SwiftUI Essentials

iOS 15 Edition

SwiftUI Essentials – iOS 15 Edition

ISBN-13: 978-1-951442-44-6

Rev: 1.0

Contents

Table of Contents

Chapter 1

1. Start Here

The goal of this book is to teach the skills necessary to build iOS 15 applications using SwiftUI, Xcode 13 and the Swift 5.5 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 5.5 programming language including data types, control flow, functions, object-oriented programming, property wrappers, structured concurrency, 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 in addition to observable, state and environment objects, as are key user interface design concepts such as modifiers, lists, tabbed views, context menus, user interface navigation, and outline groups.

The book also includes chapters covering graphics drawing, user interface animation, view transitions and gesture handling, WidgetKit, document-based apps, Core Data, CloudKit, and SiriKit integration.

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 15 using SwiftUI. Assuming you are ready to download the iOS 15 SDK and Xcode 13 and have an Apple Mac system 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 5.5 programming language, you can probably skip the Swift specific chapters. If you are not yet familiar with the SwiftUI specific language features of Swift, 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 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-ios15/

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-ios15.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*.

Chapter 2

2. Joining the Apple Developer Program

The first step in the process of learning to develop iOS 15 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 13 and the iOS 15 SDK

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

2.2 Apple Developer Program

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

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 Siri integration, iCloud access, Apple Pay, Game Center and In-App Purchasing can only be enabled and tested with Apple Developer Program membership.

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

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

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

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

2.3 When to Enroll in the Apple Developer Program?

Clearly, there are many benefits to Apple Developer Program membership and, eventually, membership will be necessary to begin selling your apps. As to whether to pay the enrollment fee now or later will depend on individual circumstances. If you are still in the early stages of learning to develop iOS apps or have yet to come

up with a compelling idea for an app to develop then much of what you need is provided without program membership. As your skill level increases and your ideas for apps to develop take shape you can, after all, always enroll in the developer program later.

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

2.4 Enrolling in the Apple Developer Program

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

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

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

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

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

https://developer.apple.com/membercenter

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

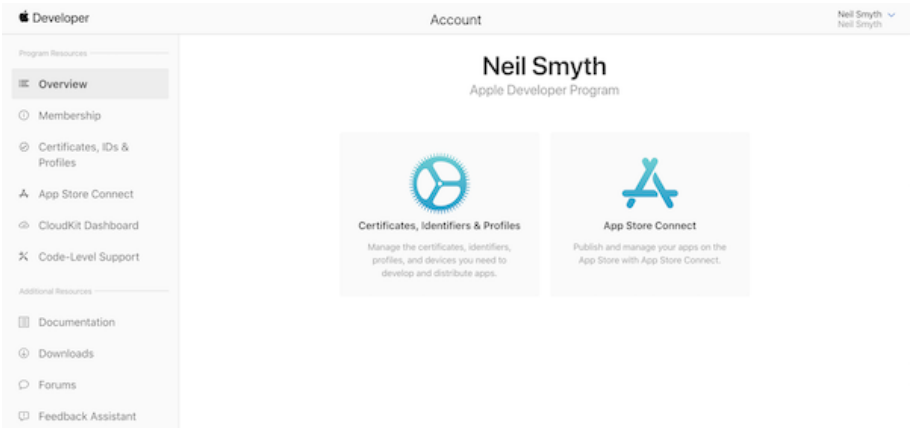

Figure 2-1

2.5 Summary

An important early step in the iOS 15 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 15 SDK and Xcode 13 development environment.

Chapter 3

3. Installing Xcode 13 and the iOS 15 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.

All of the examples in this book are based on Xcode version 12.2 and make use of features unavailable in earlier Xcode versions. In this chapter we will cover the steps involved in installing both Xcode 13.2 and the iOS 15 SDK on macOS.

3.1 Identifying Your macOS Version

When developing with SwiftUI, the Xcode 13.2 environment requires a system running macOS Big Sur (version 11.0) 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 11.0 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 13 and the iOS 15 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. This will install both Xcode and the iOS SDK.

3.3 Starting Xcode

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

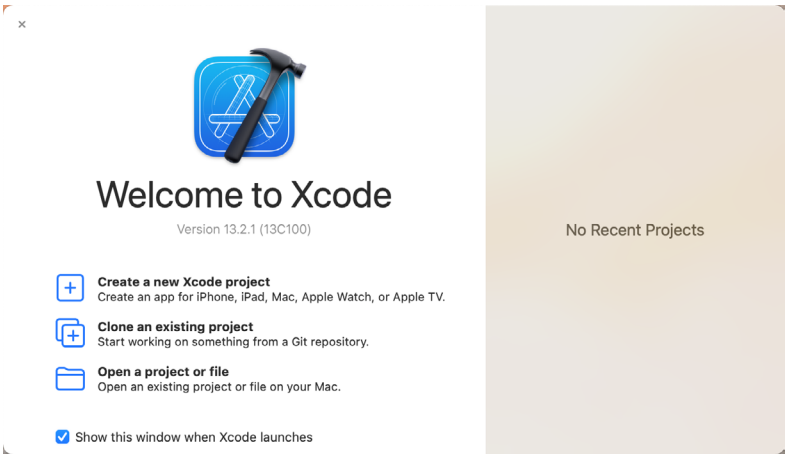

Figure 3-2

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 password before clicking on the *Sign In* button to add the account to the preferences.

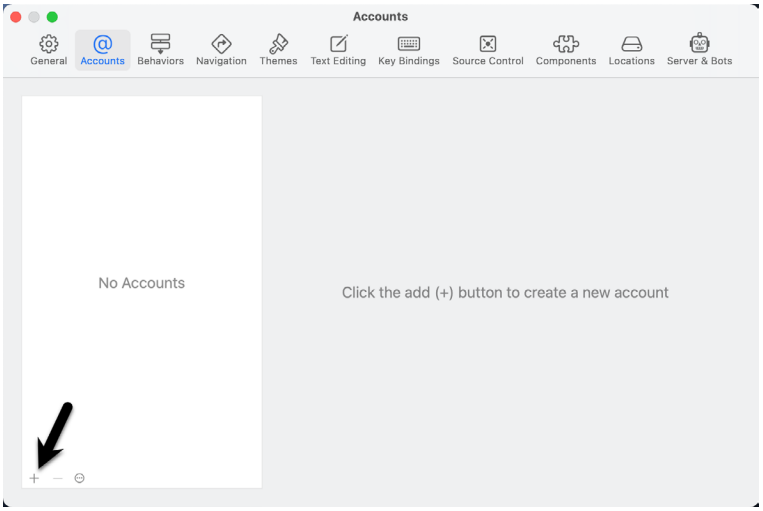

Figure 3-3

3.5 Developer and Distribution Signing Identities

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

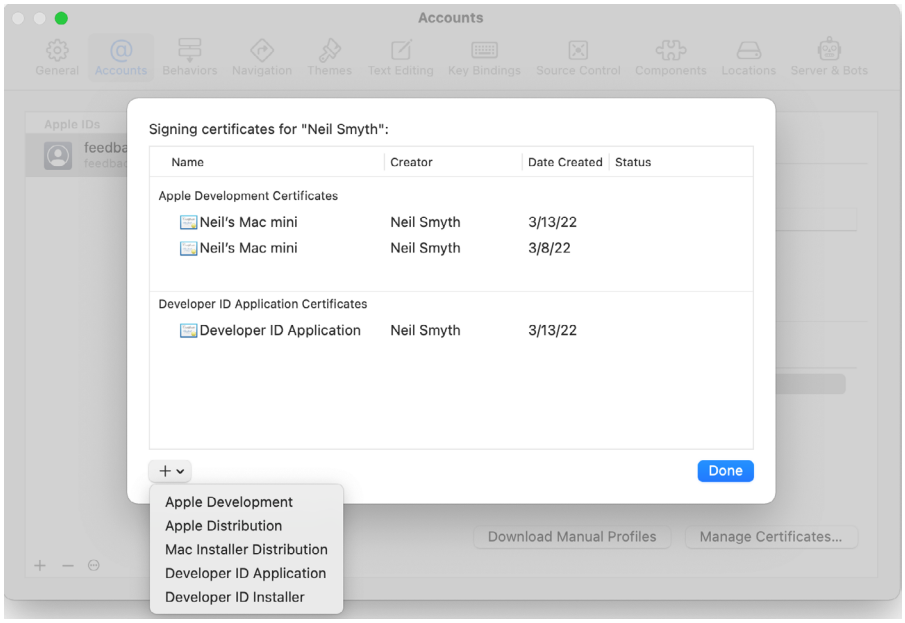

Figure 3-4

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

3.6 Summary

This book was written using Xcode version 13.2.1 and the iOS 15 SDK running on macOS 11.0 (Big Sur). Before beginning SwiftUI development, the first step is to install Xcode and configure it with your Apple ID via the accounts section of the Preferences screen. Once these steps have been performed, a development certificate must be generated which will be used to sign apps developed within Xcode. This will allow you to build and test your apps on physical iOS-based devices.

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

Having installed the iOS SDK and successfully launched Xcode 13 we can now look at Xcode in more detail, starting with Playgrounds.

Chapter 4

4. An Introduction to Xcode 13 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 *File -> New -> Playground…* menu option. Choose the iOS option on the resulting panel and select the Blank template.

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

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

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

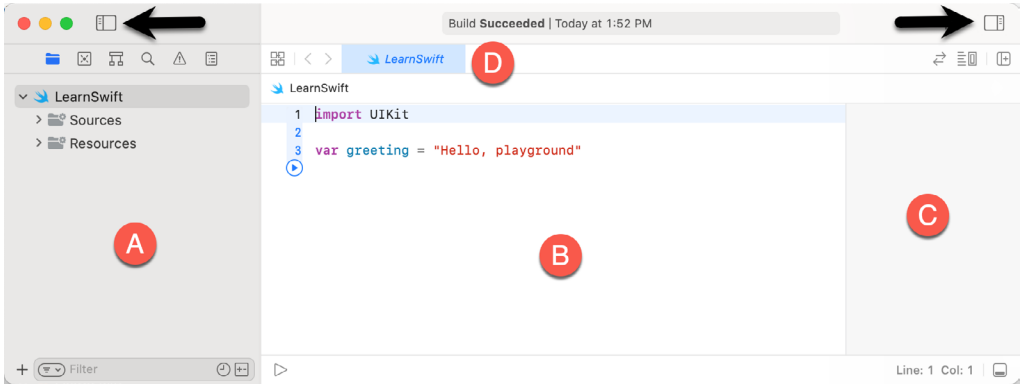

Figure 4-1

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

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

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

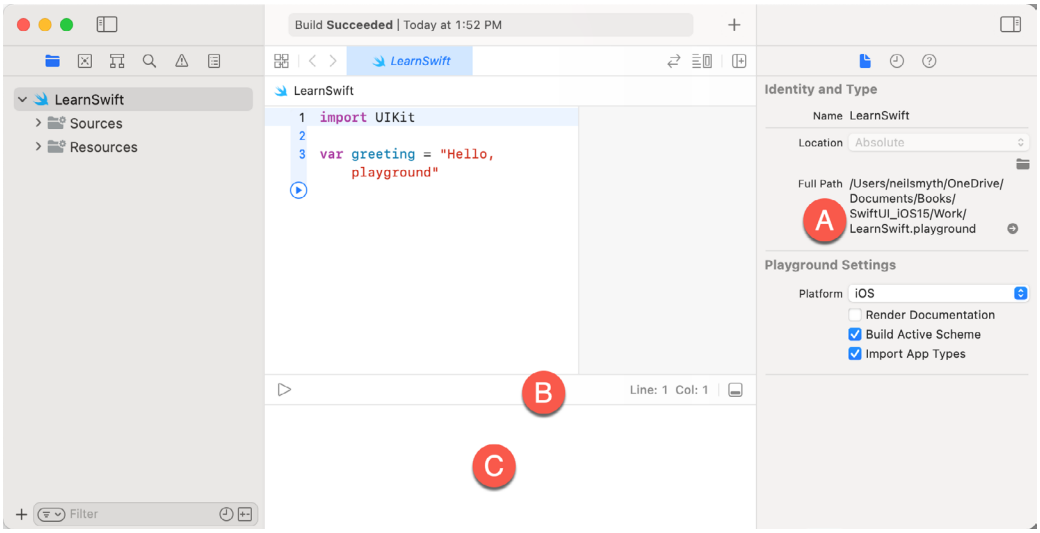

Figure 4-2

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

4.3 A Swift Playground Example

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

```
import UIKit


var 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:

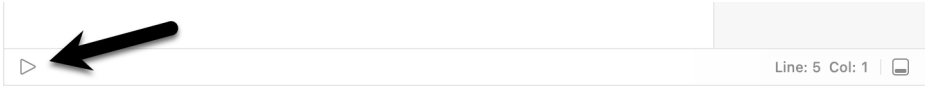

Figure 4-3

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:

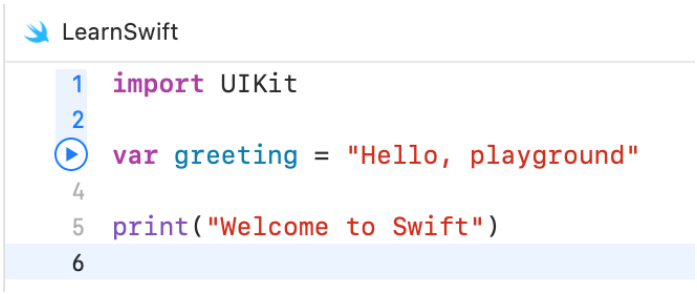

Figure 4-4

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

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

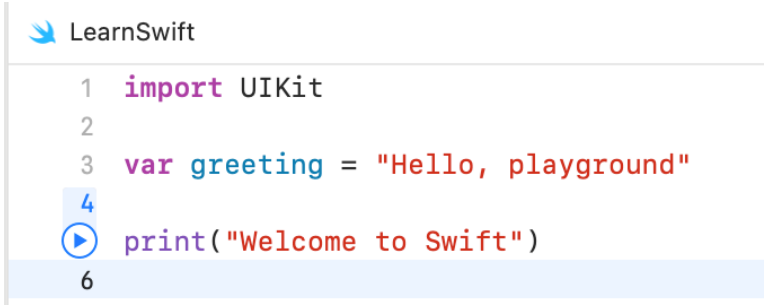

Figure 4-5

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

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

Figure 4-6

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

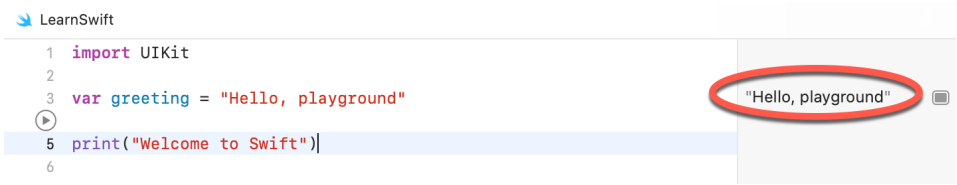

Figure 4-7

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

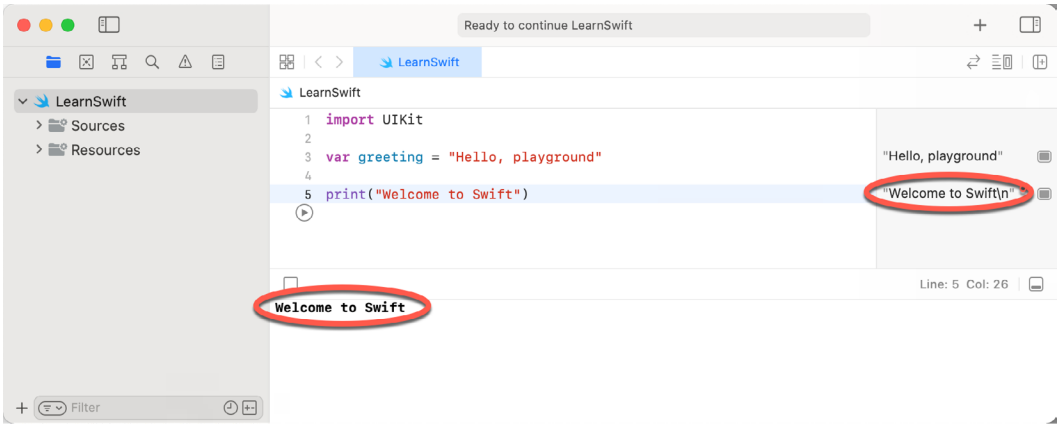

Figure 4-8

4.4 Viewing Results

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

```
var x = 10

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

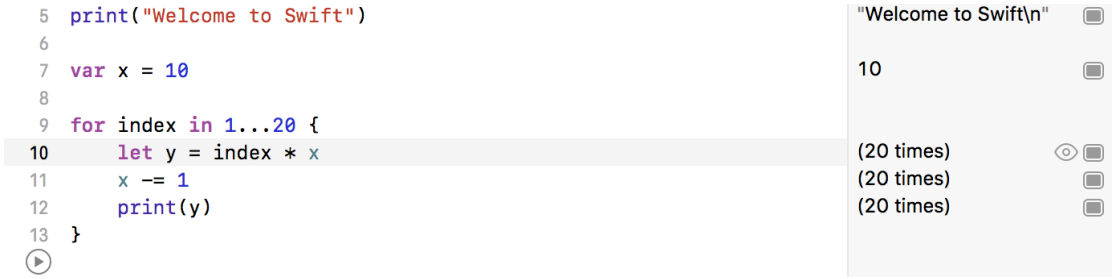

Figure 4-9

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:

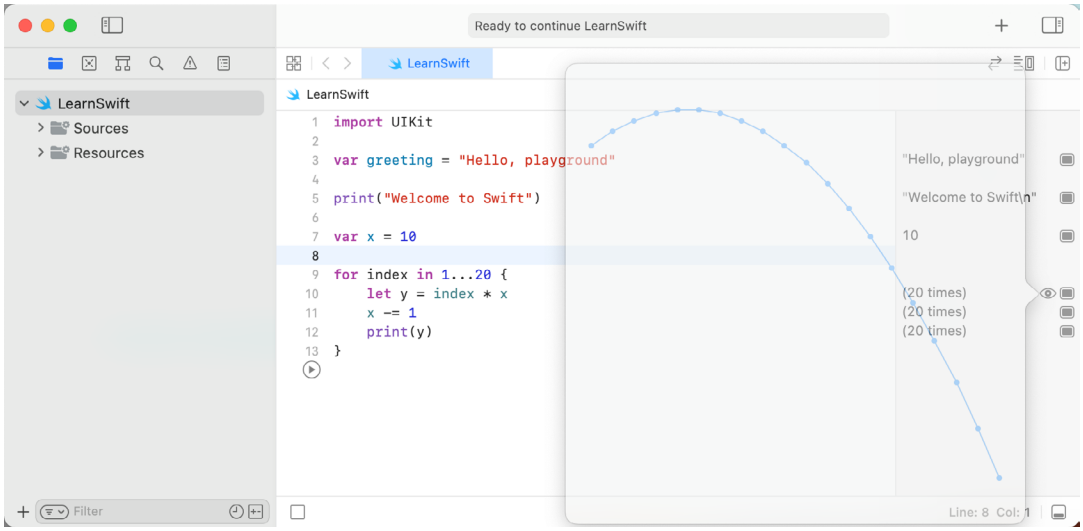

Figure 4-10

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

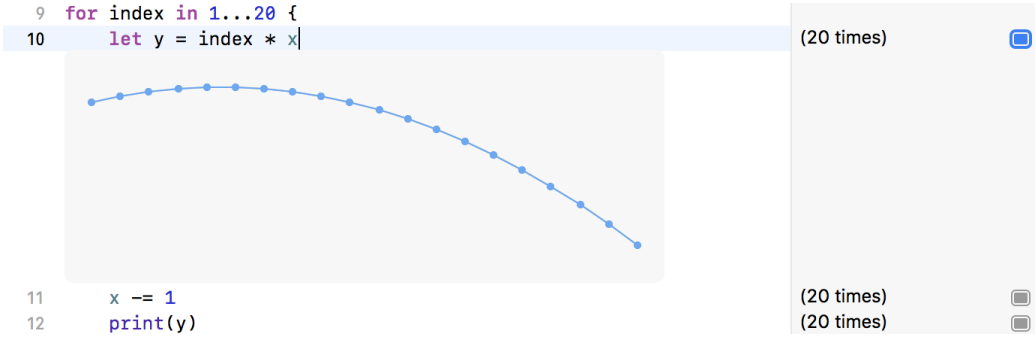

Figure 4-11

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 A in Figure 4-2). If the Inspector panel is not currently visible, click on the button indicated by the right-most arrow in Figure 4-1 to display it. Once rendered, the above rich text should appear as illustrated in Figure 4-12:

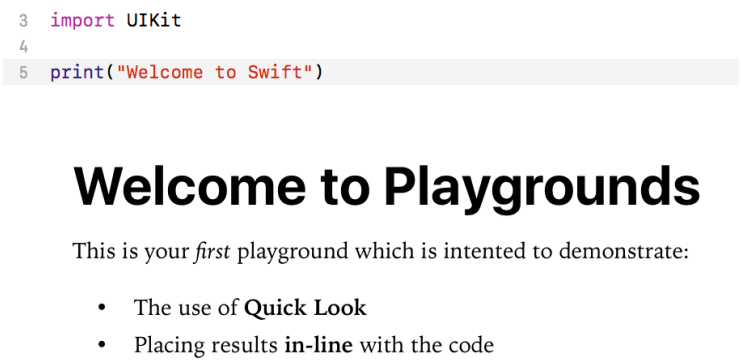

Figure 4-12

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 the button indicated by the left-most arrow in Figure 4-1 above to display it. Note that two pages are now listed in the Navigator

named “Untitled Page” and “Untitled Page 2”. Select and then click a second time on the “Untitled Page 2” entry so that the name becomes editable and change the name to *SwiftUI Example* as outlined in Figure 4-13:

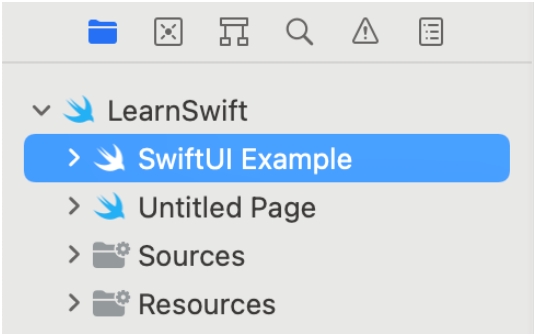

Figure 4-13

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

4.7 Working with SwiftUI and Live View in Playgrounds

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

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

```
import SwiftUI
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 (rest assured, all of the techniques used in this example will be thoroughly explained in later chapters):

```
struct ExampleView: View {

    var body: some View {

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

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

The PlaygroundSupport module includes a class named PlaygroundPage which allows playground code to interact with the pages that make up a playground. This is achieved through a range of methods and properties of the class, one of which is the *current* property. This property, in turn, provides access to the current playground page. 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 4-14:

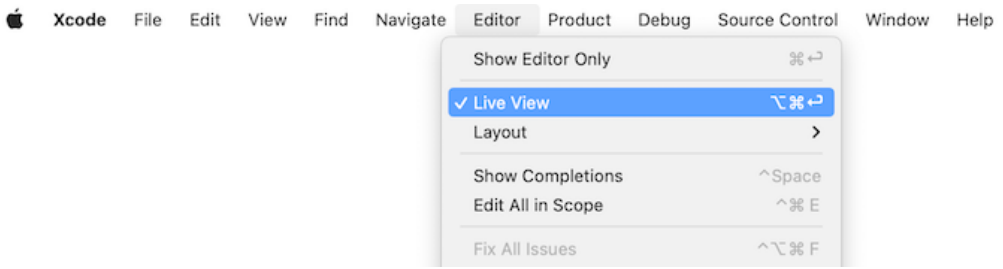

Figure 4-14

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

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

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

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

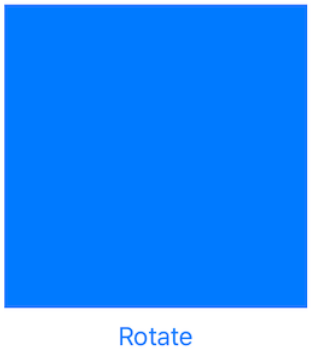

Figure 4-15

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

```
import SwiftUI
import PlaygroundSupport

struct ExampleView: View {

    @State private var rotation: Double = 0

    var body: some View {

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

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

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

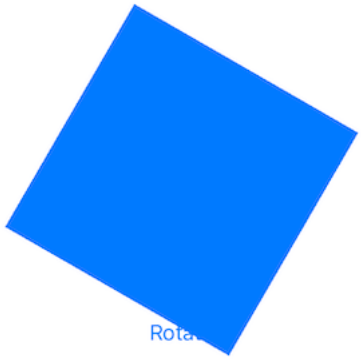

Figure 4-16

4.8 Summary

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

Chapter 5

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 *implicit returns from single expressions*, *opaque return types* and *property wrappers* you should at least read the sections and chapters relating to these features 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, iPadOS, macOS, watchOS and tvOS with the advantage that much of the syntax will be familiar to those with experience of other programming languages.

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 Xcode 13 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 subjective 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. Alternatively, the Float16 type may be used to store 16-bit floating point values. Float16 provides greater performance at the expense of lower precision.

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 Double value and a String as follows:

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

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

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

Alternatively, all 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 species (Swift arrays will be covered n 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:

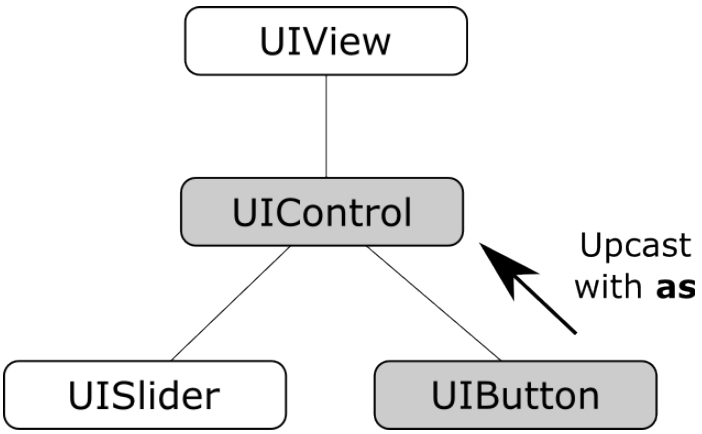

Figure 5-1

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:

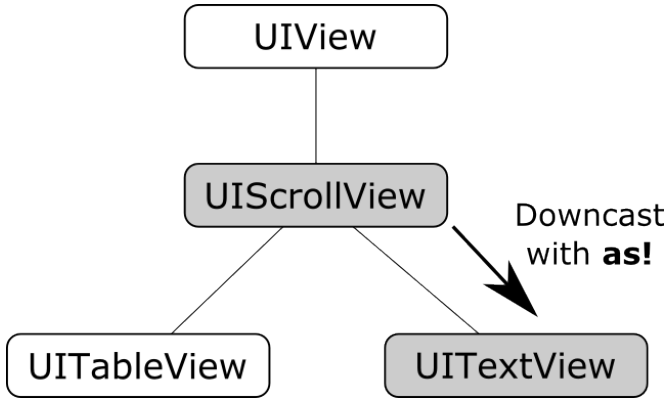

Figure 5-2

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.

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.

Chapter 7

7. Swift Control Flow

Regardless of the programming language used, application development is largely an exercise in applying logic, and much of the art of programming involves writing code that makes decisions based on one or more criteria. Such decisions define which code gets executed, how many times it is executed and, conversely, which code gets by-passed when the program is executing. This is often referred to as *control flow* since it controls the *flow* of program execution. Control flow typically falls into the categories of *looping control* (how often code is executed) and *conditional control flow* (whether code is executed). This chapter is intended to provide an introductory overview of both types of control flow in Swift.

7.1 Looping Control Flow

This chapter will begin by looking at control flow in the form of loops. Loops are essentially sequences of Swift statements which are to be executed repeatedly until a specified condition is met. The first looping statement we will explore is the *for-in* loop.

7.2 The Swift for-in Statement

The *for-in* loop is used to iterate over a sequence of items contained in a collection or number range and provides a simple to use looping option.

The syntax of the for-in loop is as follows:

```
for constant name in collection or range {
    // code to be executed
}
```

In this syntax, *constant name* is the name to be used for a constant that will contain the current item from the collection or range through which the loop is iterating. The code in the body of the loop will typically use this constant name as a reference to the current item in the loop cycle. The *collection* or *range* references the item through which the loop is iterating. This could, for example, be an array of string values, a range operator or even a string of characters (the topic of collections will be covered in greater detail within the chapter entitled "*Working with Array and Dictionary Collections in Swift*").

Consider, for example, the following for-in loop construct:

```
for index in 1...5 {
      print("Value of index is \(index)")
}
```

The loop begins by stating that the current item is to be assigned to a constant named *index*. The statement then declares a closed range operator to indicate that the for loop is to iterate through a range of numbers, starting at 1 and ending at 5. The body of the loop simply prints out a message to the console panel indicating the current value assigned to the *index* constant, resulting in the following output:

```
Value of index is 1
Value of index is 2
Value of index is 3
Value of index is 4
Value of index is 5
```

As will be demonstrated in the *"Working with Array and Dictionary Collections in Swift"* chapter of this book, the *for-in* loop is of particular benefit when working with collections such as arrays and dictionaries.

The declaration of a constant name in which to store a reference to the current item is not mandatory. In the event that a reference to the current item is not required in the body of the *for* loop, the constant name in the *for* loop declaration can be replaced by an underscore character. For example:

```
var count = 0

for _ in 1...5 {
    // No reference to the current value is required.
    count += 1
}
```

7.2.1 The while Loop

The Swift *for* loop described previously works well when it is known in advance how many times a particular task needs to be repeated in a program. There will, however, be instances where code needs to be repeated until a certain condition is met, with no way of knowing in advance how many repetitions are going to be needed to meet that criteria. To address this need, Swift provides the *while* loop.

Essentially, the *while* loop repeats a set of tasks while a specified condition is met. The *while* loop syntax is defined as follows:

```
while condition {
      // Swift statements go here
}
```

In the above syntax, *condition* is an expression that will return either *true* or *false* and the *// Swift statements go here* comment represents the code to be executed while the *condition* expression is *true*. For example:

```
var myCount = 0

while  myCount < 100 {
      myCount += 1
}
```

In the above example, the *while* expression will evaluate whether the *myCount* variable is less than 100. If it is already greater than 100, the code in the braces is skipped and the loop exits without performing any tasks.

If, on the other hand, *myCount* is not greater than 100 the code in the braces is executed and the loop returns to the *while* statement and repeats the evaluation of *myCount*. This process repeats until the value of *myCount* is greater than 100, at which point the loop exits.

7.3 The repeat ... while loop

The *repeat … while* loop replaces the Swift 1.x *do .. while* loop. It is often helpful to think of the *repeat ... while* loop as an inverted *while* loop. The *while* loop evaluates an expression before executing the code contained in the body of the loop. If the expression evaluates to *false* on the first check then the code is not executed. The *repeat ... while* loop, on the other hand, is provided for situations where you know that the code contained in the body of the loop will *always* need to be executed at least once. For example, you may want to keep stepping through the items in an array until a specific item is found. You know that you have to at least check the first item in the array to have any hope of finding the entry you need. The syntax for the *repeat ... while* loop is as follows:

```
repeat {
       // Swift statements here
```

```
} while conditional expression
```

In the *repeat ... while* example below the loop will continue until the value of a variable named *i* equals 0:

```
var i = 10

repeat {
       i -= 1
} while (i > 0)
```

7.4 Breaking from Loops

Having created a loop, it is possible that under certain conditions you might want to break out of the loop before the completion criteria have been met (particularly if you have created an infinite loop). One such example might involve continually checking for activity on a network socket. Once activity has been detected it will most likely be necessary to break out of the monitoring loop and perform some other task.

For the purpose of breaking out of a loop, Swift provides the *break* statement which breaks out of the current loop and resumes execution at the code directly after the loop. For example:

```
var j = 10

for _ in 0 ..< 100
{
    j += j

    if j > 100 {
        break
    }

    print("j = \(j)")
}
```

In the above example the loop will continue to execute until the value of j exceeds 100 at which point the loop will exit and execution will continue with the next line of code after the loop.

7.5 The continue Statement

The *continue* statement causes all remaining code statements in a loop to be skipped, and execution to be returned to the top of the loop. In the following example, the print function is only called when the value of variable *i* is an even number:

```
var i = 1

while i < 20
{
        i += 1

        if (i % 2) != 0 {
            continue
        }

        print("i = \(i)")
```

```
}
```

The *continue* statement in the above example will cause the print call to be skipped unless the value of *i* can be divided by 2 with no remainder. If the *continue* statement is triggered, execution will skip to the top of the while loop and the statements in the body of the loop will be repeated (until the value of *i* exceeds 19).

7.6 Conditional Control Flow

In the previous chapter we looked at how to use logical expressions in Swift to determine whether something is *true* or *false*. Since programming is largely an exercise in applying logic, much of the art of programming involves writing code that makes decisions based on one or more criteria. Such decisions define which code gets executed and, conversely, which code gets by-passed when the program is executing.

7.7 Using the if Statement

The *if* statement is perhaps the most basic of control flow options available to the Swift programmer. Programmers who are familiar with C, Objective-C, C++ or Java will immediately be comfortable using Swift *if* statements.

The basic syntax of the Swift *if* statement is as follows:

```
if boolean expression {
    // Swift code to be performed when expression evaluates to true
}
```

Unlike some other programming languages, it is important to note that the braces ({}) are mandatory in Swift, even if only one line of code is executed after the *if* expression.

Essentially if the *Boolean expression* evaluates to *true* then the code in the body of the statement is executed. The body of the statement is enclosed in braces ({}). If, on the other hand, the expression evaluates to *false* the code in the body of the statement is skipped.

For example, if a decision needs to be made depending on whether one value is greater than another, we would write code similar to the following:

```
let x = 10

if x > 9 {
      print("x is greater than 9!")
}
```

Clearly, x is indeed greater than 9 causing the message to appear in the console panel.

7.8 Using if ... else … Statements

The next variation of the *if* statement allows us to also specify some code to perform if the expression in the *if* statement evaluates to *false*. The syntax for this construct is as follows:

```
if boolean expression {
    // Code to be executed if expression is true
} else {
    // Code to be executed if expression is false
}
```

Using the above syntax, we can now extend our previous example to display a different message if the comparison expression evaluates to be *false*:

```
let x = 10
```

```
if x > 9 {
         print("x is greater than 9!")
} else {
         print("x is less than 9!")
}
```

In this case, the second print statement would execute if the value of x was less than 9.

7.9 Using if ... else if ... Statements

So far we have looked at *if* statements which make decisions based on the result of a single logical expression. Sometimes it becomes necessary to make decisions based on a number of different criteria. For this purpose, we can use the *if ... else if ...* construct, an example of which is as follows:

```
let x = 9

if x == 10 {
        print("x is 10")
} else if x == 9 {
        print("x is 9")
} else if x == 8 {
        print("x is 8")
}
```

This approach works well for a moderate number of comparisons but can become cumbersome for a larger volume of expression evaluations. For such situations, the Swift *switch* statement provides a more flexible and efficient solution. For more details on using the *switch* statement refer to the next chapter entitled *"The Swift Switch Statement"*.

7.10 The guard Statement

The *guard* statement is a Swift language feature introduced as part of Swift 2. A guard statement contains a Boolean expression which must evaluate to true in order for the code located *after* the guard statement to be executed. The guard statement must include an *else* clause to be executed in the event that the expression evaluates to false. The code in the else clause must contain a statement to exit the current code flow (i.e. a *return*, *break*, *continue* or *throw* statement). Alternatively, the *else* block may call any other function or method that does not itself return.

The syntax for the guard statement is as follows:

```
guard <boolean expressions> else {
   // code to be executed if expression is false
   <exit statement here>
}

// code here is executed if expression is true
```

The guard statement essentially provides an "early exit" strategy from the current function or loop in the event that a specified requirement is not met.

The following code example implements a guard statement within a function:

```
func multiplyByTen(value: Int?) {
```

```
    guard let number = value, number < 10 else {
        print("Number is too high")
        return
    }

    let result = number * 10
    print(result)
}

multiplyByTen(value: 5)
multiplyByTen(value: 10)
```

The function takes as a parameter an integer value in the form of an optional. The guard statement uses optional binding to unwrap the value and verify that it is less than 10. In the event that the variable could not be unwrapped, or that its value is greater than 9, the else clause is triggered, the error message printed, and the return statement executed to exit the function.

If the optional contains a value less than 10, the code after the guard statement executes to multiply the value by 10 and print the result. A particularly important point to note about the above example is that the unwrapped *number* variable is available to the code outside of the guard statement. This would not have been the case had the variable been unwrapped using an *if* statement.

7.11 Summary

The term *control flow* is used to describe the logic that dictates the execution path that is taken through the source code of an application as it runs. This chapter has looked at the two types of control flow provided by Swift (looping and conditional) and explored the various Swift constructs that are available to implement both forms of control flow logic.

Chapter 9

9. Swift Functions, Methods and Closures

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

9.1 What is a Function?

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

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

9.2 What is a Method?

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

9.3 How to Declare a Swift Function

A Swift function is declared using the following syntax:

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

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

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

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

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

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

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

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

9.4 Implicit Returns from Single Expressions

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

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

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

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

9.5 Calling a Swift Function

Once declared, functions are called using the following syntax:

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

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

```
sayHello()
```

9.6 Handling Return Values

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

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

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

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

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

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

9.7 Local and External Parameter Names

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

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

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

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

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

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

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

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

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

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

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

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

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

9.8 Declaring Default Function Parameters

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

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

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

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

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

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

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

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

9.9 Returning Multiple Results from a Function

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

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

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

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

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

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

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

Usage of this function might read as follows:

```
let lengthTuple = sizeConverter(20)

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

9.10 Variable Numbers of Function Parameters

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

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

```
    }
}

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

9.11 Parameters as Variables

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

```
func calcuateArea(length: Float, width: Float) -> Float {

    var length = length
    var width = width

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

print(calcuateArea(length: 10, width: 20))
```

9.12 Working with In-Out Parameters

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

```
var myValue = 10

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

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

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

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

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

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

```
Before function call myValue = 10
doubleValue call returns 20
After function call myValue = 10
```

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

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

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

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

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

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

```
Before function call myValue = 10
doubleValue call returns 20
After function call myValue = 20
```

9.13 Functions as Parameters

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

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

let toFeet = inchesToFeet
```

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

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

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

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

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

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

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

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

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

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

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

    return inches * 0.0833333
}

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

    return inches * 0.0277778
}

let toFeet = inchesToFeet
let toYards = inchesToYards
```

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

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

    let result = converterFunc(value)

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

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

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

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

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

```
    } else {
        return toYards
    }
}
```

9.14 Closure Expressions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.15 Shorthand Argument Names

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

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

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

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

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

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

9.16 Closures in Swift

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

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

    var counter = 0

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

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

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

To a large extent, and particularly as it relates to Swift, the terms *closure* and *closure expression* have started to be used interchangeably. The key point to remember, however, is that both are supported in Swift.

9.17 Summary

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

Chapter 10

10. The Basics of Swift Object-Oriented Programming

Swift provides extensive support for developing object-oriented applications. The subject area of object-oriented programming is, however, large. It is not an exaggeration to state that entire books have been dedicated to the subject. As such, a detailed overview of object-oriented software development is beyond the scope of this book. Instead, we will introduce the basic concepts involved in object-oriented programming and then move on to explaining the concept as it relates to Swift application development. Once again, while we strive to provide the basic information you need in this chapter, we recommend reading a copy of Apple's *The Swift Programming Language* book for more extensive coverage of this subject area.

10.1 What is an Instance?

Objects (also referred to as class *instances*) are self-contained modules of functionality that can be easily used and re-used as the building blocks for a software application.

Instances consist of data variables (called *properties*) and functions (called *methods*) that can be accessed and called on the instance to perform tasks and are collectively referred to as *class members*.

10.2 What is a Class?

Much as a blueprint or architect's drawing defines what an item or a building will look like once it has been constructed, a class defines what an instance will look like when it is created. It defines, for example, what the methods will do and what the properties will be.

10.3 Declaring a Swift Class

Before an instance can be created, we first need to define the class 'blueprint' for the instance. In this chapter we will create a bank account class to demonstrate the basic concepts of Swift object-oriented programming.

In declaring a new Swift class we specify an optional *parent class* from which the new class is derived and also define the properties and methods that the class will contain. The basic syntax for a new class is as follows:

```
class NewClassName: ParentClass {
   // Properties
   // Instance Methods
   // Type methods
}
```

The *Properties* section of the declaration defines the variables and constants that are to be contained within the class. These are declared in the same way that any other variable or constant would be declared in Swift.

The *Instance methods* and *Type methods* sections define the methods that are available to be called on the class and instances of the class. These are essentially functions specific to the class that perform a particular operation when called upon and will be described in greater detail later in this chapter.

To create an example outline for our BankAccount class, we would use the following:

```
class BankAccount {
```

```
}
```

Now that we have the outline syntax for our class, the next step is to add some instance properties to it.

When naming classes, note that the convention is for the first character of each word to be declared in uppercase (a concept referred to as UpperCamelCase). This contrasts with property and function names where lower case is used for the first character (referred to as lowerCamelCase).

10.4 Adding Instance Properties to a Class

A key goal of object-oriented programming is a concept referred to as *data encapsulation*. The idea behind data encapsulation is that data should be stored within classes and accessed only through methods defined in that class. Data encapsulated in a class are referred to as *properties* or *instance variables*.

Instances of our BankAccount class will be required to store some data, specifically a bank account number and the balance currently held within the account. Properties are declared in the same way any other variables and constants are declared in Swift. We can, therefore, add these variables as follows:

```
class BankAccount {
    var accountBalance: Float = 0
    var accountNumber: Int = 0
}
```

Having defined our properties, we can now move on to defining the methods of the class that will allow us to work with our properties while staying true to the data encapsulation model.

10.5 Defining Methods

The methods of a class are essentially code routines that can be called upon to perform specific tasks within the context of that class.

Methods come in two different forms, *type methods* and *instance methods*. Type methods operate at the level of the class, such as creating a new instance of a class. Instance methods, on the other hand, operate only on the instances of a class (for example performing an arithmetic operation on two property variables and returning the result).

Instance methods are declared within the opening and closing braces of the class to which they belong and are declared using the standard Swift function declaration syntax.

Type methods are declared in the same way as instance methods with the exception that the declaration is preceded by the *class* keyword.

For example, the declaration of a method to display the account balance in our example might read as follows:

```
class BankAccount {

    var accountBalance: Float = 0
    var accountNumber: Int = 0

    func displayBalance()
    {
       print("Number \(accountNumber)")
       print("Current balance is \(accountBalance)")
    }
```

```
}
```

The method is an *instance method* so it is not preceded by the *class* keyword.

When designing the BankAccount class it might be useful to be able to call a type method on the class itself to identify the maximum allowable balance that can be stored by the class. This would enable an application to identify whether the BankAccount class is suitable for storing details of a new customer without having to go through the process of first creating a class instance. This method will be named *getMaxBalance* and is implemented as follows:

```
class BankAccount {

    var accountBalance: Float = 0
    var accountNumber: Int = 0

    func displayBalance()
    {
       print("Number \(accountNumber)")
       print("Current balance is \(accountBalance)")
    }

    class func getMaxBalance() -> Float {
        return 100000.00
    }
}
```

10.6 Declaring and Initializing a Class Instance

So far all we have done is define the blueprint for our class. In order to do anything with this class, we need to create instances of it. The first step in this process is to declare a variable to store a reference to the instance when it is created. We do this as follows:

```
var account1: BankAccount = BankAccount()
```

When executed, an instance of our BankAccount class will have been created and will be accessible via the *account1* variable.

10.7 Initializing and De-initializing a Class Instance

A class will often need to perform some initialization tasks at the point of creation. These tasks can be implemented by placing an *init* method within the class. In the case of the BankAccount class, it would be useful to be able to initialize the account number and balance properties with values when a new class instance is created. To achieve this, the *init* method could be written in the class as follows:

```
class BankAccount {

    var accountBalance: Float = 0
    var accountNumber: Int = 0

    init(number: Int, balance: Float)
    {
        accountNumber = number
        accountBalance = balance
```

```
    }

    func displayBalance()
    {
       print("Number \(accountNumber)")
       print("Current balance is \(accountBalance)")
    }
}
```

When creating an instance of the class, it will now be necessary to provide initialization values for the account number and balance properties as follows:

```
var account1 = BankAccount(number: 12312312, balance: 400.54)
```

Conversely, any cleanup tasks that need to be performed before a class instance is destroyed by the Swift runtime system can be performed by implementing the de-initializer within the class definition:

```
class BankAccount {

    var accountBalance: Float = 0
    var accountNumber: Int = 0

    init(number: Int, balance: Float)
    {
        accountNumber = number
        accountBalance = balance
    }

    deinit {
        // Perform any necessary clean up here
    }

    func displayBalance()
    {
       print("Number \(accountNumber)")
       print("Current balance is \(accountBalance)")
    }
}
```

10.8 Calling Methods and Accessing Properties

Now is probably a good time to recap what we have done so far in this chapter. We have now created a new Swift class named *BankAccount*. Within this new class we declared some properties to contain the bank account number and current balance together with an initializer and a method to display the current balance information. In the preceding section we covered the steps necessary to create and initialize an instance of our new class. The next step is to learn how to call the instance methods and access the properties we built into our class. This is most easily achieved using *dot notation*.

Dot notation involves accessing an instance variable, or calling an instance method by specifying a class instance followed by a dot followed in turn by the name of the property or method:

```
classInstance.propertyName
```

```
classInstance.instanceMethod()
```

For example, to get the current value of our *accountBalance* instance variable:

```
var balance1 = account1.accountBalance
```

Dot notation can also be used to set values of instance properties:

```
account1.accountBalance = 6789.98
```

The same technique is used to call methods on a class instance. For example, to call the *displayBalance* method on an instance of the BankAccount class:

```
account1.displayBalance()
```

Type methods are also called using dot notation, though they must be called on the class type instead of a class instance:

```
ClassName.typeMethod()
```

For example, to call the previously declared *getMaxBalance* type method, the BankAccount class is referenced:

```
var maxAllowed = BankAccount.getMaxBalance()
```

10.9 Stored and Computed Properties

Class properties in Swift fall into two categories referred to as *stored properties* and *computed properties*. Stored properties are those values that are contained within a constant or variable. Both the account name and number properties in the BankAccount example are stored properties.

A computed property, on the other hand, is a value that is derived based on some form of calculation or logic at the point at which the property is set or retrieved. Computed properties are implemented by creating *getter* and optional corresponding *setter* methods containing the code to perform the computation. Consider, for example, that the BankAccount class might need an additional property to contain the current balance less any recent banking fees. Rather than use a stored property, it makes more sense to use a computed property which calculates this value on request. The modified BankAccount class might now read as follows:

```
class BankAccount {

    var accountBalance: Float = 0
    var accountNumber: Int = 0;
    let fees: Float = 25.00

    var balanceLessFees: Float {
        get {
            return accountBalance - fees
        }
    }

    init(number: Int, balance: Float)
    {
        accountNumber = number
        accountBalance = balance
    }
.
.
```

```
.
}
```

The above code adds a getter that returns a computed property based on the current balance minus a fee amount. An optional setter could also be declared in much the same way to set the balance value less fees:

```
var balanceLessFees: Float {
    get {
        return accountBalance - fees
    }

    set(newBalance)
    {
        accountBalance = newBalance - fees
    }
}
```

The new setter takes as a parameter a floating-point value from which it deducts the fee value before assigning the result to the current balance property. Although these are computed properties, they are accessed in the same way as stored properties using dot-notation. The following code gets the current balance less the fees value before setting the property to a new value:

```
var balance1 = account1.balanceLessFees
account1.balanceLessFees = 12123.12
```

10.10 Lazy Stored Properties

There are several different ways in which a property can be initialized, the most basic being direct assignment as follows:

```
var myProperty = 10
```

Alternatively, a property may be assigned a value within the initializer:

```
class MyClass {
  let title: String

  init(title: String) {
    self.title = title
  }
}
```

For more complex requirements, a property may be initialized using a closure:

```
class MyClass {

    var myProperty: String = {
        var result = resourceIntensiveTask()
        result = processData(data: result)
        return result
    }()
.
.
}
```

Particularly in the case of a complex closure, there is the potential for the initialization to be resource intensive and time consuming. When declared in this way, the initialization will be performed every time an instance of the class is created, regardless of when (or even if) the property is actually used within the code of the app. Also, situations may arise where the value assigned to the property may not be known until a later stage in the execution process, for example after data has been retrieved from a database or user input has been obtained from the user. A far more efficient solution in such situations would be for the initialization to take place only when the property is first accessed. Fortunately, this can be achieved by declaring the property as *lazy* as follows:

```
class MyClass {

    lazy var myProperty: String = {
        var result = resourceIntensiveTask()
        result = processData(data: result)
        return result
    }()
.
.
}
```

When a property is declared as being lazy, it is only initialized when it is first accessed, allowing any resource intensive activities to be deferred until the property is needed and any initialization on which the property is dependent to be completed.

Note that lazy properties must be declared as variables (*var*).

10.11 Using self in Swift

Programmers familiar with other object-oriented programming languages may be in the habit of prefixing references to properties and methods with *self* to indicate that the method or property belongs to the current class instance. The Swift programming language also provides the *self* property type for this purpose and it is, therefore, perfectly valid to write code which reads as follows:

```
class MyClass {
    var myNumber = 1

    func addTen() {
        self.myNumber += 10
    }
}
```

In this context, the *self* prefix indicates to the compiler that the code is referring to a property named *myNumber* which belongs to the MyClass class instance. When programming in Swift, however, it is no longer necessary to use self in most situations since this is now assumed to be the default for references to properties and methods. To quote The Swift Programming Language guide published by Apple, "in practice you don't need to write *self* in your code very often". The function from the above example, therefore, can also be written as follows with the *self* reference omitted:

```
func addTen() {
    myNumber += 10
}
```

In most cases, use of self is optional in Swift. That being said, one situation where it is still necessary to use *self* is when referencing a property or method from within a closure expression. The use of self, for example, is

mandatory in the following closure expression:

```
document?.openWithCompletionHandler({(success: Bool) -> Void in
    if success {
        self.ubiquityURL = resultURL
    }
})
```

It is also necessary to use self to resolve ambiguity such as when a function parameter has the same name as a class property. In the following code, for example, the first print statement will output the value passed through to the function via the myNumber parameter while the second print statement outputs the number assigned to the myNumber class property (in this case 10):

```
class MyClass {

    var myNumber = 10 // class property

    func addTen(myNumber: Int) {
        print(myNumber) // Output the function parameter value
        print(self.myNumber) // Output the class property value
    }
}
```

Whether or not to use self in most other situations is largely a matter of programmer preference. Those who prefer to use self when referencing properties and methods can continue to do so in Swift. Code that is written without use of the *self* property type (where doing so is not mandatory) is, however, just as valid when programming in Swift.

10.12 Understanding Swift Protocols

By default, there are no specific rules to which a Swift class must conform as long as the class is syntactically correct. In some situations, however, a class will need to meet certain criteria in order to work with other classes. This is particularly common when writing classes that need to work with the various frameworks that comprise the iOS SDK. A set of rules that define the minimum requirements which a class must meet is referred to as a *Protocol*. A protocol is declared using the *protocol* keyword and simply defines the methods and properties that a class must contain in order to be in conformance. When a class *adopts* a protocol, but does not meet all of the protocol requirements, errors will be reported stating that the class fails to conform to the protocol.

Consider the following protocol declaration. Any classes that adopt this protocol must include both a readable String value called *name* and a method named *buildMessage()* which accepts no parameters and returns a String value:

```
protocol MessageBuilder {

    var name: String { get }
    func buildMessage() -> String
}
```

Below, a class has been declared which adopts the MessageBuilder protocol:

```
class MyClass: MessageBuilder {


}
```

Unfortunately, as currently implemented, MyClass will generate a compilation error because it contains neither the *name* variable nor the *buildMessage()* method as required by the protocol it has adopted. To conform to the protocol, the class would need to meet both requirements, for example:

```
class MyClass: MessageBuilder {

    var name: String

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

    func buildMessage() -> String {
        "Hello " + name
    }
}
```

10.13 Opaque Return Types

Now that protocols have been explained it is a good time to introduce the concept of opaque return types. As we have seen in previous chapters, if a function returns a result, the type of that result must be included in the function declaration. The following function, for example, is configured to return an Int result:

```
func doubleFunc1 (value: Int) -> Int {
    return value * 2
}
```

Instead of specifying a specific return type (also referred to as a *concrete type*), opaque return types allow a function to return any type as long as it conforms to a specified protocol. Opaque return types are declared by preceding the protocol name with the *some* keyword. The following changes to the *doubleFunc1()* function, for example, declare that a result will be returned of any type that conforms to the Equatable protocol:

```
func doubleFunc1(value: Int) -> some Equatable {
    value * 2
}
```

To conform to the Equatable protocol, which is a standard protocol provided with Swift, a type must allow the underlying values to be compared for equality. Opaque return types can, however, be used for any protocol, including those you create yourself.

Given that both the Int and String concrete types are in conformance with the Equatable protocol, it is possible to also create a function that returns a String result:

```
func doubleFunc2(value: String) -> some Equatable {
    value + value
}
```

Although these two methods return entirely different concrete types, the only thing known about these types is that they conform to the Equatable protocol. We therefore know the capabilities of the type, but not the actual type.

In fact, we only know the concrete type returned in these examples because we have access to the source code of the functions. If these functions resided in a library or API framework for which the source is not available to us, we would not know the exact type being returned. This is intentional and designed to hide the underlying

return type used within public APIs. By masking the concrete return type, programmers will not come to rely on a function returning a specific concrete type or risk accessing internal objects which were not intended to be accessed. This also has the benefit that the developer of the API can make changes to the underlying implementation (including returning a different protocol compliant type) without having to worry about breaking dependencies in any code that uses the API.

This raises the question of what happens when an incorrect assumption is made when working with the opaque return type. Consider, for example, that the assumption could be made that the results from the *doubleFunc1()* and *doubleFunc2()* functions can be compared for equality:

```
let intOne = doubleFunc1(value: 10)
let stringOne = doubleFunc2(value: "Hello")



if (intOne == stringOne) {
    print("They match")
}
```

Working on the premise that we do not have access to the source code for these two functions there is no way to know whether the above code is valid. Fortunately, although we, as programmers, have no way of knowing the concrete type returned by the functions, the Swift compiler has access to this hidden information. The above code will, therefore, generate the following syntax error long before we get to the point of trying to execute invalid code:

```
Binary operator '==' cannot be applied to operands of type 'some
 Equatable' (result of 'doubleFunc1(value:)') and 'some Equatable'
(result of 'doubleFunc2(value:)')
```

Opaque return types are a fundamental foundation of the implementation of the SwiftUI APIs and are used widely when developing apps in SwiftUI (the *some* keyword will appear frequently in SwiftUI View declarations). SwiftUI advocates the creation of apps by composing together small, reusable building blocks and refactoring large view declarations into collections of small, lightweight subviews. Each of these building blocks will typically conform to the View protocol. By declaring these building blocks as returning opaque types that conform to the View protocol, these building blocks become remarkably flexible and interchangeable, resulting in code that is cleaner and easier to reuse and maintain.

10.14 Summary

Object-oriented programming languages such as Swift encourage the creation of classes to promote code reuse and the encapsulation of data within class instances. This chapter has covered the basic concepts of classes and instances within Swift together with an overview of stored and computed properties and both instance and type methods. The chapter also introduced the concept of protocols which serve as templates to which classes must conform and explained how they form the basis of opaque return types.

Chapter 12

12. An Introduction to Swift Structures and Enumerations

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:

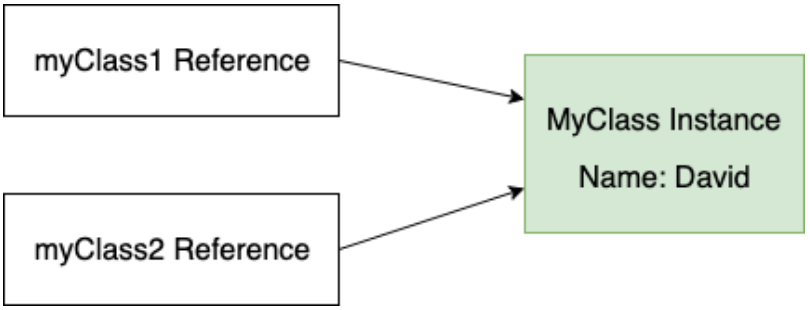

Figure 12-2

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 An Overview of Enumerations

Enumerations (typically referred to as enums) are used to create custom data types consisting of pre-defined sets of values. Enums are typically used for making decisions within code such as when using switch statements. An enum might, for example be declared as follows:

```
enum Temperature {
    case hot
    case warm
    case cold
}
```

Note that in this example, none of the cases are assigned a value. An enum of this type is essentially used to reference one of a pre-defined set of states (in this case the current temperature being hot, warm or cold). Once declared, the enum may, for example, be used within a switch statement as follows:

```
func displayTempInfo(temp: Temperature) {
    switch temp {
        case .hot:
            print("It is hot.")
        case .warm:
            print("It is warm.")
        case .cold:
            print("It is cold.")
    }
}
```

It is also worth noting that because an enum has a definitive set of valid member values, the switch statement does not need to include a default case. An attempt to pass an invalid enum case through the switch will be

caught by the compiler long before it has a chance to cause a runtime error.

To test out the enum, the *displayTempInfo()* function must be passed an instance of the Temperature enum with one of the following three possible states selected:

```
Temperature.hot
Temperature.warm
Temperature.cold
```

For example:

```
displayTempInfo(temp: Temperature.warm)
```

When executed, the above function call will output the following information:

```
It is warm.
```

Individual cases within an enum may also have *associated values*. Assume, for example, that the "cold" enum case needs to have associated with it a temperature value so that the app can differentiate between cold and freezing conditions. This can be defined within the enum declaration as follows:

```
enum Temperature {
    case hot
    case warm
    case cold(centigrade: Int)
}
```

This allows the switch statement to also check for the temperature for the cold case as follows:

```
func displayTempInfo(temp: Temperature) {
    switch temp {
        case .hot:
            print("It is hot")
        case .warm:
            print("It is warm")
        case.cold(let centigrade) where centigrade <= 0:
            print("Ice warning: \(centigrade) degrees.")
        case .cold:
            print("It is cold but not freezing.")
    }
}
```

When the cold enum value is passed to the function, it now does so with a temperature value included:

```
displayTempInfo(temp: Temperature.cold(centigrade: -10))
```

The output from the above function will read as follows:

```
Ice warning: -10 degrees
```

12.5 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.

Enumerations are used to create custom types consisting of a pre-defined set of state values and are of particular use in identifying state within switch statements.

Chapter 14

14. Working with Array and Dictionary Collections in Swift

Arrays and dictionaries in Swift are objects that contain collections of other objects. In this chapter, we will cover some of the basics of working with arrays and dictionaries in Swift.

14.1 Mutable and Immutable Collections

Collections in Swift come in mutable and immutable forms. The contents of immutable collection instances cannot be changed after the object has been initialized. To make a collection immutable, assign it to a *constant* when it is created. Collections are mutable, on the other hand, if assigned to a *variable*.

14.2 Swift Array Initialization

An array is a data type designed specifically to hold multiple values in a single ordered collection. An array, for example, could be created to store a list of String values. Strictly speaking, a single Swift based array is only able to store values that are of the same type. An array declared as containing String values, therefore, could not also contain an Int value. As will be demonstrated later in this chapter, however, it is also possible to create mixed type arrays. The type of an array can be specified specifically using type annotation or left to the compiler to identify using type inference.

An array may be initialized with a collection of values (referred to as an *array literal*) at creation time using the following syntax:

var variableName: [type] = [value 1, value2, value3, …….]

The following code creates a new array assigned to a variable (thereby making it mutable) that is initialized with three string values:

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

Alternatively, the same array could have been created immutably by assigning it to a constant:

```
let treeArray = ["Pine", "Oak", "Yew"]
```

In the above instance, the Swift compiler will use type inference to decide that the array contains values of String type and prevent values of other types being inserted into the array elsewhere within the application code.

Alternatively, the same array could have been declared using type annotation:

```
var treeArray: [String] = ["Pine", "Oak", "Yew"]
```

Arrays do not have to have values assigned at creation time. The following syntax can be used to create an empty array:

```
var variableName = [type]()
```

Consider, for example, the following code which creates an empty array designated to store floating point values and assigns it to a variable named priceArray:

```
var priceArray = [Float]()
```

Another useful initialization technique allows an array to be initialized to a certain size with each array element

pre-set with a specified default value:

```
var nameArray = [String](repeating: "My String", count: 10)
```

When compiled and executed, the above code will create a new 10 element array with each element initialized with a string that reads "My String".

Finally, a new array may be created by adding together two existing arrays (assuming both arrays contain values of the same type). For example:

```
let firstArray = ["Red", "Green", "Blue"]
let secondArray = ["Indigo", "Violet"]

let thirdArray = firstArray + secondArray
```

14.3 Working with Arrays in Swift

Once an array exists, a wide range of methods and properties are provided for working with and manipulating the array content from within Swift code, a subset of which is as follows:

14.3.1 Array Item Count

A count of the items in an array can be obtained by accessing the array's count property:

```
var treeArray = ["Pine", "Oak", "Yew"]
var itemCount = treeArray.count

print(itemCount)
```

Whether or not an array is empty can be identified using the array's Boolean *isEmpty* property as follows:

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

if treeArray.isEmpty {
    // Array is empty
}
```

14.3.2 Accessing Array Items

A specific item in an array may be accessed or modified by referencing the item's position in the array index (where the first item in the array has index position 0) using a technique referred to as *index subscripting*. In the following code fragment, the string value contained at index position 2 in the array (in this case the string value "Yew") is output by the print call:

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

print(treeArray[2])
```

This approach can also be used to replace the value at an index location:

```
treeArray[1] = "Redwood"
```

The above code replaces the current value at index position 1 with a new String value that reads "Redwood".

14.3.3 Random Items and Shuffling

A call to the *shuffled()* method of an array object will return a new version of the array with the item ordering randomly shuffled, for example:

```
let shuffledTrees = treeArray.shuffled()
```

To access an array item at random, simply make a call to the *randomElement()* method:

```
let randomTree = treeArray.randomElement()
```

14.3.4 Appending Items to an Array

Items may be added to an array using either the *append* method or + and += operators. The following, for example, are all valid techniques for appending items to an array:

```
treeArray.append("Redwood")
treeArray += ["Redwood"]
treeArray += ["Redwood", "Maple", "Birch"]
```

14.3.5 Inserting and Deleting Array Items

New items may be inserted into an array by specifying the index location of the new item in a call to the array's *insert(at:)* method. An insertion preserves all existing elements in the array, essentially moving them to the right to accommodate the newly inserted item:

```
treeArray.insert("Maple", at: 0)
```

Similarly, an item at a specific array index position may be removed using the *remove(at:)* method call:

```
treeArray.remove(at: 2)
```

To remove the last item in an array, simply make a call to the array's *removeLast* method as follows:

```
treeArray.removeLast()
```

14.3.6 Array Iteration

The easiest way to iterate through the items in an array is to make use of the for-in looping syntax. The following code, for example, iterates through all of the items in a String array and outputs each item to the console panel:

```
let treeArray = ["Pine", "Oak", "Yew", "Maple", "Birch", "Myrtle"]

for tree in treeArray {
    print(tree)
}
```

Upon execution, the following output would appear in the console:

```
Pine
Oak
Yew
Maple
Birch
Myrtle
```

The same result can be achieved by calling the *forEach()* array method. When this method is called on an array, it will iterate through each element and execute specified code. For example:

```
treeArray.forEach { tree in
    print(tree)
}
```

Note that since the task to be performed for each array element is declared in a closure expression, the above example may be modified as follows to take advantage of shorthand argument names:

```
treeArray.forEach {
    print($0)
```

```
}
```

14.4 Creating Mixed Type Arrays

A mixed type array is an array that can contain elements of different class types. Clearly an array that is either declared or inferred as being of type String cannot subsequently be used to contain non-String class object instances. Interesting possibilities arise, however, when taking into consideration that Swift includes the *Any* type. Any is a special type in Swift that can be used to reference an object of a non-specific class type. It follows, therefore, that an array declared as containing Any object types can be used to store elements of mixed types. The following code, for example, declares and initializes an array containing a mixture of String, Int and Double elements:

```
let mixedArray: [Any] = ["A String", 432, 34.989]
```

The use of the Any type should be used with care since the use of Any masks from Swift the true type of the elements in such an array thereby leaving code prone to potential programmer error. It will often be necessary, for example, to manually cast the elements in an Any array to the correct type before working with them in code. Performing the incorrect cast for a specific element in the array will most likely cause the code to compile without error but crash at runtime. Consider, for the sake of an example, the following mixed type array:

```
let mixedArray: [Any] = [1, 2, 45, "Hello"]
```

Assume that, having initialized the array, we now need to iterate through the integer elements in the array and multiply them by 10. The code to achieve this might read as follows:

```
for object in mixedArray {
    print(object * 10)
}
```

When entered into Xcode, however, the above code will trigger a syntax error indicating that it is not possible to multiply operands of type Any and Int. In order to remove this error it will be necessary to downcast the array element to be of type Int:

```
for object in mixedArray {
    print(object as! Int * 10)
}
```

The above code will compile without error and work as expected until the final String element in the array is reached at which point the code will crash with the following error:

```
Could not cast value of type 'Swift.String' to 'Swift.Int'
```

The code will, therefore, need to be modified to be aware of the specific type of each element in the array. Clearly, there are both benefits and risks to using Any arrays in Swift.

14.5 Swift Dictionary Collections

String dictionaries allow data to be stored and managed in the form of key-value pairs. Dictionaries fulfill a similar purpose to arrays, except each item stored in the dictionary has associated with it a unique key (to be precise, the key is unique to the particular dictionary object) which can be used to reference and access the corresponding value. Currently only String, Int, Double and Bool data types are suitable for use as keys within a Swift dictionary.

14.6 Swift Dictionary Initialization

A dictionary is a data type designed specifically to hold multiple values in a single unordered collection. Each item in a dictionary consists of a key and an associated value. The data types of the key and value elements type may be specified specifically using type annotation, or left to the compiler to identify using type inference.

A new dictionary may be initialized with a collection of values (referred to as a *dictionary literal*) at creation time using the following syntax:

var *variableName*: [*key type*: *value type*] = [*key 1: value 1*, *key 2: value2* ….]

The following code creates a new dictionary assigned to a variable (thereby making it mutable) that is initialized with four key-value pairs in the form of ISBN numbers acting as keys for corresponding book titles:

```
var bookDict = ["100-432112" : "Wind in the Willows",
                "200-532874" : "Tale of Two Cities",
                "202-546549" : "Sense and Sensibility",
                "104-109834" : "Shutter Island"]
```

In the above instance, the Swift compiler will use type inference to decide that both the key and value elements of the dictionary are of String type and prevent values or keys of other types being inserted into the dictionary.

Alternatively, the same dictionary could have been declared using type annotation:

```
var bookDict: [String: String] =
               ["100-432112" : "Wind in the Willows",
                "200-532874" : "Tale of Two Cities",
                "202-546549" : "Sense and Sensibility",
                "104-109834" : "Shutter Island"]
```

As with arrays, it is also possible to create an empty dictionary, the syntax for which reads as follows:

var *variableName* = [*key type*: *value type*]()

The following code creates an empty dictionary designated to store integer keys and string values:

```
var myDictionary = [Int: String]()
```

14.7 Sequence-based Dictionary Initialization

Dictionaries may also be initialized using sequences to represent the keys and values. This is achieved using the Swift *zip()* function, passing through the keys and corresponding values. In the following example, a dictionary is created using two arrays:

```
let keys = ["100-432112", "200-532874", "202-546549", "104-109834"]
let values = ["Wind in the Willows", "Tale of Two Cities",
                "Sense and Sensibility", "Shutter Island"]

let bookDict = Dictionary(uniqueKeysWithValues: zip(keys, values))
```

This approach allows keys and values to be generated programmatically. In the following example, a number range starting at 1 is being specified for the keys instead of using an array of predefined keys:

```
let values = ["Wind in the Willows", "Tale of Two Cities",
               "Sense and Sensibility", "Shutter Island"]

var bookDict = Dictionary(uniqueKeysWithValues: zip(1..., values))
```

The above code is a much cleaner equivalent to the following dictionary declaration:

```
var bookDict = [1 : "Wind in the Willows",
                2 : "Tale of Two Cities",
                3 : "Sense and Sensibility",
```

```
                4 : "Shutter Island"]
```

14.8 Dictionary Item Count

A count of the items in a dictionary can be obtained by accessing the dictionary's count property:

```
print(bookDict.count)
```

14.9 Accessing and Updating Dictionary Items

A specific value may be accessed or modified using key subscript syntax to reference the corresponding value. The following code references a key known to be in the bookDict dictionary and outputs the associated value (in this case the book entitled "A Tale of Two Cities"):

```
print(bookDict["200-532874"])
```

When accessing dictionary entries in this way, it is also possible to declare a default value to be used in the event that the specified key does not return a value:

```
print(bookDict["999-546547", default: "Book not found"])
```

Since the dictionary does not contain an entry for the specified key, the above code will output text which reads "Book not found".

Indexing by key may also be used when updating the value associated with a specified key, for example, to change the title of the same book from "A Tale of Two Cities" to "Sense and Sensibility"):

```
bookDict["200-532874"] = "Sense and Sensibility"
```

The same result is also possible by making a call to the *updateValue(forKey:)* method, passing through the key corresponding to the value to be changed:

```
bookDict.updateValue("The Ruins", forKey: "200-532874")
```

14.10 Adding and Removing Dictionary Entries

Items may be added to a dictionary using the following key subscripting syntax:

dictionaryVariable[*key*] = *value*

For example, to add a new key-value pair entry to the books dictionary:

```
bookDict["300-898871"] = "The Overlook"
```

Removal of a key-value pair from a dictionary may be achieved either by assigning a *nil* value to the entry, or via a call to the *removeValueForKey* method of the dictionary instance. Both code lines below achieve the same result of removing the specified entry from the books dictionary:

```
bookDict["300-898871"] = nil
bookDict.removeValue(forKey: "300-898871")
```

14.11 Dictionary Iteration

As with arrays, it is possible to iterate through dictionary entries by making use of the for-in looping syntax. The following code, for example, iterates through all of the entries in the books dictionary, outputting both the key and value for each entry:

```
for (bookid, title) in bookDict {
  print("Book ID: \(bookid) Title: \(title)")
}
```

Upon execution, the following output would appear in the console:

```
Book ID: 100-432112 Title: Wind in the Willows
```

```
Book ID: 200-532874 Title: The Ruins
Book ID: 104-109834 Title: Shutter Island
Book ID: 202-546549 Title: Sense and Sensibility
```

14.12 Summary

Collections in Swift take the form of either dictionaries or arrays. Both provide a way to collect together multiple items within a single object. Arrays provide a way to store an ordered collection of items where those items are accessed by an index value corresponding to the item position in the array. Dictionaries provide a platform for storing key-value pairs, where the key is used to gain access to the stored value. Iteration through the elements of Swift collections can be achieved using the for-in loop construct.

Chapter 16

16. An Overview of SwiftUI

Now that Xcode has been installed and the basics of the Swift programing 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 are 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"*).

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 not yet available in SwiftUI.

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 changes drive 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.

Chapter 21

21. 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.

21.1 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 21-1:

Figure 21-1

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 21-2) select the appropriate option :

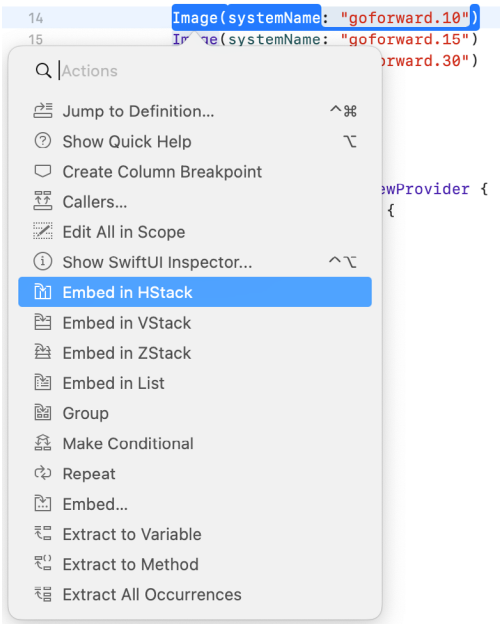

Figure 21-2

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 21-3:

Figure 21-3

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.

21.2 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("$1000")
                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:

Figure 21-4

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

21.3 Container Child Limit

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

```
Extra arguments at positions #11, #12 in call
```

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).

21.4 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 21-5

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 21-6

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 21-7:

Figure 21-7

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 21-8:

Figure 21-8

21.5 Traditional vs. Lazy Stacks

So far in this chapter we have only covered the HStack, VStack and ZStack views. Although the stack examples shown so far contain relatively few child views, it is possible for a stack to contain large quantities of views. This is particularly common when a stack is embedded in a ScrollView. ScrollView is a view which allows the user to scroll through content that extends beyond the visible area of either the containing view or device the screen.

When using the traditional HStack and VStack views, the system will create all the views child views at initialization, regardless of whether those views are currently visible to the user. While this may not be an issue for most requirements, this can lead to performance degradation in situations where a stack has thousands of child views.

To address this issue, SwiftUI also provides "lazy" vertical and horizontal stack views. These views (named LazyVStack and LazyHStack) use exactly the same declaration syntax as the traditional stack views, but are

designed to only create child views as they are needed. For example, as the user scrolls through a stack, views that are currently off screen will only be created once they approach the point of becoming visible to the user. Once those views pass out of the viewing area, SwiftUI releases those views so that they no longer take up system resources.

When deciding whether to use traditional or lazy stacks, it is generally recommended to start out using the traditional stacks and to switch to lazy stacks if you encounter performance issues relating to a high number of child views.

21.6 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 21-9

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 21-10

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 21-11

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 21-10 above. Longer text, however, will be displayed as follows:

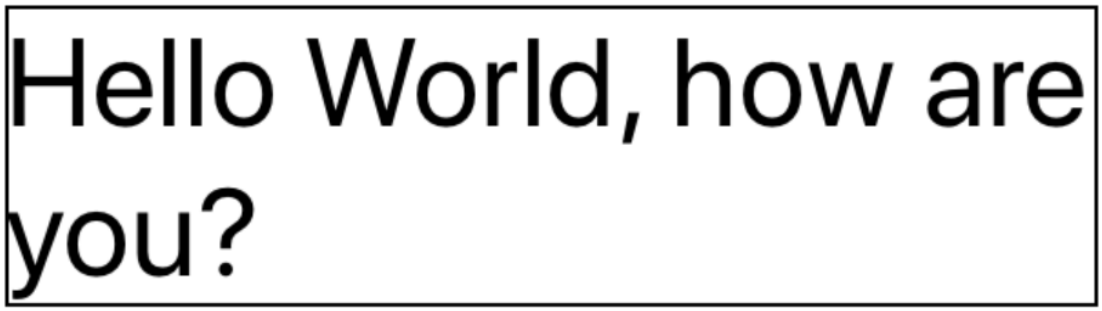

Figure 21-12

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)
```

21.7 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.

21.8 Summary

User interface design mostly involves gathering 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.