install and run on our mobile devices have come to be referred to as “apps”. In this chapter, reference will be made both to these apps and the App element in the above figure. To avoid confusion, we will use “application” to refer to the completed, installed, and running app while referring to the App element as “App”. The remainder of the book will revert to using the more common “app” when discussing applications.

18.2 App

The App object is the top-level element within the structure of a SwiftUI application and is responsible for handling the launching and lifecycle of each running instance of the application.

The App element is also responsible for managing the various Scenes that make up the application’s user interface. An application will include only one App instance.

18.3 Scenes

Each SwiftUI application will contain one or more scenes. A scene represents a section or region of the application’s user interface. On iOS and watchOS, a scene will typically take the form of a window that takes up the entire device screen. On the other hand, SwiftUI applications running on macOS and iPadOS will likely be comprised of multiple scenes. Different scenes might, for example, contain context-specific layouts to be displayed when the user selects tabs within a dialog or to design applications that consist of multiple windows.

SwiftUI includes some pre-built primitive scene types that can be used when designing applications, the most common being WindowGroup and DocumentGroup. It is also possible to group scenes together to create your own custom scenes.

18.4 Views

Views are the basic building blocks that make up the visual elements of the user interface, such as buttons, labels, and text fields. Each scene will contain a hierarchy of the views that make up a section of the application's user interface. Views can either be individual visual elements, such as text views or buttons or take the form of containers that manage other views. The Vertical Stack view, for example, is designed to display child views in a vertical layout. In addition to the Views provided with SwiftUI, you will also create custom views when developing SwiftUI applications. These custom views will comprise groups of other views together with customizations to the appearance and behavior of those views to meet the requirements of the application's user interface.

Figure 18-2, for example, illustrates a scene containing a simple view hierarchy consisting of a Vertical Stack containing a Button and TextView combination:

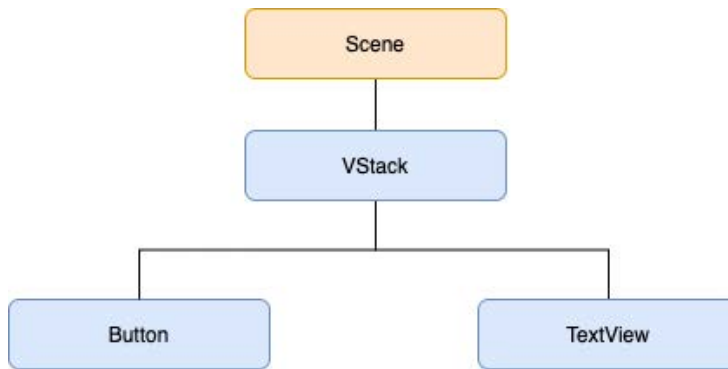


Figure 18-2

18.5 Take the Knowledge Test



Click the link below or scan the QR code to test your knowledge and understanding of SwiftUI's architecture:

<https://www.answertopia.com/z871>



18.6 Summary

SwiftUI applications are constructed hierarchically. At the top of the hierarchy is the App instance, which is responsible for the launching and lifecycle of the application. One or more child Scene instances contain hierarchies of the View instances that make up the application's user interface. These scenes can either be derived from one of the SwiftUI primitive Scene types, such as WindowGroup, or custom-built.

On iOS or watchOS, an application typically contains a single scene, which takes the form of a window occupying the entire display. However, on a macOS or iPadOS system, an application may comprise multiple scene instances, often represented by separate windows that can be displayed simultaneously or grouped together in a tabbed interface.

23. A SwiftUI Example Tutorial

Now that some of the fundamentals of SwiftUI development have been covered, this chapter will begin to put this theory into practice by building an example SwiftUI-based project.

This chapter aims to demonstrate using Xcode to design a simple interactive user interface using views, modifiers, state variables, and some basic animation effects. This tutorial will use various techniques to add and modify views. While this may appear inconsistent, the objective is to gain familiarity with the options available.

23.1 Creating the Example Project

Start Xcode and select the option to create a new project. Then, on the template selection screen, make sure Multiplatform is selected and choose the App option, as shown in Figure 23-1, before proceeding to the next screen:

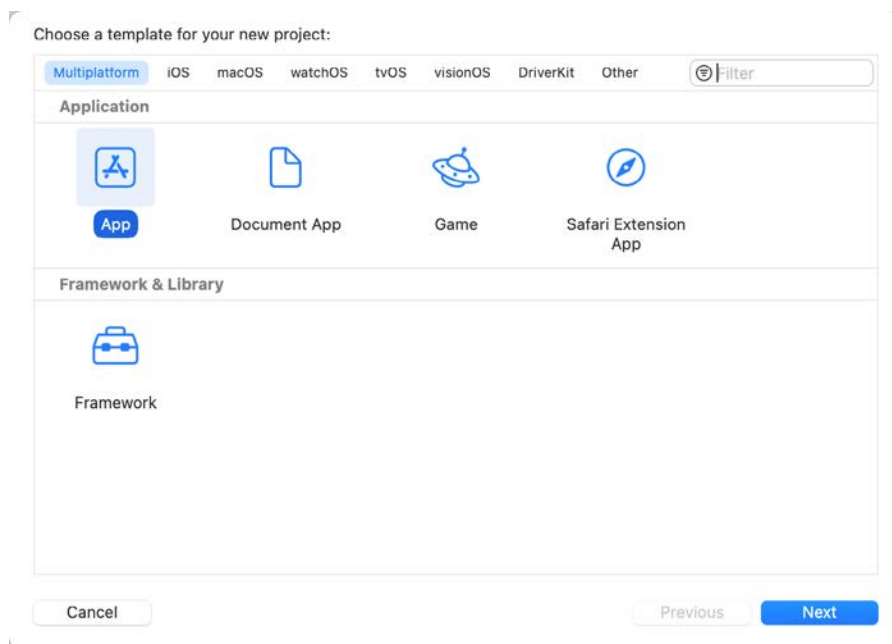


Figure 23-1

On the project options screen, name the project *SwiftUIDemo* and set the Testing System menu to “None” before clicking Next to proceed to the final screen. Choose a suitable filesystem location for the project and click the Create button.

23.2 Reviewing the Project

Once the project has been created, it will contain the *SwiftUIDemoApp.swift* file along with a SwiftUI View file named *ContentView.swift*, which should have loaded into the editor and preview canvas ready for modification (if it has not loaded, select it in the project navigator panel). Next, from the target device menu (Figure 23-2), select an iPhone 15 simulator:

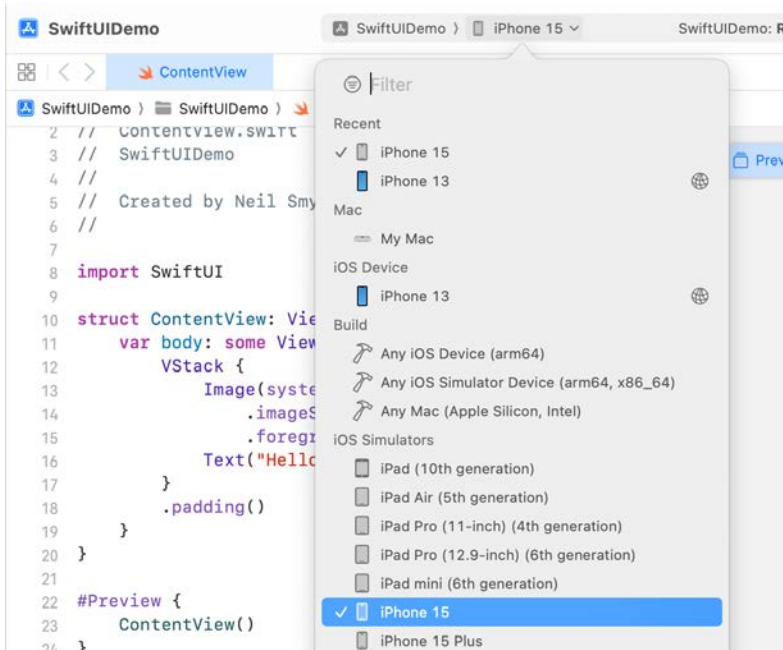


Figure 23-2

If the preview canvas is in the paused state, click on the Resume button to build the project and display the preview:

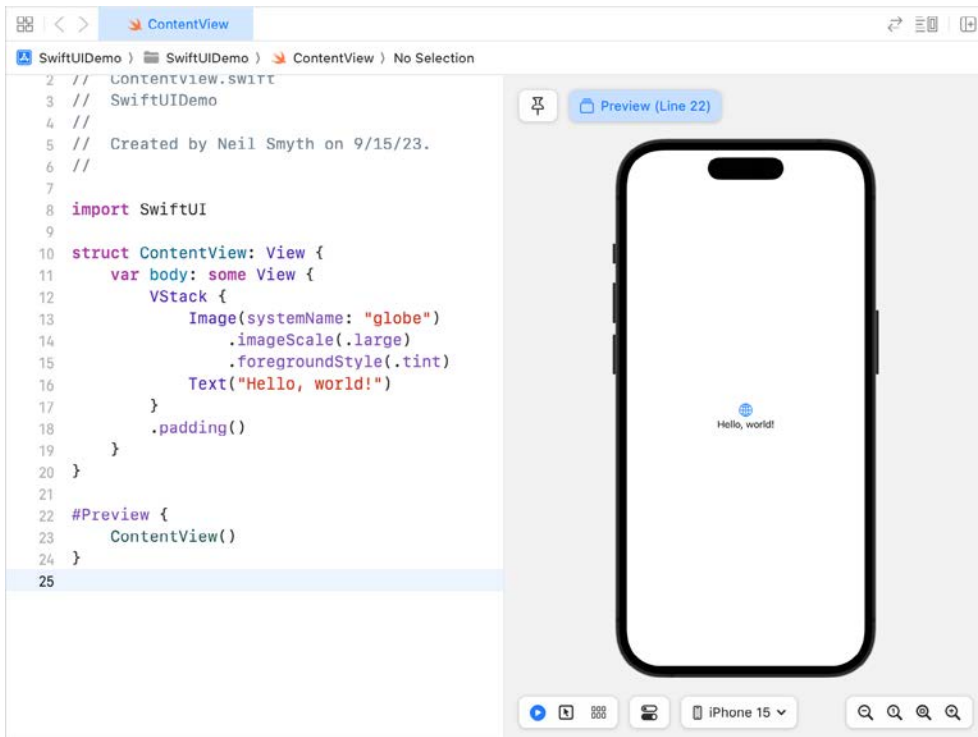


Figure 23-3

23.3 Modifying the Layout

The view body currently consists of a vertical stack layout (VStack) containing an Image and a Text view. Although we could reuse some of the existing layout for our example, we will learn more by deleting the current views and starting over. Within the Code Editor, delete the existing views from the ContentView body:

```
import SwiftUI

struct ContentView: View {
    var body: some View {
        VStack {
            Image(systemName: "globe")
                .imageScale(.large)
                .foregroundColor(.accentColor)
            Text("Hello, world!")
        }
        .padding()
    }
}
```

Next, add a Text view to the layout as follows:

```
struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
    }
}
```

Right-click on the Text view entry within the code editor, and select the Embed in VStack option from the resulting menu:

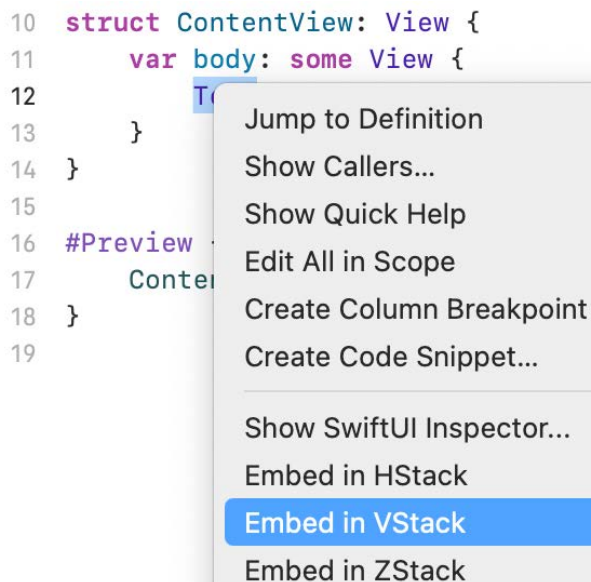


Figure 23-4

A SwiftUI Example Tutorial

Once the Text view has been embedded into the VStack, the declaration will read as follows:

```
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Hello, world!")
        }
    }
}
```

23.4 Adding a Slider View to the Stack

The next item to be added to the layout is a Slider view. Display the Library panel by clicking on the '+' button highlighted in Figure 23-5, locating the Slider in the View list, and dragging it into position beneath the Text view in the editor. Ensure that the Slider view will be inserted into the existing stack before dropping the view into place:

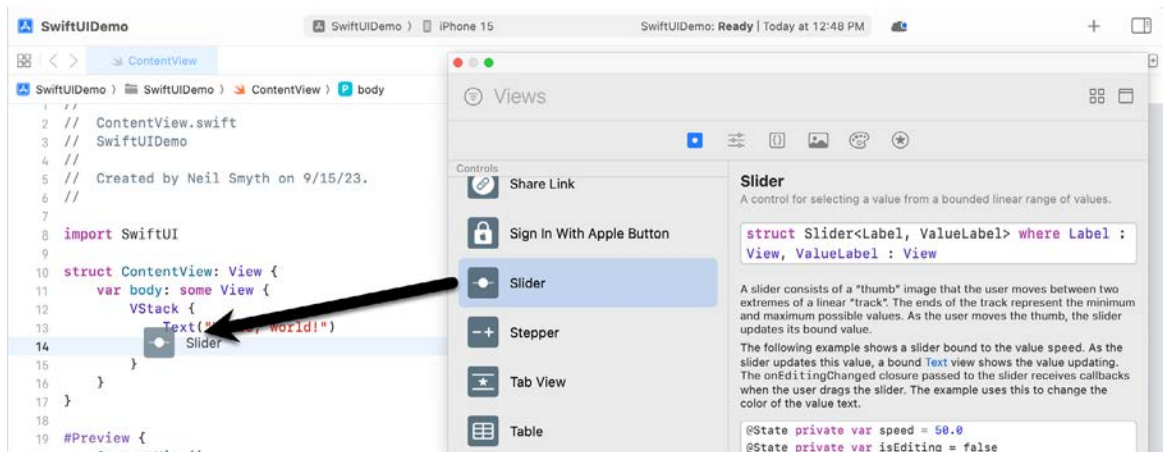


Figure 23-5

Once the slider has been dropped into place, the view implementation should read as follows:

```
struct ContentView: View {
    var body: some View {
        VStack {
            VStack {
                Text("Hello, world!")
                Slider(value: Value)
            }
        }
    }
}
```

23.5 Adding a State Property

The Slider will control the degree to which the Text view will be rotated. As such, a binding must be established between the Slider view and a state property into which the current rotation angle will be stored. Within the code editor, declare this property and configure the Slider to use a range between 0 and 360 in increments of 0.1:

```
struct ContentView: View {
```

```

@State private var rotation: Double = 0

var body: some View {
    VStack {
        VStack {
            Text("Hello, world!")
            Slider(value: $rotation, in: 0 ... 360, step: 0.1)
        }
    }
}

```

Note that since we declare a binding between the Slider view and the rotation state property, it is prefixed by a '\$' character.

23.6 Adding Modifiers to the Text View

The next step is to add some modifiers to the Text view to change the font and adopt the rotation value stored by the Slider view. Begin by displaying the Library panel, switch to the modifier list, and drag and drop a font modifier onto the Text view entry in the code editor:

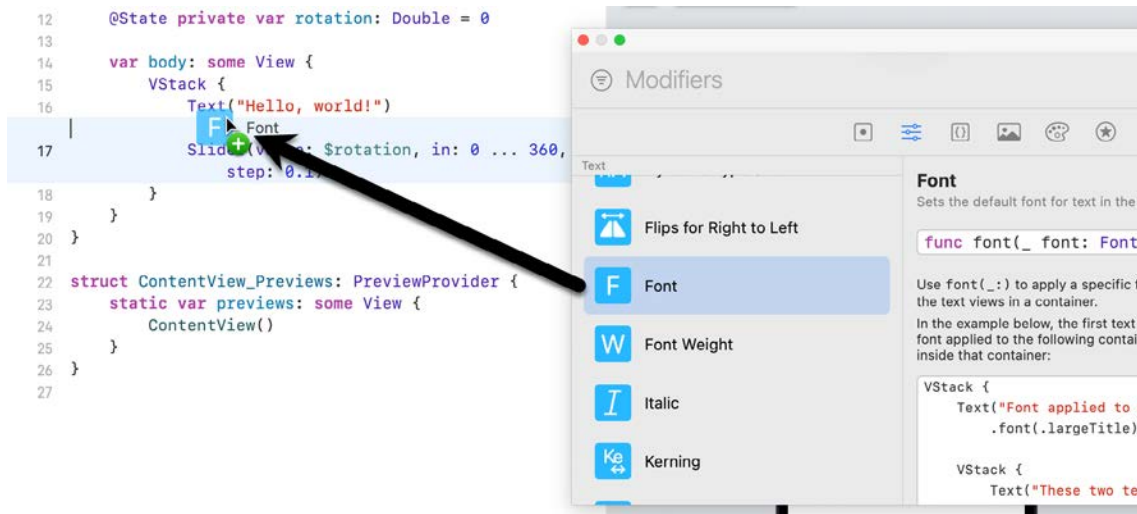


Figure 23-6

Select the modifier line in the editor, refer to the Attributes inspector panel, and change the font property from Title to Large Title, as shown in Figure 23-7:

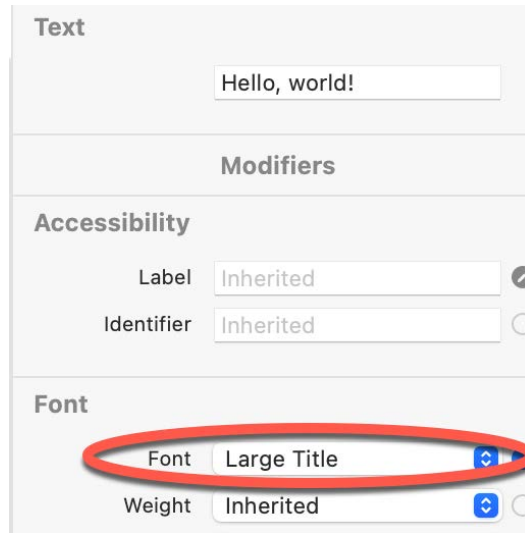


Figure 23-7

Note that the modifier added above does not change the font weight. Since modifiers may also be added to a view from within the Attributes inspector, take this opportunity to change the setting of the Weight menu from Inherited to Heavy.

On completion of these steps, the View body should read as follows:

```
var body: some View {
    VStack {
        VStack {
            Text("Hello, world!")
                .font(.largeTitle)
                .fontWeight(.heavy)
            Slider(value: $rotation, in: 0 ... 360, step: 0.1)
        }
    }
}
```

23.7 Adding Rotation and Animation

The next step is to add the rotation and animation effects to the Text view using the value stored by the Slider (animation is covered in greater detail in the “*SwiftUI Animation and Transitions*” chapter). This can be implemented using a modifier as follows:

```
Text("Hello, world!")
    .font(.largeTitle)
    .fontWeight(.heavy)
    .rotationEffect(.degrees(rotation))
```

Note that since we are simply reading the value assigned to the rotation state property, instead of establishing a binding, the property name is not prefixed with the ‘\$’ sign notation.

Click on the Live button (indicated by the arrow in Figure 23-8), wait for the code to compile, then use the slider to rotate the Text view:

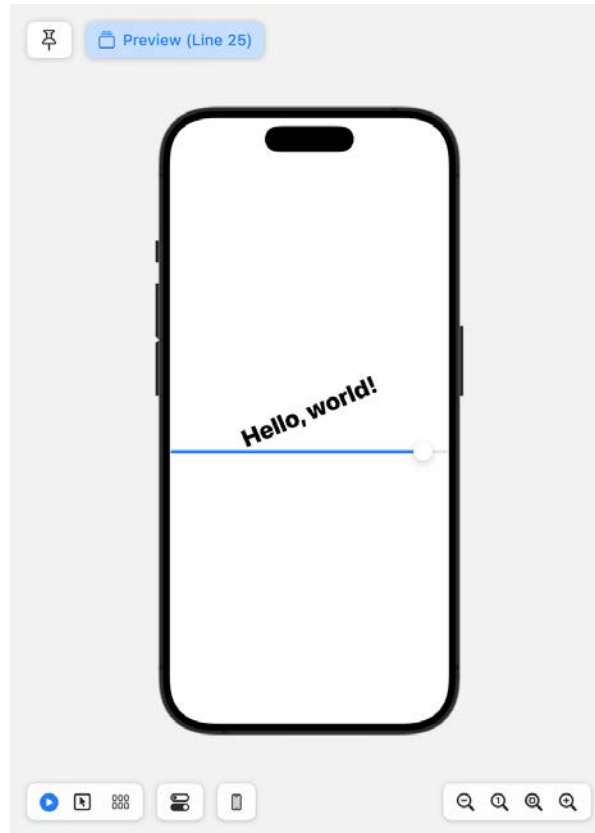


Figure 23-8

Next, add an animation modifier to the Text view to animate the rotation over 5 seconds using the Ease In Out effect:

```
Text("Hello, world!")
    .font(.largeTitle)
    .fontWeight(.heavy)
    .rotationEffect(.degrees(rotation))
    .animation(.easeInOut(duration: 5), value: rotation)
```

Use the slider once again to rotate the text, and note that rotation is now smoothly animated.

23.8 Adding a TextField to the Stack

In addition to supporting text rotation, the app will allow custom text to be entered and displayed on the Text view. This will require the addition of a TextField view to the project. To achieve this, either directly edit the View structure or use the Library panel to add a TextField so that the structure reads as follows (also note the addition of a state property in which to store the custom text string and the change to the Text view to use this property):

```
struct ContentView: View {

    @State private var rotation: Double = 0
    @State private var text: String = "Welcome to SwiftUI"
```

A SwiftUI Example Tutorial

```
var body: some View {
    VStack {
        VStack {
            Text(text)
                .font(.largeTitle)
                .fontWeight(.heavy)
                .rotationEffect(.degrees(rotation))
                .animation(.easeInOut(duration: 5))

            Slider(value: $rotation, in: 0 ... 360, step: 0.1)

            TextField("Enter text here", text: $text)
                .textFieldStyle(RoundedBorderTextFieldStyle())
        }
    }
}
```

When the user enters text into the TextField view, that text will be stored in the *text* state property and will automatically appear on the Text view via the binding.

Return to the preview canvas and ensure that the changes work as expected.

23.9 Adding a Color Picker

A Picker view is the final view to be added to the stack before we tidy up the layout. This view will allow the user to choose the foreground color of the Text view from a range of color options. Begin by adding some arrays of color names and Color objects, together with a state property to hold the current array index value as follows:

```
import SwiftUI

struct ContentView: View {

    var colors: [Color] = [.black, .red, .green, .blue]
    var colornames = ["Black", "Red", "Green", "Blue"]

    @State private var colorIndex = 0
    @State private var rotation: Double = 0
    @State private var text: String = "Welcome to SwiftUI"
```

With these variables configured, display the Library panel, locate the Picker in the Views screen, and drag and drop it beneath the TextField view in the code editor to embed it in the existing VStack layout. Once added, the view entry will read as follows:

```
Picker(selection: .constant(1), label: Text("Picker")) {
    Text("1").tag(1)
    Text("2").tag(2)
}
```

The Picker view needs to be configured to store the current selection in the *colorIndex* state property and to display an option for each color name in the *colorNames* array. In addition, to make the Picker more visually

appealing, we will change the background color for each Text view to the corresponding color in the *colors* array.

To iterate through the *colorNames* array, the code will use the SwiftUI *ForEach* structure. At first glance, *ForEach* looks like just another Swift programming language control flow statement. In fact, *ForEach* differs greatly from the Swift *forEach()* array method outlined earlier in the book.

ForEach is a SwiftUI view structure designed to generate multiple views by looping through a data set such as an array or range. We may also configure the Picker view to display the color choices in various ways. For this project, we must select the *WheelPickerStyle* (*.wheel*) style via the *pickerStyle()* modifier. Within the editor, modify the Picker view declaration so that it reads as follows:

```
Picker(selection: $colorIndex, label: Text("Color")) {
    ForEach (0 ..< colornames.count, id:\.self) { color in
        Text(colornames[color])
            .foregroundColor(colors[color])
    }
}
.pickerStyle(.wheel)
```

In the above implementation, *ForEach* is used to loop through the elements of the *colornames* array, generating a Text view for each color and setting the displayed text and background color on each view accordingly.

The *ForEach* loop in the above example is contained within a closure expression. As outlined in the “*Swift Functions, Methods, and Closures*” chapter, this expression can be simplified using *shorthand argument names*. Using this technique, modify the Picker declaration so that it reads as follows:

```
Picker(selection: $colorIndex, label: Text("Color")) {
    ForEach (0 ..< colornames.count, id:\.self) { color in
        Text(colornames[$0])
            .foregroundColor(colors[$0])
    }
}
.pickerStyle(.wheel)
```

Remaining in the code editor, locate the Text view and add a foreground color modifier to set the foreground color based on the current Picker selection value:

```
Text(text)
    .font(.largeTitle)
    .fontWeight(.heavy)
    .rotationEffect(.degrees(rotation))
    .animation(.easeInOut(duration: 5), value: rotation)
    .foregroundColor(colors[colorIndex])
```

Test the app in the preview canvas and confirm that the Picker view appears with all of the color names using the corresponding foreground color and that color selections are reflected in the Text view.

23.10 Tidying the Layout

Until this point, the focus of this tutorial has been on the appearance and functionality of the individual views. Aside from making sure the views are stacked vertically, however, no attention has been paid to the overall appearance of the layout. At this point, the layout should resemble that shown in Figure 23-9:



Figure 23-9

The first improvement needed is to add some space around the Slider, TextField, and Picker views so that they are not so close to the edge of the device display. To implement this, we will add some padding modifiers to the views:

```
Slider(value: $rotation, in: 0 ... 360, step: 0.1)
    .padding()

TextField("Enter text here", text: $text)
    .textFieldStyle(RoundedBorderTextFieldStyle())
    .padding()

Picker(selection: $colorIndex, label: Text("Color")) {
    ForEach (0 ..< colornames.count, id:\.self) {
        Text(colornames[$0])
            .foregroundColor(colors[$0])
    }
}
```

```
.pickerStyle(.wheel)
.padding()
```

Next, the layout would look better if the Views were evenly spaced. One way to implement this is to add some Spacer views before and after the Text view:

```
.
.
VStack {
    Spacer()
    Text(text)
        .font(.largeTitle)
        .fontWeight(.heavy)
        .rotationEffect(.degrees(rotation))
        .animation(.easeInOut(duration: 5), value: rotation)
        .foregroundColor(colors[colorIndex])
    Spacer()
    Slider(value: $rotation, in: 0 ... 360, step: 0.1)
        .padding()
.
.
```

The Spacer view provides a flexible space between views that will expand and contract based on the requirements of the layout. If a Spacer is contained in a stack, it will resize along the stack axis. A Spacer view can resize horizontally and vertically when used outside a stack container.

To make the separation between the Text view and the Slider more obvious, also add a Divider view to the layout:

```
.
.
VStack {
    Spacer()
    Text(text)
        .font(.largeTitle)
        .fontWeight(.heavy)
        .rotationEffect(.degrees(rotation))
        .animation(.easeInOut(duration: 5), value: rotation)
        .foregroundColor(colors[colorIndex])
    Spacer()
    Divider()
.
.
```

The Divider view draws a line to indicate the separation between two views in a stack container.

With these changes made, the layout should now appear in the preview canvas, as shown in Figure 23-10:



Figure 23-10

23.11 Take the Knowledge Test



Click the link below or scan the QR code to test your knowledge and understanding so far of SwiftUI:

<https://www.answertopia.com/ukob>



23.12 Summary

The goal of this chapter has been to put into practice some of the theory covered in the previous chapters through the creation of an example app project. In particular, the tutorial used various techniques for adding views to a layout and using modifiers and state property bindings. The chapter also introduced the Spacer and Divider views and used the ForEach structure to generate views from a data array dynamically.

31. SwiftUI Lists and Navigation

The SwiftUI List view provides a way to present information to the user as a vertical list of rows. Often the items within a list will navigate to another area of the app when tapped by the user. Behavior of this type is implemented in SwiftUI using the `NavigationStack` and `NavLink` components.

The List view can present both static and dynamic data and may also be extended to allow for the addition, removal, and reordering of row entries.

This chapter will provide an overview of the List View used in conjunction with `NavigationStack` and `NavLink` in preparation for the tutorial in the next chapter entitled “*A SwiftUI List and NavigationStack Tutorial*”.

31.1 SwiftUI Lists

The SwiftUI List control provides similar functionality to the UIKit `TableView` class in that it presents information in a vertical list of rows with each row containing one or more views contained within a cell. Consider, for example, the following List implementation:

```
struct ContentView: View {
    var body: some View {

        List {
            Text("Wash the car")
            Text("Vacuum house")
            Text("Pick up kids from school bus @ 3pm")
            Text("Auction the kids on eBay")
            Text("Order Pizza for dinner")
        }
    }
}
```

When displayed in the preview, the above list will appear, as shown in Figure 31-1:

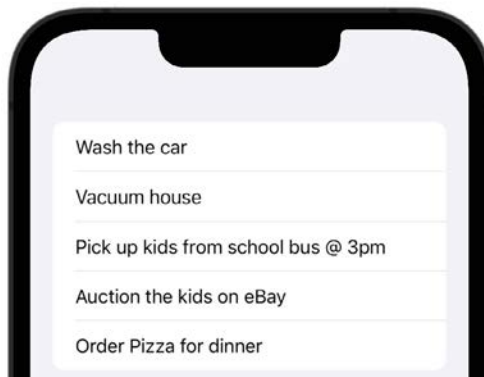


Figure 31-1

A list cell is not restricted to containing a single component. In fact, any combination of components can be displayed in a list cell. Each row of the list in the following example consists of an image and text component within an HStack:

```
List {
    HStack {
        Image(systemName: "trash.circle.fill")
        Text("Take out the trash")
    }
    HStack {
        Image(systemName: "person.2.fill")
        Text("Pick up the kids")
    }
    HStack {
        Image(systemName: "car.fill")
        Text("Wash the car")
    }
}
```

The preview canvas for the above view structure will appear, as shown in Figure 31-2 below:

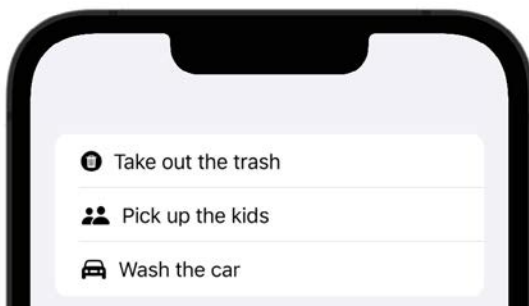


Figure 31-2

31.2 Modifying List Separators and Rows

The lines used by the List view to separate rows can be hidden by applying the `listRowSeparator()` modifier to the cell content views. The `listRowSeparatorTint()` modifier, on the other hand, can be used to change the color of the lines. It is even possible to assign a view to appear as the background of a row using the `listRowBackground()` modifier. The following code, for example, hides the first separator, changes the tint of the next two separators, and displays a background image on the final row:

```
List {
    Text("Wash the car")
        .listRowSeparator(.hidden)
    Text("Pick up kids from school bus @ 3pm")
        .listRowSeparatorTint(.green)
    Text("Auction the kids on eBay")
        .listRowSeparatorTint(.red)
    Text("Order Pizza for dinner")
        .listRowBackground(Image("MyBackgroundImage"))
}
```

The above examples demonstrate the use of a List to display static information. To display a dynamic list of items a few additional steps are required.

31.3 SwiftUI Dynamic Lists

A list is considered to be dynamic when it contains a set of items that can change over time. In other words, items can be added, edited, and deleted and the list updates dynamically to reflect those changes.

To support a list of this type, each data element to be displayed must be contained within a class or structure that conforms to the Identifiable protocol. The Identifiable protocol requires that the instance contain a property named *id* which can be used to uniquely identify each item in the list. The *id* property can be any Swift or custom type that conforms to the Hashable protocol which includes the String, Int, and UUID types in addition to several hundred other standard Swift types. If you opt to use UUID as the type for the property, the *UUID()* method can be used to automatically generate a unique ID for each list item.

The following code implements a simple structure for the To Do list example that conforms to the Identifiable protocol. In this case, the *id* is generated automatically via a call to *UUID()*:

```
struct ToDoItem : Identifiable {
    var id = UUID()
    var task: String
    var imageName: String
}
```

For example, an array of *ToDoItem* objects can be used to simulate the supply of data to the list which can now be implemented as follows:

```
struct ContentView: View {

    @State var listData: [ToDoItem] = [
        ToDoItem(task: "Take out trash", imageName: "trash.circle.fill"),
        ToDoItem(task: "Pick up the kids", imageName: "person.2.fill"),
        ToDoItem(task: "Wash the car", imageName: "car.fill")
    ]

    var body: some View {

        List(listData) { item in
            HStack {
                Image(systemName: item.imageName)
                Text(item.task)
            }
        }
    }
}
```

Now the list no longer needs a view for each cell. Instead, the list iterates through the data array and reuses the same *HStack* declaration, simply plugging in the appropriate data for each array element.

In situations where dynamic and static content needs to be displayed together within a list, the *ForEach* statement

can be used within the body of the list to iterate through the dynamic data while also declaring static entries. The following example includes a static toggle button together with a ForEach loop for the dynamic content:

```
struct ContentView: View {  
  
    @State private var toggleStatus = true  
    .  
    .  
    var body: some View {  
  
        List {  
            Toggle(isOn: $toggleStatus) {  
                Text("Allow Notifications")  
            }  
  
            ForEach (listData) { item in  
                HStack {  
                    Image(systemName: item.imageName)  
                    Text(item.task)  
                }  
            }  
        }  
    }  
}
```

Note the appearance of the toggle button and the dynamic list items in Figure 31-3:

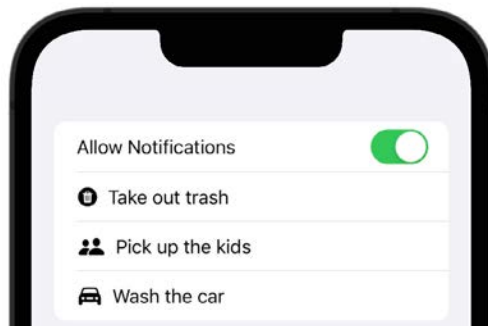


Figure 31-3

A SwiftUI List implementation may also be divided into sections using the Section view, including headers and footers if required. Figure 31-4 shows the list divided into two sections, each with a header:

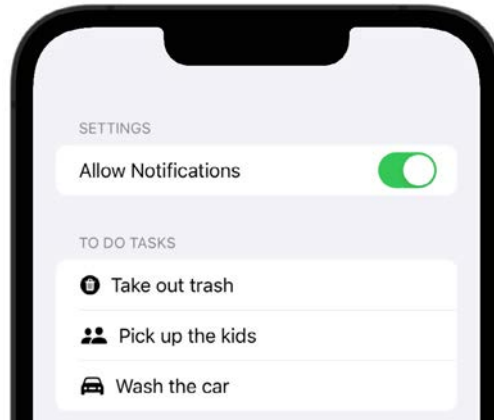


Figure 31-4

The changes to the view declaration to implement these sections are as follows:

```
List {
    Section(header: Text("Settings")) {
        Toggle(isOn: $toggleStatus) {
            Text("Allow Notifications")
        }
    }

    Section(header: Text("To Do Tasks")) {
        ForEach (listData) { item in
            HStack {
                Image(systemName: item.imageName)
                Text(item.task)
            }
        }
    }
}
```

Often the items within a list will navigate to another area of the app when tapped by the user. Behavior of this type is implemented in SwiftUI using the `NavigationStack` and `NavigationLink` views.

31.4 Creating a Refreshable List

The data displayed on a screen is often derived from a dynamic source which is subject to change over time. The standard paradigm within iOS apps is for the user to perform a downward swipe to refresh the displayed data. During the refresh process, the app will typically display a spinning progress indicator after which the latest data is displayed. To make it easy to add this type of refresh behavior to your apps, SwiftUI provides the `refreshable()` modifier. When applied to a view, a downward swipe gesture on that view will display the progress indicator and execute the code in the modifier closure. For example, we can add refresh support to our list as follows:

```
List {
    Section(header: Text("Settings")) {
        Toggle(isOn: $toggleStatus) {
            Text("Allow Notifications")
        }
    }
}
```

```

    }
}

Section(header: Text("To Do Tasks")) {
    ForEach (listData) { item in
        HStack {
            Image(systemName: item.imageName)
            Text(item.task)
        }
    }
}

.refreshable {
    listData = [
        ToDoItem(task: "Order dinner", imageName: "dollarsign.circle.fill"),
        ToDoItem(task: "Call financial advisor", imageName: "phone.fill"),
        ToDoItem(task: "Sell the kids", imageName: "person.2.fill")
    ]
}

```

Figure 31-5 demonstrates the effect of performing a downward swipe gesture within the List view after adding the above modifier. Note both the progress indicator at the top of the list and the appearance of the updated to-do list items:

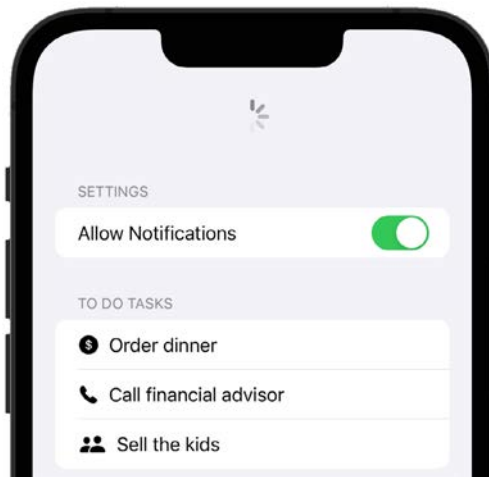


Figure 31-5

When using the *refreshable()* modifier, be sure to perform any time-consuming activities as an asynchronous task using structured concurrency (covered previously in the chapter entitled “*An Overview of Swift Structured Concurrency*”). This will ensure that the app remains responsive during the refresh.

31.5 SwiftUI NavigationStack and NavigationLink

To make items in a list navigable, the first step is to embed the entire list within a `NavigationStack`. Once the list is embedded, the individual rows must be wrapped in a `NavigationLink` control which is, in turn, passed a value that uniquely identifies each navigation link within the context of the `NavigationStack`.

The following changes to our example code embed the List view in a NavigationStack and wrap the row content in a NavigationLink:

```

NavigationStack {
    List {
        Section(header: Text("Settings")) {
            Toggle(isOn: $toggleStatus) {
                Text("Allow Notifications")
            }
        }

        Section(header: Text("To Do Tasks")) {
            ForEach (listData) { item in
                NavigationLink(value: item.task) {
                    HStack {
                        Image(systemName: item.imageName)
                        Text(item.task)
                    }
                }
            }
        }
    }
}

```

Note that we have used the item task string as the NavigationLink value to uniquely identify each row. The next step is to specify the destination view to which the user is to be taken when the row is tapped. We achieve this by applying the *navigationDestination(for:)* modifier to the list. When adding this modifier, we need to pass it the value type for which it is to provide navigation. In our example we are using the task string, so we need to specify *String.self* as the value type. Within the trailing closure of the *navigationDestination(for:)* call we need to call the view that is to be displayed when the row is selected. This closure is passed the value from the NavigationLink, allowing us to display the appropriate view:

```

NavigationStack {
    List {
        .
        .

        Section(header: Text("To Do Tasks")) {
            ForEach (listData) { item in
                NavigationLink(value: item.task) {
                    HStack {
                        Image(systemName: item.imageName)
                        Text(item.task)
                    }
                }
            }
        }
    }
}
.navigationDestination(for: String.self) { task in

```

```

        Text("Selected task = \(task)")
    }
}

```

In this example, the navigation link will simply display a new screen containing the destination Text view displaying the *item.task* string value. The finished list will appear, as shown in Figure 31-6 with the title and chevrons on the far right of each row now visible indicating that navigation is available. Tapping the links will navigate to and display the destination Text view.



Figure 31-6

31.6 Navigation by Value Type

The *navigationDestination()* modifier is particularly useful for adding navigation support to lists containing values of different types, with each type requiring navigation to a specific view. Suppose, for example, that in addition to the string-based task navigation link, we also have a *NavigationLink* which is passed an integer value indicating the number of tasks in the list. This could be implemented in our example as follows:

```

NavigationStack {

    List {

        Section(header: Text("Settings")) {
            Toggle(isOn: $toggleStatus) {
                Text("Allow Notifications")
            }

            NavigationLink(value: listData.count) {
                Text("View Task Count")
            }
        }

        .
        .
    }
}

```

When this link is selected, we need the app to navigate to a Text view that displays the current task count. All this requires is a second *navigationDestination()* modifier, this time configured to handle Int instead of String values:

```

.
}
.navigationDestination(for: String.self) { task in
    Text("Selected task = \(task)")
}
.navigationDestination(for: Int.self) { count in
    Text("Number of tasks = \(count)")
}
.
.

```

This technique allows us to configure multiple navigation destinations within a single navigation stack based solely on the value type passed to each navigation link.

31.7 Working with Navigation Paths

As the name suggests, `NavigationStack` provides a stack on which navigation targets are stored as the user navigates through the screens of an app. When a user navigates from one view to another, a reference to the originating view is *pushed* onto the stack. If the user then navigates to another view, the current view will also be placed onto the stack. At any point, the user can tap the back arrow displayed in the navigation bar to move back to the previous view. As the user navigates back through the views, each one is *popped* off the stack until the view from which navigation began is reached.

The views through which a user navigates are called the *navigation path*. SwiftUI allows us to provide our own path by passing an instance of `NavigationPath` to the `NavigationStack` instance as follows:

```

struct ContentView: View {

    @State private var stackPath = NavigationPath()

    var body: some View {

        NavigationStack(path: $stackPath) {
.
.

```

With `NavigationStack` using our path, we can perform tasks such as manually popping targets off the stack to jump back multiple navigation levels instead of making the user navigate through the targets individually. We could, for example, configure a button on a view deep within the stack to take the user directly back to the home screen. We can do this by identifying how many navigation targets are in the stack and then removing them via a call to the `removeLast()` method of the path instance, for example:

```

var stackCount = stackPath.count
stackPath.removeLast(stackCount)

```

We can also programmatically navigate to specific destination views by calling the navigation path's `append()` method and passing through the navigation value associated with the destination:

```

stackPath.append(value)

```

31.8 Navigation Bar Customization

The `NavigationStack` title bar may also be customized using modifiers on the `List` component to set the title and add buttons to perform additional tasks. The following example calls the `navigationTitle()` modifier to set the

title to “To Do List.” The code also adds a button labeled “Add” to the toolbar. This button is applied to the layout as the child of a `ToolbarItem` instance using the `toolbar()` modifier and configured to call a hypothetical method named `addTask()`:

```
NavigationStack {
    List {
        .
        .
    }
    .navigationTitle(Text("To Do List"))
    .toolbar {
        ToolbarItem(placement: .navigationBarLeading) {
            Button(action: addTask) {
                Text("Add")
            }
        }
    }
}
```

31.9 Making the List Editable

It is common for an app to allow the user to delete items from a list and, in some cases, even move an item from one position to another. Deletion can be enabled by adding an `onDelete()` modifier to each list cell, specifying a method to be called which will delete the item from the data source. When this method is called it will be passed an `IndexSet` object containing the offsets of the rows being deleted and it is the responsibility of this method to remove the selected data from the data source. Once implemented, the user will be able to swipe left on rows in the list to reveal the Delete button, as shown in Figure 31-7:

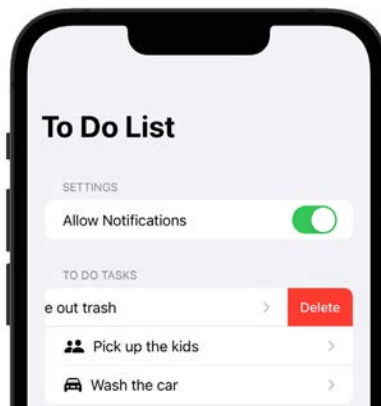


Figure 31-7

The changes to the example List to implement this behavior might read as follows:

```
.
.
List {
    Section(header: Text("Settings")) {
```

```

        Toggle(isOn: $toggleStatus) {
            Text("Allow Notifications")
        }
    }

    Section(header: Text("To Do Tasks")) {
        ForEach (listData) { item in
            NavigationLink(value: item.task) {
                HStack {
                    Image(systemName: item.imageName)
                    Text(item.task)
                }
            }
        }
        .onDelete(perform: deleteItem)
    }
}
.
.
func deleteItem(at offsets: IndexSet) {
    // Delete items from the data source here
}

```

To allow the user to move items up and down in the list the *onMove()* modifier must be applied to the cell, once again specifying a method to be called to modify the ordering of the source data. In this case, the method will be passed an *IndexSet* object containing the positions of the rows being moved and an integer indicating the destination position.

In addition to adding the *onMove()* modifier, an *EditButton* instance needs to be added to the *List*. When tapped, this button automatically switches the list into editable mode and allows items to be moved and deleted by the user. The *List* declaration can be modified as follows to add this functionality:

```

List {
    Section(header: Text("Settings")) {
        Toggle(isOn: $toggleStatus) {
            Text("Allow Notifications")
        }
    }

    Section(header: Text("To Do Tasks")) {
        ForEach (listData) { item in
            NavigationLink(value: item.task) {
                HStack {
                    Image(systemName: item.imageName)
                    Text(item.task)
                }
            }
        }
    }
}

```

SwiftUI Lists and Navigation

```
.onDelete(perform: deleteItem)
.onMove(perform: moveItem)
}

}
.navigationTitle(Text("To Do List"))
.toolbar {
    ToolbarItem(placement: .navigationBarLeading) {
        NavigationLink(value: "Add Car") { Text("Add") }
    }
    ToolbarItem(placement: .navigationBarTrailing) {
        EditButton()
    }
}
.
.
func moveItem(from source: IndexSet, to destination: Int) {
    // Reorder items in source data here
}
```

Viewed within the preview canvas, the list will appear, as shown in Figure 31-8 when the Edit button is tapped. Clicking and dragging the three lines on the right side of each row allows the row to be moved to a different list position (in the figure below the “Pick up the kids” entry is in the process of being moved):



Figure 31-8

31.10 Hierarchical Lists

SwiftUI also includes support for organizing hierarchical data for display in list format, as shown in Figure 31-9 below:

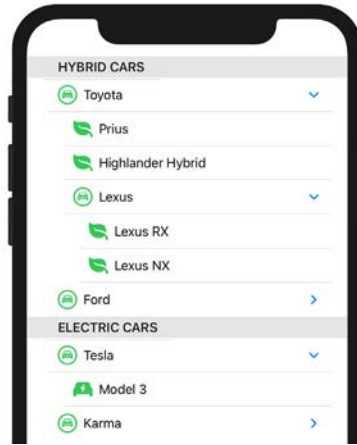


Figure 31-9

This behavior is achieved using features of the List view together with the OutlineGroup and DisclosureGroup views which automatically analyze the parent-child relationships within a data structure to create a browsable list containing controls to expand and collapse branches of data. This topic is covered in detail beginning with the chapter titled *“An Overview of List, OutlineGroup and DisclosureGroup”*.

31.11 Multicolumn Navigation

NavigationStack provides navigation between views where each destination occupies the entire device screen. SwiftUI also supports multicolumn navigation where the destinations appear together on the screen with each appearing in a separate column. Multicolumn navigation is provided by the NavigationSplitView component and will be covered beginning with the chapter titled *“An Overview of Split View Navigation”*.

31.12 Take the Knowledge Test



Click the link below or scan the QR code to test your knowledge and understanding of SwiftUI list navigation:

<https://www.answerstopia.com/utji>



31.13 Summary

The SwiftUI List view provides a way to order items in a single column of rows, each containing a cell. Each cell, in turn, can contain multiple views when those views are encapsulated in a container view such as a stack layout. The List view provides support for displaying both static and dynamic items or a combination of both. Lists may also be used to group, organize and display hierarchical data.

List views are used to allow the user to navigate to other screens. This navigation is implemented by wrapping the List declaration in a NavigationStack and each row in a NavigationLink, using the `navigationDestination()` modifier to define the navigation target view.

Lists can be divided into titled sections and assigned a navigation bar containing a title and buttons. Lists may also be configured to allow rows to be added, deleted, and moved.

55. An Introduction to SwiftData

The preceding chapters covered database storage using Core Data. While Core Data is a powerful and flexible solution to data storage, it was created long before the introduction of SwiftUI and lacks the simplicity of SwiftUI’s approach to app development. Introduced in iOS 17, SwiftData addresses this shortcoming by providing a declarative approach to persistent data storage that is tightly integrated with SwiftUI.

This chapter introduces SwiftData and provides a broad overview of the key elements required to store and manage persistent data within iOS apps.

55.1 Introducing SwiftData

The SwiftData framework integrates seamlessly with SwiftUI code and offers a declarative way to store persistent data within apps. Implemented as a layer on top of Core Data, SwiftData provides access to many of its features without the need to write complex code.

The rest of this chapter will introduce the SwiftData framework classes and outline how to integrate SwiftData into your iOS app projects. In the next chapter, titled “*A SwiftData Tutorial*”, we will create a project that demonstrates persistent data storage using SwiftData.

55.2 Model Classes

The SwiftData model classes represent the schema for the data to be stored and are declared as Swift classes. Consider the following class representing the data structure of an address book app:

```
class Contact {
    var firstname: String
    var lastname: String
    var address: String

    init(firstname: String, lastname: String, address: String) {
        self.firstname = firstname
        self.lastname = lastname
        self.address = address
    }
}
```

To store the contact information using SwiftData, we need to designate this class as a SwiftData model. To make this declaration, all that is required is to import the SwiftData framework and add the `@Model` macro to the class:

```
import SwiftData
```

```
@Model
```

```
class Contact {
    var firstname: String
    var lastname: String
    var address: String
}
```

```
init(firstname: String, lastname: String, address: String) {
    self.firstname = firstname
    self.lastname = lastname
    self.address = address
}
}
```

55.3 Model Container

The purpose of the Model Container class is to collect the model schema and generate a database in which to store instances of the data model objects. Essentially, the model container provides an interface between the model schema and the underlying database storage.

Model containers may be created directly or by applying the *modelContainer(for:)* modifier to a Scene or WindowGroup. In both cases, the container must be passed a list of the models to be managed. The following code, for example, creates a model container for our Contact model:

```
let modelContainer = try? ModelContainer(for: Contact.self)
```

In the following example, the model container is initialized using three models:

```
let modelContainer = try? ModelContainer(for: Contact.self, Message.self,
CallLog.self)
```

The following code, on the other hand, uses the *modelContainer(for:)* modifier to create a model container for a WindowGroup:

```
var body: some Scene {
    WindowGroup {
        ContentView()
    }
    .modelContainer(for: Contact.self)
}
```

55.4 Model Configuration

Model configurations can be applied to model containers to configure how the persistent data is stored and accessed. A model container might, for example, be configured to store the data in memory, in a specific file, or to access data in read-only mode. The following code creates a model configuration for in-memory data storage and applies it to a new model container:

```
let modelConfig = ModelConfiguration(isStoredInMemoryOnly: true)
let modelContainer = try? ModelContainer(
    for: Contact.self,
    configurations: modelConfig)
```

55.5 Model Context

When a model container is created, SwiftUI creates a binding to the container's model context. The model context tracks changes to the underlying data and provides the programming interface through which the app code performs operations on the stored data, such as adding, updating, fetching, and deleting model objects.

When a model container is created, a binding to the model context is placed into the app's environment, where it can be accessed from within scenes and views as follows:

```
@Environment(\.modelContext) var modelContext
```

The model context provides several methods for accessing the database, for example:

```
// Insert a model object
modelContext.insert(contact)

// Delete a model object
modelContext.delete(contact)

// Save all changes
modelContext.save()
```

55.6 Predicates and FetchDescriptors

Predicates define the criteria for fetching matching data from a database and take the form of logical expressions that evaluate to true or false. The following code creates a predicate to filter the contacts whose last name is “Smith”:

```
let namePredicate = #Predicate<Contact> { $0.lastname.contains("Smith") }
```

Once the predicate has been declared, it is used to create a `FetchDescriptor` as follows:

```
let descriptor = FetchDescriptor<Visitor>(predicate: namePredicate)
```

Finally, the fetch descriptor is passed to the model context’s `fetch()` method to obtain a list of matching objects:

```
let theSmiths = try? modelContext.fetch(descriptor)
```

In addition to filtering fetch results, the fetch descriptor can also be used to sort the returned matches. The following descriptor, for example, sorts the fetch results by contact last name:

```
let descriptor = FetchDescriptor<Visitor>(predicate: namePredicate,
                                         sortBy: [SortDescriptor(\Contact.lastname)])
```

The `SortDescriptor` may also be used to specify the sorting order of the fetch results. The following `SortDescriptor` example will reverse the sorting order when used in a `fetch()` call:

```
let descriptor = FetchDescriptor<Visitor>(predicate: namePredicate,
                                         sortBy: [SortDescriptor(\Contact.lastname, order: .reverse)])
```

55.7 The @Query Macro

The `@Query` macro provides a convenient way to fetch objects from storage and uses the observability features of SwiftUI to ensure that the results are always up to date. In the simplest form, the `@Query` macro can be used to fetch all of the stored contact objects from the database:

```
@Query var contacts: [Contact]
```

Once declared, the `contacts` array will automatically update to contain the latest contacts without the need to call the `fetch()` method on the model context.

The `@Query` macro can also be used to filter results using predicates and sort descriptors, for example:

```
@Query(filter: #Predicate<Contact> { $0.lastname.contains("Smith") }, sort:
[SortDescriptor(\Contact.lastname, order: .reverse)]) var theSmiths: [Visitor]
```

55.8 Model Relationships

Relationships between SwiftData models are declared using the `@Relationship` macro. Suppose, for example, that our address book app keeps a phone call log for each of our contacts. This will require a model class containing the date and time of the call:

```
@Model
```

```
class CallDate {  
  
    var date: Date  
  
    init(date: Date) {  
        self.date = date  
    }  
}
```

To associate a contact with the list of calls, we need to establish a relationship between the `Contact` and `CallDate` models. This is achieved using the `@Relationship` macro in the `Contact` model as follows:

```
@Model  
class Contact {  
    var firstname: String  
    var lastname: String  
    var address: String  
  
    @Relationship var calls = [CallDate]()  
  
    init(firstname: String, lastname: String, address: String) {  
        self.firstname = firstname  
        self.lastname = lastname  
        self.address = address  
    }  
}
```

With the relationship established, we can access the `calls` array property of contact model objects to access the list of associated calls. In the following code, for example, a new call entry is added to a contact's call log:

```
contact.calls.append(CallDate(date: Date.now))
```

When a model object is deleted, the default behavior is for any related objects to remain in the database. This means that if we deleted a contact, all of their calls would remain in the database. While this may be the desired behavior in other situations, it does not make sense to keep the log entries in our address book example. To delete all of the call data when a contact is removed, we can specify a deletion rule. In this case, the `cascade` option is used to remove all related data down through the entire chain of relationships:

```
@Relationship(deleteRule: .cascade) var calls = [CallDate]()
```

The full list of deletion rules is as follows:

- **cascade** - Removes all related objects.
- **deny** - Prevents the removal of objects containing relationships with other objects.
- **noAction** - Leaves the related objects unchanged, leaving in place references to the deleted objects.
- **nullify** - Does not remove the related objects but nullifies references to the deleted objects.

55.9 Model Attributes

The `@Attributes` macro applies behavior to individual properties in a model class. A common use is to specify unique properties. For example, to prevent duplicate last names, the `@Attribute` macro would be used as follows:

```

@Model
class Contact {
    var firstname: String
    @Attribute(.unique) var lastname: String
    var address: String

    @Relationship var calls = [CallDate]()

    init(firstname: String, lastname: String, address: String) {
        self.firstname = firstname
        self.lastname = lastname
        self.address = address
    }
}

```

Finally, a property within a model class may be excluded from being stored in the database using the `@Transient` macro:

```

@Model
class Contact {
    var firstname: String
    var lastname: String
    var address: String
    @Transient var tempAddr: String
}

```

55.10 Take the Knowledge Test



Click the link below or scan the QR code to test your knowledge and understanding of SwiftData:

<https://www.answerstopia.com/a7fx>



55.11 Summary

SwiftData combines many of the features of Core Data with the convenience of SwiftUI to provide a simple way to store persistent data in iOS apps. The database schema are declared as Swift classes and adapted into SwiftData models using the `@Model` macro. The model container collects the model classes and uses them to create and manage the underlying database system. The model context tracks changes to the data and provides a programming interface for adding, searching, and modifying the stored data objects. Data is fetched using predicates, fetch descriptors, and the `@Query` macro. Relationships between models are established using the `@Relationship` macro, while the `@Attributes` macro allows rules to be applied to individual model class properties.

62. An Overview of Live Activities in SwiftUI

The previous chapters introduced WidgetKit and demonstrated how it can be used to display widgets that provide information to the user on the home screen, lock screen, and Today view. Widgets of this type present information based on a timeline you create and pass to WidgetKit. In this chapter, we will introduce ActivityKit and Live Activities and explore how these can be used to present dynamic information to the user via widgets on the lock screen and Dynamic Island.

62.1 Introducing Live Activities

Live Activities are created using the ActivityKit and WidgetKit frameworks and present dynamic information in glanceable form without restricting updates to a predefined timeline.

A single app can have multiple Live Activities, and the information presented can be sourced locally within the app or delivered from a remote server via push notifications. One important caveat is that updates to the Live Activity will not necessarily occur in real-time. Both the local and remote push notification options use background modes of execution, the timing and frequency of which are dictated by the operating system based on various factors, including battery status, the resource-intensive nature of the update task, and user behavior patterns. We will cover this in more detail in the next chapter.

In addition to displaying information, Live Activities may contain Button and Toggle views to add interactive behavior.

62.2 Creating a Live Activity

Once a Widget Extension has been added to an Xcode app project, the process of creating a Live Activity can be separated into the following steps, each of which will be covered in this chapter and put to practical use in the next chapter:

- Declare static and dynamic Activity Attributes.
- Design the Live Activity presentations for the lock screen and Dynamic Island.
- Configure and start the Live Activity.
- Update the Live Activity with the latest information.
- End the Live Activity when updates are no longer required.

62.3 Live Activity Attributes

The purpose of Live Activities is to present information to the user when the corresponding app has been placed in the background. The Live Activity attributes declare the data structure to be presented and are created using ActivityKit's `ActivityAttributes` class. Two types of attributes can be included. The first type declares the data that will change over the lifecycle of the Live Activity, such as the latest scores of a live sporting event or an estimated flight arrival time. The second attribute type declares values that will remain static while the Live Activity executes, such as the name of the sports teams or the airline and flight number of a tracked flight.

Within the `ActivityAttributes` declaration, the dynamic attributes are embedded in a `ContentState` structure using the following syntax:

```
struct DemoWidgetAttributes: ActivityAttributes {
    public struct ContentState: Codable, Hashable {
        // dynamic attributes here
        var arrivalTime: Date
    }

    // static attributes here
    var airlineName: String = "Pending"
    var flightNumber: String = "Pending"
}
```

62.4 Designing the Live Activity Presentations

Live Activities present data to the user via lock screen, Dynamic Island, and banner widgets, each of which must be designed to complete the Live Activity. These presentations are created using SwiftUI views. While the lock screen presentation (also used for the banner widget) consists of a single layout, the Dynamic Island presentations are separated into regions.

The layouts for the Live Activity widgets are defined in a configuration structure subclassed from the `WidgetKit` framework's `Widget` class and must conform to the following syntax:

```
struct DemoWidgetLiveActivity: Widget {
    var body: some WidgetConfiguration {
        ActivityConfiguration(for: DemoWidgetAttributes.self) { context in

            } dynamicIsland: { context in

                DynamicIsland {

                    DynamicIslandExpandedRegion(.leading) {

                    }

                    DynamicIslandExpandedRegion(.trailing) {

                    }

                    DynamicIslandExpandedRegion(.bottom) {

                    }

                    DynamicIslandExpandedRegion(.center) {

                    }

                } compactLeading: {

                } compactTrailing: {

                } minimal: {
```

```

    }
  }
}

```

Each element is passed a context object from which static and current dynamic data values can be accessed for inclusion in the presentation views. For example, the arrival time and flight number from the previous activity attributes declaration could be displayed by the widget as follows:

```

Text("Arrival: \(context.state.arrivalTime)")
Text("Flight: \(context.attributes.flightNumber)")

```

62.4.1 Lock Screen/Banner

Starting at the top of the Widget declaration, the layout for the lock screen and banner presentation consists of an area the size of a typical lock screen notification. The following example will display two Text views in a VStack layout:

```

struct DemoWidgetLiveActivity: Widget {
    var body: some WidgetConfiguration {
        ActivityConfiguration(for: DemoWidgetAttributes.self) { context in
            VStack {
                Text("Arrival: \(context.state.arrivalTime)")
                Text("Flight: \(context.attributes.flightNumber)")
            }
        } dynamicIsland: { context in
            .
            .
        }
    }
}

```

62.4.2 Dynamic Island Expanded Regions

The Live Activity will display data using compact layouts on devices with a Dynamic Island. However, a long press performed on the island will display the expanded widget. Unlike the lock screen widget, the expanded Dynamic Island presentation is divided into four regions, as illustrated in Figure 62-1:

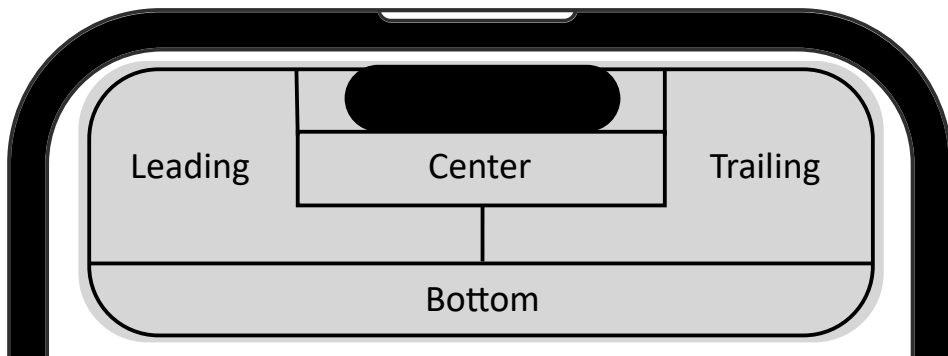


Figure 62-1

The following example highlights the code locations for each Dynamic Island region:

```

.
.

```

An Overview of Live Activities in SwiftUI

```
} dynamicIsland: { context in

    DynamicIsland {

        DynamicIslandExpandedRegion(.leading) {
            Text("Leading")
        }
        DynamicIslandExpandedRegion(.trailing) {
            Text("Trailing")
        }
        DynamicIslandExpandedRegion(.bottom) {
            Text("Bottom")
        }
        DynamicIslandExpandedRegion(.center) {
            Text("Center")
        }
    } compactLeading: {
        .
        .
    }
}
```

The default sizing behavior of each region can be changed using priorities. In the following code, for example, the leading and trailing region sizes are set to 25% and 75% of the available presentation width, respectively:

```
DynamicIslandExpandedRegion(.leading, priority: 0.25) {
    Text("Leading")
}
DynamicIslandExpandedRegion(.trailing, priority: 0.75) {
    Text("Trailing")
}
```

62.4.3 Dynamic Island Compact Regions

The compact presentation is divided into regions located on either side of the camera, as illustrated in Figure 62-2:

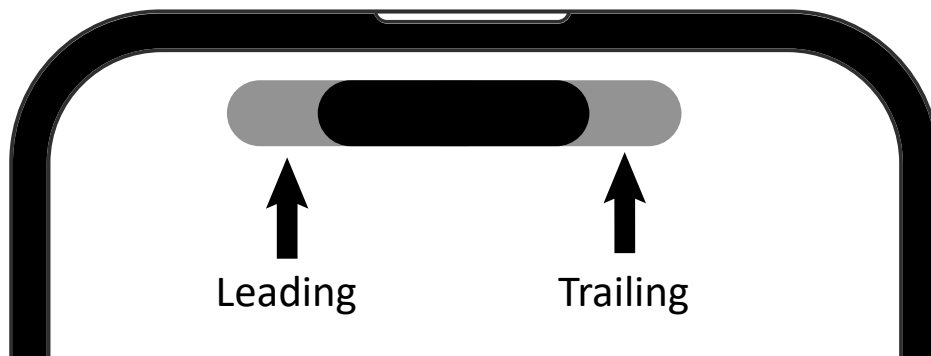


Figure 62-2

An example compact declaration might read as follows:

.

```

.
} compactLeading: {
    Text("L")
} compactTrailing: {
    Text("T")
} minimal: {
.
.

```

62.4.4 Dynamic Island Minimal

The Live Activity uses minimal presentations when multiple Live Activities are running concurrently. In this situation, the minimal presentation for one Live Activity will appear in the compact leading region (referred to as the *attached minimal*), while another appears as a *detached minimal* positioned to the right of the camera:

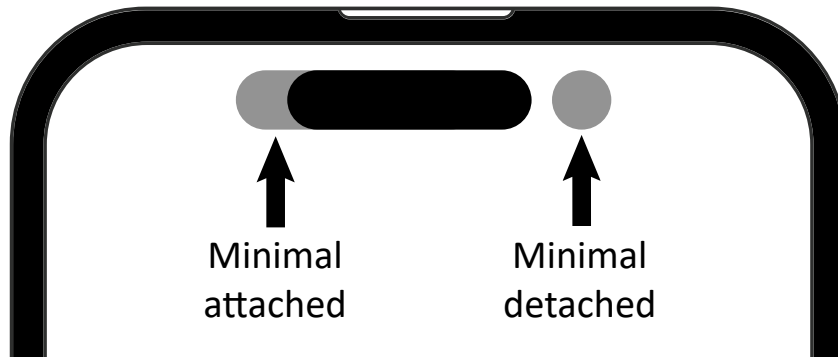


Figure 62-3

For example:

```

.
.
} minimal: {
    Text("M")
}
.
.

```

62.5 Starting a Live Activity

Once the data model has been defined and the presentations designed, the next step is to request and start the Live Activity. This is achieved by a call to the `Activity.request()` method. When the request method is called, an activity attributes instance, an initialized `ContentState`, and a push type must be provided. The push type should be set to `token` if the data updates will be received via push notifications or `nil` if updates are coming from the app.

An optional *stale date* may also be included. When the stale date is reached, the state of the Live Activity context will update to reflect that the information is out of date, allowing you to notify the user within the widget presentation. To check if the Live Activity is out of date, access the context's `isStale` property. The following code, for example, displays a message in the Dynamic Island expanded presentation when the data needs to be refreshed:

```
DynamicIslandExpandedRegion(.leading) {
```

An Overview of Live Activities in SwiftUI

```
VStack {
    Text("Arrival: \(context.state.arrivalTime)")
    Text("Flight: \(context.attributes.flightNumber)")

    if (context.isStale) {
        Text("Out of date")
    }
}
```

Set the *staleDate* parameter to *nil* if you do not plan to check the Live Activity status for this property.

Based on the above requirements, the first step is to create an activity attributes object and initialize any static properties, for example:

```
var attributes = DemoWidgetAttributes()
attributes.flightNumber = "Loading..."
```

The second requirement is a *ContentState* instance configured with initial dynamic values:

```
let contentState = DemoWidgetAttributes.ContentState(arrivalTime: Date.now + 60)
```

With the requirements met, the *Activity.request()* method can be called as follows:

```
private var activity: Activity<DemoWidgetAttributes>?
.
.
do {
    activity = try Activity.request(
        attributes: attributes,
        content: .init(state: contentState, staleDate: nil),
        pushType: nil
    )
} catch (let error) {
    print("Error requesting live activity: \(error.localizedDescription).")
}
}
```

If the request is successful, the Live Activity will launch and be ready to receive updates. In the above example, the push type has been set to *nil* to indicate the data is generated within the app. This would need to be changed to *token* to support updates using push notifications.

62.6 Updating a Live Activity

To refresh a Live Activity with updated data, a call is made to the *update()* method of the activity instance returned by the earlier call to the *Activity.request()* method. The update call must be passed an *ActivityContent* instance containing a *ContentState* initialized with the updated dynamic data values and an optional stale date value. For example:

```
let flightState = DemoWidgetAttributes.ContentState(arrivalTime: newTime)
```

```
Task {

    await activity?.update(
```

```

        ActivityContent<DemoWidgetAttributes.ContentState>(
            state: flightState,
            staleDate: Date.now + 120,
            relevanceScore: 0
        ),
        alertConfiguration: nil
    )
}

```

If your app starts multiple concurrent Live Activities, the system will display the one with the highest `relevanceScore`. When working with push notifications, the content state is updated automatically, and the update call is unnecessary.

62.7 Activity Alert Configurations

Alert configurations are passed to the `update()` method to notify the user of significant events in the Live Activity data. When an alert is triggered, a banner (based on the lock screen presentation layout) appears on the device screen, accompanied by an optional alert sound. The following code example creates an alert configuration when a tracked flight has been significantly delayed:

```

var alertConfig: AlertConfiguration? = nil

if (arrivalTime > Date.now + 84000) {
    alertConfig = AlertConfiguration(
        title: "Flight Delay",
        body: "Flight now arriving tomorrow",
        sound: .default
    )
}

```

Note that the title and body text will only appear on Apple Watch devices.

Once an alert configuration has been created, it can be passed to the `update()` method:

```

await activity?.update(
    ActivityContent<DemoWidgetAttributes.ContentState>(
        state: flightState,
        staleDate: Date.now + 120,
        relevanceScore: 0
    ),
    alertConfiguration: alertConfig
)

```

62.8 Stopping a Live Activity

Live Activities are stopped by calling the `end()` method of the activity instance. The call is passed a `ContentState` instance initialized with the final data values and a dismissal policy setting. For example:

```

let finalState = DemoWidgetAttributes.ContentState(arrivalTime: Date.now)

await activity?.end(
    .init(state: finalState, staleDate: nil),
    dismissalPolicy: .default
)

```

)

When the `dismissalPolicy` is set to *default*, the Live Activity widget will remain on the lock screen for four hours unless the user removes it. Use *immediate* to instantly remove the Live Activity from the lock screen or *after()* to dismiss the Live Activity at a specific time within the four-hour window.

62.9 Take the Knowledge Test



Click the link below or scan the QR code to test your knowledge and understanding of SwiftUI Live Activities:

<https://www.answerstopia.com/oh2m>



62.10 Summary

Live Activities provide users with timely updates via widgets on the device lock screen and Dynamic Island. Updated information can be generated locally within the app or sent from a remote server using push notifications. A Live Activity consists of a set of attributes that define the data to be presented and SwiftUI-based layouts for each of the widget presentations. Live Activity instances are started, stopped, and updated using calls to the corresponding Activity object. When working with push notifications, the activity will update automatically on receipt of a notification. Updates may also include an optional alert to attract the user's attention.

Index

Symbols

& 37
 ^ 38
 ^= 39
 << 39
 <<= 39
 &= 39
 >> 39
 >>= 39
 | 38
 |= 39
 ~ 37
 \$ 156
 \$0 61
 @AppStorage 231, 232, 235, 237
 @Attributes 436
 @Bindable 161
 @Binding 157, 259
 @Environment 162
 @FetchRequest 412, 416
 @GestureState 364
 @main 130
 @MainActor 217
 @Model 433, 439
 @Observable 160
 @ObservedObject 159
 ?? operator 36
 @Parameter 485
 @Published 158
 @Query 435, 441, 445
 @Relationship 435, 441
 @SceneStorage 231, 233, 235, 237
 @State 155
 @StateObject 159
 Text Styles

body 139
 callout 139
 caption 139
 footnote 139
 headline 139
 subheadline 139
 @Transient 437
 @ViewBuilder 320

A

Actors 213
 data isolation 214
 Declaring 213
 example 215
 @MainActor 216
 MainActor 216
 nonisolated keyword 214
 adaptable padding 147
 addArc() 342
 addCurve() 342
 addLine() 342
 addLines() 342
 addQuadCurve() 342
 addTask() function 207
 Alignment 147
 Cross Stack 185
 alignmentGuide() modifier 181
 Alignment Guides 177, 179
 AlignmentID protocol 183
 Alignment Types
 custom 182
 AND (&&) operator 35
 AND operator 37
 Animation 170, 351
 automatically starting 356
 autoreverse 353
 easeIn 352
 easeInOut 352
 easeOut 352

Index

- explicit 354
 - implicit 351
 - linear 352
 - repeating 353
 - animation() modifier 351
 - AnyObject 94
 - AnyTransition 359
 - APNs 521
 - apns-expiration 533
 - APNs Key
 - registering 522
 - apns-priority 533
 - apns-topic 533
 - apns-unique-id 536
 - App 127
 - app delegate 524
 - append() method 247
 - AppEntity protocol 484
 - App Hierarchy 127
 - App Icons 571
 - App Intent Configuration 450
 - AppIntentConfiguration 484
 - App Intents framework 517
 - AppIntentTimelineProvider 483
 - AppIntentTimelineProvider protocol 452
 - Apple Developer Program 3
 - Apple Push Notification service 521
 - Application Performance 122
 - AppStorage 231
 - App Store
 - creating archive 572
 - submission 569
 - App Store Connect 573
 - Architecture
 - overview 127
 - AreaMark 375
 - Array
 - forEach() 93
 - mixed type 94
 - Array Initialization 91
 - Array Item Count 92
 - Array Items
 - accessing 92
 - appending 93
 - inserting and deleting 93
 - Array Iteration 93
 - Arrays
 - immutable 91
 - mutable 91
 - as! keyword 31
 - Assets.xcassets 130
 - Assistant Editor 561
 - async
 - suspend points 201
 - async/await 201
 - asynchronous functions 200
 - Asynchronous Properties 210
 - async keyword 201
 - async-let bindings 203
 - AsyncSequence protocol 209
 - Attributes inspector 118
 - await keyword 201, 202
- ## B
- backgroundAccessoryView 393
 - Background Notifications
 - enabling 427
 - BarMark 375
 - Bézier curves 342
 - binary operators 33
 - bit operators 37
 - Bitwise AND 37
 - Bitwise Left Shift 38
 - bitwise OR 38
 - bitwise right shift 39
 - bitwise XOR 38
 - body 139
 - Boolean Logical Operators 35
 - break statement 43
 - Build Errors 122
- ## C
- callout 139
 - cancelAll() function 208

- Capsule() 340
- caption 139
- cardinal 379
- case Statement 48
- catch statement 101
 - multiple matches 101
- catmullRom 379
- CGRect 342, 343
- Character data type 23
- chartPlotStyle() 379
- Charts 375, 381
 - chartPlotStyle() 379
 - foregroundColor() modifier 378, 379, 383
 - interpolationMethod() modifier 379
 - interpolation options 379
 - mark type combining 377
 - mark types 375
 - multiple graphs 378
 - passing data to 376
 - PlottableValue 375
 - PointMark 384
 - symbol() modifier 379
- checkCancellation() method 207
- Child Limit 149
- CircularProgressViewStyle 370
- Class Extensions 76
- closed range operator 35
- closeSubPath() 342
- Closure Expressions 60
 - shorthand argument names 61
- closures 53
- Closures 61
- CloudKit 421
 - add container 426
 - Console 423, 428
 - Containers 421
 - Data Storage Quotas 422
 - enabling in Xcode 425
 - filtering and sorting 429
 - NSPersistentCloudKitContainer 427
 - Persistence Container 427
 - Record IDs 423
 - Records 422
 - Record Zones 423
 - References 423
 - Sharing 424
 - Subscriptions 424
- CloudKit console 531
 - push notifications 531
- CloudKit Console 423, 428
- CloudKit Sharing 424
- CloudKit Subscriptions 424
- code editor 111
 - context menu 118
- Color
 - drop() modifier 345
 - .gradient 345
 - inner() modifier 345
 - .mix() modifier 344
 - shadow() modifier 345
- Color Space 344
 - device 344
 - perceptual 344
- Combine framework 158
- Comparable protocol 88
- Comparison Operators 34
- Completion Handlers 199
- Compound Bitwise Operators 39
- computed properties 67
- concrete type 71
- Concurrent Tasks
 - launching 223
- Conditional Control Flow 44
- Configuration Intent UI 488
- constants 25
- Container Alignment 177
- Container Child Limit 149
- Containers 421
- Container Views 141
- ContentView.swift file 111, 130
- Context Menus 335
- continue Statement 43
- Coordinator 543
- Core Data 403, 409

Index

- enabling in Xcode 409
 - Entity Description 405
 - Fetches property 404
 - Fetch request 404
 - @FetchRequest 412, 416
 - loadPersistentStores() method 406
 - Managed Object 406
 - Managed Object Context 404, 406
 - Managed Object Model 404
 - Managed Objects 404
 - NSFetchRequest 407, 419
 - NSPersistentContainer 406
 - NSSortDescriptor 416
 - Persistence Controller 411
 - Persistent Container 404
 - Persistent Object Store 405
 - Persistent Store Coordinator 405
 - Private Databases 422
 - Public Database 421
 - Relationships 404
 - tutorial 409
 - View Context 411
 - viewController property 406
 - Core Data Stack 403
 - CPU cores 199
 - curl tool 531
 - Custom Alignment Types 182
 - custom container views
 - Sections 320
 - Custom container views 319
 - Custom Container Views 141
 - custom fonts 139
 - Custom Paths 342
 - Custom Shapes 342
- ## D
- data encapsulation 64
 - Data Isolation 214
 - data race 213
 - Data Races 208
 - Data Storage Quotas 422
 - Debug Navigator 122
 - debug panel 111
 - Debug View Hierarchy 123
 - Declarative Syntax 105
 - Deep Links 477, 480
 - Default Function Parameters 55
 - defer statement 102
 - Detached Tasks 206
 - Developer Mode setting 120
 - Developer Program 3
 - Devices
 - managing 120
 - Dictionary Collections 94
 - Dictionary Entries
 - adding and removing 96
 - Dictionary Initialization 94
 - Dictionary Item Count 96
 - Dictionary Items
 - accessing and updating 96
 - Dictionary Iteration 96
 - didFinishLaunchingWithOptions 525
 - DisclosureGroup 251, 277, 290
 - syntax 282
 - tutorial 285
 - using 281
 - Disclosures 277
 - dismantleUIView() 542
 - Divider view 175
 - do-catch statement 101
 - multiple matches 101
 - Document App
 - creating 385
 - Document Content Type Identifier 387
 - DocumentGroup 128, 385, 386
 - Content Type Identifier 387
 - Document Structure 389
 - Filename Extensions 387
 - File Type Support 387
 - Handler Rank 387
 - Info.plist 397
 - navigation 391
 - overview 385
 - tutorial 397, 409

- Type Identifiers 387
- DocumentGroupLaunchScene 393
 - backgroundAccessoryView 393
 - .frame 393
 - overlayAccessoryView 393
 - .titleLabelFrame 393
- DocumentGroups
 - Exported Type Identifiers 388
 - Imported Type Identifiers 388
- Double 22
- downcasting 31
- DragGesture.Value 364
- drop() modifier 345
- Dynamic Lists 241

E

- easeIn 352
- easeInOut 352
- easeOut 352
- EditButton view 264
- Entity Description 405, 409
 - defining 405, 409
- EntityQuery 485
- enum 82, 99
 - associated values 83
- Enumeration 82
- environment() 163
- environmentObject() 163
- Environment Object 161
 - example 225, 229
- Errata 2
- Error
 - throwing 100
- Error Catching
 - disabling 102
- Error Object
 - accessing 102
- ErrorType protocol 99
- Event handling 141
- exclusive OR 38
- Explicit Animation 354
- Expression Syntax 33

- external parameter names 55

F

- fallthrough statement 50
- FetchDescriptors 435
- Fetches property 404
- fetch() method 407
- Fetch request 404
- FileDocument class 390
- FileWrapper 390
- fill() modifier 339
- Flexible frames. *See* Frames
- Float 22
- flow control 41
- font
 - create custom 139
- footnote 139
- for-await 209
- forced unwrapping 27
- forEach() 93
- ForEach 173, 241, 255, 319
- ForEach(section:) 321
- ForEach(subview:) 320
- foregroundColor() modifier 340
- foregroundColorStyle() modifier 378, 379, 383
- for loop 41
- Form container 256
- Frames 145, 152
 - Geometry Reader 154
 - infinity 153
- function 483
 - arguments 53
 - parameters 53
- Function Parameters
 - variable number of 56
- functions 53
 - as parameters 58
 - default function parameters 55
 - external parameter names 55
 - In-Out Parameters 57
 - parameters as variables 57
 - return multiple results 56

Index

G

GeometryReader 154
gesture() modifier 361
gesture recognizer
 removal of 362
Gesture Recognizers 361
 exclusive 365
 onChanged 362
 sequenced 365
 simultaneous 365
 updating 364
Gestures
 composing 365
getSnapshot() 452
getTimeline() 452
gradient 345
Gradients
 drawing 345
 LinearGradient 346
 RadialGradient 346
Graphics
 drawing 339
 overlays 341
Graphics Drawing 339
Grid 307
 adaptive 296
 alignment 312
 column spanning 312
 empty cells 310
 fixed 296
 flexible 296
 spacing 312
gridCellAnchor() modifier 317
gridCellColumns() modifier 312
gridCellUnsizeAxes() modifier 311
GridItems 295
 adaptive 300
 fixed 301
Grid Layouts 295
GridRow 307
 alignment 315
 empty cells 311

 .gridCellAnchor() modifier 317
 .gridCellColumns() modifier 312
 .gridCellUnsizeAxes() modifier 311
guard statement 45

H

half-closed range operator 36
Handler Rank 387
headline 139
Hierarchical data
 displaying 278
HorizontalAlignment 182, 183
Hosting Controller 557
 adding 560
HStack 136, 145
 conversion to VStack 149

I

if ... else if ... Statements 45
if ... else ... Statements 44
if-let 28
if Statement 44
Image view 145
implicit alignment 177
Implicit Animation 351
implicitly unwrapped 30
Inheritance, Classes and Subclasses 73
init method 65
in keyword 60
inner() modifier 345
inout keyword 58
In-Out Parameters 57
Instance Properties 64
Integers 22
 signed 22
 unsigned 22
IntentTimelineProvider 478
Interface Builder 105
Interpolation
 cardinal 379
 catmullRom 379
 monotone 380

- stepCenter 380
- stepEnd 380
- stepStart 380
- interpolationMethod() modifier 379
- iOS Distribution Certificate 569
- iOS SDK
 - installation 7
 - system requirements 7
- isCancelled property 207
- isEmpty property 208
- is keyword 32

L

- Label view 143
- Layout Hierarchy 123
- Layout Priority 150
- lazy
 - keyword 69
- LazyHGrid 295, 303
- LazyHStack 151
- Lazy properties 68
- Lazy Stacks 151
 - vs. traditional 151
- LazyVGrid 295
- LazyVStack 151
- Left Shift Operator 38
- Library panel 116
- Lifecycle Events 219
- linear 352, 380
- LinearGradient 346
- lineLimit() modifier 150
- LineMark 375
- listRowSeparator() modifier 240
- listRowSeparatorTint() modifier 240
- Lists 239
 - dynamic 241
 - hierarchical 250
 - listRowSeparator() modifier 240
 - listRowSeparatorTint() modifier 240
 - making editable 248
 - refreshable 243
 - separators 240

- listStyle() modifier 288
- List view
 - adding navigation 258
 - .listStyle() modifier 288
 - SidebarListStyle 288
- List view
 - tutorial 253
- Live Activity
 - adding interactivity 517
 - App Intent 517
 - frequent updates 524
 - isStale 519
 - payload 533
 - push notifications 531
 - Push Notifications 521
 - Push Token 527
 - pushTokenUpdates() 527
 - stale date 518
- LiveActivityIntent protocol 517
- Live View 17
- loadPersistentStores() method 406
- localizedStandardContains() 446
- local parameter names 55
- Loops
 - breaking from 43

M

- MainActor 216
- Main.storyboard file 559
- Main Thread , 199
- makeBody() 372
- makeCoordinator() 547, 548
- makeUIView() 542
- Managed Object
 - fetch() method 407
 - saving a 406
 - setting attributes 406
- Managed Object Context 404, 406
- Managed Object Model 404
- Managed Objects 404
 - retrieving 407
- mathematical expressions 33

Index

MeshGradient
 animation 349
 SIMD2 vectors 348

Methods
 declaring 64

minimap 112

Mixed Type Arrays 94

mix() modifier 344

Model Attributes 436

Model Classes 433

Model Configuration 434

Model Container 434, 440

modelContainer(for:) 440

modelContainer(for:) modifier 434

Model Context 434, 440
 delete() 435
 fetch() 435
 insert() 435
 save() 435

Model Relationships 435

modifier() method 140

Modifiers 140

monotone 380

N

Navigation 239
 implementing 228
 tutorial 253

navigationDestination(for:) modifier 245

navigationDestination() modifier 258, 263

NavigationLink 239, 243, 244, 258, 262

navigation path 247

NavigationPath 247, 263
 append() method 247
 removeLast() method 247

NavigationSplitView 251

NavigationStack 239, 244, 258
 navigationDestination(for:) modifier 245
 NavigationPath 247
 path 247

navigationTitle() modifier 247, 261

Network Testing 121

NewDocumentButton 393

new line 24

nil coalescing operator 36

nonisolated keyword 214

NOT (!) operator 35

NSFetchRequest 407, 419

NSPersistentCloudKitContainer 427

NSPersistentContainer 406

NSSortDescriptor 416

O

Objective-C 21

Observable Object
 example 225

ObservableObject 155

ObservableObject protocol 158

Observation 158
 @Bindable 161

Observation Framework 160

onAppear() 357

onAppear modifier 220

onChanged() 362

onChange modifier 221

onDelete() 248, 264

onDisappear modifier 220

onMove() 249, 264

onOpenUrl() 480

Opaque Return Types 71

operands 33

optional
 implicitly unwrapped 30

optional binding 28

Optional Type 27

OR (||) operator 35

OR operator 38

OutlineGroup 251, 277, 289
 tutorial 285
 using 280

overlayAccessoryView 393

Overlays 341

P

- Padding 147
 - padding() modifier 147
 - PageTabViewStyle() 332
 - Parameter Names 55
 - external 55
 - local 55
 - parent class 63
 - Path object 342
 - Paths 342
 - Performance
 - monitoring 122
 - Persistence Container
 - switching to 427
 - Persistence Controller
 - creating 411
 - Persistent Container 404, 406
 - initialization 406
 - Persistent Object Store 405
 - Persistent Store Coordinator 405
 - Physical iOS Device 119
 - running app on 119
 - Picker view 155
 - placeholder() 452, 478
 - Playground 11
 - creating a 11
 - Live View 17
 - pages 17
 - rich text comments 16
 - Rich Text Comments 16
 - Playground editor 12
 - PlaygroundPage 18
 - PlaygroundSupport module 17
 - Playground Timelines 14
 - PlottableValue 375, 378
 - PlottableValue.value 376
 - PointMark 375, 384
 - Predicates 435
 - Predictive Code Completion 193
 - comments 196
 - enabling 193
 - examples 194
 - installing 193
 - prompts 196
 - system requirements 193
 - preferred text size 138
 - Preview Canvas 113
 - Preview on Device 115
 - Preview Pinning 114
 - Private Databases 422
 - Profile in Instruments 123
 - ProgressView 369
 - circular 369, 370
 - CircularProgressViewStyle 370
 - customization 371
 - indeterminate 369, 371
 - linear 369, 370
 - makeBody() 371, 372
 - progressViewStyle() 371
 - ProgressViewStyle 371
 - styles 369
 - progressViewStyle() 371
 - ProgressViewStyle 371
 - Property Wrappers 85
 - example 85
 - Multiple Variables and Types 87
 - Protocols 70
 - Public Database 421
 - push notifications 531
 - curl tool 531
 - troubleshooting 537
 - Push Notifications 521
 - enabling 523
 - Push Token 527
 - pushTokenUpdates() 527
- ## R
- Range Operators 35
 - Record IDs 423
 - Record Zones 423
 - Rectangle() 339
 - RectangleMark 375
 - Reference Types 80
 - Refreshable lists 243
 - refreshable() modifier 243

Index

removeLast() method 247
repeatCount() modifier 353
repeatForever() modifier 353
repeat ... while loop 42
Resume button 114
Right Shift Operator 39
Rotation 170
RuleMark 375
running an app 119

S

scale 359
Scene 127
ScenePhase 221
SceneStorage 231
ScrollView 299
searchable() modifier 444
Segue Action 561
self 69
SF Symbols 143
 macOS app 143
shadow() modifier 345
Shapes 342
 drawing 339
shorthand argument names 61, 93, 173
SidebarListStyle 288
sign bit 39
Signing Identities 9
SIMD2 vectors 348
Simulator
 running app 119
Simulators
 managing 120
sleep() method 200
slide 359
Slider view 168
snapshot() 452, 467, 478
some
 keyword 71
SortDescriptor 435
source code
 download 2
 Spacers 147
 Spacer view 175
 Spacer View 147
 spring() modifier 353
SQLite 403
Stacks 145
 alignment 177
 alignment guides 177
 child limit 149
 cross stack alignment 185
 implicit alignment 177
 Layout Priority 150
State Binding 157
State Objects 159
State properties 155
 binding 156
 example 168
stepCenter 380
stepStart 380
Stored and Computed Properties 67
stored properties 67
String
 data type 23
stroke() modifier 340
StrokeStyle 340
struct keyword 79
Structured Concurrency 199, 200, 209
 addTask() function 207
 async/await 201
 Asynchronous Properties 210
 async keyword 201
 async-let bindings 203
 await keyword 201, 202
 cancelAll() function 208
 cancel() method 207
 Data Races 208
 detached tasks 206
 error handling 204
 for-await 209
 isCancelled property 207
 isEmpty property 208
 priority 206

- suspend point 203
- suspend points 201
- synchronous code 200
- Task Groups 207
- task hierarchy 205
- Task object 202
- Tasks 205
- throw/do/try/catch 204
- withTaskGroup() 207
- withThrowingTaskGroup() 207
- yield() method 207
- Structures 79
- subheadline 139
- subtraction operator 33
- Subviews 136
- suspend points 201, 203
- Swift
 - Actors 213
 - Arithmetic Operators 33
 - array iteration 93
 - arrays 91
 - Assignment Operator 33
 - async/await 201
 - async keyword 201
 - async-let bindings 203
 - await keyword 201, 202
 - base class 73
 - Binary Operators 34
 - Bitwise AND 37
 - Bitwise Left Shift 38
 - Bitwise NOT 37
 - Bitwise Operators 37
 - Bitwise OR 38
 - Bitwise Right Shift 39
 - Bitwise XOR 38
 - Bool 23
 - Boolean Logical Operators 35
 - break statement 43
 - calling a function 54
 - case statement 47
 - character data type 23
 - child class 73
 - class declaration 63
 - class deinitialization 65
 - class extensions 76
 - class hierarchy 73
 - class initialization 65
 - Class Methods 64
 - class properties 63
 - closed range operator 35
 - Closure Expressions 60
 - Closures 61
 - Comparison Operators 34
 - Compound Bitwise Operators 39
 - constant declaration 25
 - constants 25
 - continue statement 43
 - control flow 41
 - data types 21
 - Dictionaries 94
 - do ... while loop 42
 - error handling 99
 - Escape Sequences 24
 - exclusive OR 38
 - expressions 33
 - floating point 22
 - for Statement 41
 - function declaration 53
 - functions 53
 - guard statement 45
 - half-closed range operator 36
 - if ... else ... Statements 44
 - if Statement 44
 - implicit returns 21, 54
 - Inheritance, Classes and Subclasses 73
 - Instance Properties 64
 - instance variables 64
 - integers 22
 - methods 63
 - opaque return types 71
 - operators 33
 - optional binding 28
 - optional type 27
 - Overriding 74

Index

- parent class 73
 - Property Wrappers 85
 - protocols 70
 - Range Operators 35
 - Reference Types 80
 - root class 73
 - single expression functions 54
 - single expression returns 54
 - single inheritance 73
 - Special Characters 24
 - Stored and Computed Properties 67
 - String data type 23
 - structured concurrency 199
 - structures 79
 - subclass 73
 - suspend points 201
 - switch fallthrough 50
 - switch statement 47
 - syntax 47
 - Ternary Operator 36
 - tuples 26
 - type annotations 25
 - type casting 30
 - type checking 30
 - type inference 25
 - Value Types 80
 - variable declaration 25
 - variables 25
 - while loop 42
 - Swift Actors 213
 - SwiftData 433, 439
 - @Attributes 436
 - FetchDescriptors 435
 - @Model 433
 - Model Attributes 436
 - Model Classes 433
 - Model Container 434, 440
 - modelContainer(for:) 440
 - modelContainer(for:) modifier 434
 - Model Context 434, 440
 - Model Relationships 435
 - Predicates 435
 - @Query 441, 445
 - @Relationship 435, 441
 - SortDescriptor 435
 - @Transient 437
 - Swift Playground 11
 - Swift Structures 79
 - SwiftUI
 - create project 109
 - custom views 133
 - data driven 106
 - Declarative Syntax 105
 - example project 165
 - overview 105
 - Subviews 136
 - Views 133
 - SwiftUI Project
 - anatomy of 129
 - creating 109
 - SwiftUI Views 133
 - SwiftUI View template 227
 - SwiftUI vs. UIKit 106
 - switch statement 47
 - example 47
 - switch Statement 47
 - example 47
 - range matching 49
 - symbol() modifier 379
 - synchronous code 200
- ## T
- Tabbed Views 331
 - tabItem() 333
 - Tab Items 333
 - Tab Item Tags 333
 - TabView 331
 - PageTabViewStyle() 332
 - page view style 332
 - tab items 333
 - tag() 333
 - Task.detached() method 206
 - Task Groups 207
 - addTask() function 207

- cancelAll() function 208
- isEmpty property 208
- withTaskGroup() 207
- withThrowingTaskGroup() 207
- Task Hierarchy 205
- task modifier 223
- Task object 202
- Tasks 206
 - cancel() 207
 - detached tasks 206
 - isCancelled property 207
 - overview 205
 - priority 206
- ternary operator 36
- TextField view 171
- Text Styles 138
- Text view
 - adding modifiers 169
 - line limits 150
- Threads
 - overview , 199
- throw statement 100
- timeline() 452, 483, 486
- timeline entries 450
- TimelineEntryRelevance 453
- timeline() method 467
- ToggleButton view 156
- ToolbarItem 248
- toolbar() modifier 248, 261
- transition() modifier 359
- Transitions 351, 358
 - asymmetrical 360
 - combining 359
 - .opacity 359
 - .scale 359
 - .slide 359
- try statement 100
- try! statement 102
- Tuple 26
- TupleView 134
- Tutorial
 - Charts 381

- Type Annotations 25
- type casting 30
- Type Checking 30
- Type Identifiers 387
- Type Inference 25
- type safe programming 25

U

- UIHostingController 557
- UIImagePickerController 549
- UIKit 105
- UIKit integration
 - data sources 544
 - delegates 544
- UIKit Integration 541
 - Coordinator 543
- UINotificationCenter 525
- UInt8 22
- UInt16 22
- UInt32 22
- UInt64 22
- UIRefreshControl 543
- UIScrolledView 544
- UIView 541
 - SwiftUI integration 541
- UIViewController 549
 - SwiftUI integration 549
- UIViewControllerRepresentable protocol 549
- UIViewRepresentable protocol 543
 - makeCoordinator() 543
- unary negative operator 33
- Unicode scalar 25
- Uniform Type Identifier 387
- Unstructured Concurrency 205
 - cancel() method 207
 - detached tasks 206
 - isCancelled property 207
 - priority 206
 - yield() method 207
- upcasting 31
- updateView() 542
- UserDefaults 232

Index

UTI 387

UTType 390

UUID() method 241

V

Value Types 80

variables 25

variadic parameters 56

VerticalAlignment 182, 183

View 128

ViewBuilder 142, 319

 closures 319

View Context 411

viewController property 406

ViewDimensions 183

ViewDimensions object 181

View Hierarchy

 exploring the 123

ViewModifier protocol 140

Views

 adding 227

 as properties 137

 modifying 137

VStack 145

 conversion to HStack 149

W

where clause 30

where statement 49

while Loop 42

WidgetCenter 453

Widget Configuration 449

WidgetConfiguration 449

WidgetConfigurationIntent 450, 483

WidgetConfiguration protocol 449

Widget Configuration Types 450

Widget Entry View 449, 451

Widget Extension 449

widgetFamily 472

Widget kind 449

WidgetKit 471, 477

 Configuration Intent UI 488

 Deep Links 477, 480

 Intent Configuration 449, 450

 introduction 449

 Reload Policy 452

 ReloadPolicy

 .after(Date) 452

 .atEnd 452

 .never 452

 size families 471

 snapshot() 452

 Static Configuration 449, 450

 timeline() 452

 timeline entries 450

 TimelineEntryRelevance 453

 timeline example 462

 timeline() method 467

 Timeline Reload 453

 tutorial 457

 Widget Configuration 449

 WidgetConfiguration protocol 449

 widget entry view 465

 Widget Entry View 449, 451

 Widget Extension 449, 460

 widgetFamily 472

 Widget kind 449

 Widget Provider 452, 467

 Widget Sizes 454

 widget timeline 450

Widget Provider 452

Widget Sizes 454

widget timeline 450

widgetUrl() 479

WindowGroup 128, 130

withAnimation() closure 354

withTaskGroup() 207

withThrowingTaskGroup() 207

X

Xcode

 Attributes inspector 118

 code editor 111

 create project 109

- debug panel 111
- device log 539
- enabling CloudKit 425
- entity editor 405
- installation 7
- Library panel 116
- preferences 8
- preview canvas 113
- Preview Resume button 114
- project navigation panel 111
- SwiftUI mode 109

XOR operator 38

Y

yield() method 207

Z

ZStack 145, 177

- alignment 187

ZStack Custom Alignment 187

