

SwiftUI Essentials

iOS Edition

SwiftUI Essentials – iOS Edition

ISBN-13: 978-1-951442-06-4

© 2019 Neil Smyth / Payload Media, Inc. All Rights Reserved.

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

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

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

Rev: 1.0

Table of Contents

1. Start Here	1
1.1 For Swift Programmers	1
1.2 For Non-Swift Programmers	2
1.3 Source Code Download	2
1.4 Feedback	2
1.5 Errata.....	2
2. Joining the Apple Developer Program	3
2.1 Downloading Xcode 11 and the iOS 13 SDK	3
2.2 Apple Developer Program.....	3
2.3 When to Enroll in the Apple Developer Program?	4
2.4 Enrolling in the Apple Developer Program	4
2.5 Summary	5
3. Installing Xcode 11 and the iOS 13 SDK	7
3.1 Identifying Your macOS Version	7
3.2 Installing Xcode 11 and the iOS 13 SDK	8
3.3 Starting Xcode	8
3.4 Adding Your Apple ID to the Xcode Preferences	8
3.5 Developer and Distribution Signing Identities	9
4. An Introduction to Xcode 11 Playgrounds	11
4.1 What is a Playground?	11
4.2 Creating a New Playground.....	11
4.3 A Basic Swift Playground Example	13
4.4 Viewing Results	15
4.5 Adding Rich Text Comments	16
4.6 Working with Playground Pages	18
4.7 Working with UIKit in Playgrounds	18
4.8 Adding Resources to a Playground	20
4.9 Working with Enhanced Live Views	21
4.10 Summary	23
5. Swift Data Types, Constants and Variables	25
5.1 Using a Swift Playground	26
5.2 Swift Data Types.....	26
5.2.1 Integer Data Types.....	26
5.2.2 Floating Point Data Types.....	27
5.2.3 Bool Data Type.....	27
5.2.4 Character Data Type	27

5.2.5 String Data Type	28
5.2.6 Special Characters/Escape Sequences	29
5.3 Swift Variables	30
5.4 Swift Constants	30
5.5 Declaring Constants and Variables	30
5.6 Type Annotations and Type Inference.....	30
5.7 The Swift Tuple	31
5.8 The Swift Optional Type.....	32
5.9 Type Casting and Type Checking.....	36
5.10 Summary	38
6. Swift Operators and Expressions.....	39
6.1 Expression Syntax in Swift	39
6.2 The Basic Assignment Operator	39
6.3 Swift Arithmetic Operators	39
6.4 Compound Assignment Operators	40
6.5 Comparison Operators.....	41
6.6 Boolean Logical Operators	42
6.7 Range Operators	42
6.8 The Ternary Operator	43
6.9 Bitwise Operators	44
6.9.1 Bitwise NOT.....	44
6.9.2 Bitwise AND.....	45
6.9.3 Bitwise OR	45
6.9.4 Bitwise XOR.....	45
6.9.5 Bitwise Left Shift	46
6.9.6 Bitwise Right Shift.....	46
6.10 Compound Bitwise Operators.....	47
6.11 Summary	47
7. Swift Control Flow.....	49
7.1 Looping Control Flow	49
7.2 The Swift for-in Statement.....	49
7.2.1 The while Loop	50
7.3 The repeat ... while loop	51
7.4 Breaking from Loops	51
7.5 The continue Statement	52
7.6 Conditional Flow Control	52
7.7 Using the if Statement	53
7.8 Using if ... else ... Statements	53
7.9 Using if ... else if ... Statements.....	54
7.10 The guard Statement	54
7.11 Summary	55

8. The Swift Switch Statement.....	57
8.1 Why Use a switch Statement?	57
8.2 Using the switch Statement Syntax	57
8.3 A Swift switch Statement Example	58
8.4 Combining case Statements.....	58
8.5 Range Matching in a switch Statement	59
8.6 Using the where statement	60
8.7 Fallthrough	60
8.8 Summary	61
9. An Overview of Swift 5 Functions, Methods and Closures.....	63
9.1 What is a Function?.....	63
9.2 What is a Method?.....	63
9.3 How to Declare a Swift Function.....	63
9.4 Implicit Returns from Single Expressions.....	64
9.5 Calling a Swift Function	65
9.6 Handling Return Values.....	65
9.7 Local and External Parameter Names.....	65
9.8 Declaring Default Function Parameters.....	66
9.9 Returning Multiple Results from a Function	67
9.10 Variable Numbers of Function Parameters	67
9.11 Parameters as Variables.....	68
9.12 Working with In-Out Parameters.....	68
9.13 Functions as Parameters.....	69
9.14 Closure Expressions.....	71
9.15 Closures in Swift.....	73
9.16 Summary	73
10. The Basics of Object-Oriented Programming in Swift	75
10.1 What is an Instance?	75
10.2 What is a Class?.....	75
10.3 Declaring a Swift Class	75
10.4 Adding Instance Properties to a Class.....	76
10.5 Defining Methods.....	76
10.6 Declaring and Initializing a Class Instance	78
10.7 Initializing and Deinitializing a Class Instance	78
10.8 Calling Methods and Accessing Properties	79
10.9 Stored and Computed Properties	80
10.10 Lazy Stored Properties	81
10.11 Using self in Swift	82
10.12 Understanding Swift Protocols	84
10.13 Opaque Return Types	85

10.14 Summary	86
11. An Introduction to Swift Subclassing and Extensions	87
11.1 Inheritance, Classes and Subclasses	87
11.2 A Swift Inheritance Example	87
11.3 Extending the Functionality of a Subclass	88
11.4 Overriding Inherited Methods	89
11.5 Initializing the Subclass	90
11.6 Using the SavingsAccount Class	91
11.7 Swift Class Extensions	91
11.8 Summary	92
12. An Introduction to Swift Structures	93
12.1 An Overview of Swift Structures	93
12.2 Value Types vs. Reference Types	94
12.3 When to use Structures or Classes	96
12.4 Summary	96
13. An Introduction to Swift Property Wrappers	97
13.1 Understanding Property Wrappers	97
13.2 A Simple Property Wrapper Example	97
13.3 Supporting Multiple Variables and Types	99
13.4 Summary	102
14. Working with Array and Dictionary Collections in Swift	103
14.1 Mutable and Immutable Collections	103
14.2 Swift Array Initialization	103
14.3 Working with Arrays in Swift	104
14.3.1 Array Item Count	104
14.3.2 Accessing Array Items	105
14.4 Random Items and Shuffling	105
14.5 Appending Items to an Array	105
14.5.1 Inserting and Deleting Array Items	105
14.6 Array Iteration	106
14.7 Creating Mixed Type Arrays	106
14.8 Swift Dictionary Collections	107
14.9 Swift Dictionary Initialization	107
14.10 Sequence-based Dictionary Initialization	108
14.11 Dictionary Item Count	108
14.12 Accessing and Updating Dictionary Items	109
14.13 Adding and Removing Dictionary Entries	109
14.14 Dictionary Iteration	109
14.15 Summary	110

15. Understanding Error Handling in Swift 5	111
15.1 Understanding Error Handling	111
15.2 Declaring Error Types	111
15.3 Throwing an Error	112
15.4 Calling Throwing Methods and Functions	113
15.5 Accessing the Error Object.....	114
15.6 Disabling Error Catching.....	114
15.7 Using the defer Statement.....	114
15.8 Summary	115
16. An Overview of SwiftUI	117
16.1 UIKit and Interface Builder.....	117
16.2 SwiftUI Declarative Syntax.....	118
16.3 SwiftUI is Data Driven	118
16.4 SwiftUI vs. UIKit.....	119
16.5 Summary	120
17. Using Xcode in SwiftUI Mode.....	121
17.1 Starting Xcode 11	121
17.2 Creating a SwiftUI Project	122
17.3 Xcode in SwiftUI Mode.....	122
17.4 The Preview Canvas	123
17.5 Preview Pinning.....	125
17.6 Modifying the Design	125
17.7 Editor Context Menu.....	129
17.8 Previewing on Multiple Device Configurations.....	129
17.9 Running the App on a Simulator	131
17.10 Running the App on a Physical iOS Device.....	132
17.11 Managing Devices and Simulators	134
17.12 Enabling Network Testing	135
17.13 Dealing with Build Errors	135
17.14 Monitoring Application Performance	135
17.15 Exploring the User Interface Layout Hierarchy	136
17.16 Summary	139
18. The Anatomy of a Basic SwiftUI Project.....	141
18.1 Creating an Example Project.....	141
18.2 UIKit and SwiftUI	141
18.3 The AppDelegate.swift File	141
18.4 The SceneDelegate.swift File	142
18.5 ContentView.swift File	144
18.6 Assets.xcassets	144
18.7 Info.plist	144

18.8 LaunchScreen.storyboard	144
18.9 Summary	144
19. Creating Custom Views with SwiftUI	145
19.1 SwiftUI Views	145
19.2 Creating a Basic View	145
19.3 Adding Additional Views	146
19.4 Working with Subviews	148
19.5 Views as Properties	149
19.6 Modifying Views	150
19.7 Working with Text Styles	150
19.8 Modifier Ordering	152
19.9 Custom Modifiers	153
19.10 Basic Event Handling	153
19.11 The onAppear and onDisappear Methods	154
19.12 Building Custom Container Views	155
19.13 Summary	156
20. SwiftUI Stacks and Frames	157
20.1 SwiftUI Stacks	157
20.2 Spacers, Alignment and Padding	159
20.3 Container Child Limit	161
20.4 Text Line Limits and Layout Priority	162
20.5 SwiftUI Frames	164
20.6 Frames and the Geometry Reader	166
20.7 Summary	167
21. Working with SwiftUI State, Observable and Environment Objects	169
21.1 State Properties	169
21.2 State Binding	171
21.3 Observable Objects	172
21.4 Environment Objects	174
21.5 Summary	176
22. A SwiftUI Example Tutorial	177
22.1 Creating the Example Project	177
22.2 Reviewing the Project	178
22.3 Adding a VStack to the Layout	180
22.4 Adding a Slider View to the Stack	180
22.5 Adding a State Property	181
22.6 Adding Modifiers to the Text View	182
22.7 Adding Rotation and Animation	183
22.8 Adding a TextField to the Stack	185

22.9 Adding a Color Picker	185
22.10 Tidying the Layout.....	187
22.11 Summary	190
23. SwiftUI Observable and Environment Objects – A Tutorial.....	191
23.1 About the ObservableDemo Project.....	191
23.2 Creating the Project	191
23.3 Adding the Observable Object.....	191
23.4 Designing the ContentView Layout.....	192
23.5 Adding the Second View	194
23.6 Adding Navigation.....	195
23.7 Using an Environment Object	196
23.8 Summary	198
24. SwiftUI Stack Alignment and Alignment Guides	199
24.1 Container Alignment.....	199
24.2 Alignment Guides.....	201
24.3 Using the Alignment Guides Tool.....	205
24.4 Custom Alignment Types	206
24.5 Cross Stack Alignment.....	209
24.6 ZStack Custom Alignment	212
24.7 Summary	215
25. SwiftUI Lists and Navigation	217
25.1 SwiftUI Lists	217
25.2 SwiftUI Dynamic Lists.....	219
25.3 SwiftUI NavigationView and NavigationLink.....	221
25.4 Making the List Editable.....	223
25.5 Summary	226
26. A SwiftUI List and Navigation Tutorial	227
26.1 About the ListNavDemo Project	227
26.2 Creating the ListNavDemo Project.....	227
26.3 Preparing the Project	227
26.4 Adding the Car Structure	228
26.5 Loading the JSON Data.....	229
26.6 Adding the Data Store.....	230
26.7 Designing the Content View.....	230
26.8 Designing the Detail View	233
26.9 Adding Navigation to the List.....	235
26.10 Designing the Add Car View.....	236
26.11 Implementing Add and Edit Buttons.....	239
26.12 Adding the Edit Button Methods	241

26.13 Summary	242
27. Building Tabbed Views in SwiftUI	243
27.1 An Overview of SwiftUI TabView	243
27.2 Creating the TabViewDemo App	244
27.3 Adding the TabView Container	244
27.4 Adding the Content Views	244
27.5 Adding the Tab Items	244
27.6 Adding Tab Item Tags	245
27.7 Summary	246
28. Building Context Menus in SwiftUI	247
28.1 Creating the ContextMenuDemo Project	247
28.2 Preparing the Content View	247
28.3 Adding the Context Menu	248
28.4 Testing the Context Menu	249
28.5 Summary	250
29. Basic SwiftUI Graphics Drawing	251
29.1 Creating the DrawDemo Project	251
29.2 SwiftUI Shapes	251
29.3 Using Overlays	254
29.4 Drawing Custom Paths and Shapes	254
29.5 Drawing Gradients	257
29.6 Summary	260
30. SwiftUI Animation and Transitions	261
30.1 Creating the AnimationDemo Example Project	261
30.2 Implicit Animation	261
30.3 Repeating an Animation	264
30.4 Explicit Animation	264
30.5 Animation and State Bindings	265
30.6 Automatically Starting an Animation	266
30.7 SwiftUI Transitions	269
30.8 Combining Transitions	270
30.9 Asymmetrical Transitions	271
30.10 Summary	271
31. Working with Gesture Recognizers in SwiftUI	273
31.1 Creating the GestureDemo Example Project	273
31.2 Basic Gestures	273
31.3 The onChange Action Callback	275
31.4 The updating Callback Action	276

31.5 Composing Gestures	278
31.6 Summary	280
32. Integrating UIViews with SwiftUI	281
32.1 SwiftUI and UIKit Integration	281
32.2 Integrating UIViews into SwiftUI	282
32.3 Adding a Coordinator	283
32.4 Handling UIKit Delegation and Data Sources	284
32.5 An Example Project	286
32.6 Wrapping the UIScrollView	286
32.7 Implementing the Coordinator	287
32.8 Using MyScrollView	288
32.9 Summary	289
33. Integrating UIViewController with SwiftUI	291
33.1 UIViewControllers and SwiftUI	291
33.2 Creating the ViewControllerDemo project	291
33.3 Wrapping the UIImagePickerController	291
33.4 Designing the Content View	293
33.5 Completing MyImagePicker	294
33.6 Completing the Content View	296
33.7 Testing the App	297
33.8 Summary	297
34. Integrating SwiftUI with UIKit	299
34.1 An Overview of the Hosting Controller	299
34.2 A UIHostingController Example Project	300
34.3 Adding the SwiftUI Content View	300
34.4 Preparing the Storyboard	301
34.5 Adding a Hosting Controller	302
34.6 Configuring the Segue Action	304
34.7 Embedding a Container View	306
34.8 Embedding SwiftUI in Code	308
34.9 Summary	310
35. Preparing and Submitting an iOS 13 Application to the App Store	311
35.1 Verifying the iOS Distribution Certificate	311
35.2 Adding App Icons	313
35.3 Designing the Launch Screen	314
35.4 Assign the Project to a Team	314
35.5 Archiving the Application for Distribution	315
35.6 Configuring the Application in iTunes Connect	316
35.7 Validating and Submitting the Application	317

35.8 Configuring and Submitting the App for Review320

Index321

1. Start Here

The goal of this book is to teach the skills necessary to build iOS 13 applications using SwiftUI, Xcode 11 and the Swift 5 programming language.

Beginning with the basics, this book provides an outline of the steps necessary to set up an iOS development environment together with an introduction to the use of Swift Playgrounds to learn and experiment with Swift.

The book also includes in depth chapters introducing the Swift 5 programming language including data types, control flow, functions, object-oriented programming, property wrappers and error handling.

An introduction to the key concepts of SwiftUI and project architecture is followed by a guided tour of Xcode in SwiftUI development mode. The book also covers the creation of custom SwiftUI views and explains how these views are combined to create user interface layouts including the use of stacks, frames and forms.

Other topics covered include data handling using state properties and both observable and environment objects, as are key user interface design concepts such as modifiers, lists, tabbed views, context menus and user interface navigation.

The book also includes chapters covering graphics drawing, user interface animation, view transitions and gesture handling.

Chapters are also provided explaining how to integrate SwiftUI views into existing UIKit-based projects and explains the integration of UIKit code into SwiftUI.

Finally, the book explains how to package up a completed app and upload it to the App Store for publication.

Along the way, the topics covered in the book are put into practice through detailed tutorials, the source code for which is also available for download.

The aim of this book, therefore, is to teach you the skills necessary to build your own apps for iOS 13 using SwiftUI. Assuming you are ready to download the iOS 13 SDK and Xcode 11 and have an Intel-based Mac you are ready to get started.

1.1 For Swift Programmers

This book has been designed to address the needs of both existing Swift programmers and those who are new to both Swift and iOS app development. If you are familiar with the Swift 5.1 programming language, you can probably skip the Swift specific chapters. If you are not yet familiar

Start Here

with the new language features of Swift 5.1, however, we recommend that you at least read the sections covering *implicit returns from single expressions*, *opaque return types* and *property wrappers*. These features are central to the implementation and understanding of SwiftUI.

1.2 For Non-Swift Programmers

If you are new to programming in Swift (or programming in general) then the entire book is appropriate for you. Just start at the beginning and keep going.

1.3 Source Code Download

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

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

1.4 Feedback

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

1.5 Errata

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

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

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

2. Joining the Apple Developer Program

The first step in the process of learning to develop iOS 13 based applications involves gaining an understanding of the advantages of enrolling in the Apple Developer Program and deciding the point at which it makes sense to pay to join. With these goals in mind, this chapter will outline the costs and benefits of joining the developer program and, finally, walk through the steps involved in enrolling.

2.1 Downloading Xcode 11 and the iOS 13 SDK

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

2.2 Apple Developer Program

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

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

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

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

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

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

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

2.3 When to Enroll in the Apple Developer Program?

Clearly, there are many benefits to Apple Developer Program membership and, eventually, membership will be necessary to begin selling applications. As to whether to pay the enrollment fee now or later will depend on individual circumstances. If you are still in the early stages of learning to develop iOS applications or have yet to come up with a compelling idea for an application to develop then much of what you need is provided without program membership. As your skill level increases and your ideas for applications to develop take shape you can, after all, always enroll in the developer program later.

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

2.4 Enrolling in the Apple Developer Program

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

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

Apple provides enrollment options for businesses and individuals. To enroll as an individual, you will need to provide credit card information in order to verify your identity. To enroll as a company, you must have legal signature authority (or access to someone who does) and be able to provide documentation such as a Dun & Bradstreet D-U-N-S number and documentation confirming legal entity status.

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

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

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

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

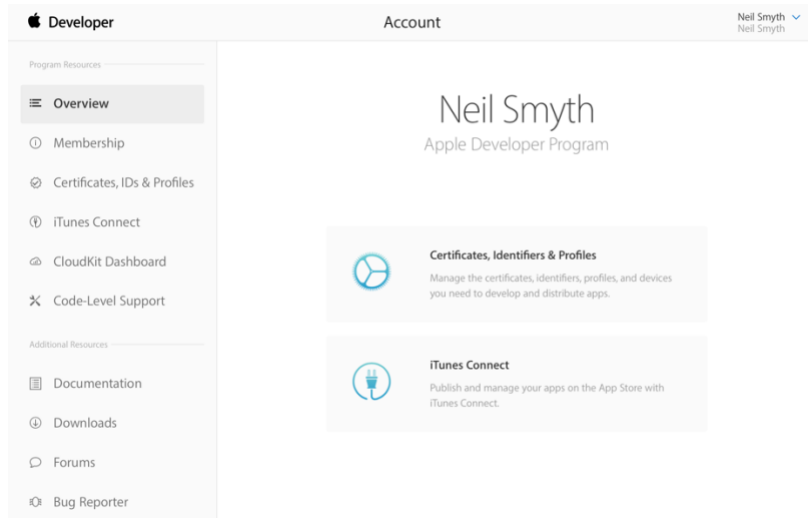


Figure 2-1

2.5 Summary

An important early step in the iOS 13 application development process involves identifying the best time to enroll in the Apple Developer Program. This chapter has outlined the benefits of joining the program, provided some guidance to keep in mind when considering developer program membership and walked briefly through the enrollment process. The next step is to download and install the iOS 13 SDK and Xcode 11 development environment.

3. Installing Xcode 11 and the iOS 13 SDK

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

In this chapter we will cover the steps involved in installing both Xcode 11 and the iOS 13 SDK on macOS.

3.1 Identifying Your macOS Version

When developing with SwiftUI, the Xcode 11 environment requires that the version of macOS running on the system be version 10.15 or later. If you are unsure of the version of macOS on your Mac, you can find this information by clicking on the Apple menu in the top left-hand corner of the screen and selecting the *About This Mac* option from the menu. In the resulting dialog check the *Version* line.

If the "About This Mac" dialog does not indicate that macOS 10.15 or later is running, click on the *Software Update...* button to download and install the appropriate operating system upgrades.



Figure 3-1

3.2 Installing Xcode 11 and the iOS 13 SDK

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

3.3 Starting Xcode

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

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

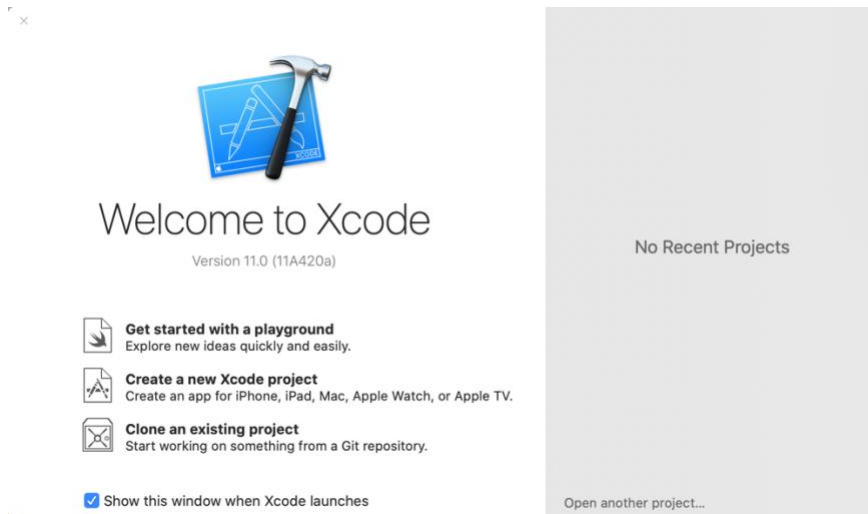


Figure 3-2

3.4 Adding Your Apple ID to the Xcode Preferences

Regardless of whether or not you choose to enroll in the Apple Developer Program it is worth adding your Apple ID to Xcode now that it is installed and running. Select the *Xcode -> Preferences...* menu option 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 associated password and click on the *Sign In* button to add the account to the preferences.

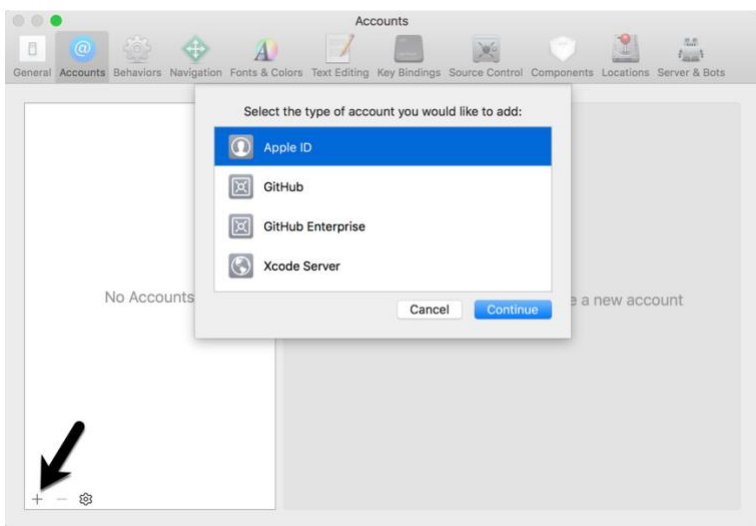


Figure 3-3

3.5 Developer and Distribution Signing Identities

Once the Apple ID has been entered the next step is to generate signing identities. To view the current signing identities, select the newly added Apple ID in the Accounts panel and click on the *Manage Certificates...* button at which point a list of available signing identities will be listed. To create a signing identity, simply click on the + button highlighted in Figure 3-4 and make the appropriate selection from the menu:

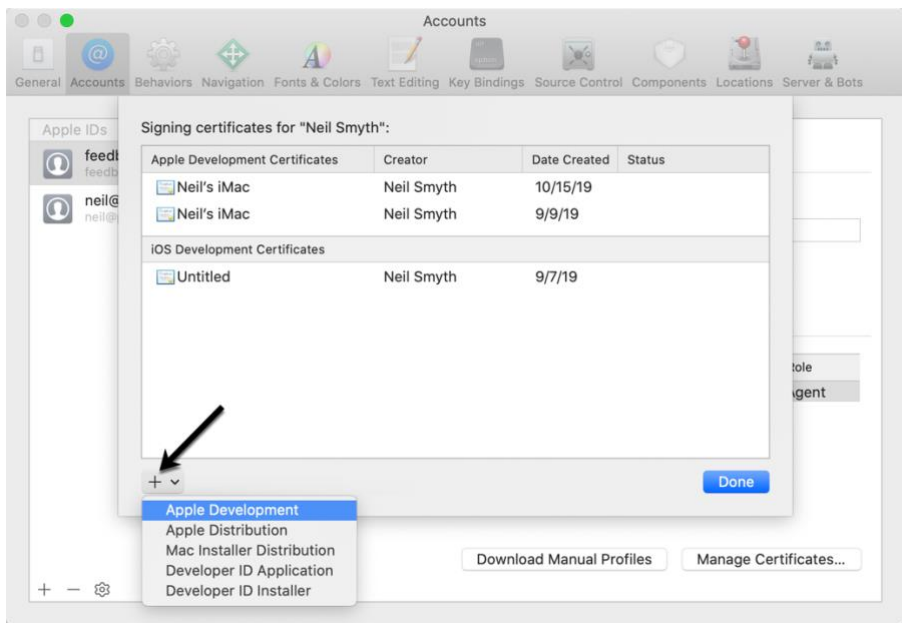


Figure 3-4

Installing Xcode 11 and the iOS 13 SDK

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. 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.

Having installed the iOS SDK and successfully launched Xcode 11 we can now look at Xcode in more detail, starting with Playgrounds.

4. An Introduction to Xcode 11 Playgrounds

Before introducing the Swift programming language in the chapters that follow, it is first worth learning about a feature of Xcode known as *Playgrounds*. 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 *Get started with a playground* option from the welcome screen or select the *File -> New -> Playground...* menu option. Choose the iOS option on the resulting panel and select the Blank template.

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

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

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

An Introduction to Xcode 11 Playgrounds

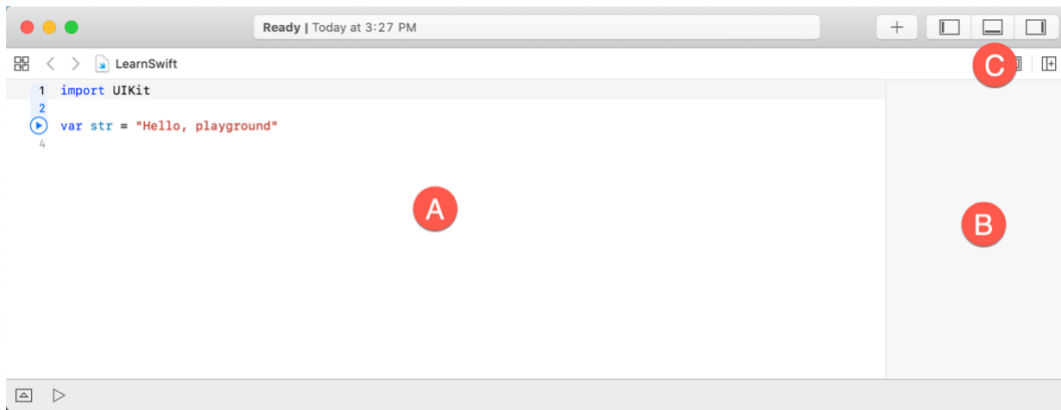


Figure 4-1

The panel on the left-hand side of the window (marked A in Figure 4-1) is the *playground editor* where the lines of Swift code are entered. The right-hand panel (marked B) is referred to as the *results panel* and is where the results of each Swift expression entered into the playground editor panel are displayed.

The cluster of three buttons at the right-hand side of the toolbar (marked C) are used to hide and display other panels within the playground window. The left most button displays the Navigator panel which provides access to the folders and files that make up the playground (marked A in Figure 4-2 below). The middle button, on the other hand, displays the Debug view (B) which displays code output and information about coding or runtime errors. The right most button displays the Utilities panel (C) where a variety of properties relating to the playground may be configured.

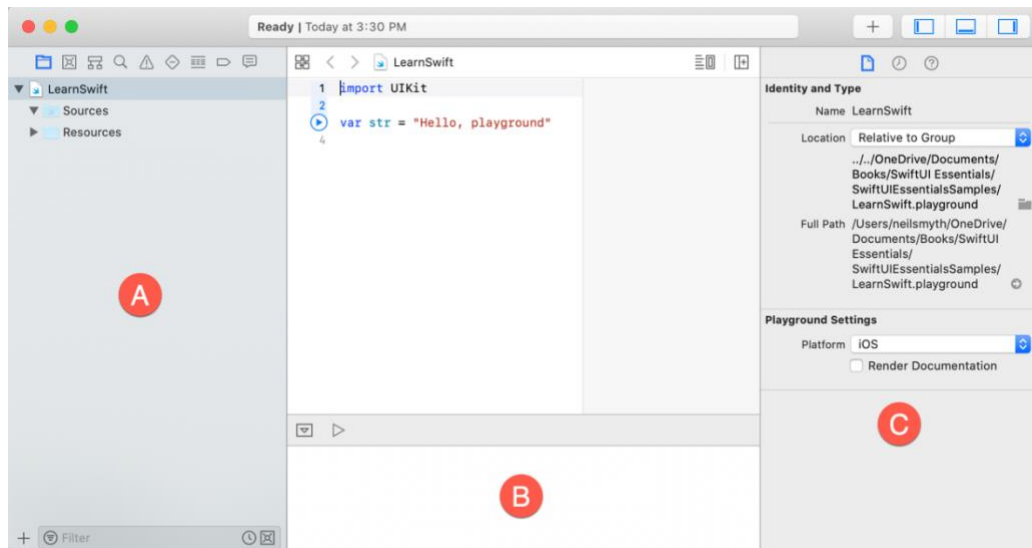


Figure 4-2

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

4.3 A Basic Swift Playground Example

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

```
import UIKit

var str = "Hello , playground"

print("Welcome to Swift")
```

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

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

An Introduction to Xcode 11 Playgrounds

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:

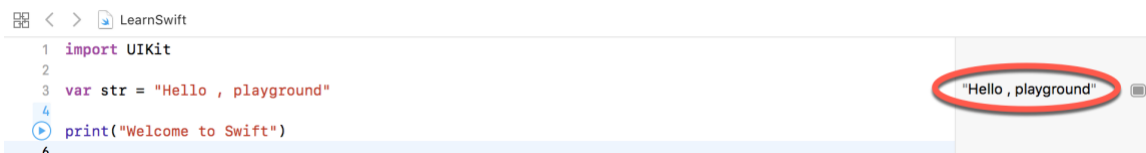


Figure 4-7

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 panel:

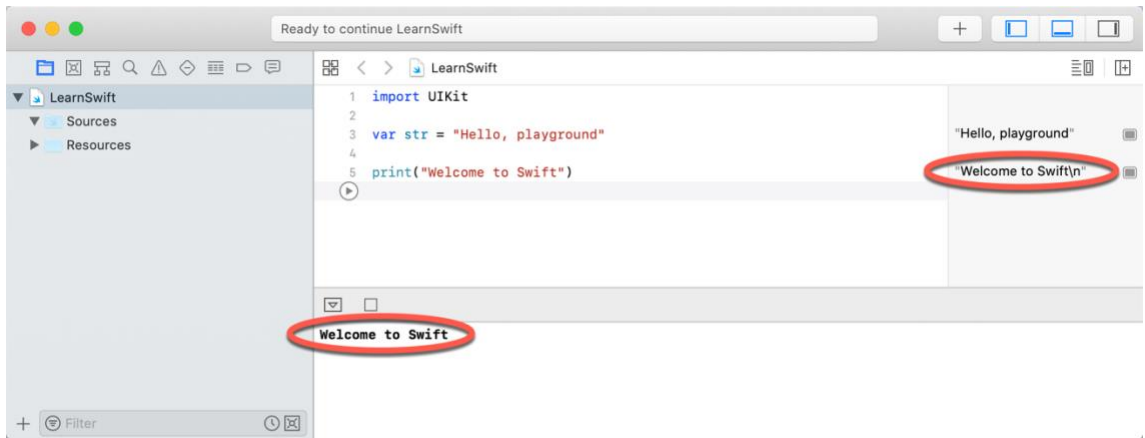


Figure 4-8

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
    print(y)
}
```

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 number of times the loop was performed. More interesting information, however, may be obtained by hovering the mouse pointer over the results line so that two additional buttons appear as shown in Figure 4-9:

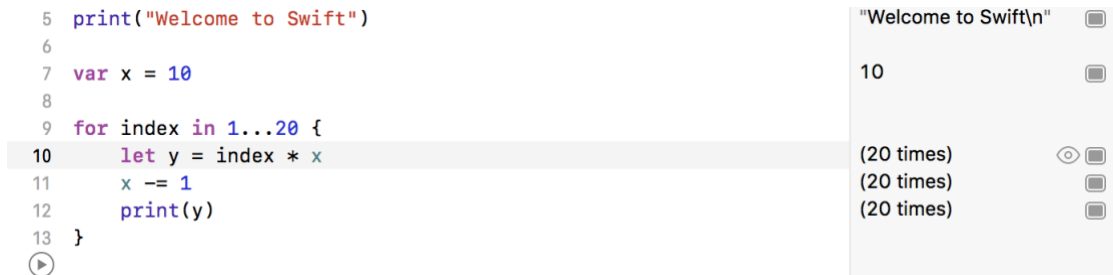


Figure 4-9

An Introduction to Xcode 11 Playgrounds

The left most of the two buttons is the *Quick Look* button which, when selected, will show a popup panel displaying the results as shown in Figure 4-10:

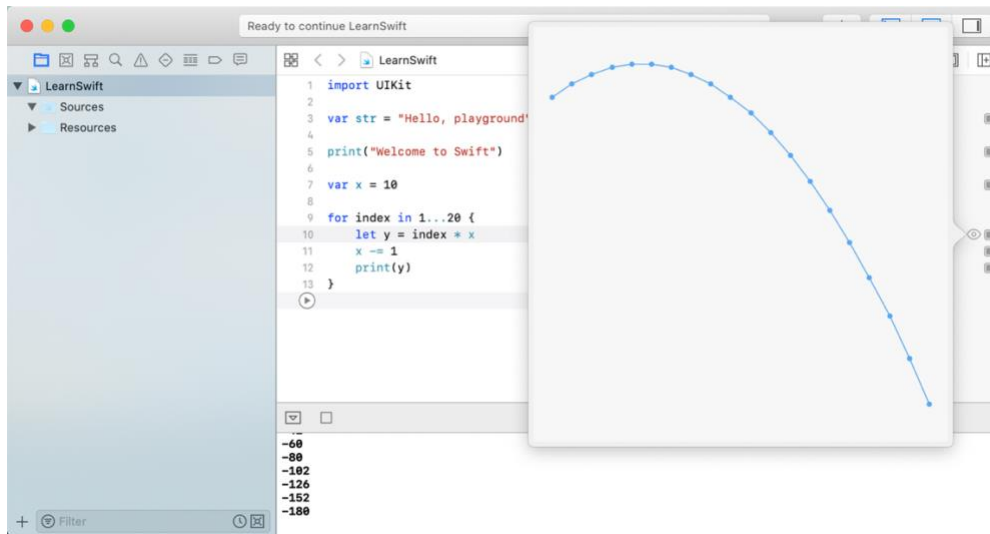


Figure 4-10

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

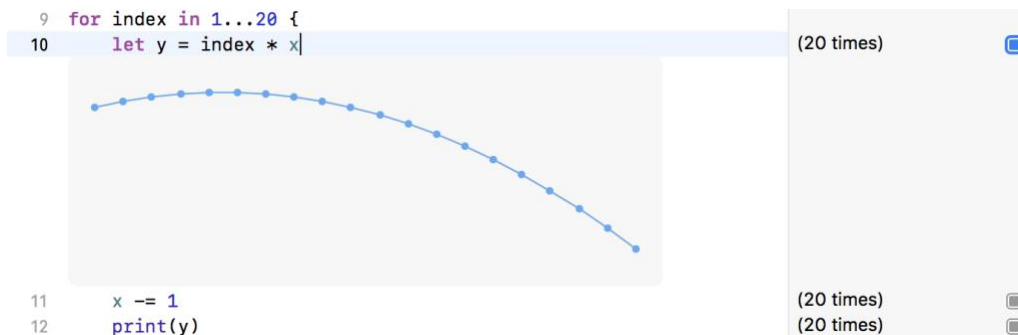


Figure 4-11

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 are the ability to embed images and hyperlinks into the content of a rich text comment.

To see rich text comments in action, enter the following 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 C in Figure 4-2). If the Inspector panel is not currently visible, click on the right most of the three view buttons (marked C 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 an additional page to the playground now by selecting the LearnSwift entry at the top of the Navigator panel, right-clicking and selecting the *New Playground Page* menu option. If the Navigator panel is not currently visible, click on the left most of the three view buttons (marked C in Figure 4-1) 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 *UIKit Examples* as outlined in Figure 4-13:

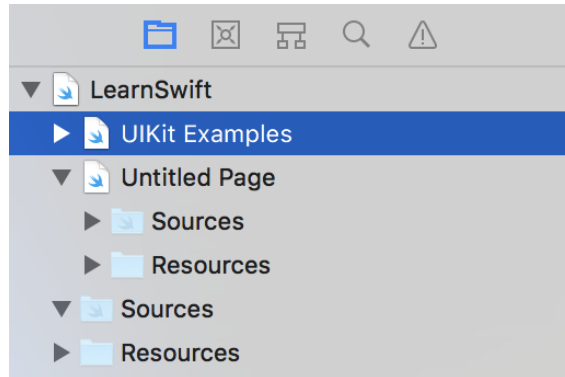


Figure 4-13

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 UIKit in Playgrounds

Prior to the introduction of SwiftUI, iOS apps were developed using UIKit and a range of UIKit-based frameworks. Unsurprisingly, UIKit can be used within Xcode playgrounds.

While it is also possible to use SwiftUI within a playground, this requires some additional steps that involve integrating SwiftUI into UIKit (a topic which is not covered until much later in this book). In practice, however, Xcode provides a Live Preview canvas which provides many of the features of playgrounds when developing with SwiftUI. As will be discussed later, it may still be necessary to use UIKit when working with SwiftUI, so for completeness, the use of UIKit in a playground will be covered in this section.

When working with UIKit within a playground page it is necessary to import the iOS UIKit Framework. The UIKit Framework contains most of the classes necessary to implement user interfaces for iOS applications when not using SwiftUI. An extremely powerful feature of playgrounds is that it is also possible to work with UIKit along with many of the other frameworks that comprise the iOS SDK.

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

```
import UIKit

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

myLabel.backgroundColor = UIColor.red
myLabel.text = "Hello Swift"
myLabel.textAlignment = .center
myLabel.font = UIFont(name: "Georgia", size: 24)
myLabel
```

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

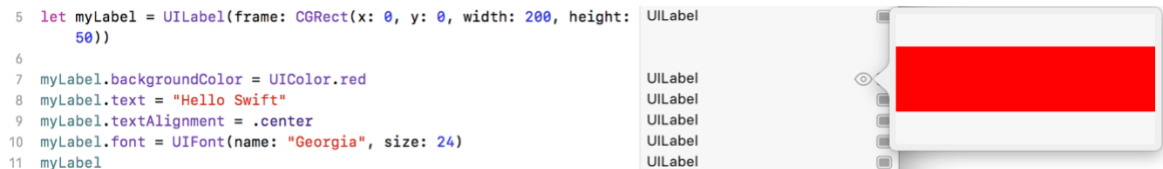


Figure 4-14

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

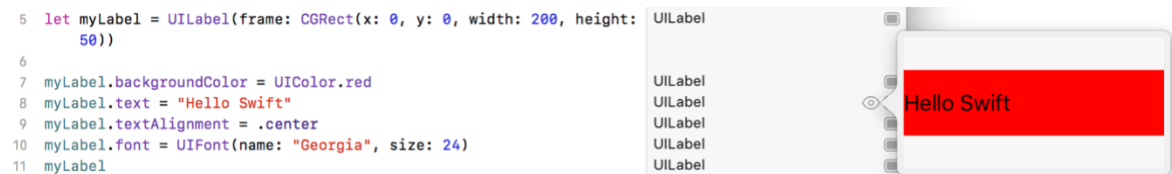


Figure 4-15

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

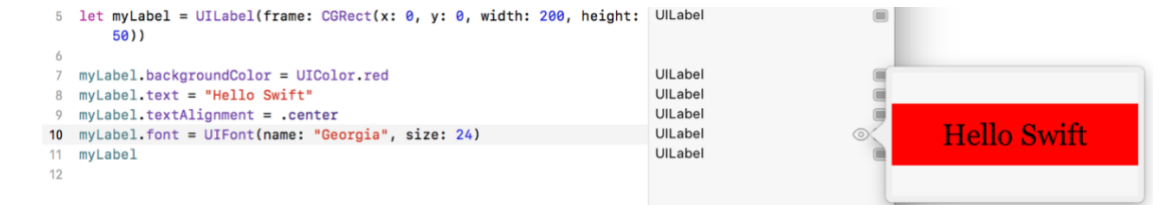


Figure 4-16

4.8 Adding Resources to a Playground

Another useful feature of playgrounds is the ability to bundle and access resources such as image files in a playground. Within the Navigator panel, click on the right facing arrow (known as a *disclosure arrow*) to the left of the UIKit Examples page entry to unfold the page contents (Figure 4-17) and note the presence of a folder named *Resources*:

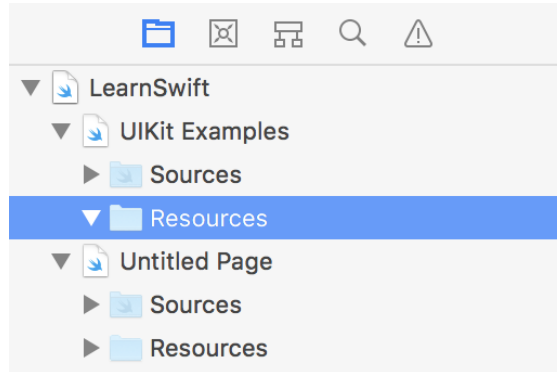


Figure 4-17

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

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

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

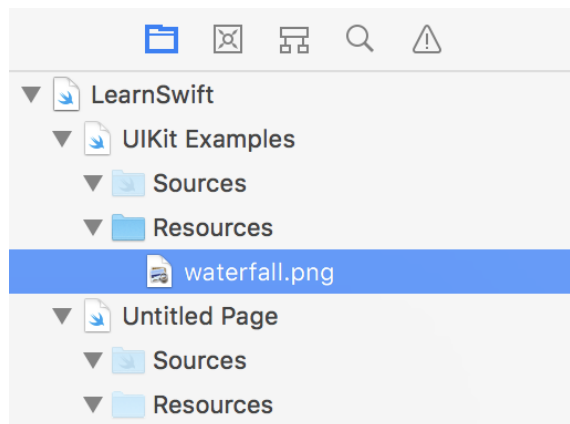


Figure 4-18

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

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


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



Figure 4-19

4.9 Working with Enhanced Live Views

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

```
import UIKit
import PlaygroundSupport
```

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

Beneath the import statements, add the following code:

```
import UIKit
import PlaygroundSupport

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

container.addSubview(square)

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

An Introduction to Xcode 11 Playgrounds

The code creates a `UIView` object to act as a container view and assigns it a white background color. A smaller view is then drawn positioned in the center of the container view and colored red. The second view is then added as a child of the container view. An animation is then used to change the color of the smaller view to blue and to rotate it through 360 degrees.

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

The `PlaygroundSupport` module includes a class named `PlaygroundPage` that allows playground code to interact with the pages that make up a playground. This is achieved through a range of methods and properties of the class, one of which is the *current* property. This property, in turn, provides access to the current playground page. In order to execute the 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 4-20:

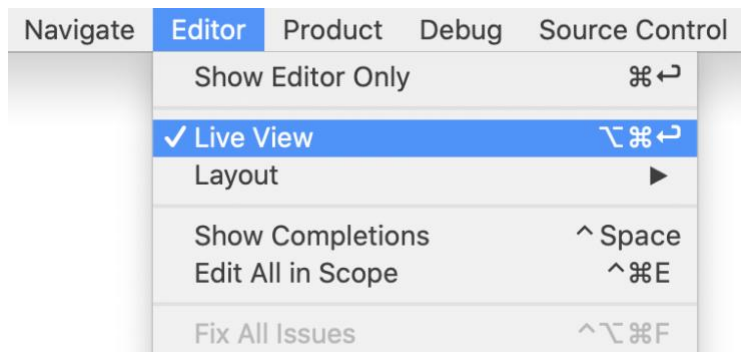


Figure 4-20

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

```
import UIKit
import PlaygroundSupport

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

PlaygroundPage.current.liveView = container

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

container.addSubview(square)

UIView.animate(withDuration: 5.0, animations: {
```

```

square.backgroundColor = UIColor.blue
let rotation = CGAffineTransform(rotationAngle: 3.14)
square.transform = rotation
})

```

Once the call has been added, re-execute the code at which point the views should appear in the timeline (Figure 4-21). During the 5 second animation duration, the red square should rotate through 360 degrees while gradually changing color to blue:

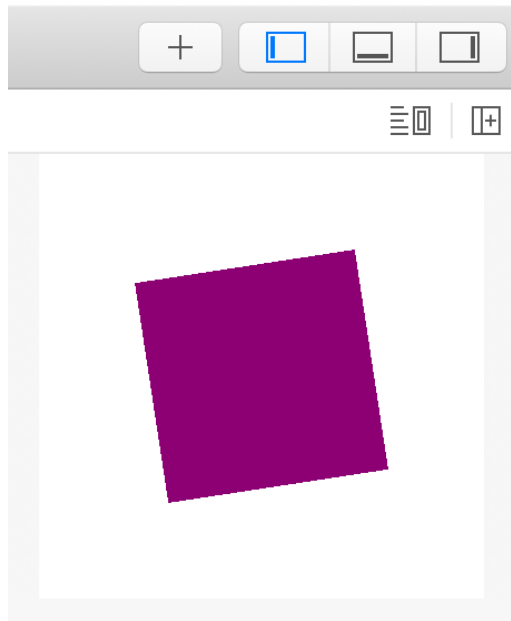


Figure 4-21

To repeat the execution of the code in the playground page, click on the stop button highlighted in Figure 4-6 to reset the playground and change the stop button into the run button (Figure 4-3). Click the run button to repeat the execution.

4.10 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.

5. Swift Data Types, Constants and Variables

If you are new to the Swift programming language then the next few chapters are recommended reading. Although SwiftUI makes the development of apps easier, it will still be necessary to learn Swift programming both to understand SwiftUI and develop fully functional apps.

If, on the other hand, you are familiar with the Swift programming language you can skip the Swift specific chapters that follow (though if you are not familiar with the new features of Swift 5.1 you should at least read the sections and chapters relating to *implicit returns from single expressions*, *opaque return types* and *property wrappers* before moving on to the SwiftUI chapters).

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

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

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

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

5.1 Using a Swift Playground

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

5.2 Swift Data Types

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

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

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

```
var mynumber = 10
```

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

```
1010
```

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

5.2.1 Integer Data Types

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

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

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

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

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

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

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

5.2.2 Floating Point Data Types

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

5.2.3 Bool Data Type

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

5.2.4 Character Data Type

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

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

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

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

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

5.2.5 String Data Type

The String data type is a sequence of characters that typically make up a word or sentence. In addition to providing a storage mechanism, the String data type also includes a range of string manipulation features allowing strings to be searched, matched, concatenated and modified. Strings in Swift are represented internally as collections of characters (where a character is, as previously discussed, comprised of one or more Unicode scalar values).

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

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

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

print(message)
```

When executed, the code will output the following message:

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

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

```
var multiline = """

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

    "I thought you said you knew how to fly this!" yelled Mary.

    "It was much easier on the simulator" replied her brother,
    trying to keep the panic out of his voice.

    """

print(multiline)
```

The above code will generate the following output when run:

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



```
"I thought you said you knew how to fly this!" yelled Mary.
```

```
"It was much easier on the simulator" replied her brother,  
trying to keep the panic out of his voice.
```

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

5.2.6 Special Characters/Escape Sequences

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

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

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

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

Commonly used special characters supported by Swift are as follows:

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

5.3 Swift Variables

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

5.4 Swift Constants

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

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

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

5.5 Declaring Constants and Variables

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

```
var userCount = 10
```

Constants are declared using the `let` keyword.

```
let maxUserCount = 20
```

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

5.6 Type Annotations and Type Inference

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

There are two ways in which the type of a constant or variable will be identified. One approach is to use a *type annotation* at the point the variable or constant is declared in the code. This is achieved

by placing a colon after the constant or variable name followed by the type declaration. The following line of code, for example, declares a variable named `userCount` as being of type `Int`:

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

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

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

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

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

```
let bookTitle = "SwiftUI Essentials"
```

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

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

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

5.7 The Swift Tuple

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

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

Swift Data Types, Constants and Variables

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

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

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

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

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

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

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

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

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

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

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

5.8 The Swift Optional Type

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

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

```
var index: Int?
```

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

An optional can easily be tested (typically using an `if` statement) to identify whether it has a value assigned to it as follows:

```
var index: Int?

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

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

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

```
var index: Int?

index = 3

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

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

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

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

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

```
Value of optional type 'Int?' must be unwrapped to a value of type 'Int'
```

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

```
if let constantname = optionalName {

}

if var variablename = optionalName {
```

```
}
```

The above constructs perform two tasks. In the first instance, the statement ascertains whether the designated optional contains a value. Second, in the event that the optional has a value, that value is assigned to the declared constant or variable and the code within the body of the statement is executed. The previous forced unwrapping example could, therefore, be modified as follows to use optional binding instead:

```
var index: Int?

index = 3

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

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

In this case the value assigned to the `index` variable is unwrapped and assigned to a temporary constant named *myvalue* which is then used as the index reference into the array. Note that the `myvalue` constant is described as temporary since it is only available within the scope of the `if` statement. Once the `if` statement completes execution, the constant will no longer exist. For this reason, there is no conflict in using the same temporary name as that assigned to the optional. The following is, for example, valid code:

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

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

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

}
```

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

```

var pet1: String?
var pet2: String?

pet1 = "cat"
pet2 = "dog"

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

```

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

```

if let firstPet = pet1, let secondPet = pet2, petCount > 1 {
    print(firstPet)
    print(secondPet)
} else {
    print("insufficient pets")
}

```

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

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

```

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

index = 3

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

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

```

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

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

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

5.9 Type Casting and Type Checking

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

The following code, for example, lets the compiler know that the value returned from the `object(forKey:)` method needs to be treated as a `String` type:

```
let myValue = record.object(forKey: "comment") as! String
```

In fact, there are two types of casting which are referred to as *upcasting* and *downcasting*. Upcasting occurs when an object of a particular class is cast to one of its superclasses. Upcasting is performed using the `as` keyword and is also referred to as *guaranteed conversion* since the compiler can tell from the code that the cast will be successful. The `UIButton` class, for example, is a subclass of the `UIControl` class as shown in the fragment of the `UIKit` class hierarchy shown in Figure 5-1:

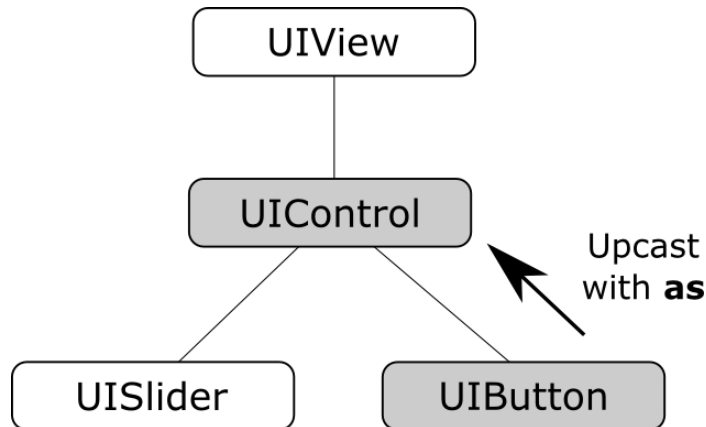


Figure 5-1

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

```
let myButton: UIButton = UIButton()

let myControl = myButton as UIControl
```


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

Downcasting usually involves converting from a class to one of its subclasses. Downcasting is performed using the *as!* keyword syntax and is also referred to as *forced conversion*. Consider, for example, the UIKit UIScrollView class which has as subclasses both the UITableView and UITextView classes as shown in Figure 5-2:

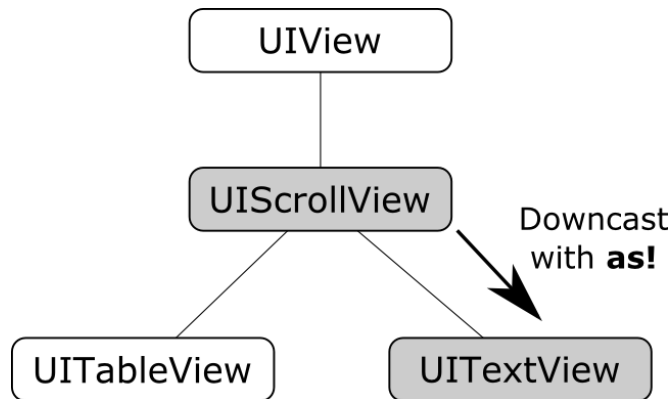


Figure 5-2

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

```
let myScrollView: UIScrollView = UIScrollView()

let myTextView = myScrollView as UITextView
```

The above code will result in the following error:

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

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

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

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

Swift Data Types, Constants and Variables

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

```
if let myTextView = myScrollView as? UITextView {  
    print("Type cast to UITextView succeeded")  
} else {  
    print("Type cast to UITextView failed")  
}
```

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

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

5.10 Summary

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

12. An Introduction to Swift Structures

Having covered Swift classes in the preceding chapters, this chapter will introduce the use of structures in Swift. Although at first glance structures and classes look similar, there are some important differences that need to be understood when deciding which to use. This chapter will outline how to declare and use structures, explore the differences between structures and classes and introduce the concepts of value and reference types.

12.1 An Overview of Swift Structures

As with classes, structures form the basis of object-oriented programming and provide a way to encapsulate data and functionality into re-usable instances. Structure declarations resemble classes with the exception that the *struct* keyword is used in place of the *class* keyword. The following code, for example, declares a simple structure consisting of a String variable, initializer and method:

```
struct SampleStruct {  
  
    var name: String  
  
    init(name: String) {  
        self.name = name  
    }  
  
    func buildHelloMsg() {  
        "Hello " + name  
    }  
}
```

Consider the above structure declaration in comparison to the equivalent class declaration:

```
class SampleClass {  
  
    var name: String  
  
    init(name: String) {  
        self.name = name  
    }  
  
    func buildHelloMsg() {  
        "Hello " + name  
    }  
}
```

```
}  
}
```

Other than the use of the *struct* keyword instead of *class*, the two declarations are identical. Instances of each type are also created using the same syntax:

```
let myStruct = SampleStruct(name: "Mark")  
let myClass = SampleClass(name: "Mark")
```

In common with classes, structures may be extended and are also able to adopt protocols and contain initializers.

Given the commonality between classes and structures, it is important to gain an understanding of how the two differ. Before exploring the most significant difference it is first necessary to understand the concepts of *value types* and *reference types*.

12.2 Value Types vs. Reference Types

While on the surface structures and classes look alike, major differences in behavior occur when structure and class instances are copied or passed as arguments to methods or functions. This occurs because structure instances are value type while class instances are reference type.

When a structure instance is copied or passed to a method, an actual copy of the instance is created, together with any data contained within the instance. This means that the copy has its own version of the data which is unconnected with the original structure instance. In effect, this means that there can be multiple copies of a structure instance within a running app, each with its own local copy of the associated data. A change to one instance has no impact on any other instances.

In contrast, when a class instance is copied or passed as an argument, the only thing duplicated or passed is a reference to the location in memory where that class instance resides. Any changes made to the instance using those references will be performed on the same instance. In other words, there is only one class instance but multiple references pointing to it. A change to the instance data using any one of those references changes the data for all other references.

To demonstrate reference and value types in action, consider the following code:

```
struct SampleStruct {  
  
    var name: String  
  
    init(name: String) {  
        self.name = name  
    }  
  
    func buildHelloMsg() {  
        "Hello " + name  
    }  
}
```

```
let myStruct1 = SampleStruct(name: "Mark")
print(myStruct1.name)
```

When the code executes, the name “Mark” will be displayed. Now change the code so that a copy of the myStruct1 instance is made, the name property changed and the names from each instance displayed:

```
let myStruct1 = SampleStruct(name: "Mark")
var myStruct2 = myStruct1
myStruct2.name = "David"

print(myStruct1.name)
print(myStruct2.name)
```

When executed, the output will read as follows:

```
Mark
David
```

Clearly, the change of name only applied to myStruct2 since this is an actual copy of myStruct1 containing its own copy of the data as shown in Figure 12-1:



Figure 12-1

Contrast this with the following class example:

```
class SampleClass {

    var name: String

    init(name: String) {
        self.name = name
    }

    func buildHelloMsg() {
        "Hello " + name
    }
}

let myClass1 = SampleClass(name: "Mark")
var myClass2 = myClass1
```

```
myClass2.name = "David"

print(myClass1.name)
print(myClass2.name)
```

When this code executes, the following output will be generated:

```
David
David
```

In this case, the name property change is reflected for both `myClass1` and `myClass2` because both are references pointing to the same class instance as illustrated in Figure 12-2 below:

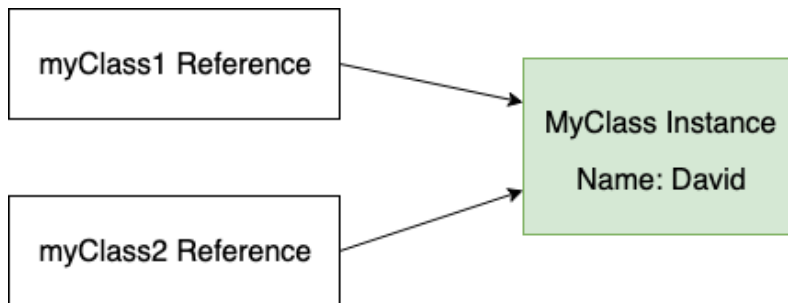


Figure 12-2

In addition to these value and reference type differences, structures do not support inheritance and sub-classing in the way that classes do. In other words, it is not possible for one structure to inherit from another structure. Unlike classes, structures also cannot contain a de-initializer (deinit) method. Finally, while it is possible to identify the type of a class instance at runtime, the same is not true of a struct.

12.3 When to use Structures or Classes

In general, structures are recommended whenever possible because they are both more efficient than classes and safer to use in multi-threaded code. Classes should be used when inheritance is needed, only one instance of the encapsulated data is required, or extra steps need to be taken to free up resources when an instance is de-initialized.

12.4 Summary

Swift structures and classes both provide a mechanism for creating instances that define properties, store values and define methods. Although the two mechanisms appear to be similar, there are significant behavioral differences when structure and class instances are either copied or passed to a method. Classes are categorized as being reference type instances while structures are value type. When a structure instance is copied or passed, an entirely new copy of the instance is created containing its own data. Class instances, on the other hand, are passed and copied by reference, with each reference pointing to the same class instance. Other features unique to classes include support for inheritance and deinitialization and the ability to identify the class type at runtime. Structures should typically be used in place of classes unless specific class features are required.

16. An Overview of SwiftUI

Now that Xcode has been installed and the basics of the Swift programming language covered, it is time to start introducing SwiftUI.

First announced at Apple's Worldwide Developer Conference in 2019, SwiftUI is an entirely new approach to developing apps for all Apple operating system platforms. The basic goals of SwiftUI are to make app development easier, faster and less prone to the types of bugs that typically appear when developing software projects. These elements have been combined with SwiftUI specific additions to Xcode that allow SwiftUI projects to be tested in near real-time using a live preview of the app during the development process.

Many of the advantages of SwiftUI originate from the fact that it is both *declarative* and *data driven*, topics which will be explained in this chapter.

The discussion in this chapter is intended as a high-level overview of SwiftUI and does not cover the practical aspects of implementation within a project. Implementation and practical examples will be covered in detail in the remainder of the book.

16.1 UIKit and Interface Builder

To understand the meaning and advantages of SwiftUI's declarative syntax, it helps to understand how user interface layouts were designed before the introduction of SwiftUI. Up until the introduction of SwiftUI, iOS apps were built entirely using UIKit together with a collection of associated frameworks that make up the iOS Software Development Kit (SDK).

To aid in the design of the user interface layouts that make up the screens of an app, Xcode includes a tool called Interface Builder. Interface Builder is a powerful tool that allows storyboards to be created which contain the individual scenes that make up an app (with a scene typically representing a single app screen).

The user interface layout of a scene is designed within Interface Builder by dragging components (such as buttons, labels, text fields and sliders) from a library panel to the desired location on the scene canvas. Selecting a component in a scene provides access to a range of inspector panels where the attributes of the components can be changed.

The layout behavior of the scene (in other words how it reacts to different device screen sizes and changes to device orientation between portrait and landscape) is defined by configuring a range of constraints that dictate how each component is positioned and sized in relation to both the containing window and the other components in the layout.

Finally, any components that need to respond to user events (such as a button tap or slider motion) are connected to methods in the app source code where the event is handled.

At various points during this development process, it is necessary to compile and run the app on a simulator or device to test that everything is working as expected.

16.2 SwiftUI Declarative Syntax

SwiftUI introduces a declarative syntax that provides an entirely different way of implementing user interface layouts and behavior from the UIKit and Interface Builder approach. Instead of manually designing the intricate details of the layout and appearance of components that make up a scene, SwiftUI allows the scenes to be described using a simple and intuitive syntax. In other words, SwiftUI allows layouts to be created by declaring how the user interface should appear without having to worry about the complexity of how the layout is actually built.

This essentially involves declaring the components to be included in the layout, stating the kind of layout manager in which they to be contained (vertical stack, horizontal stack, form, list etc.) and using modifiers to set attributes such as the text on a button, the foreground color of a label, or the method to be called in the event of a tap gesture. Having made these declarations, all the intricate and complicated details of how to position, constrain and render the layout are handled automatically by SwiftUI.

SwiftUI declarations are structured hierarchically, which also makes it easy to create complex views by composing together small, re-usable custom subviews.

While the view layout is being declared and tested, Xcode provides a preview canvas which changes in real-time to reflect the appearance of the layout. Xcode also includes a *live preview* mode which allows the app to be launched within the preview canvas and fully tested without the need to build and run on a simulator or device.

Coverage of the SwiftUI declaration syntax begins with the chapter entitled *Creating Custom Views with SwiftUI*.

16.3 SwiftUI is Data Driven

When we say that SwiftUI is data driven, this is not to say that it is no longer necessary to handle events generated by the user (in other words the interaction between the user and the app user interface). It is still necessary, for example, to know when the user taps a button and to react in some app specific way. Being data driven relates more to the relationship between the underlying app data and the user interface and logic of the app.

Prior to the introduction of SwiftUI, an iOS app would contain code responsible for checking the current values of data within the app. If data is likely to change over time, code has to be written to ensure that the user interface always reflects the latest state of the data (perhaps by writing code to frequently check for changes to the data, or by providing a refresh option for the user to request a data update). Similar problems arise when keeping the user interface state consistent and making sure issues like toggle button settings are stored appropriately. Requirements such as these can become increasingly complex when multiple areas of an app depend on the same data sources.

SwiftUI addresses this complexity by providing several ways to *bind* the data model of an app to the user interface components and logic that provide the app functionality.

When implemented, the data model *publishes* data variables to which other parts of the app can then *subscribe*. Using this approach, changes to the published data are automatically reported to all subscribers. If the binding is made from a user interface component, any data changes will automatically be reflected within the user interface by SwiftUI without the need to write any additional code.

16.4 SwiftUI vs. UIKit

With the choice of using UIKit and SwiftUI now available, the obvious question arises as to which is the best option. When making this decision it is important to understand that SwiftUI and UIKit are not mutually exclusive. In fact, several integration solutions are available (a topic area covered starting with the chapter entitled *Integrating UIViews with SwiftUI*).

The first factor to take into consideration during the decision process is that any app that includes SwiftUI-based code will only run on devices running iOS 13 or later. This means, for example, that your app will only be available to users with the following iPhone models:

- iPhone 11
- iPhone 11 Pro
- iPhone 11 Pro Max
- iPhone Xs
- iPhone Xs Max
- iPhone XR
- iPhone X
- iPhone 8
- iPhone 8 Plus
- iPhone 7
- iPhone 7 Plus
- iPhone 6s
- iPhone 6s Plus
- iPhone SE

Apple reported on October 15, 2019 that, based on App Store measurements, 50% of all iPhone devices were running iOS 13, a percentage that will continue to increase with the passage of time.

If supporting devices running older versions of iOS is not of concern and you are starting a new project, it makes sense to use SwiftUI wherever possible. Not only does SwiftUI provide a faster, more efficient app development environment, it also makes it easier to make the same app available on multiple Apple platforms (iOS, iPadOS, macOS, watchOS and tvOS) without making significant code changes.

If you have an existing app developed using UIKit there is no easy migration path to convert that code to SwiftUI, so it probably makes sense to keep using UIKit for that part of the project. UIKit will

continue to be a valuable part of the app development toolset and will be extended, supported and enhanced by Apple for the foreseeable future. When adding new features to an existing project, however, consider doing so using SwiftUI and integrating it into the existing UIKit codebase.

When adopting SwiftUI for new projects, it will probably not be possible to avoid using UIKit entirely. Although SwiftUI comes with a wide array of user interface components, it will still be necessary to use UIKit for certain functionality such as map and web view integration.

In addition, for extremely complex user interface layout designs, it may also be necessary to use Interface Builder in situations where layout needs cannot be satisfied using the SwiftUI layout container views.

16.5 Summary

SwiftUI introduces a different approach to app development than that offered by UIKit and Interface Builder. Rather than directly implement the way in which a user interface is to be rendered, SwiftUI allows the user interface to be declared in descriptive terms and then does all the work of deciding the best way to perform the rendering when the app runs.

SwiftUI is also data driven in that data change drives the behavior and appearance of the app. This is achieved through a publisher and subscriber model.

This chapter has provided a very high-level view of SwiftUI. The remainder of this book will explore SwiftUI in greater depth.

17. Using Xcode in SwiftUI Mode

When creating a new project, Xcode now provides a choice of creating either a Storyboard or SwiftUI based user interface for the project. When creating a SwiftUI project, Xcode appears and behaves significantly differently when designing the user interface for an app project compared to the UIKit Storyboard mode.

When working in SwiftUI mode, most of your time as an app developer will be spent in the code editor and preview canvas, both of which will be explored in detail in this chapter.

17.1 Starting Xcode 11

As with all iOS examples in this book, the development of our example will take place within the Xcode 11 development environment. If you have not already installed this tool together with the latest iOS SDK refer first to the *Installing Xcode 11 and the iOS 13 SDK* chapter of this book. Assuming the installation is complete, launch Xcode either by clicking on the icon on the dock (assuming you created one) or use the macOS Finder to locate Xcode in the Applications folder of your system.

When launched for the first time, and until you turn off the *Show this window when Xcode launches* toggle, the screen illustrated in Figure 17-1 will appear by default:

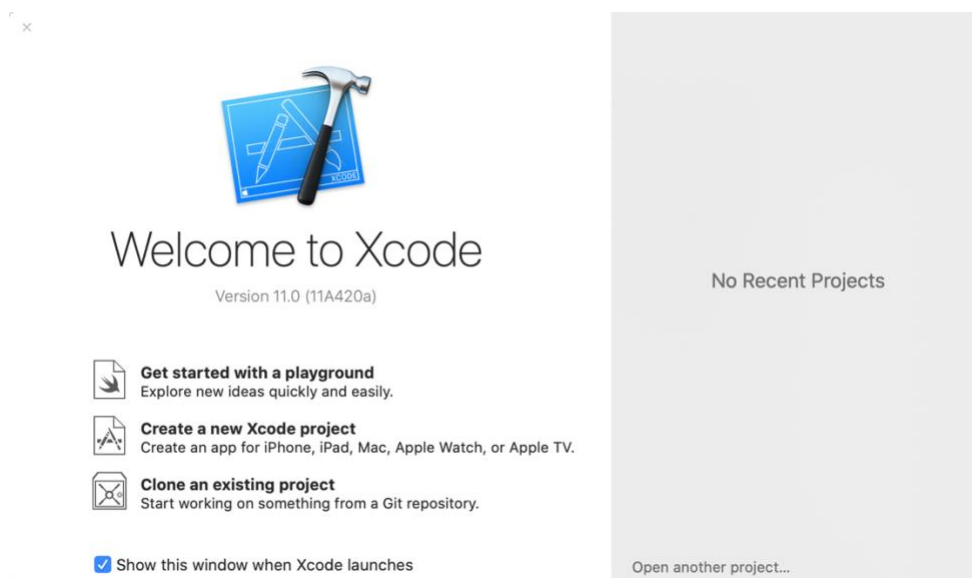


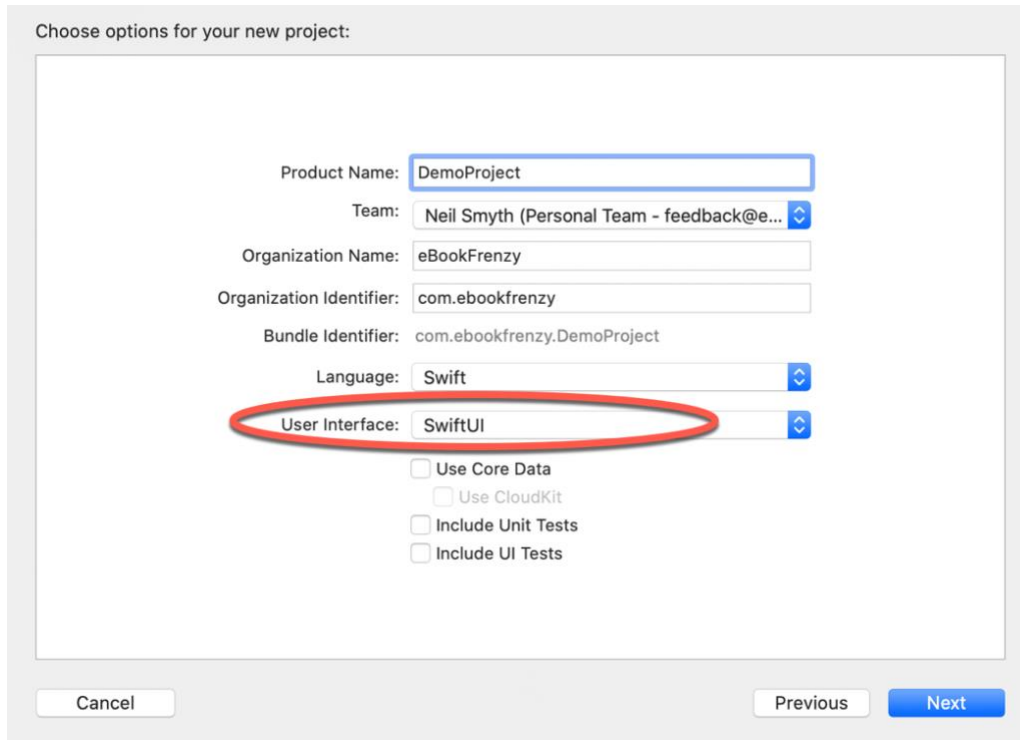
Figure 17-1

Using Xcode in SwiftUI Mode

If you do not see this window, simply select the *Window -> Welcome to Xcode* menu option to display it. From within this window, click on the option to *Create a new Xcode project*.

17.2 Creating a SwiftUI Project

When creating a new project, the project options screen includes an option to select how the user interface is to be implemented. To use SwiftUI, simply change the menu option highlighted in Figure 17-2 to *SwiftUI*:



The image shows the 'Choose options for your new project' dialog in Xcode. The 'Product Name' is 'DemoProject', 'Team' is 'Neil Smyth (Personal Team - feedback@e...)', 'Organization Name' is 'eBookFrenzy', 'Organization Identifier' is 'com.ebookfrenzy', and 'Bundle Identifier' is 'com.ebookfrenzy.DemoProject'. The 'Language' is 'Swift'. The 'User Interface' is set to 'SwiftUI', which is highlighted with a red oval. Below this, there are four unchecked checkboxes: 'Use Core Data', 'Use CloudKit', 'Include Unit Tests', and 'Include UI Tests'. At the bottom, there are 'Cancel', 'Previous', and 'Next' buttons.

Figure 17-2

Once a new project has been created with SwiftUI selected, the main Xcode panel will appear with the default layout for SwiftUI development displayed.

17.3 Xcode in SwiftUI Mode

Before beginning work on a SwiftUI user interface, it is worth taking some time to gain familiarity with how Xcode works in SwiftUI mode. A newly created project includes a single SwiftUI View file named *ContentView.swift* which, when selected from the project navigation panel, will appear within Xcode as shown in Figure 17-3 below:

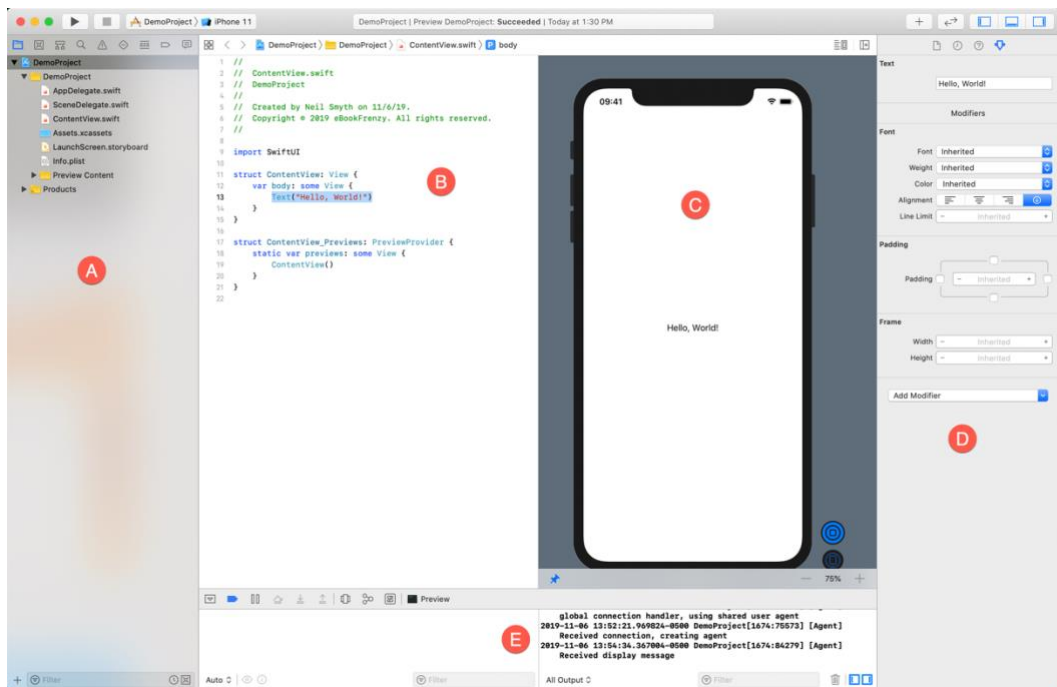


Figure 17-3

Located to the right of the project navigator (A) is the code editor (B). To the right of this is the preview canvas (C) where any changes made to the SwiftUI layout declaration will appear in real-time.

Selecting a view in the canvas will automatically select and highlight the corresponding entry in the code editor, and vice versa. Attributes for the currently selected item will appear in the attributes inspector panel (D).

During debugging, the debug panel (E) will appear containing debug output from both the iOS frameworks and any diagnostic print statements you have included in your code.

The three panels (A, D and E) can be displayed and hidden using the three buttons located on the right-hand side of the toolbar as shown in Figure 17-4:



Figure 17-4

17.4 The Preview Canvas

The preview canvas provides both a visual representation of the user interface design and a tool for adding and modifying views within the layout design. The canvas may also be used to perform live testing of the running app without the need to launch an iOS simulator. Figure 17-5 illustrates a typical preview canvas for a newly created project:

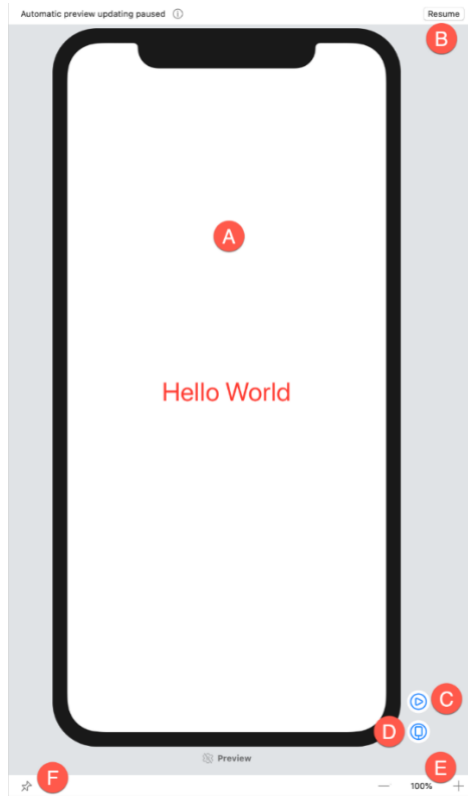


Figure 17-5

If the canvas is not visible it can be displayed using the Xcode *Editor* -> *Canvas* menu option.

The main canvas area (A) represents the current view as it will appear when running on a physical device. When changes are made to the code in the editor, those changes are reflected within the preview canvas. To avoid continually updating the canvas, and depending on the nature of the changes being made, the preview will occasionally pause live updates. When this happens, the Resume button (B) will appear which, when clicked, will once again begin updating the preview.

By default, the preview displays a static representation of the user interface. To test the user interface in a running version of the app, simply click on the Live Preview button (C). Xcode will then build the app and run it within the preview canvas where you can interact with it as you would in a simulator or on a physical device. When in Live Preview mode, the button changes to a stop button which can be used to exit live mode.

The current version of the app may also be previewed on an attached physical device by clicking on the Preview on Device button (D). As with the preview canvas, the running app on the device will update dynamically as changes are made to the code in the editor.

Right-clicking on either the Live Preview or Preview on Device buttons will provide the option to run in debug mode, attaching the process to the debugger and allowing diagnostic output to appear in the debug area (marked E in Figure 17-3 above).

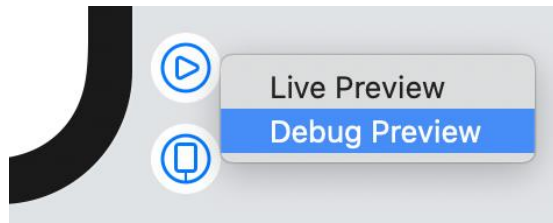


Figure 17-6

17.5 Preview Pinning

When building an app in Xcode it is likely that it will consist of several SwiftUI View files in addition to the default *ContentView.swift* file. When a SwiftUI View file is selected from the project navigator, both the code editor and preview canvas will change to reflect the currently selected file. Sometimes you may want the user interface layout for one SwiftUI file to appear in the preview canvas while editing the code in a different file. This can be particularly useful if the layout from one file is dependent on or embedded in another view. The pin button (labelled F in Figure 17-5 above) pins the current preview to the canvas so that it remains visible on the canvas after navigating to a different view. The view to which you have navigated will appear beneath the pinned view in the canvas and can be viewed by scrolling.

Finally, the size buttons (E) can be used to zoom in and out of the canvas.

17.6 Modifying the Design

Working with SwiftUI primarily involves adding additional views, customizing those views using modifiers, adding logic and interacting with state and other data instance bindings. All of these tasks can be performed exclusively by modifying the structure in the code editor. The font used to display the “Hello World” Text view, for example, can be changed by adding the appropriate modifier in the editor:

```
Text("Hello World")
    .font(.largeTitle)
```

An alternative to this is to make changes to the SwiftUI views by dragging and dropping items from the Library panel. The Library panel is displayed by clicking on the toolbar button highlighted in Figure 17-7:

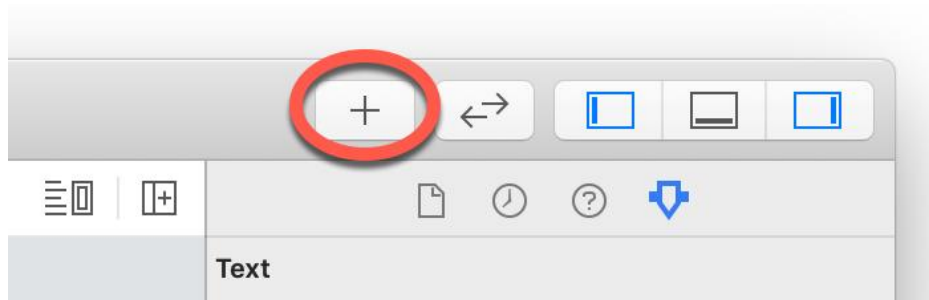


Figure 17-7

Using Xcode in SwiftUI Mode

When displayed, the Library panel will appear as shown in Figure 17-8:

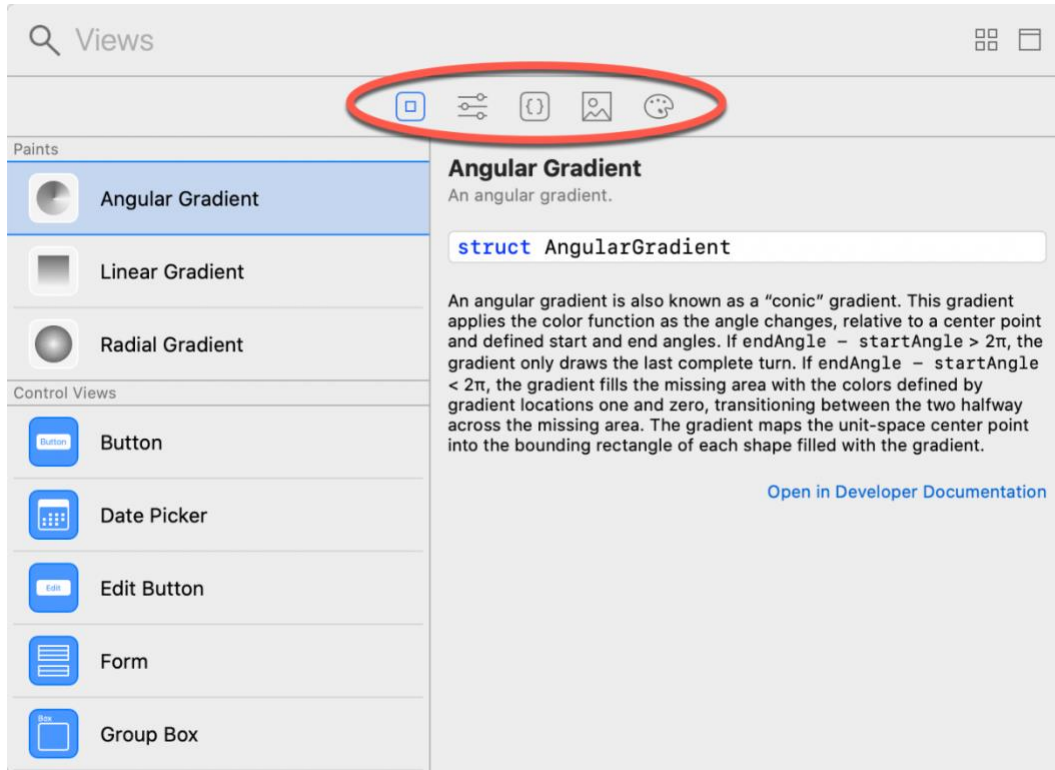


Figure 17-8

When launched in this way, the Library panel is transient and will disappear either after a selection has been made, or a click is performed outside of the panel. To keep the panel displayed, hold down the Option key when clicking on the Library button.

When first opened, the panel displays a list of views available for inclusion in the user interface design. The list can be browsed, or the search bar used to narrow the list to specific views. The toolbar (highlighted in the above figure) can be used to switch to other categories such as modifiers, commonly used code snippets, images and color resources.

An item within the library can be applied to the user interface design in a number of ways. To apply a font modifier to the “Hello World” Text view, one option is to select the view in either the code or preview canvas, locate the font modifier in the Library panel and double-click on it. Xcode will then automatically apply the font modifier.

Another option is to locate the Library item and then drag and drop it onto the desired location either in the code editor or the preview canvas. In Figure 17-9 below, for example, the font modifier is being dragged to the Text view within the editor:

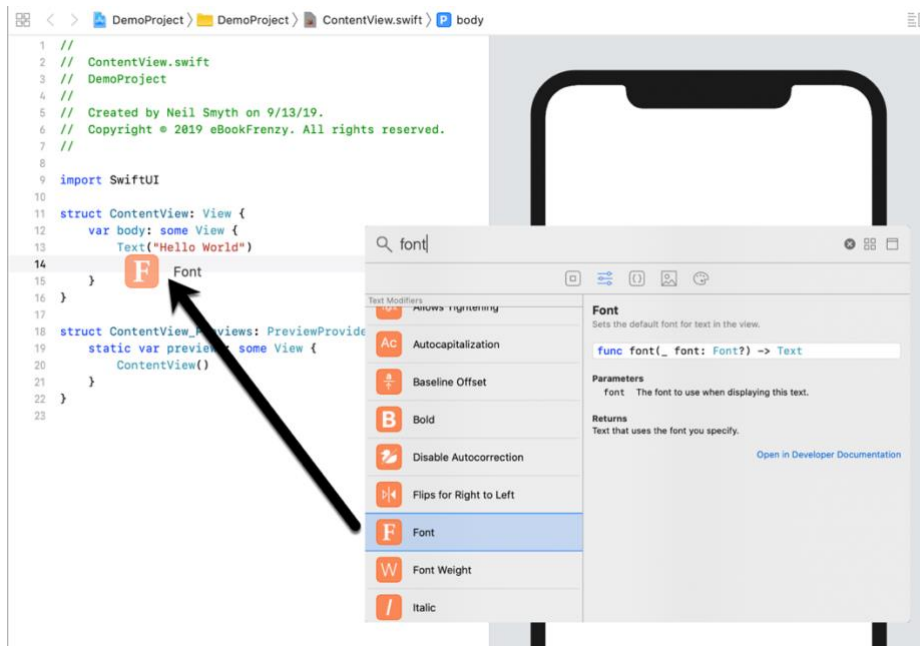


Figure 17-9

The same result can be achieved by dragging an item from the library onto the preview canvas. In the case of Figure 17-10, a Button view is being added to the layout beneath the existing Text view:

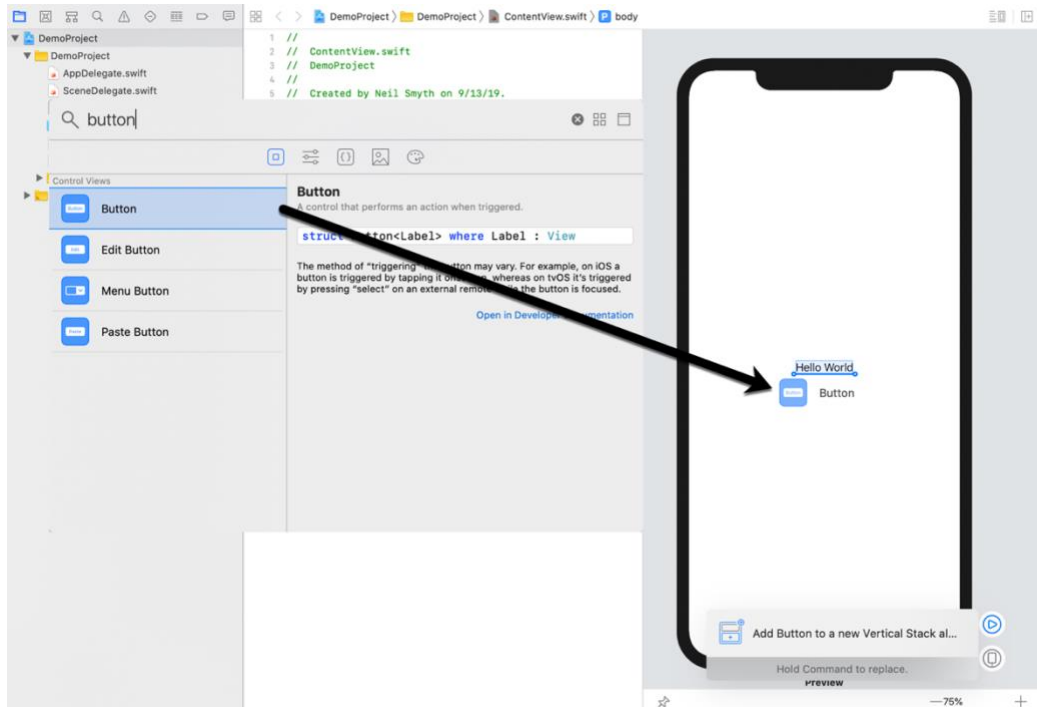


Figure 17-10

Using Xcode in SwiftUI Mode

In this example, the side along which the view will be placed if released highlights and the preview canvas displays a notification that the Button and existing Text view will automatically be placed in a Vertical Stack container view (stacks will be covered later in the chapter entitled *SwiftUI Stacks and Frames*).

Once a view or modifier has been added to the SwiftUI view file it is highly likely that some customization will be necessary, such as specifying the color for a foreground modifier. One option is, of course, to simply make the changes within the editor, for example:

```
Text("Hello World")  
    .font(.largeTitle)  
    .foregroundColor(.red)
```

Another option is to select the view in either the editor or preview panel and then make the necessary changes within the Attributes inspector panel:

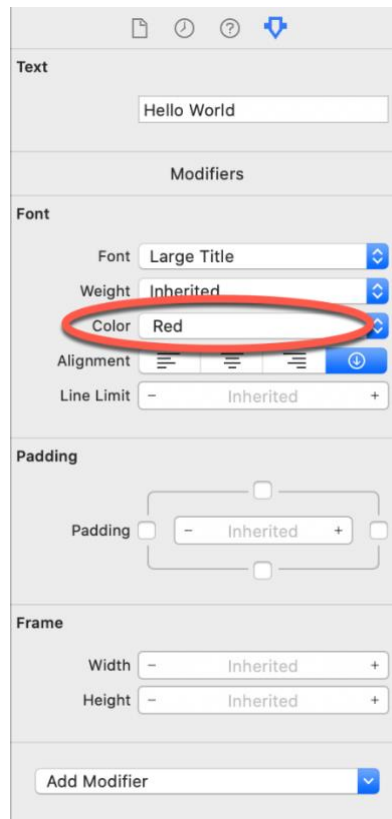


Figure 17-11

The Attributes inspector will provide the option to make changes to any modifiers already applied to the selected view.

Before moving on to the next topic, it is also worth noting that the Attributes inspector provides yet another way to add modifiers to a view via the Add Modifier menu located at the bottom of the

panel. When clicked, this menu will display a long list of modifiers available for the current view type. To apply a modifier, simply select it from the menu. An entry for the new modifier will subsequently appear in the inspector where it can be configured with the required properties.

17.7 Editor Context Menu

Holding down the Command key while clicking on the item in the code editor will display the menu shown in Figure 17-12:

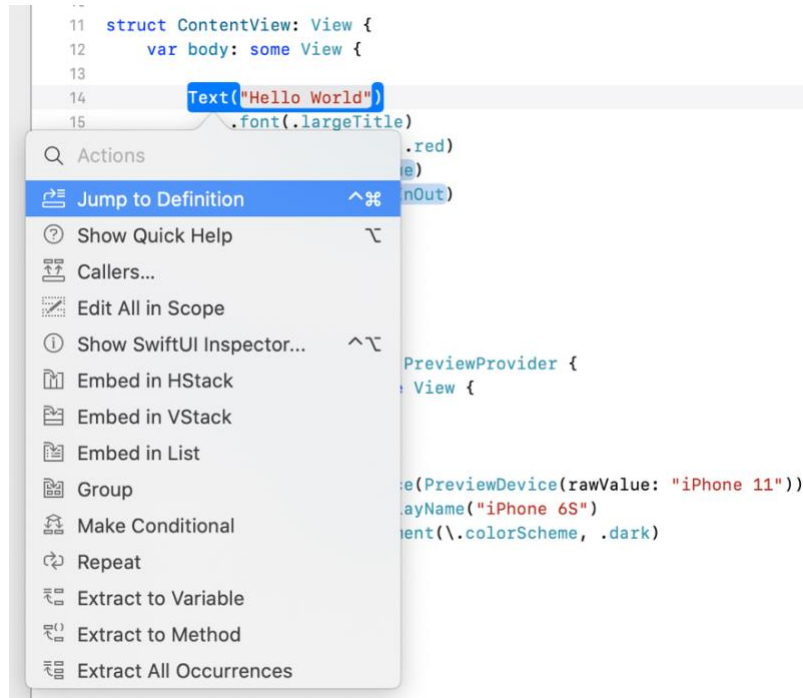


Figure 17-12

This menu provides a list of options which will vary depending on the type of item selected. Options typically include a shortcut to a popup version of the Attributes inspector for the current view, together with options to embed the current view in a stack or list container. This menu is also useful for extracting part of a view into its own self-contained subview. Creating subviews is strongly encouraged to promote reuse, improve performance and unclutter complex design structures.

17.8 Previewing on Multiple Device Configurations

Every newly created SwiftUI View file includes an additional declaration at the bottom of the file that resembles the following:

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}
```

Using Xcode in SwiftUI Mode

This structure, which conforms to the `PreviewProvider` protocol, returns an instance of the primary view within the file. This instructs Xcode to display the preview for that view within the preview canvas (without this declaration, nothing will appear in the canvas).

By default, the preview canvas shows the user interface on a single device based on the current selection in the run target menu to the right of the run and stop button in the Xcode toolbar. To preview on other device models, one option is to simply change the run target and wait for the preview canvas to change.

A better option, however, is to modify the previews structure to specify a different device. In the following example, the canvas previews the user interface on an iPhone SE:

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
            .previewDevice(PreviewDevice(rawValue: "iPhone SE"))
            .previewDisplayName("iPhone SE")
    }
}
```

In fact, it is possible using this technique to preview multiple device types simultaneously by placing them into a `Group` view as follows:

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {

        Group {
            ContentView()
                .previewDevice(PreviewDevice(rawValue: "iPhone SE"))
                .previewDisplayName("iPhone SE")

            ContentView()
                .previewDevice(PreviewDevice(rawValue: "iPhone 11"))
                .previewDisplayName("iPhone 11")
        }
    }
}
```

When multiple devices are previewed, they appear in a scrollable list within the preview canvas as shown in Figure 17-13:

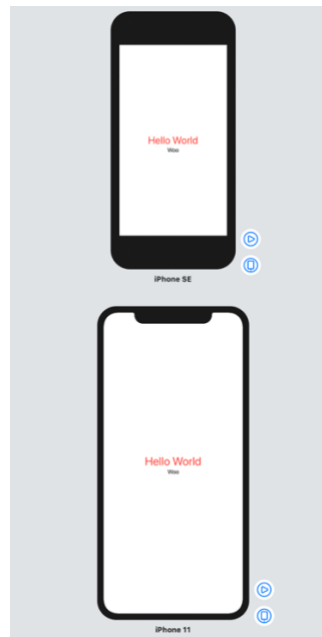


Figure 17-13

The environment modifier may also be used to preview the layout in other configurations, for example, to preview in dark mode:

```
ContentView()
    .previewDevice(PreviewDevice(rawValue: "iPhone SE"))
    .previewDisplayName("iPhone SE")
    .environment(\.colorScheme, .dark)
```

This preview structure is also useful for passing sample data into the enclosing view for testing purposes within the preview canvas, a technique that will be used in later chapters. For example:

```
struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView(sampleData: mySampleData)
    }
}
```

17.9 Running the App on a Simulator

Although much can be achieved using the preview canvas, there is no substitute for running the app on physical devices and simulators during testing.

Within the main Xcode project window, the menu located in the top left-hand corner of the window (marked C in Figure 17-14) is used to choose a target simulator. This menu will include both simulators that have been configured and any physical devices connected to the development system:



Figure 17-14

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

The simulator includes a number of options not available in the live preview for testing different aspects of the app. The Hardware and Debug menus, for example, include options for rotating the simulator through portrait and landscape orientations, testing Face ID authentication and simulating geographical location changes for navigation and map-based apps.

17.10 Running the App on a Physical iOS Device

Although the Simulator environment provides a useful way to test an app on a variety of different iOS device models, it is important to also test on a physical iOS device.

If you have entered your Apple ID in the Xcode preferences screen as outlined in the *Joining the Apple Developer Program* chapter and selected a development team for the project, it is possible to run the app on a physical device simply by connecting it to the development Mac system with a USB cable and selecting it as the run target within Xcode.

With a device connected to the development system, and an application ready for testing, refer to the device menu located in the Xcode toolbar. There is a reasonable chance that this will have defaulted to one of the iOS Simulator configurations. Switch to the physical device by selecting this menu and changing it to the device name as shown in Figure 17-5:

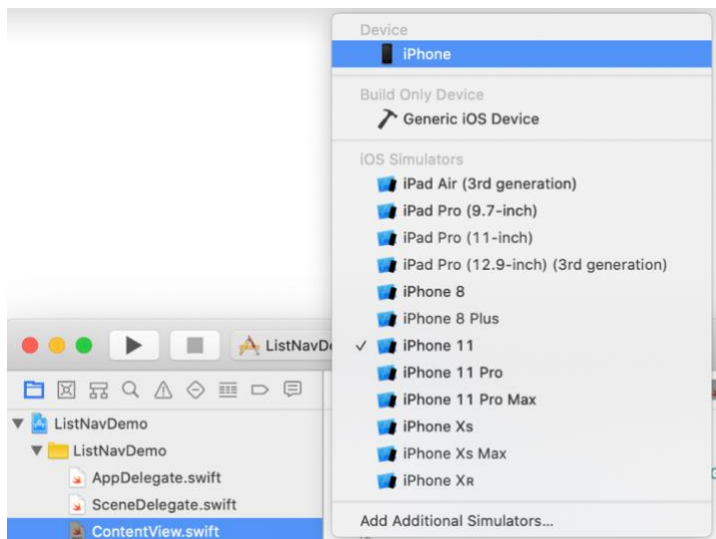


Figure 17-15

With the target device selected, make sure the device is unlocked and click on the run button at which point Xcode will install and launch the app on the device. If you have not yet joined the Apple Developer Program, the following dialog may appear within Xcode indicating that you need to configure your device to trust the developer certificate used to build the app:

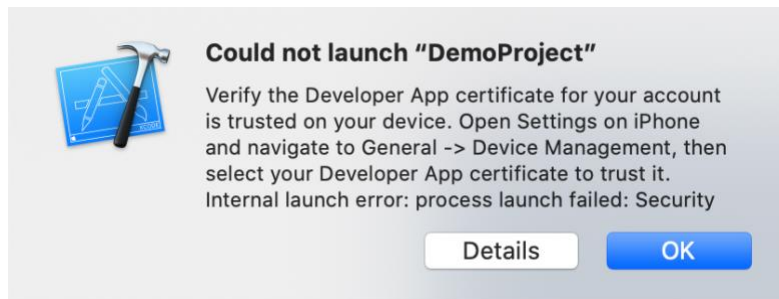


Figure 17-16

Following the instructions in the dialog, open the Settings app on the device, navigate to *General* -> *Profiles and Device Management* and select the developer certificate on the resulting screen:



Figure 17-17

In the subsequent certificate screen, tap the *Trust "Apple Development: <email address>"* button followed by the Trust button in the confirmation dialog:

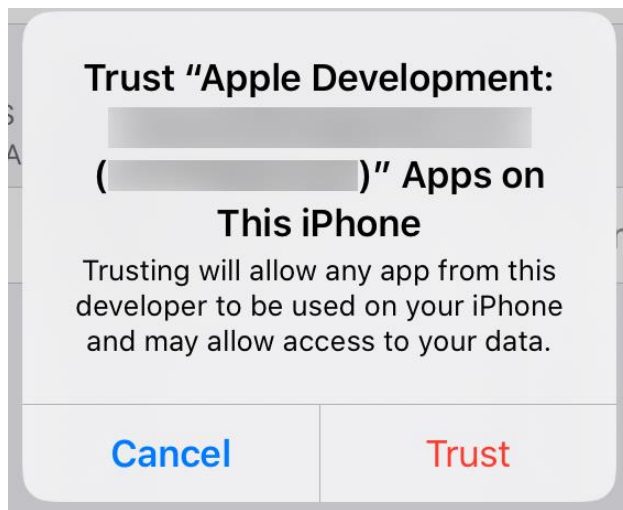


Figure 17-18

Once the certificate is trusted, it should be possible to install and run the app on the device.

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

17.11 Managing Devices and Simulators

Currently connected iOS devices and the simulators configured for use with Xcode can be viewed and managed using the Xcode Devices window which is accessed via the *Window -> Devices and Simulators* menu option. Figure 17-19 for example, shows a typical Device screen on a system where an iPhone has been detected:

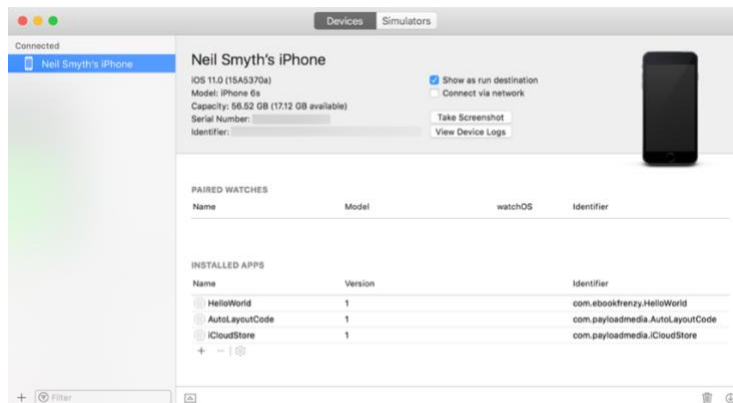


Figure 17-19

A wide range of simulator configurations are set up within Xcode by default and can be viewed by selecting the *Simulators* tab at the top of the dialog. Other simulator configurations can be added

by clicking on the + button located in the bottom left-hand corner of the window. Once selected, a dialog will appear allowing the simulator to be configured in terms of device, iOS version and name.

17.12 Enabling Network Testing

In addition to testing an app on a physical device connected to the development system via a USB cable, Xcode also supports testing via a network connection. This option is enabled on a per device basis within the Devices and Simulators dialog introduced in the previous section. With the device connected via the USB cable, display this dialog, select the device from the list and enable the *Connect via network* option as highlighted in Figure 17-20:

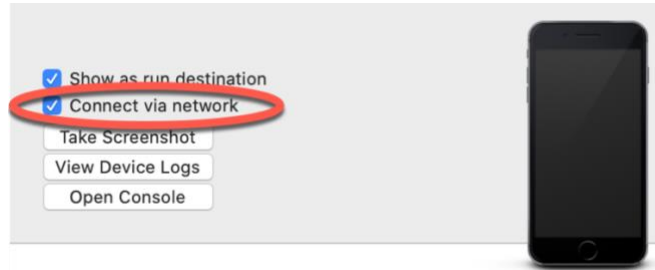


Figure 17-20

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

17.13 Dealing with Build Errors

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

17.14 Monitoring Application Performance

Another useful feature of Xcode is the ability to monitor the performance of an application while it is running, either on a device or simulator or within the live preview canvas. This information is accessed by displaying the *Debug Navigator*.

When Xcode is launched, the project navigator is displayed in the left-hand panel by default. Along the top of this panel is a bar with a range of other options. The seventh option from the left displays the debug navigator when selected as illustrated in Figure 17-21. When displayed, this panel shows a number of real-time statistics relating to the performance of the currently running application such as memory, CPU usage, disk access, energy efficiency, network activity and iCloud storage access.

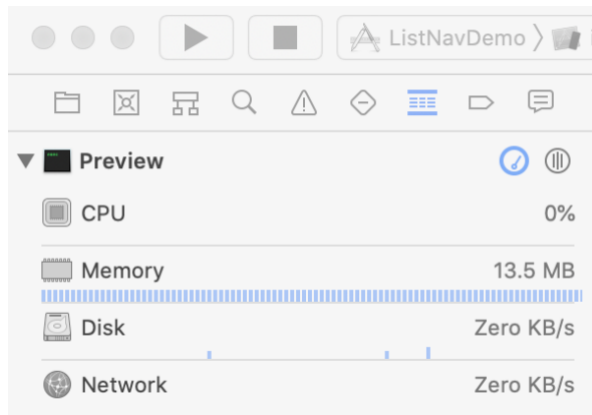


Figure 17-21

When one of these categories is selected, the main panel (Figure 17-22) updates to provide additional information about that particular aspect of the application’s performance:

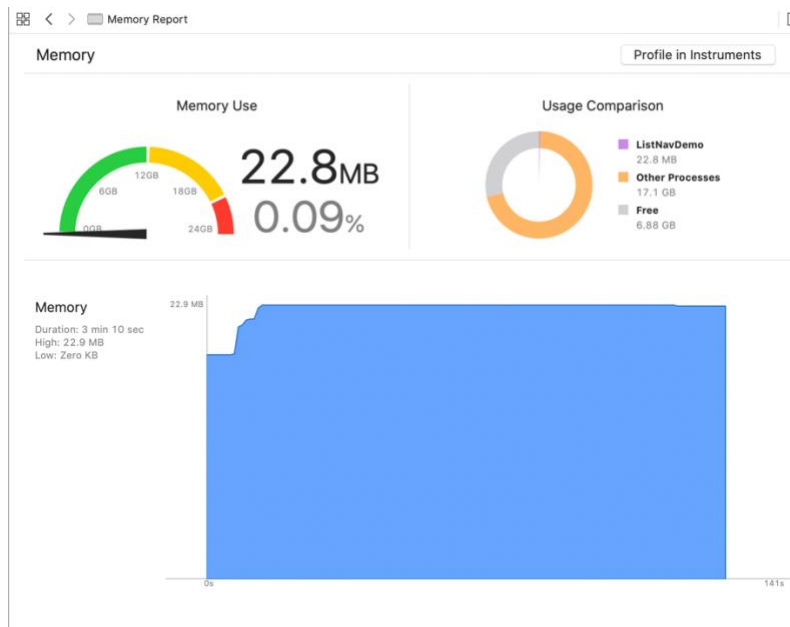


Figure 17-22

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

17.15 Exploring the User Interface Layout Hierarchy

Xcode also provides an option to break the user interface layout out into a rotatable 3D view that shows how the view hierarchy for a user interface is constructed. This can be particularly useful for identifying situations where one view instance is obscured by another appearing on top of it or a layout is not appearing as intended. This is also useful for learning how SwiftUI works behind the

scenes to construct a SwiftUI layout, if only to appreciate how much work SwiftUI is saving us from having to do.

To access the view hierarchy in this mode, begin by previewing the view in debug mode as illustrated in Figure 17-6 above. Once the preview is live, click on the *Debug View Hierarchy* button indicated in Figure 17-23:

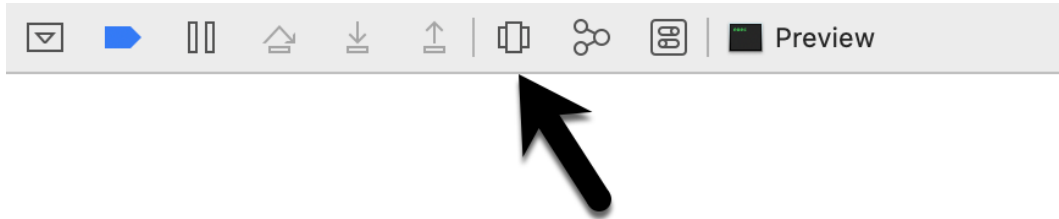


Figure 17-23

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

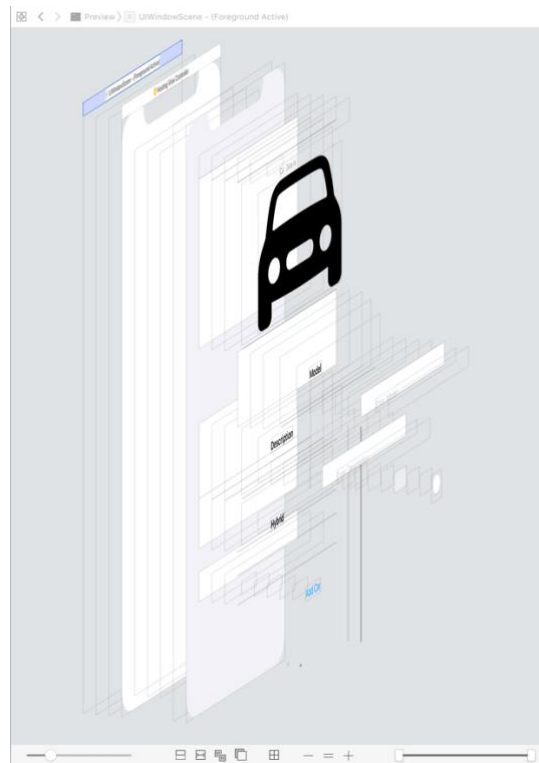


Figure 17-24

Moving the slider in the bottom left-hand corner of the panel will adjust the spacing between the different views in the hierarchy. The two markers in the right-hand slider (Figure 17-25) may also

Using Xcode in SwiftUI Mode

be used to narrow the range of views visible in the rendering. This can be useful, for example, to focus on a subset of views located in the middle of the hierarchy tree:



Figure 17-25

While the hierarchy is being debugged, the left-hand panel will display the entire view hierarchy tree for the layout as shown in Figure 17-26 below:

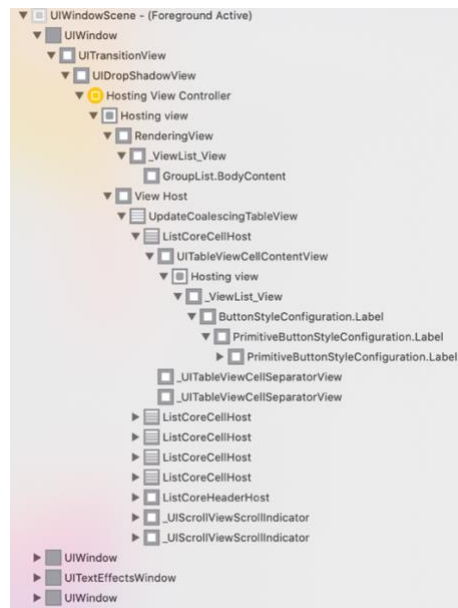


Figure 17-26

Selecting an object in the hierarchy tree will highlight the corresponding item in the 3D rendering and vice versa. The far right-hand panel will also display the attributes of the selected object. If the panel is not currently visible it can be displayed by clicking on the toolbar button indicated in Figure 17-27:



Figure 17-27

Figure 17-28, for example, shows the inspector panel while a Text view is selected within the view hierarchy.

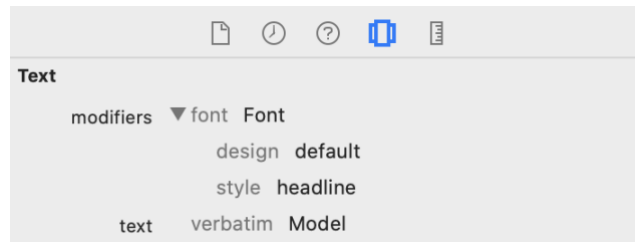


Figure 17-28

17.16 Summary

When creating a new project, Xcode provides the option to use either UIKit Storyboards or SwiftUI as the basis of the user interface of the app. When in SwiftUI mode, most of the work involved in developing an app takes place in the code editor and the preview canvas. New views can be added to the user interface layout and configured either by typing into the code editor or dragging and dropping components from the Library either onto the editor or the preview canvas.

The preview canvas will usually update in real time to reflect code changes as they are typed into the code editor, though will frequently pause updates in response to larger changes. When in the paused state, clicking the Resume button will restart updates. The Attribute inspector allows the properties of a selected view to be changed and new modifiers added. Holding down the Command key while clicking on a view in the editor or canvas displays the context menu containing a range of options such as embedding the view in a container or extracting the selection to a subview.

The preview structure at the end of the SwiftUI View file allows previewing to be performed on multiple device models simultaneously and with different environment settings.

18. The Anatomy of a Basic SwiftUI Project

When a new SwiftUI project is created in Xcode using the Single View App template, Xcode generates a number of different files and folders which form the basis of the project, and on which the finished app will eventually be built.

Although it is not necessary to know in detail about the purpose of each of these files when beginning with SwiftUI development, each of them will become useful as you progress to developing more complex applications.

The goal of this chapter, therefore, is to provide a brief overview of each element of a basic project structure.

18.1 Creating an Example Project

It may be useful to create a sample project to review while working through this chapter. To do so, launch Xcode and, on the welcome screen, select the option to create a new project. On the resulting template selection panel, choose the Single View App option before proceeding to the next screen. On the project options screen, name the project *ProjectDemo* and change the User Interface menu to SwiftUI. Click Next to proceed to the final screen, choose a suitable filesystem location for the project and click on the Create button.

18.2 UIKit and SwiftUI

As discussed previously, before the introduction of SwiftUI, iOS apps were developed using UIKit. In recognition of this reality, Apple has provided a number of ways in which SwiftUI and UIKit code can be integrated within the same project.

It may not be obvious initially, but when creating a new SwiftUI based project, Xcode actually creates a UIKit-based app which uses these integration techniques to host the SwiftUI views that ultimately make up the app. Some of the files described in this chapter are, therefore, UIKit-based and all class names prefixed with “UI” are UIKit classes.

18.3 The AppDelegate.swift File

Every iOS app has one instance of the UIApplication class which is responsible for handling events and managing the different UIWindow objects that will be used by the app to display user interfaces to the user. UIWindow instances are not visible to the user but instead provide containers to hold the visible objects that make up the user interface.

The `UIApplication` instance has associated with it a delegate which it notifies via method calls of significant events relating to the lifecycle of the app such as the app launching, incoming notifications, low device memory, the pending termination of the app and the creation of new scenes within the app.

By default, the `AppDelegate.swift` file generated by Xcode contains only the methods that are mandatory to comply with the `AppDelegate` protocol but others can be added for the app to receive notification of other app lifecycle events. These methods can be useful for implementing early app specific initialization tasks such as establishing a network connection or setting up database access. The `didFinishLaunchingWithOptions` method is particularly useful for adding initialization code since it is the first method to be called after the app has finished launching.

18.4 The `SceneDelegate.swift` File

The entire user interface of an app is represented as a scene in the form of a `UIWindowScene` object with a `UIWindow` child. It is important not to confuse this with a UIKit *Storyboard scene* which represents only a single screen within an app user interface. By default, an app will have only one scene, but with the introduction of multi-window support with iOS 13 it is also possible to configure an app to allow the creation of multiple instances of its user interface. On iPhone devices, the user switches between user interface copies using the app switcher while on the iPad, copies of the user interface can also appear side by side.

While multiple scenes all share the same `UIApplication` object, each of the `UIWindowScene` instances in a multi-window configuration has its own scene delegate instance.

The `SceneDelegate` class file implements the `UIWindowSceneDelegate` protocol and contains methods to handle events such as a new scene object connecting to the current session, the scene transitioning between background and foreground or a scene disconnecting from the app.

All of the `SceneDelegate` methods are useful for performing initialization and deinitialization tasks during the lifecycle of the app. The most important delegate method in this file, however, is the `willConnectTo` method which is called each time a new scene object is added to the app.

By default, the `willConnectTo` delegate method will have been implemented by Xcode to create an instance of the SwiftUI `ContentView` view declared in the `ContentView.swift` file and make it visible to the user. It is within this method that the gap between the UIKit architecture and SwiftUI is bridged.

In order to embed a SwiftUI view into a UIKit project, the SwiftUI view is embedded into a `UIHostingController` instance (a topic covered in detail starting with the chapter entitled *Integrating UIViews with SwiftUI*). To achieve this, the `willConnectTo` delegate method performs the following tasks:

1. Creates an instance of `ContentView`.
2. Creates a new `UIWindow` instance.
3. Embeds the `ContentView` instance into a `UIHostingController` instance.
4. Assigns the `UIHostingController` as the root view controller for the newly created `UIWindow` instance.

5. Replaces the scene's current UIWindow instance with the new one.
6. Makes the window visible to the user.

Figure 18-1 illustrates the hierarchy of a single window app:

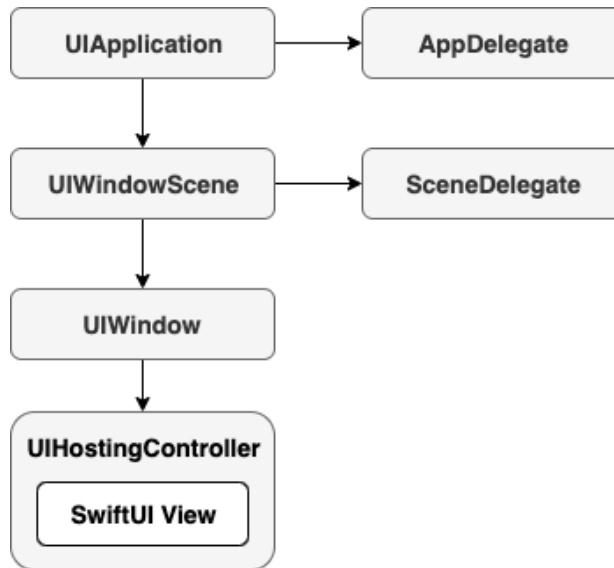


Figure 18-1

A multi-window app hierarchy, on the other hand, can be represented as shown in Figure 18-2 below. Note that while there is only one **AppDelegate**, each scene has its own **SceneDelegate** instance:

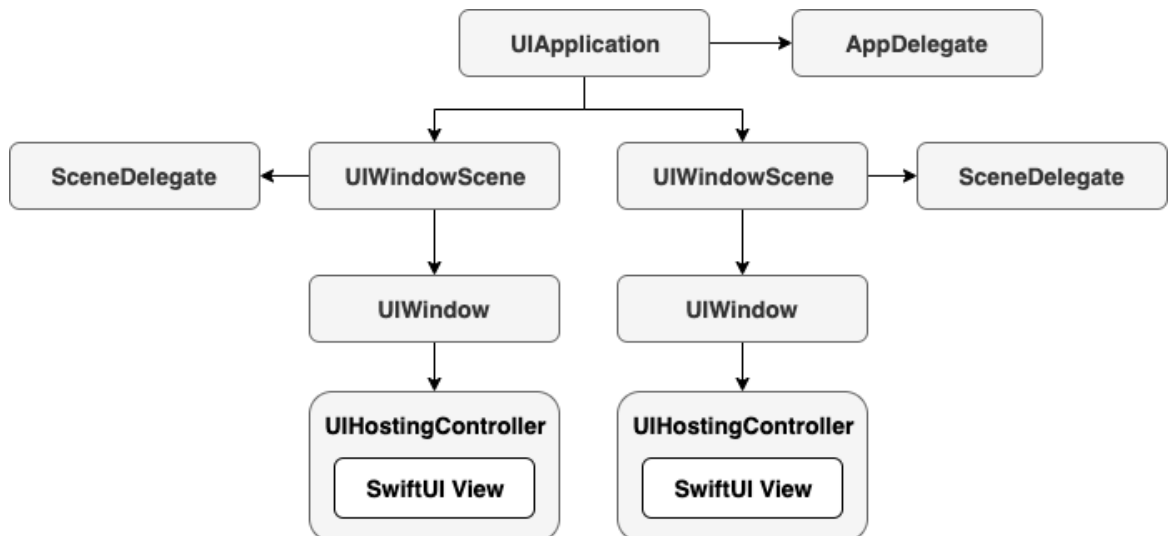


Figure 18-2

18.5 **ContentView.swift File**

This is a SwiftUI View file that contains the content of the first screen to appear when the app starts. This file and others like it are where most of the work is performed when developing apps in SwiftUI. By default, it contains a single Text view displaying the words “Hello World”.

18.6 **Assets.xcassets**

The Assets.xcassets folder contains the asset catalog that is used to store resources used by the app such as images, icons and colors.

18.7 **Info.plist**

The information property list file is an XML file containing key-value pairs used to configure the app. The setting to enable multi-window support, for example, is contained within this file.

18.8 **LaunchScreen.storyboard**

Contains the storyboard file containing the user interface layout for the screen displayed to the user while the app is launching. Since this is a UIKit Storyboard scene, it is designed using the Interface Builder tool rather than SwiftUI.

18.9 **Summary**

When a new SwiftUI project is created in Xcode using the Single View App template, Xcode automatically generates a number of files required for the app to function. All of these files and folders can be modified to add functionality to the app, both in terms of adding resource assets, performing initialization and deinitialization tasks and building the user interface and logic of the app. This chapter has provided a high-level overview of each of these files together with an outline of the internal architecture of a SwiftUI-based iOS app.

```
    }  
    .font(.largeTitle)  
  }  
}
```

Note that this declaration still returns an instance that complies with the View protocol and that the body contains the VStack declaration from the previous subview. Instead of including static views to be included in the stack, however, the child views of the stack will be passed to the initializer, handled by ViewBuilder and embedded into the VStack as child views. The custom MyVStack view can now be initialized with different child views wherever it is used in a layout, for example:

```
MyVStack {  
    Text("Text 1")  
    Text("Text 2")  
    HStack {  
        Image(systemName: "star.fill")  
        Image(systemName: "star.fill")  
        Image(systemName: "star")  
    }  
}
```

19.13 Summary

SwiftUI user interfaces are declared in SwiftUI View files and are composed of components that conform to the View protocol. To conform with the View protocol a structure must contain a property named body which is itself a View.

SwiftUI provides a library of built-in components that can be used to design user interface layouts. The appearance and behavior of a view can be configured by applying modifiers and views can be modified and grouped together to create custom views and subviews. Similarly, custom container views can be created using the ViewBuilder closure property.

When a modifier is applied to a view, a new modified view is returned and subsequent modifiers are then applied to this modified view. This can have significant implications for the order in which modifiers are applied to a view.

20. SwiftUI Stacks and Frames

User interface design is largely a matter of selecting the appropriate interface components, deciding how those views will be positioned on the screen, and then implementing navigation between the different screens and views of the app.

As is to be expected, SwiftUI includes a wide range of user interface components to be used when developing an app such as button, label, slider and toggle views. SwiftUI also provides a set of layout views for the purpose of defining both how the user interface is organized and the way in which the layout responds to changes in screen orientation and size.

This chapter will introduce the Stack container views included with SwiftUI and explain how they can be used to create user interface designs with relative ease.

Once stack views have been explained, this chapter will cover the concept of flexible frames and explain how they can be used to control the sizing behavior of views in a layout.

20.1 SwiftUI Stacks

SwiftUI includes three stack layout views in the form of VStack (vertical), HStack (horizontal) and ZStack (views are layered on top of each other).

A stack is declared by embedding child views into a stack view within the SwiftUI View file. In the following view, for example, three Image views have been embedded within an HStack:

```
struct ContentView: View {
    var body: some View {
        HStack {
            Image(systemName: "goforward.10")
            Image(systemName: "goforward.15")
            Image(systemName: "goforward.30")
        }
    }
}
```

Within the preview canvas, the above layout will appear as illustrated in Figure 20-1:



Figure 20-1

SwiftUI Stacks and Frames

A similarly configured example using a `VStack` would accomplish the same results with the images stacked vertically:

```
VStack {  
    Image(systemName: "goforward.10")  
    Image(systemName: "goforward.15")  
    Image(systemName: "goforward.30")  
}
```

To embed an existing component into a stack, either wrap it manually within a stack declaration, or hover the mouse pointer over the component in the editor so that it highlights, hold down the Command key on the keyboard and left-click on the component. From the resulting menu (Figure 20-2) select the appropriate option:

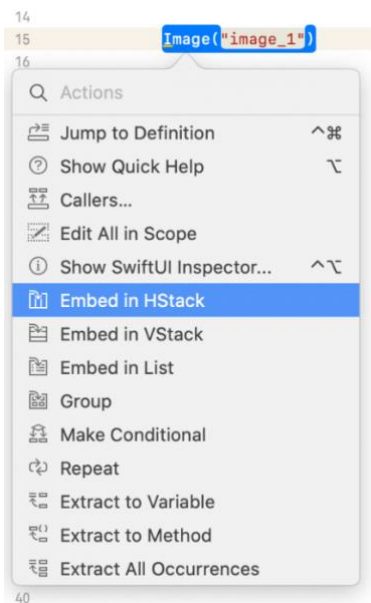


Figure 20-2

Layouts of considerable complexity can be designed simply by embedding stacks within other stacks, for example:

```
VStack {  
    Text("Financial Results")  
        .font(.title)  
  
    HStack {  
        Text("Q1 Sales")  
            .font(.headline)  
  
        VStack {  
            Text("January")  
        }  
    }  
}
```

```

        Text("February")
        Text("March")
    }

    VStack {
        Text("$1000")
        Text("$200")
        Text("$3000")
    }
}

```

The above layout will appear as shown in Figure 20-3:

Financial Results

January \$1000

Q1 Sales February \$200

March \$3000

Figure 20-3

As currently configured the layout clearly needs some additional work, particularly in terms of alignment and spacing. The layout can be improved in this regard using a combination of alignment settings, the `Spacer` component and the padding modifier.

20.2 Spacers, Alignment and Padding

To add space between views, SwiftUI includes the `Spacer` component. When used in a stack layout, the spacer will flexibly expand and contract along the axis of the containing stack (in other words either horizontally or vertically) to provide a gap between views positioned on either side, for example:

```

HStack(alignment: .top) {

    Text("Q1 Sales")
        .font(.headline)
    Spacer()
    VStack(alignment: .leading) {
        Text("January")
        Text("February")
        Text("March")
    }
    Spacer()
}

```

```
.  
.
```

In terms of aligning the content of a stack, this can be achieved by specifying an alignment value when the stack is declared, for example:

```
VStack(alignment: .center) {  
    Text("Financial Results")  
        .font(.title)
```

Alignments may also be specified with a corresponding spacing value:

```
VStack(alignment: .center, spacing: 15) {  
    Text("Financial Results")  
        .font(.title)
```

Spacing around the sides of any view may also be implemented using the *padding()* modifier. When called without a parameter SwiftUI will automatically use the best padding for the layout, content and screen size (referred to as *adaptable padding*). The following example sets adaptable padding on all four sides of a Text view:

```
Text("Hello World!")  
    .padding()
```

Alternatively, a specific amount of padding may be passed as a parameter to the modifier as follows:

```
Text("Hello World!")  
    .padding(15)
```

Padding may also be applied to a specific side of a view with or without a specific value. In the following example a specific padding size is applied to the top edge of a Text view:

```
Text("Hello World!")  
    .padding(.top, 10)
```

Making use of these options, the example layout created earlier in the chapter can be modified as follows:

```
VStack(alignment: .center, spacing: 15) {  
    Text("Financial Results")  
        .font(.title)  
  
    HStack(alignment: .top) {  
        Text("Q1 Sales")  
            .font(.headline)  
        Spacer()  
        VStack(alignment: .leading) {  
            Text("January")  
            Text("February")
```

```

        Text("March")
    }
    Spacer()
    VStack(alignment: .leading) {
        Text("$10000")
        Text("$200")
        Text("$3000")
    }
    .padding(5)
}
.padding(5)
}
.padding(5)
}

```

With the alignments, spacers and padding modifiers added, the layout should now resemble the following figure:

Financial Results

Q1 Sales	January	\$10000
	February	\$200
	March	\$3000

Figure 20-4

More advanced stack alignment topics will be covered in a later chapter entitled *SwiftUI Stack Alignment and Alignment Guides*.

20.3 Container Child Limit

Container views are limited to 10 direct descendent views. If a stack contains more than 10 direct children, Xcode will likely display the following syntax error:

```
Argument passed to call that takes no arguments
```

If a stack exceeds the 10 direct children limit, the views will need to be embedded into multiple containers. This can, of course, be achieved by adding stacks as subviews, but another useful container is the Group view. In the following example, a VStack can contain 12 Text views by splitting the views between Group containers giving the VStack only two direct descendants:

```

VStack {
    Group {

```



```
        Text("Sample Text")
        Text("Sample Text")
        Text("Sample Text")
        Text("Sample Text")
        Text("Sample Text")
        Text("Sample Text")
    }

    Group {
        Text("Sample Text")
        Text("Sample Text")
        Text("Sample Text")
        Text("Sample Text")
        Text("Sample Text")
        Text("Sample Text")
    }
}
```

In addition to providing a way to avoid the 10-view limit, groups are also useful when performing an operation on multiple views (for example, a set of related views can all be hidden in a single operation by embedding them in a `Group` and hiding that view).

20.4 Text Line Limits and Layout Priority

By default, an `HStack` will attempt to display the text within its `Text` view children on a single line. Take, for example, the following `HStack` declaration containing an `Image` view and two `Text` views:

```
HStack {
    Image(systemName: "airplane")
    Text("Flight times:")
    Text("London")
}
.font(.largeTitle)
```

If the stack has enough room, the above layout will appear as follows:



Figure 20-5

If a stack has insufficient room (for example if it is constrained by a frame or is competing for space with sibling views) the text will automatically wrap onto multiple lines when necessary:



Figure 20-6

While this may work for some situations, it may become an issue if the user interface is required to display this text in a single line. The number of lines over which text can flow can be restricted using the *lineCount()* modifier. The example HStack could, therefore, be limited to 1 line of text with the following change:

```
HStack {
    Image(systemName: "airplane")
    Text("Flight times:")
    Text("London")
}
.font(.largeTitle)
.lineLimit(1)
```

When an HStack has insufficient space to display the full text and is not permitted to wrap the text over enough lines, the view will resort to truncating the text, as is the case in Figure 20-7:



Figure 20-7

In the absence of any priority guidance, the stack view will decide how to truncate the Text views based on the available space and the length of the views. Obviously, the stack has no way of knowing whether the text in one view is more important than the text in another unless the text view declarations include some priority information. This is achieved by making use of the *layoutPriority()* modifier. This modifier can be added to the views in the stack and passed values indicating the level of priority for the corresponding view. The higher the number, the greater the layout priority and the less the view will be subjected to truncation.

Assuming the flight destination city name is more important than the “Flight times:” text, the example stack could be modified as follows:

```
HStack {
    Image(systemName: "airplane")
```

```
Text("Flight times:")
Text("London").layoutPriority(1)
}
.font(.largeTitle)
.lineLimit(1)
```

With a higher priority assigned to the city Text view (in the absence of a layout priority the other text view defaults to a priority of 0) the layout will now appear as illustrated in Figure 20-8:



Figure 20-8

20.5 SwiftUI Frames

By default, a view will be sized automatically based on its content and the requirements of any layout in which it may be embedded. Although much can be achieved using the stack layouts to control the size and positioning of a view, sometimes a view is required to be a specific size or to fit within a range of size dimensions. To address this need, SwiftUI includes the flexible frame modifier.

Consider the following Text view which has been modified to display a border:

```
Text("Hello World")
    .font(.largeTitle)
    .border(Color.black)
```

Within the preview canvas, the above text view will appear as follows:



Figure 20-9

In the absence of a frame, the text view has been sized to accommodate its content. If the Text view was required to have height and width dimensions of 100, however, a frame could be applied as follows:

```
Text("Hello World")
    .font(.largeTitle)
    .border(Color.black)
    .frame(width: 100, height: 100, alignment: .center)
```

Now that the Text view is constrained within a frame, the view will appear as follows:



Figure 20-10

In many cases, fixed dimensions will provide the required behavior. In other cases, such as when the content of a view changes dynamically, this can cause problems. Increasing the length of the text, for example, might cause the content to be truncated:



Figure 20-11

This can be resolved by creating a frame with minimum and maximum dimensions:

```
Text("Hello World, how are you?")  
    .font(.largeTitle)  
    .border(Color.black)  
    .frame(minWidth: 100, maxWidth: 300, minHeight: 100,  
           maxHeight: 100, alignment: .center)
```

Now that the frame has some flexibility, the view will be sized to accommodate the content within the defined minimum and maximum limits. When the text is short enough, the view will appear as shown in Figure 20-10 above. Longer text, however, will be displayed as follows:



Figure 20-12

Frames may also be configured to take up all the available space by setting the minimum and maximum values to 0 and infinity respectively:

```
.frame(minWidth: 0, maxWidth: .infinity, minHeight: 0,
       maxHeight: .infinity)
```

Remember that the order in which modifiers are chained often impacts the appearance of a view. In this case, if the border is to be drawn at the edges of the available space it will need to be applied to the frame:

```
Text("Hello World, how are you?")
    .font(.largeTitle)
    .frame(minWidth: 0, maxWidth: .infinity, minHeight: 0,
           maxHeight: .infinity)
    .border(Color.black, width: 5)
```

By default, the frame will honor the safe areas on the screen when filling the display. Areas considered to be outside the safe area include those occupied by the camera notch on some device models and the bar across the top of the screen displaying the time and Wi-Fi and cellular signal strength icons. To configure the frame to extend beyond the safe area, simply use the *edgesIgnoringSafeArea()* modifier, specifying the safe area edges to ignore:

```
.edgesIgnoringSafeArea(.all)
```

20.6 Frames and the Geometry Reader

Frames can also be implemented so that they are sized relative to the size of the container within which the corresponding view is embedded. This is achieved by wrapping the view in a *GeometryReader* and using the reader to identify the container dimensions. These dimensions can then be used to calculate the frame size. The following example uses a frame to set the dimensions of two *Text* views relative to the size of the containing *VStack*:

```
GeometryReader { geometry in
    VStack {
        Text("Hello World, how are you?")
            .font(.largeTitle)
```

```

        .frame(width: geometry.size.width / 2,
              height: (geometry.size.height / 4) * 3)
Text("Goodbye World")
    .font(.largeTitle)
    .frame(width: geometry.size.width / 3,
          height: geometry.size.height / 4)
}
}

```

The topmost Text view is configured to occupy half the width and three quarters of the height of the VStack while the lower Text view occupies one third of the width and one quarter of the height.

20.7 Summary

User interface design mostly involves gathering together components and laying them out on the screen in a way that provides a pleasant and intuitive user experience. User interface layouts must also be responsive so that they appear correctly on any device regardless of screen size and, ideally, device orientation. To ease the process of user interface layout design, SwiftUI provides several layout views and components. In this chapter we have looked at layout stack views and the flexible frame.

By default, a view will be sized according to its content and the restrictions imposed on it by any view in which it may be contained. When insufficient space is available, a view may be restricted in size resulting in truncated content. Priority settings can be used to control the amount by which views are reduced in size relative to container sibling views.

For greater control of the space allocated to a view, a flexible frame can be applied to the view. The frame can be fixed in size, constrained within a range of minimum and maximum values or, using a Geometry Reader, sized relative to the containing view.

Index

- .linear · 262
- automatically starting · 266
- autoreverse · 264
- explicit · 264
- implicit · 261
- repeating · 264
- animation() modifier · 262
- AnyObject · 106
- AnyTransition · 270
- App Icons · 313
- App Store
 - creating archive · 315
 - submission · 311
- AppDelegate · 143
- AppDelegate.swift file · 141
- Apple Developer Program · 3
- Application Performance · 135
- Array
 - mixed type · 106
- Array Initialization · 103
- Array Item Count · 104
- Array Items
 - accessing · 105
 - appending · 105
 - inserting and deleting · 105
- Array Iteration · 106
- Arrays
 - immutable · 103
 - mutable · 103
- as! keyword · 37
- Assets.xcassets · 144
- Assistant Editor · 304
- Attributes inspector · 128

B

- Bézier curves · 254
- binary operators · 39
- bit operators · 44
- Bitcode · 317
- Bitwise AND · 45
- Bitwise Left Shift · 46
- bitwise OR · 45
- bitwise right shift · 46
- bitwise XOR · 45
- body · 151
- Boolean Logical Operators · 42
- break statement · 51
- Build Errors · 135

C

- callout · 151
- Capsule() · 252
- caption · 151
- case Statement · 58
- CGRect · 254, 256
- Character data type · 27
- Child Limit · 161
- Class Extensions · 91
- closed range operator · 42
- closeSubPath() · 255
- closure expressions · 71
- Closure Expressions · 71
- closures · 63
- Closures · 73
- code editor · 123
 - context menu · 129
- Combine framework · 173
- Comparable protocol · 100
- Comparison Operators · 41
- Compound Assignment Operators · 40
- Compound Bitwise Operators · 47
- concrete type · 85
- Conditional Control Flow · 52
- constants · 30
- Container Alignment · 199
- Container Child Limit · 161
- Container Views · 155
- ContentView.swift file · 122, 144
- Context Menus · 247
- continue Statement · 52
- Coordinator · 283
- Custom Alignment Types · 206
- Custom Container Views · 155
- custom fonts · 151
- Custom Paths · 254
- Custom Shapes · 254

D

- dark mode
 - previewing · 131
- data encapsulation · 76
- Debug Navigator · 135
- debug panel · 123
- Debug View Hierarchy · 137
- Declarative Syntax · 118
- Default Function Parameters · 66
- defer statement · 114

- Developer Program · 3
- Devices
 - managing · 134
- Dictionary Collections · 107
- Dictionary Entries
 - adding and removing · 109
- Dictionary Initialization · 107
- Dictionary Item Count · 108
- Dictionary Items
 - accessing and updating · 109
- Dictionary Iteration · 109
- dismantleUIView() · 282
- Divider view · 189
- do-catch statement · 113
- Double · 27
- downcasting · 36
- DragGesture.Value · 277
- Dynamic Lists · 219

E

- EditButton view · 241
- Embed in VStack · 180
- Environment Object
 - example · 169, 191, 196
- Environment Objects · 174
- EnvironmentObject · 169
- Errata · 2
- Error
 - throwing · 112
- Error Catching
 - disabling · 114
- Error Object
 - accessing · 114
- ErrorType protocol · 111
- Event handling · 153
- exclusive OR · 45
- Explicit Animation · 264
- Expression Syntax · 39
- external parameter names · 65

F

- fallthrough statement · 60
- fill() modifier · 251
- Flexible frames · *See* Frames
- Float · 27
- flow control · 49
- font
 - create custom · 151

- footnote · 151
- for loop · 49
- forced unwrapping · 33
- ForEach · 186, 190, 220, 231, 232
- foregroundColor() modifier · 252
- Form container · 234
- Frames · 157, 164
 - Geometry Reader · 166
 - infinity · 166
- function
 - arguments · 63
 - parameters · 63
- Function Parameters
 - variable number of · 67
- functions · 63
 - as parameters · 69
 - default function parameters · 66
 - external parameter names · 65
 - In-Out Parameters · 68
 - parameters as variables · 68
 - return multiple results · 67

G

- GeometryReader · 166
- gesture recognizer
 - removal of · 275
- Gesture Recognizers · 273
 - exclusive · 278
 - onChanged · 275
 - sequenced · 278
 - simultaneous · 278
 - updating · 276
- gesture() modifier · 273
- Gestures
 - composing · 278
- Gradients
 - drawing · 257
 - LinearGradient · 259
 - RadialGradient · 259
- Graphics
 - drawing · 251
 - overlays · 254
- Graphics Drawing · 251
- guard statement · 54

H

- half-closed range operator · 43
- headline · 151

Index

HorizontalAlignment · 204, 207
Hosting Controller · 299
 adding · 302
HStack · 148, 157

I

if ... else ... Statements · 53
if ... else if ... Statements · 54
if Statement · 53
Image view · 157
implicit alignment · 199
Implicit Animation · 261
implicitly unwrapped · 35
Info.plist · 144
Inheritance, Classes and Subclasses · 87
init method · 78
inout keyword · 69
In-Out Parameters · 68
Instance Properties · 76
Interface Builder · 117
iOS 13
 adoption rate · 119
iOS 13 SDK
 installation · 7
 system requirements · 7
iOS Distribution Certificate · 311
is keyword · 38
iTunes Connect · 316

J

JSON · 229
 loading · 229

L

Launch Screen · 314
LaunchScreen.storyboard · 144, 314
Layout Hierarchy · 136
Layout Priority · 162
lazy
 keyword · 82
Lazy properties · 81
Left Shift Operator · 46
Library panel · 125
LinearGradient · 259
List view · 217

 adding navigation · 235
 dynamic · 219
 making editable · 223
List view
 tutorial · 227
Live Preview · 124
local parameter names · 65
Loops
 breaking from · 51

M

Main.storyboard file · 301
makeUIView() · 282
mathematical expressions · 39
Methods
 declaring · 77
Mixed Type Arrays · 106
modifier() method · 153
Modifiers · 153
Multiple Device Configurations · 129

N

Navigation · 217
 implementing · 195
 tutorial · 227
navigationBarItems() modifier · 224, 239
navigationBarTitle() modifier · 239
NavLink · 221, 240
NavigationView · 221
Network Testing · 135
new line · 29
NOT (!) operator · 42

O

Objective-C · 25
Observable Object
 example · 191
ObservableObject · 169, 172
ObservableObject protocol · 172
onAppear() · 154, 267, 268
onChanged() · 275
onDelete() · 223, 241
onDisappear() · 154
onMove() · 224, 241
Opaque Return Types · 85

- operands · 39
- optional
 - implicitly unwrapped · 35
- optional binding · 33
- Optional Type · 32
- OR (| |) operator · 42
- OR operator · 45
- Overlays · 254

P

- Padding · 159
- padding() modifier · 160
- parent class · 75
- Path object · 254
- Paths · 254
- Performance
 - monitoring · 135
- Physical iOS Device · 132
 - running app on · 132
- Picker view · 169
 - example · 185
- playground
 - quick look · 16
 - results panel · 12
- Playground · 11
 - adding resources · 20
 - creating a · 11
 - Enhanced Live Views · 21
 - pages · 18
 - Rich Text Comments · 16
- playground editor · 12
- Playground Timelines · 15
- preferred text size · 150
- Preview Canvas · 123
- Preview on Device · 124
- Preview Pinning · 125
- PreviewProvider protocol · 130
- Profile in Instruments · 136
- Property Wrappers · 97
 - example · 97
 - Multiple Variables and Types · 99
- Protocols · 84
- published properties · 172

R

- Range Operators · 42
- Rectangle() · 251
- Reference Types · 94

- repeat ... while loop · 51
- repeatCount() modifier · 264
- repeatForever() modifier · 264
- Resume button · 124
- Right Shift Operator · 46
- root view controller · 179
- Rotation · 183
- running an app · 131

S

- SceneDelegate · 142, 143
- SceneDelegate.swift file · 142, 175
- Segue Action · 304
- self · 82
- SF Symbols · 171
 - macOS app · 171
- Shapes · 254
 - drawing · 251
- sign bit · 46
- Signing Identities · 9
- Simulator
 - running app · 131
- Simulators
 - managing · 134
- Slider view · 180
- some
 - keyword · 85
- source code
 - download · 2
- Spacer view · 188
- Spacer View · 159
- Spacers · 159
- spring() modifier · 263
- Stack
 - embed views in a · 180
- Stacks · 157
 - alignment · 199
 - alignment guides · 199
 - child limit · 161
 - cross stack alignment · 209
 - implicit alignment · 199
 - Layout Priority · 162
- State Binding · 171
- State properties · 169
 - binding · 170
 - example · 181
- Stored and Computed Properties · 80
- String
 - data type · 28
- stroke() modifier · 252

Index

- StrokeStyle · 252
- struct keyword · 93
- Structures · 93
- subheadline · 151
- subtraction operator · 40
- Subviews · 148
- Swift
 - Arithmetic Operators · 39
 - array iteration · 106
 - arrays · 103
 - Assignment Operator · 39
 - base class · 87
 - Binary Operators · 41
 - Bitwise AND · 45
 - Bitwise Left Shift · 46
 - Bitwise NOT · 44
 - Bitwise Operators · 44
 - Bitwise OR · 45
 - Bitwise Right Shift · 46
 - Bitwise XOR · 45
 - Bool · 27
 - Boolean Logical Operators · 42
 - break statement · 51
 - calling a function · 65
 - case statement · 58
 - character data type · 27
 - child class · 87
 - class declaration · 75
 - class deinitialization · 78
 - class extensions · 91
 - class hierarchy · 87
 - class initialization · 78
 - Class Methods · 76
 - class properties · 75
 - closed range operator · 42
 - Closure Expressions · 71
 - Closures · 73
 - Comparison Operators · 41
 - Compound Assignment Operators · 40
 - Compound Bitwise Operators · 47
 - Conditional Operator · 43
 - constant declaration · 30
 - constants · 30
 - continue statement · 52
 - control flow · 49
 - data types · 25, 26
 - Dictionaries · 107
 - do ... while loop · 51
 - error handling · 111
 - Escape Sequences · 29
 - exclusive OR · 45
 - expressions · 39
 - floating point · 27
 - flow control · 53
 - for Statement · 49
 - function declaration · 63
 - functions · 63
 - guard statement · 54
 - half-closed range operator · 43
 - if ... else ... Statements · 53
 - if Statement · 53
 - implicit returns* · 2, 25, 64
 - Inheritance, Classes and Subclasses · 87
 - Instance Properties · 76
 - instance variables · 76
 - integers · 26
 - methods · 75
 - object oriented programming · 75
 - opaque return types · 85
 - operators · 39
 - optional binding · 33
 - optional type · 32
 - Overriding · 89
 - parent class · 87
 - Property Wrappers · 97
 - protocols · 84
 - Range Operators · 42
 - Reference Types · 94
 - root class · 87
 - single expression functions · 64
 - single expression returns · 64
 - single inheritance · 87
 - Special Characters · 29
 - Stored and Computed Properties · 80
 - String data type · 28
 - structures · 93
 - subclass · 87
 - switch fallthrough · 60
 - switch statement · 57
 - syntax · 57
 - Ternary Operator · 43
 - tuples · 31
 - type annotations · 30
 - type casting · 36
 - type checking · 36
 - type inference · 30
 - Value Types · 94
 - variable declaration · 30
 - variables · 30
 - while loop · 50
- Swift Playground · 11
- Swift Structures · 93
- SwiftUI
 - create project · 122

- custom views · 145
- data driven · 118
- Declarative Syntax · 118
- example project · 177
- overview · 117
- Subviews · 148
- Views · 145
- SwiftUI Project
 - anatomy of · 141
 - creating · 122
- SwiftUI View template · 194
- SwiftUI Views · 145
- SwiftUI vs. UIKit · 119
- switch statement · 57
 - example · 58
- switch Statement · 57
 - example · 58
 - range matching · 59

T

- Tab Item Tags · 245
- Tab Items · 244
- Tabbed Views · 243
- tabItem() · 245
- TabView · 243
 - tab items · 244
- tag() · 245
- ternary operator · 43
- Text Styles
 - body · 151
 - callout · 151
 - caption · 151
 - footnote · 151
 - headline · 151
 - subheadline · 151
- Text Styles · 150
- Text view
 - adding modifiers · 182
 - line limits · 162
- TextField view · 185
- throw statement · 112
- ToggleButton view · 171
- transition() modifier · 269
- Transitions · 261, 269
 - .move(edge: edge) · 269
 - .opacity · 269
 - .scale · 269
 - .slide · 269
 - asymmetrical · 271
 - combining · 270
- try statement · 113
- try! statement · 114
- Tuple · 31
- type annotation · 30
- Type Annotations · 30
- type casting · 36
- Type Casting · 36
- Type Checking · 36
- type inference · 31
- Type Inference · 30
- type safe programming · 30

U

- UIApplication · 142
- UIHostingController · 142, 299
- UIImagePickerController · 291
- UIKit · 117
 - in playgrounds · 18
- UIKit integration
 - data sources · 284
 - delegates · 284
- UIKit Integration · 281
 - Coordinator · 283
- UILabel
 - set color · 18
- UInt16 · 27
- UInt32 · 27
- UInt64 · 27
- UInt8 · 27
- UIScrollView · 284
- UIView · 281
 - SwiftUI integration · 281
- UIViewController · 291
 - SwiftUI integration · 291
- UIViewControllerRepresentable protocol · 291
- UIViewRepresentable protocol · 283
 - makeCoordinator() · 283
- UIWindow · 142, 143
- UIWindowScene · 142
- UIWindowSceneDelegate protocol · 142
- unary negative operator · 40
- Unicode scalar · 29
- upcasting · 36
- updateView() · 282
- UUID() method · 219

V

- Value Types · 94

Index

variables · 30
variadic parameters · 67
VerticalAlignment · 204, 207
View Hierarchy
 exploring the · 137
ViewBuilder · 155
ViewDimensions · 207
ViewDimensions object · 203
ViewModifier protocol · 153
Views
 adding · 194
 as properties · 149
 modifying · 150
VStack · 146, 157
 adding to layout · 180

W

where clause · 35
where statement · 60
while Loop · 50
willConnectTo · 142, 179
withAnimation() closure · 264

X

Xcode

account configuration · 8
Attributes inspector · 128
code editor · 123
create project · 122
debug panel · 123
Library panel · 125
Live Preview · 124
preferences · 8
preview canvas · 123
Preview on Device · 124
Preview Resume button · 124
project navigation panel · 122
starting · 121
SwiftUI mode · 121
Xcode 11
 installation · 7
 system requirements · 7
XCPlayground module · 21
XCShowView · 22
XOR operator · 45

Z

ZStack · 157, 199
 alignment · 212
ZStack Custom Alignment · 212