iOS 16 App Development Essentials

UIKit Edition

iOS 16 App Development Essentials – UIKit Edition

ISBN-13: 978-1-951442-62-0

Rev: 1.0

Table of Contents

Chapter 1

1. Start Here

This book aims to teach the skills necessary to create iOS apps using the iOS 16 SDK, UIKit, Xcode 14, and the Swift programming language.

Beginning with the basics, this book outlines the steps necessary to set up an iOS development environment. Next, an introduction to the architecture of iOS 16 and programming in Swift 5.7 is provided, followed by an in-depth look at the design of iOS apps and user interfaces. More advanced topics such as file handling, database management, graphics drawing, and animation are also covered, as are touch screen handling, gesture recognition, multitasking, location management, local notifications, camera access, and video playback support. Other features include Auto Layout, local map search, user interface animation using UIKit dynamics, Siri integration, iMessage app development, and biometric authentication.

Additional features of iOS development using Xcode are also covered, including Swift playgrounds, universal user interface design using size classes, app extensions, Interface Builder Live Views, embedded frameworks, collection and stack layouts, CloudKit data storage, and the document browser.

Other features of iOS 16 and Xcode 14 are also covered in detail, including iOS machine learning features.

The aim of this book, therefore, is to teach you the skills necessary to build your own apps for iOS 16. Assuming you are ready to download the iOS 16 SDK and Xcode 14, have a Mac, and some ideas for some apps to develop, you are ready to get started.

1.1 Source Code Download

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

https://www.ebookfrenzy.com/retail/ios16/

1.2 Feedback

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

1.3 Errata

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

https://www.ebookfrenzy.com/errata/ios16.html

If you find an error not listed in the errata, please email our technical support team at *feedback@ebookfrenzy.com*.

Chapter 2

2. Joining the Apple Developer Program

The first step in learning to develop iOS 16-based applications involves understanding 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 14 and the iOS 16 SDK

The latest iOS SDK and Xcode versions 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 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.

Before 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; all that is now required to test apps on physical iOS devices is an Apple ID.

Clearly, much can be achieved without paying to join the Apple Developer program. There are, however, areas of app development that 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 for obtaining assistance and guidance from other iOS developers and finding solutions to problems 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 to publish an application for sale or download in the App Store.

Clearly, program membership will 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, it is first worth checking whether the company already has a 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 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:

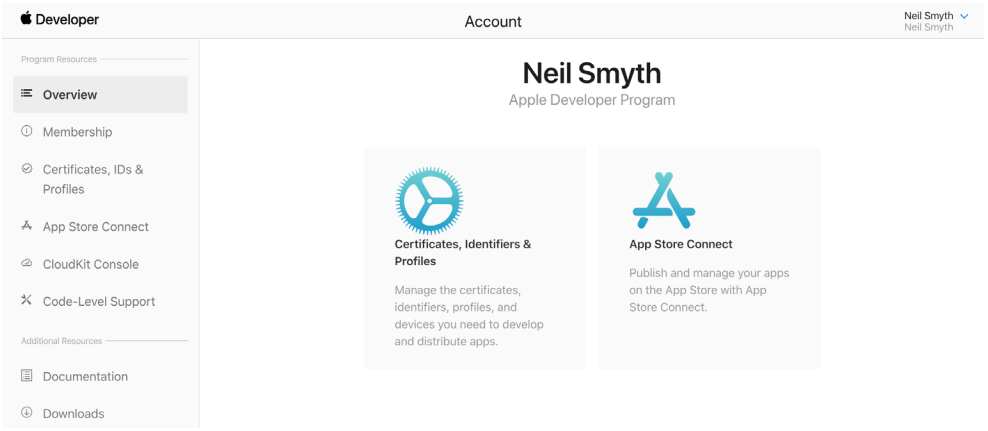

Figure 2-1

2.5 Summary

An important early step in the iOS 16 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 downloading and installing the iOS 16 SDK and Xcode 14 development environment.

Chapter 3

3. Installing Xcode 14 and the iOS 16 SDK

iOS apps are developed using the iOS SDK and Apple's Xcode development environment. Xcode is an integrated development environment (IDE) within which you will code, compile, test, and debug your iOS apps. The Xcode environment also includes a feature called Interface Builder, which enables you to graphically design your app's user interface using the UIKit Framework's components.

This chapter will cover the steps involved in installing Xcode and the iOS 16 SDK on macOS.

3.1 Identifying Your macOS Version

The Xcode 14 environment requires that the version of macOS running on the system be version 12.3 or later. If you are unsure of the version of macOS on your Mac, you can find this information by clicking on the Apple menu in the top left-hand corner of the screen and selecting the *About This Mac* option from the menu. In the resulting dialog, check the *macOS* line:

Figure 3-1

If the "About This Mac" dialog indicates that an older macOS is installed, click on the *More Info...* button to display the System Settings dialog, followed by the *General -> Software Update* option to check for operating system upgrade availability.

3.2 Installing Xcode 14 and the iOS 16 SDK

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

3.3 Starting Xcode

Having successfully installed the SDK and Xcode, the next step is to launch it to create a sample iOS 16 app. To start up Xcode, open the Finder and search for *Xcode*. Since you will be using this tool frequently, take this opportunity to drag and drop it into your dock for easier access in the future. Next, click on the Xcode icon in the dock to launch the tool. You may be prompted to install additional components the first time Xcode runs. Follow these steps, entering your username and password when prompted.

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

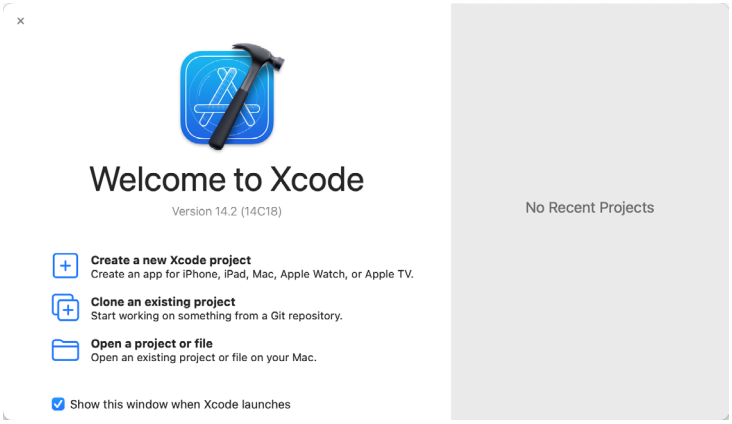

Figure 3-2

3.4 Adding Your Apple ID to the Xcode Preferences

Whether or not you enroll in the Apple Developer Program, it is worth adding your Apple ID to Xcode now that it is installed and running. First, select the *Xcode -> Settings…* menu option and select the *Accounts* tab. Then, on the Accounts screen, click the + button highlighted in Figure 3-3, choose *Apple ID* from the resulting panel, and click on the *Continue* button. When prompted, enter your Apple ID and associated password and click the *Sign In* button to add the account to the preferences.

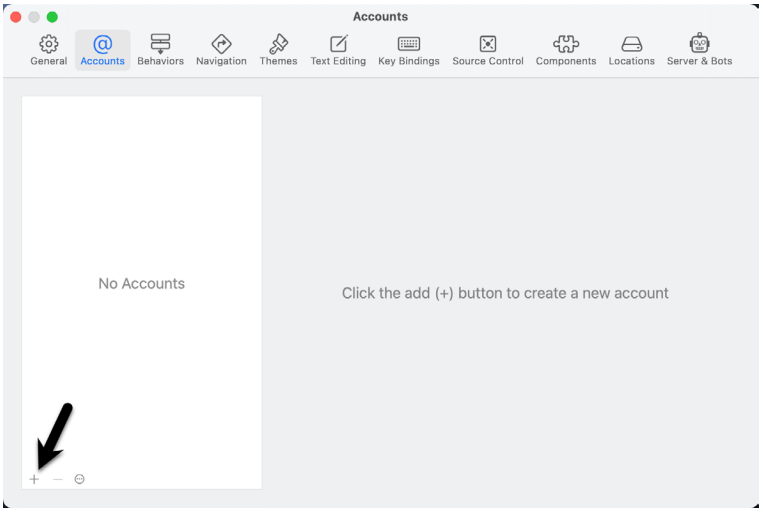

Figure 3-3

3.5 Developer and Distribution Signing Identities

Once the Apple ID has been entered, the next step is to generate signing identities. Select the newly added Apple ID in the Accounts panel to view the current signing identities and click on the *Manage Certificates…* button. At this point, a list of available signing identities will be listed. If you have not yet enrolled in the Apple Developer Program, it will only be possible to create iOS and Mac Development identities. To create the iOS Development signing identity, click on the + button highlighted in Figure 3-4 and make the appropriate selection from the menu:

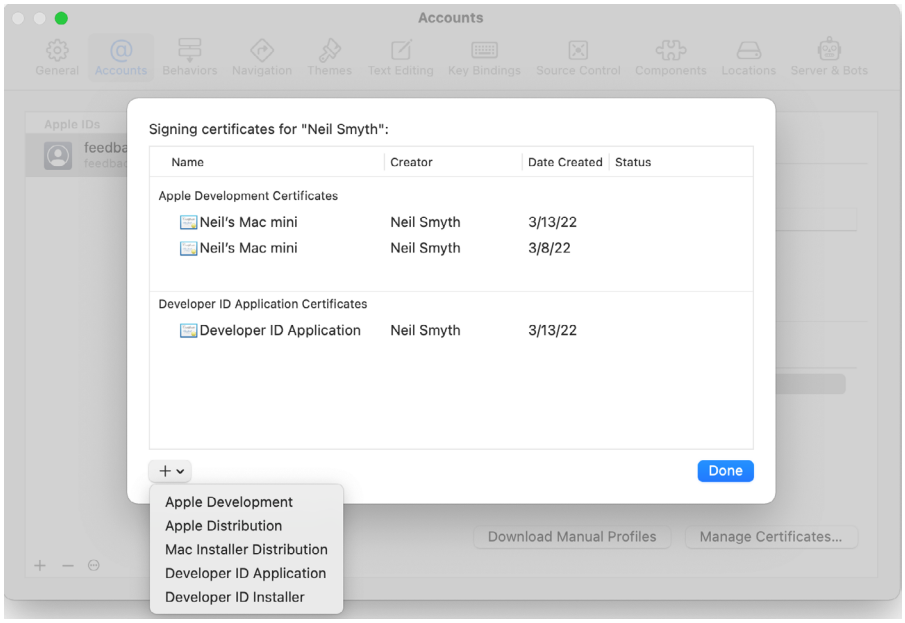

Figure 3-4

If the Apple ID has been used to enroll in the Apple Developer program, the option to create an *iOS Distribution* certificate will, when clicked, generate the signing identity required to submit the app to the Apple App Store.

Having installed the iOS SDK and successfully launched Xcode 14, we can now look at Xcode in more detail.

Chapter 4

4. A Guided Tour of Xcode 14

Just about every activity related to developing and testing iOS apps involves the use of the Xcode environment. This chapter is intended to serve two purposes. Primarily it is intended to provide an overview of many key areas that comprise the Xcode development environment. In the course of providing this overview, the chapter will also work through creating a straightforward iOS app project to display a label that reads "Hello World" on a colored background.

By the end of this chapter, you will have a basic familiarity with Xcode and your first running iOS app.

4.1 Starting Xcode 14

As with all iOS examples in this book, the development of our example will take place within the Xcode 14 development environment. Therefore, if you have not already installed this tool with the latest iOS SDK, refer first to the *"Installing Xcode 14 and the iOS 16 SDK"* chapter of this book. Then, assuming that the installation is complete, launch Xcode either by clicking on the icon on the dock (assuming you created one) or using the macOS Finder to locate Xcode in the Applications folder of your system.

When launched for the first time, and until you turn off the *Show this window when Xcode launches* toggle, the screen illustrated in Figure 4-1 will appear by default:

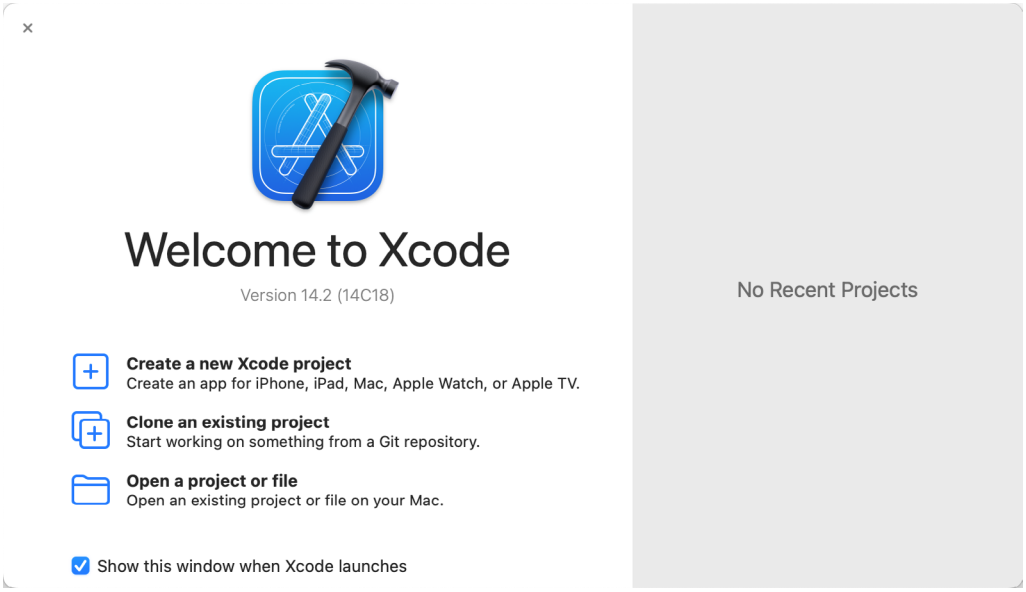

Figure 4-1

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

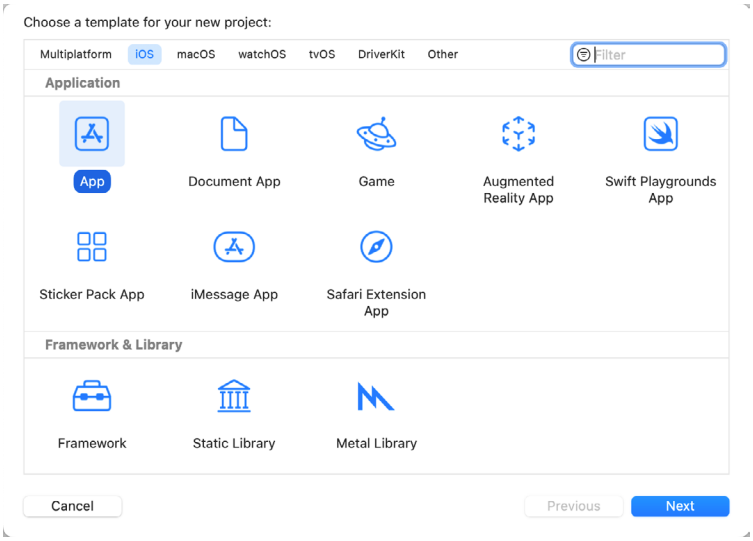

Figure 4-2

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

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

- **App** – This creates a basic template for an app containing a single view and corresponding view controller.
- **Document App** – Creates a project intended to use the iOS document browser. The document browser provides a visual environment where the user can navigate and manage local and cloud-based files from within an iOS app.
- **Game** – Creates a project configured to take advantage of Sprite Kit, Scene Kit, OpenGL ES, and Metal for developing 2D and 3D games.
- **Augmented Reality App** – Creates a template project pre-configured to use ARKit to integrate augmented reality support into an iOS app.
- **Sticker Pack App** – Allows a sticker pack app to be created and sold within the Message App Store. Sticker pack apps allow additional images to be made available for inclusion in messages sent via the iOS Messages app.
- **iMessage App** – iMessage apps are extensions to the built-in iOS Messages app that allow users to send interactive messages, such as games, to other users. Once created, iMessage apps are available through the Message App Store.
- **Safari Extension App** - This option creates a project to be used as the basis for developing an extension for the Safari web browser.

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

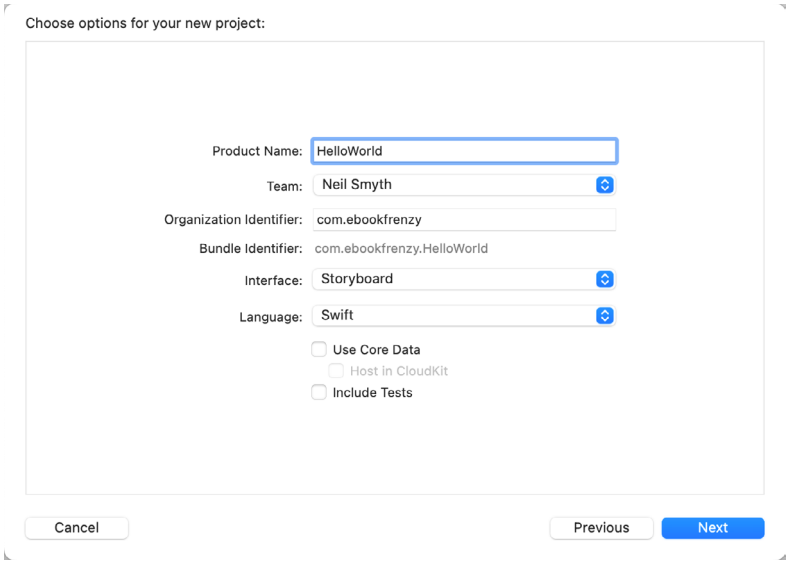

Figure 4-3

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

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

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

When developing an app in Xcode, the user interface can be designed using either Storyboards or SwiftUI. For this book, we will be using Storyboards, so make sure that the Interface menu is set to Storyboard. SwiftUI development is covered in my *SwiftUI Essentials - iOS 16 Edition* book.

Apple supports two programming languages for the development of iOS apps in the form of *Objective-C* and *Swift*. While it is still possible to program using the older Objective-C language, Apple considers Swift to be the future of iOS development. Therefore, all the code examples in this book are written in Swift, so make sure that the *Language* menu is set accordingly before clicking on the *Next* button.

On the final screen, choose a location on the file system for the new project to be created. This panel also allows placing the project under Git source code control. Source code control systems such as Git allow different project revisions to be managed and restored, and for changes made over the project's development lifecycle to be tracked. Since this is typically used for larger projects, or those involving more than one developer, this option can be turned off for this and the other projects created in the book.

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

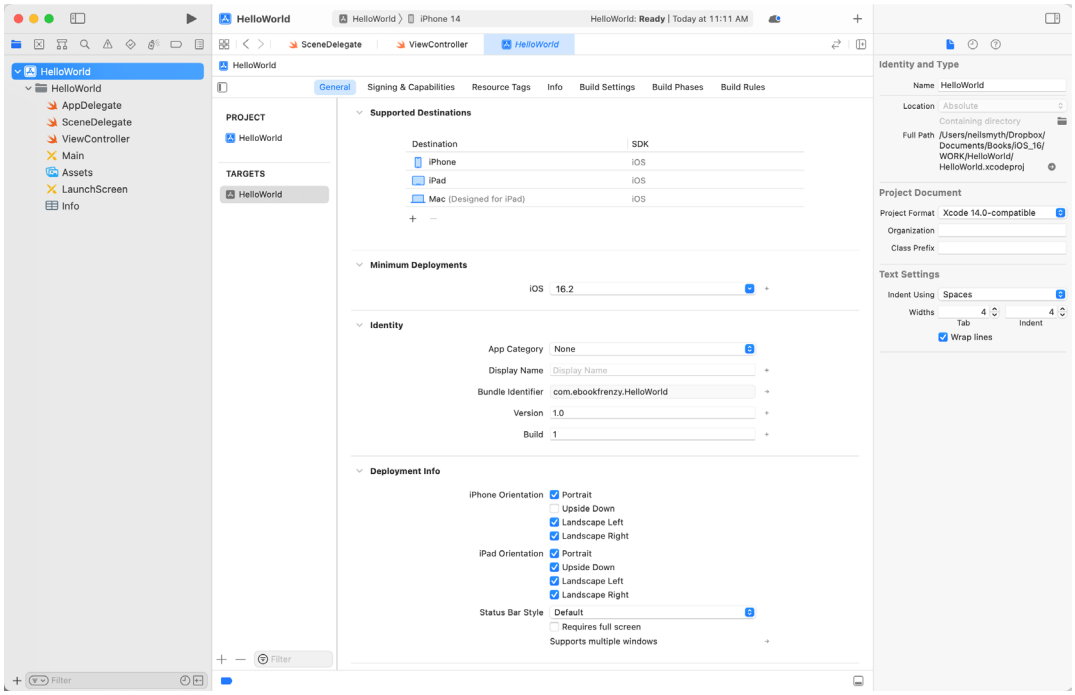

Figure 4-4

Before proceeding, we should take some time to look at what Xcode has done for us. First, it has created a group of files we will need to complete our app. Some of these are Swift source code files, where we will enter the code to make our app work.

In addition, the *Main* storyboard file is the save file used by the Interface Builder tool to hold the user interface design we will create. A second Interface Builder file named *LaunchScreen* will also have been added to the project. This file contains the user interface design for the screen that appears on the device while the app is loading.

Also present will be one or more *Property List* files that contain key/value pair information. For example, the *Info.plist* file contains resource settings relating to items such as the language, executable name, and app identifier and, as will be shown in later chapters, is the location where several properties are stored to configure the capabilities of the project (for example to configure access to the user's current geographical location). The list of files is displayed in the *Project Navigator* located in the left-hand panel of the main Xcode project window. In addition, a toolbar at the top of this panel contains options to display other information, such as build and run history, breakpoints, and compilation errors.

By default, the center panel of the window shows a general summary of the settings for the app project. This summary includes the identifier specified during the project creation process and the target devices. In addition, options are also provided to configure the orientations of the device that are to be supported by the app, together with opportunities to upload icons (the small images the user selects on the device screen to launch the app) and launch screen images (displayed to the user while the app loads) for the app.

The Signing section allows selecting an Apple identity when building the app. This identity ensures that the app is signed with a certificate when it is compiled. If you have registered your Apple ID with Xcode using the Preferences screen outlined in the previous chapter, select that identity now using the Team menu. Testing apps on physical devices will not be possible if no team is selected, though the simulator environment may still be

used.

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

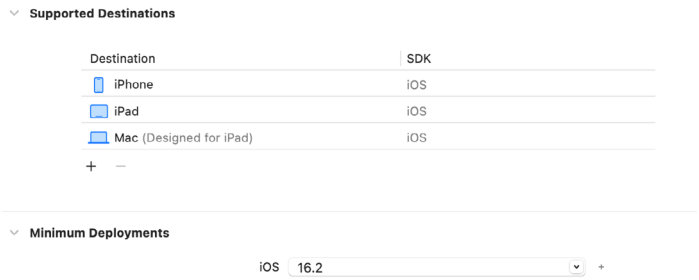

Figure 4-5

The iOS ecosystem now includes a variety of devices and screen sizes. When developing a project, it is possible to indicate that it is intended to target either the iPhone or iPad family of devices. With the gap between iPad and iPhone screen sizes now reduced by the introduction of the Pro range of devices, it no longer makes sense to create a project that targets just one device family. A much more sensible approach is to create a single project that addresses all device types and screen sizes. As will be shown in later chapters, Xcode 14 and iOS 16 include several features designed specifically to make the goal of *universal* app projects easy to achieve. With this in mind, ensure that the destination list at least includes the iPhone and iPad.

In addition to the General screen, tabs are provided to view and modify additional settings consisting of Signing & Capabilities, Resource Tags, Info, Build Settings, Build Phases, and Build Rules.

As we progress through subsequent chapters of this book, we will explore some of these other configuration options in greater detail. To return to the project settings panel at any future time, ensure the *Project Navigator* is selected in the left-hand panel and select the top item (the app name) in the navigator list.

When a source file is selected from the list in the navigator panel, the contents of that file will appear in the center panel, where it may then be edited.

4.2 Creating the iOS App User Interface

Simply by the very nature of the environment in which they run, iOS apps are typically visually oriented. Therefore, a vital component of any app involves a user interface through which the user will interact with the app and, in turn, receive feedback. While it is possible to develop user interfaces by writing code to create and position items on the screen, this is a complex and error-prone process. In recognition of this, Apple provides a tool called Interface Builder, which allows a user interface to be visually constructed by dragging and dropping components onto a canvas and setting properties to configure the appearance and behavior of those components.

As mentioned in the preceding section, Xcode pre-created several files for our project, one of which has a .storyboard filename extension. This is an Interface Builder storyboard save file, and the file we are interested in for our HelloWorld project is named *Main.storyboard*. To load this file into Interface Builder, select the *Main* item in the list in the left-hand panel. Interface Builder will subsequently appear in the center panel, as shown in Figure 4-6:

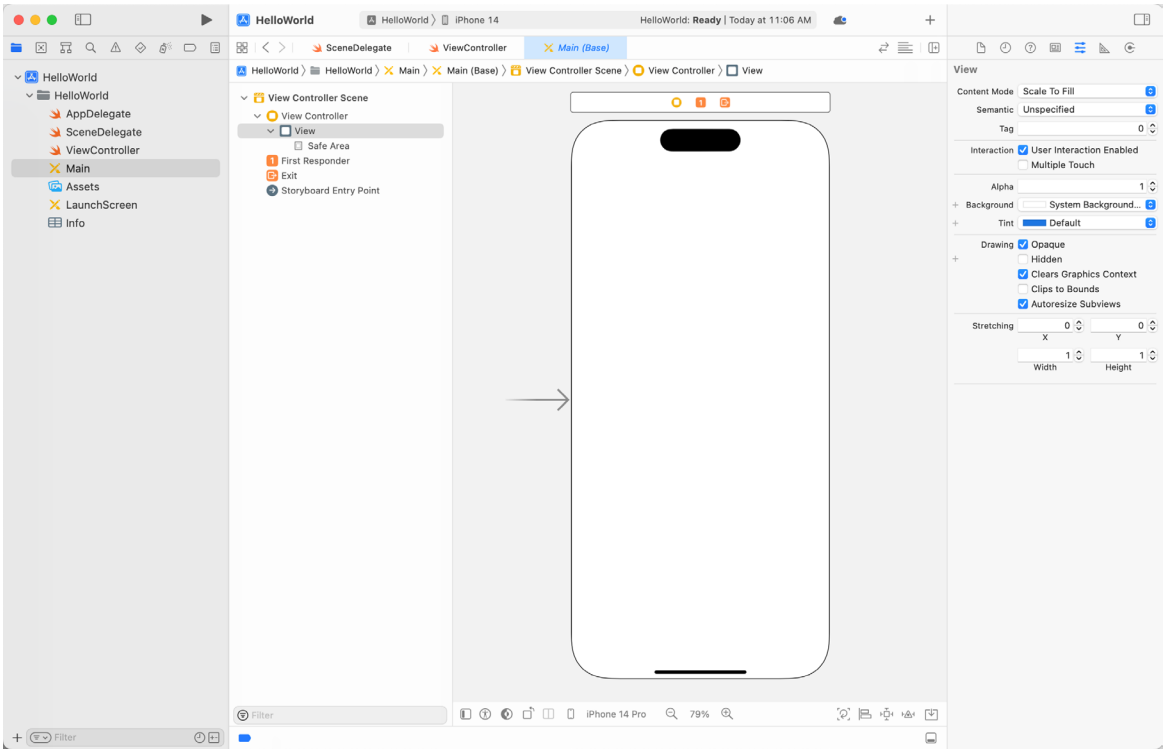

Figure 4-6

In the center panel, a visual representation of the app's user interface is displayed. Initially, this consists solely of a *View Controller* (UIViewController) containing a single View (UIView) object. This layout was added to our design by Xcode when we selected the App template option during the project creation phase. We will construct the user interface for our HelloWorld app by dragging and dropping user interface objects onto this UIView object. Designing a user interface consists primarily of dragging and dropping visual components onto the canvas and setting a range of properties. The user interface components are accessed from the Library panel, which is displayed by clicking on the Library button in the Xcode toolbar, as indicated in Figure 4-7:

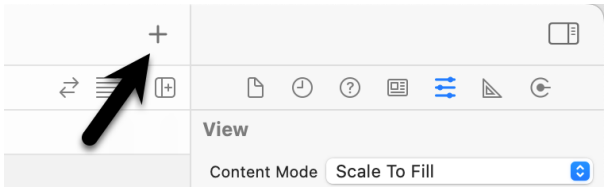

Figure 4-7

This button will display the UI components used to construct our user interface. The layout of the items in the library may also be switched from a single column of objects with descriptions to multiple columns without descriptions by clicking on the button located in the top right-hand corner of the panel and to the right of the search box.

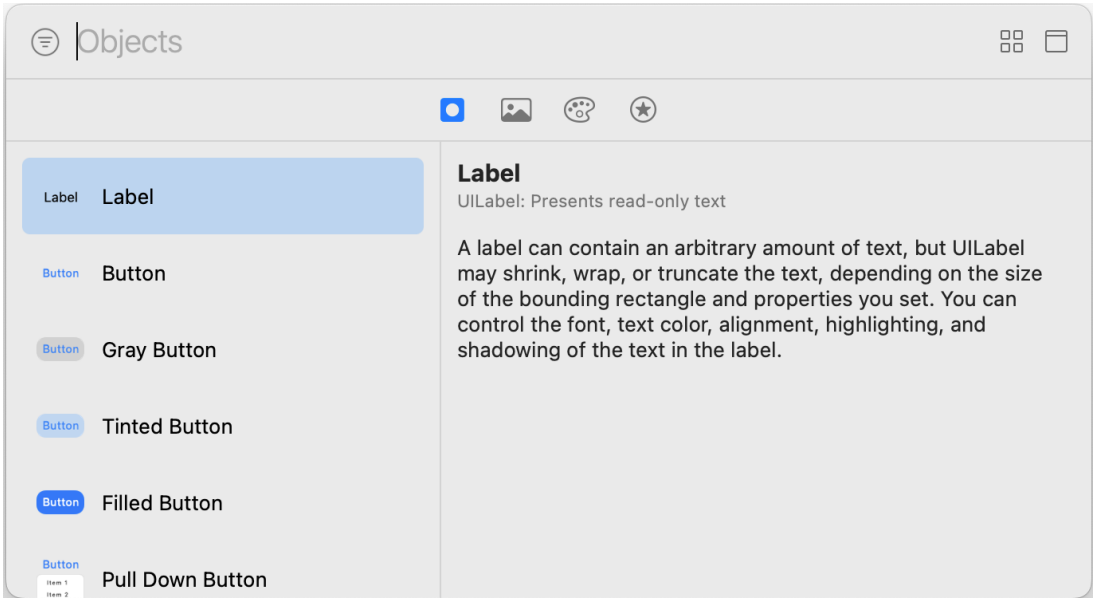

Figure 4-8

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

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

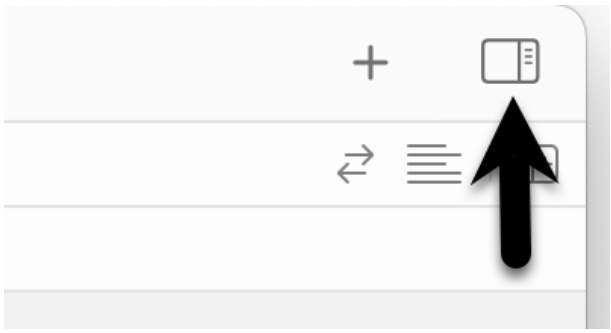

Figure 4-9

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

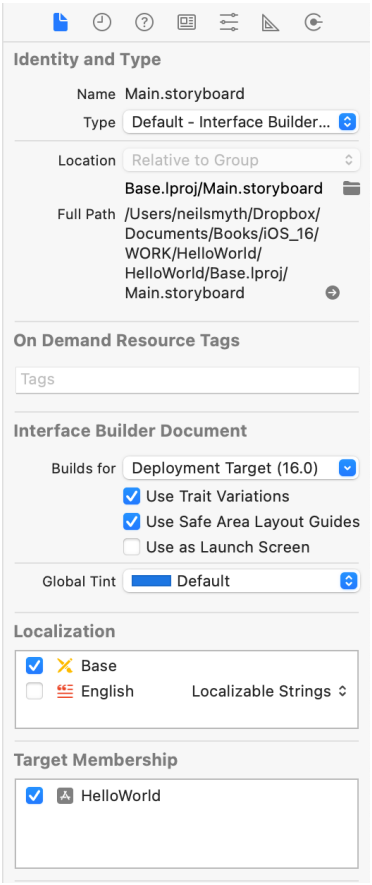

Figure 4-10

Along the top edge of the panel is a row of buttons that change the settings displayed in the upper half of the panel. By default, the *File Inspector* is typically shown. Options are also provided to display quick help, the *Identity Inspector*, *History Inspector*, *Attributes Inspector*, *Size Inspector*, and *Connections Inspector*. Take some time to review each of these selections to familiarize yourself with the configuration options each provides. Throughout the remainder of this book, extensive use of these inspectors will be made.

4.3 Changing Component Properties

With the property panel for the View selected in the main panel, we will begin our design work by changing the background color of this view. Start by ensuring the View is selected and that the Attributes Inspector (*View -> Inspectors -> Attributes*) is displayed in the Utilities panel. Next, click on the current property setting next to the *Background* setting and select the Custom option from the popup menu to display the *Colors* dialog. Finally, choose a visually pleasing color using the color selection tool and close the dialog. You will now notice that the view window has changed from white to the new color selection.

4.4 Adding Objects to the User Interface

The next step is to add a Label object to our view. To achieve this, display the Library panel as shown in Figure 4-7 above and either scroll down the list of objects in the Library panel to locate the Label object or, as illustrated in Figure 4-11, enter *Label* into the search box beneath the panel:

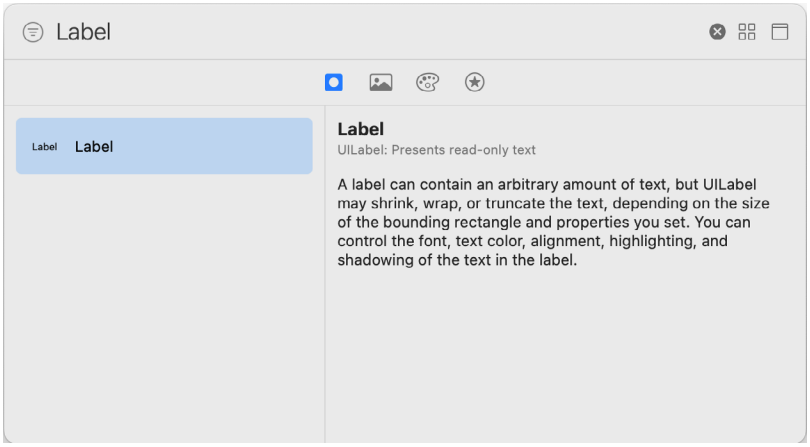

Figure 4-11

After locating the Label object, click on it and drag it to the center of the view so that the vertical and horizontal center guidelines appear. Once it is in position, release the mouse button to drop it at that location. We have now added an instance of the UILabel class to the scene. Cancel the Library search by clicking on the "x" button on the right-hand edge of the search field. Next, select the newly added label and stretch it horizontally so that it is approximately three times the current width. With the Label still selected, click on the centered alignment button in the Attributes Inspector (*View -> Inspectors -> Attributes*) to center the text in the middle of the label view:

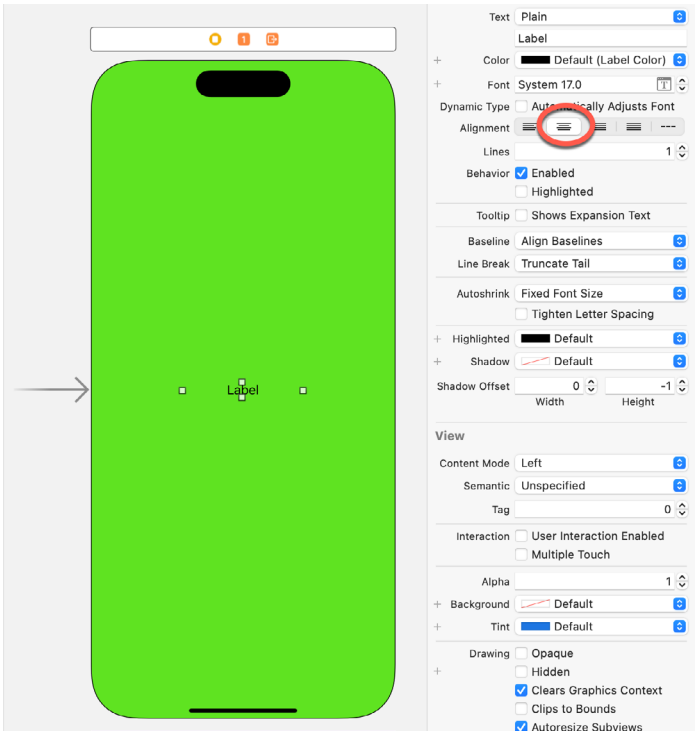

Figure 4-12

Double-click on the text in the label that currently reads "Label" and type in "Hello World". Locate the font setting property in the Attributes Inspector panel and click the "T" button to display the font selection menu

next to the font name. Change the Font setting from *System – System* to *Custom* and choose a larger font setting, for example, a Georgia bold typeface with a size of 24, as shown in Figure 4-13:

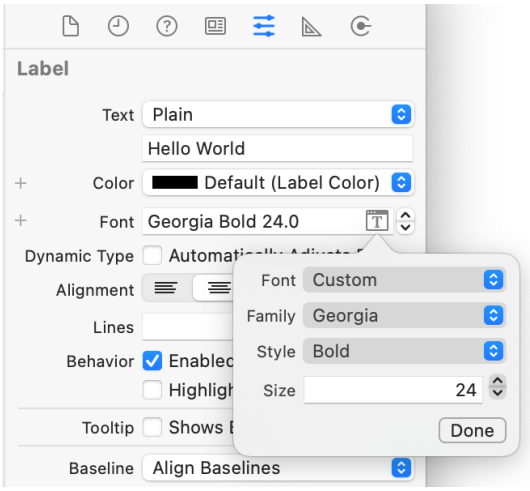

Figure 4-13

The final step is to add some layout constraints to ensure that the label remains centered within the containing view regardless of the size of the screen on which the app ultimately runs. This involves using the Auto Layout capabilities of iOS, a topic that will be covered extensively in later chapters. For this example, select the Label object, display the Align menu as shown in Figure 4-14, and enable both the *Horizontally in Container* and *Vertically in Container* options with offsets of 0 before clicking on the *Add 2 Constraints* button.

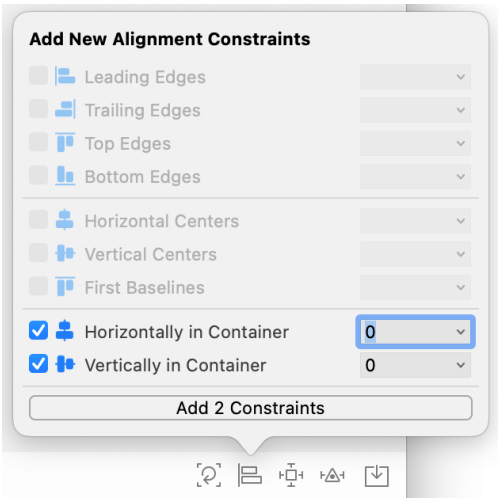

Figure 4-14

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

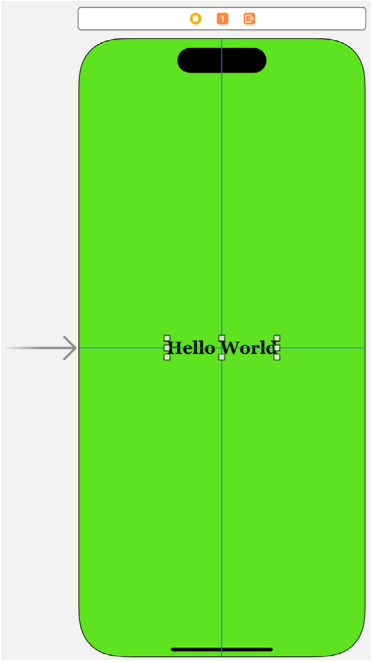

Figure 4-15

Before building and running the project, it is worth taking a short detour to look at the Xcode *Document Outline* panel. This panel appears by default to the left of the Interface Builder panel. It is controlled by the small button in the bottom left-hand corner (indicated by the arrow in Figure 4-16) of the Interface Builder panel.

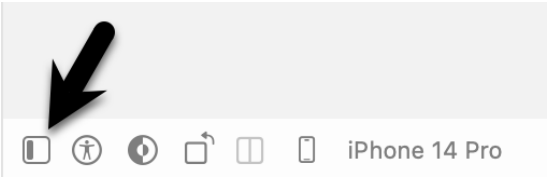

Figure 4-16

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

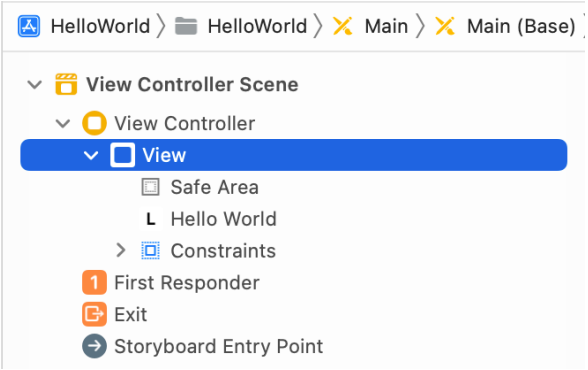

Figure 4-17

4.5 Building and Running an iOS App in Xcode

Before an app can be run, it must first be compiled. Once successfully compiled, it may be run either within a simulator or on a physical iPhone or iPad device. For this chapter, however, it is sufficient to run the app in the simulator.

Within the main Xcode project window, make sure that the menu located in the top left-hand corner of the window (marked C in Figure 4-18) has the *iPhone 14* simulator option selected:

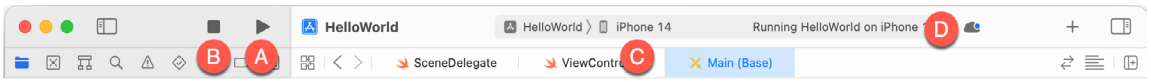

Figure 4-18

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

Figure 4-19

Note that the user interface appears as designed in the Interface Builder tool. Click on the stop button (B), change the target menu from iPhone 14 to iPad Air 2, and rerun the app. Once again, the label will appear centered on the screen even with the larger screen size. Finally, verify that the layout is correct in landscape orientation by using the *Device -> Rotate Left* menu option. This indicates that the Auto Layout constraints are working and that we have designed a *universal* user interface for the project.

4.6 Running the App on a Physical iOS Device

Although the Simulator environment provides a valuable way to test an app on various iOS device models, it is important to also test on a physical iOS device.

If you have entered your Apple ID in the Xcode preferences screen as outlined in the previous chapter and selected a development team for the project, it is possible to run the app on a physical device simply by connecting it to

the development Mac system with a USB cable and selecting it as the run target within Xcode.

With a device connected to the development system and an app ready for testing, refer to the device menu in the Xcode toolbar. There is a reasonable chance that this will have defaulted to one of the iOS Simulator configurations. Switch to the physical device by selecting this menu and changing it to the device name, as shown in Figure 4-20:

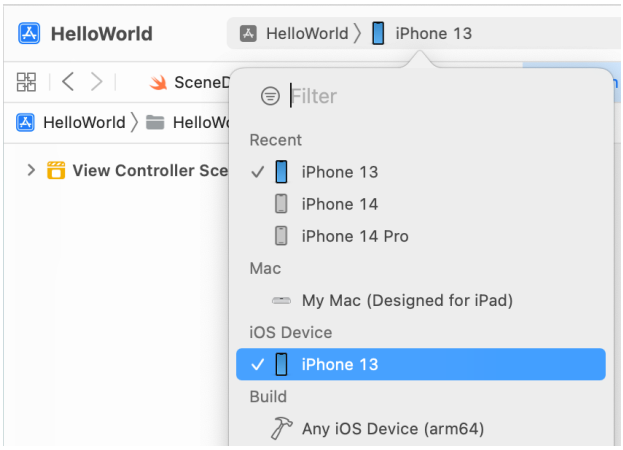

Figure 4-20

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

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

4.7 Managing Devices and Simulators

Currently connected iOS devices and the simulators configured for use with Xcode can be viewed and managed using the Xcode Devices window, accessed via the *Window -> Devices and Simulators* menu option. Figure 4-21, for example, shows a typical Device screen on a system where an iPhone has been detected:

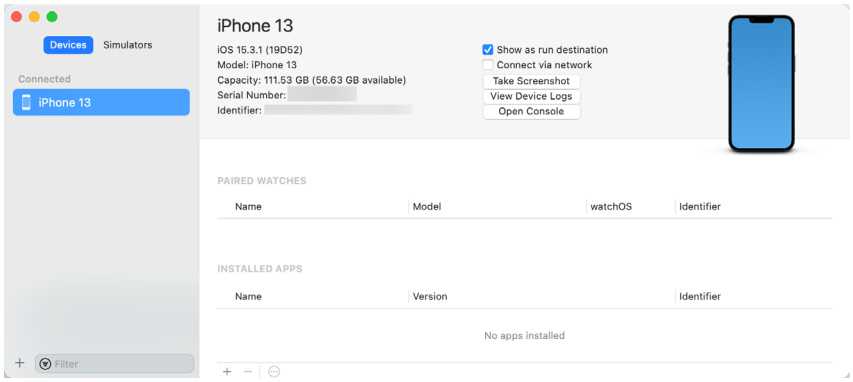

Figure 4-21

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

4.8 Enabling Network Testing

In addition to testing an app on a physical device connected to the development system via a USB cable, Xcode also supports testing via a network connection. This option is enabled on a per device basis within the Devices and Simulators dialog introduced in the previous section. With the device connected via the USB cable, display this dialog, select the device from the list and enable the *Connect via network* option as highlighted in Figure 4-22:

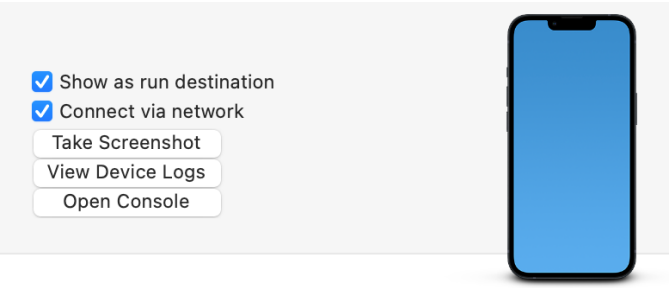

Figure 4-22

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

4.9 Dealing with Build Errors

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

4.10 Monitoring Application Performance

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

When Xcode is launched, the project navigator is displayed in the left-hand panel by default. Along the top of this panel is a bar with various of other options. The seventh option from the left displays the debug navigator when selected, as illustrated in Figure 4-23. When displayed, this panel shows real-time statistics relating to the performance of the currently running application such as memory, CPU usage, disk access, energy efficiency, network activity, and iCloud storage access.

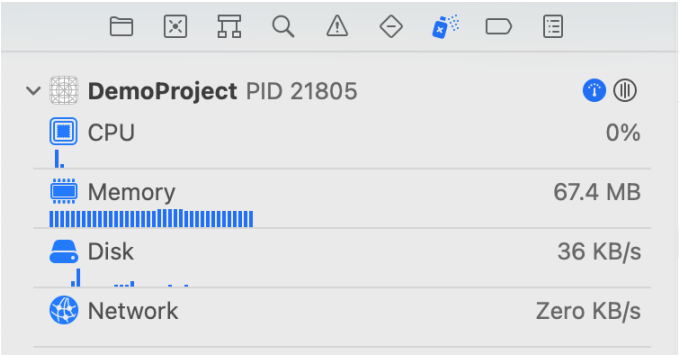

Figure 4-23

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

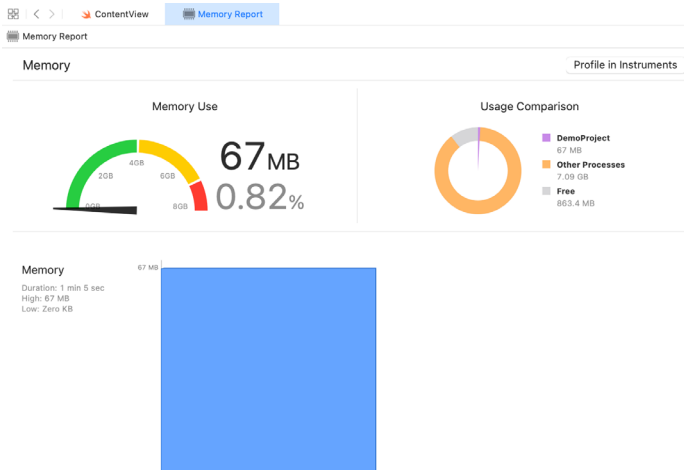

Figure 4-24

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

4.11 Exploring the User Interface Layout Hierarchy

Xcode also provides an option to break the user interface layout out into a rotatable 3D view that shows how the view hierarchy for a user interface is constructed. This can be particularly useful for identifying situations where one view instance is obscured by another appearing on top of it or a layout is not appearing as intended. This is also useful for learning how iOS works behind the scenes to construct a layout if only to appreciate how much work iOS is saving us from having to do.

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

Figure 4-25

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

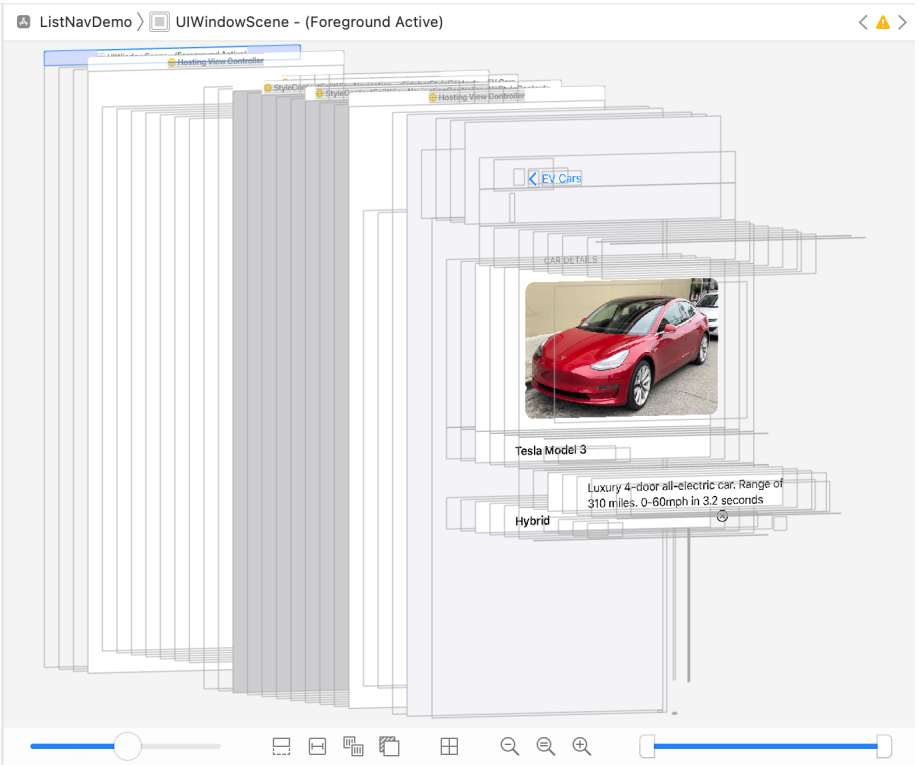

Figure 4-26

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

Figure 4-27

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

layout as shown in Figure 4-28 below:

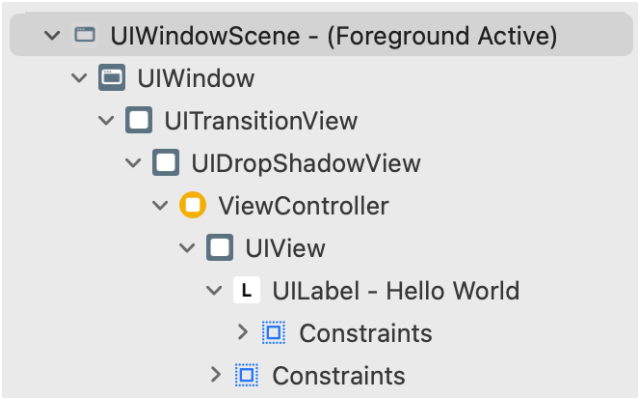

Figure 4-28

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

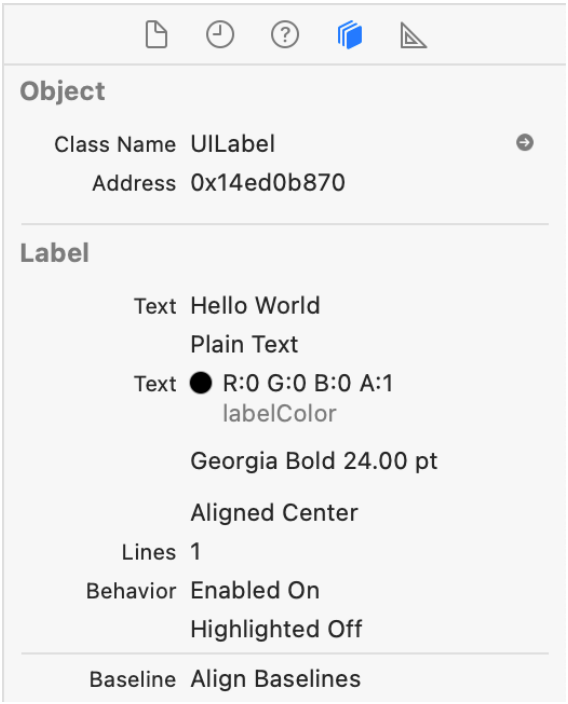

Figure 4-29

4.12 Summary

Apps are primarily created within the Xcode development environment. This chapter has provided a basic overview of the Xcode environment and worked through creating a straightforward example app. Finally, a brief overview was provided of some of the performance monitoring features in Xcode 14. In subsequent chapters of the book, many more features and capabilities of Xcode and Interface Builder will be covered.

Chapter 5

5. An Introduction to Xcode 14 Playgrounds

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

5.1 What is a Playground?

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

5.2 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:

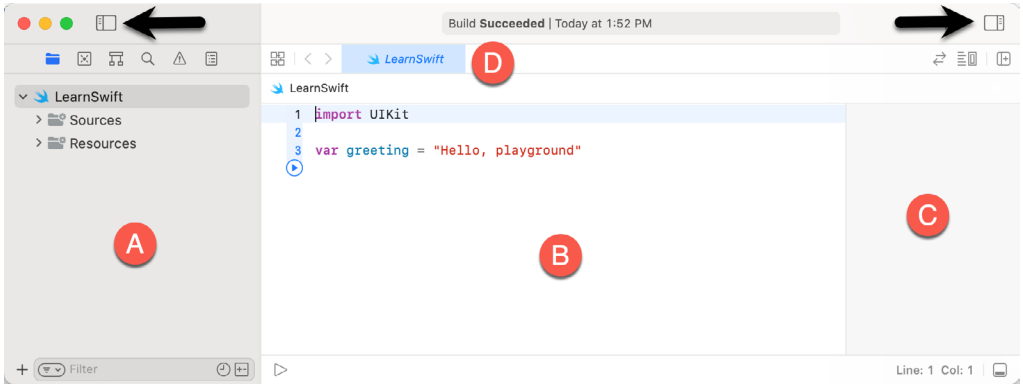

Figure 5-1

The panel on the left-hand side of the window (marked A in Figure 5-1) 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 leftmost 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 5-2 below), where various 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 the code is executed:

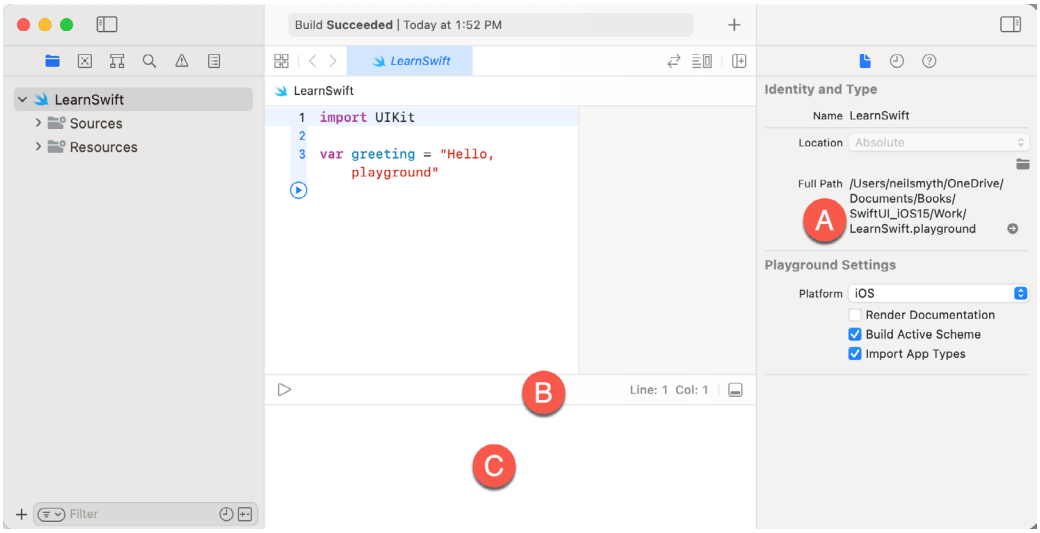

Figure 5-2

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

5.3 A Swift Playground Example

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

```
import UIKit


var greeting = "Hello, playground"

print("Welcome to Swift")
```

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

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

in the bottom left-hand corner of the main panel, as indicated by the arrow in Figure 5-3:

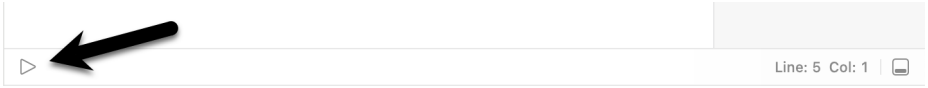

Figure 5-3

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 5-4:

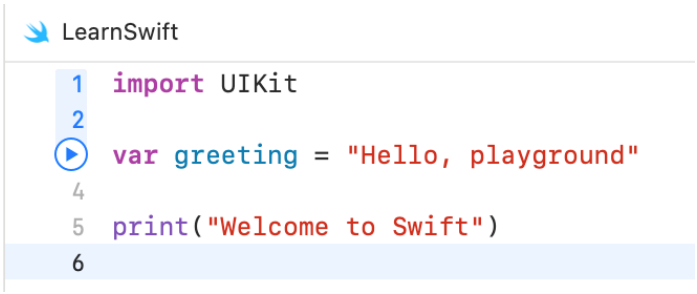

Figure 5-4

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

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

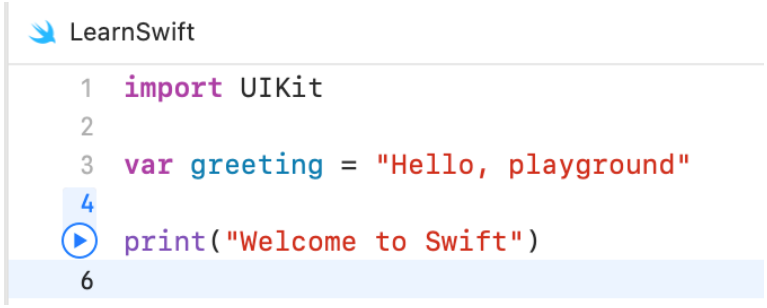

Figure 5-5

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

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

Figure 5-6

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

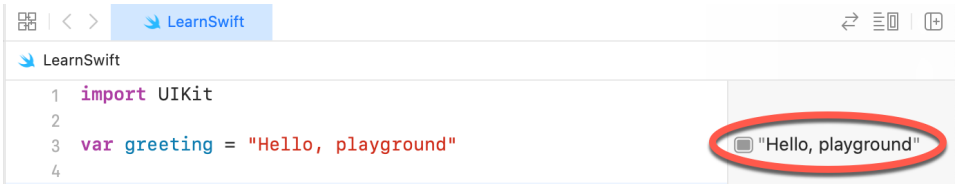

Figure 5-7

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

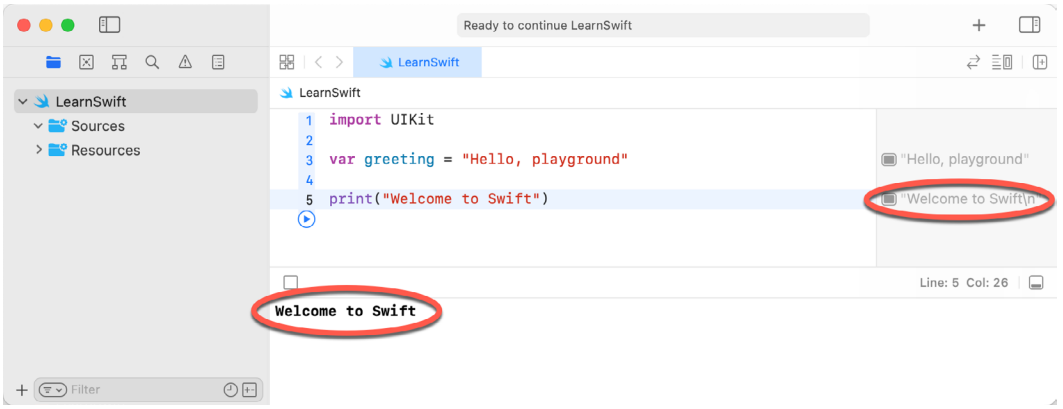

Figure 5-8

5.4 Viewing Results

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

```
var x = 10

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

This expression repeats a loop 20 times, performing 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 5-9:

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

"Welcome to Swift\n"

10

(20 times)
(20 times)
(20 times)

Figure 5-9

Hovering over the output will display the *Quick Look* button on the far right, which, when selected, will show a popup panel displaying the results as shown in Figure 5-10:

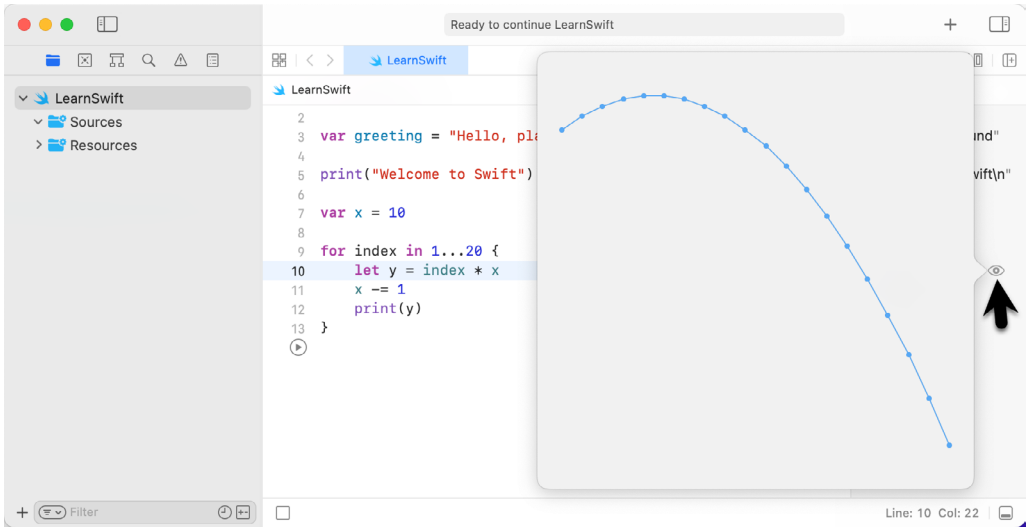

Figure 5-10

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

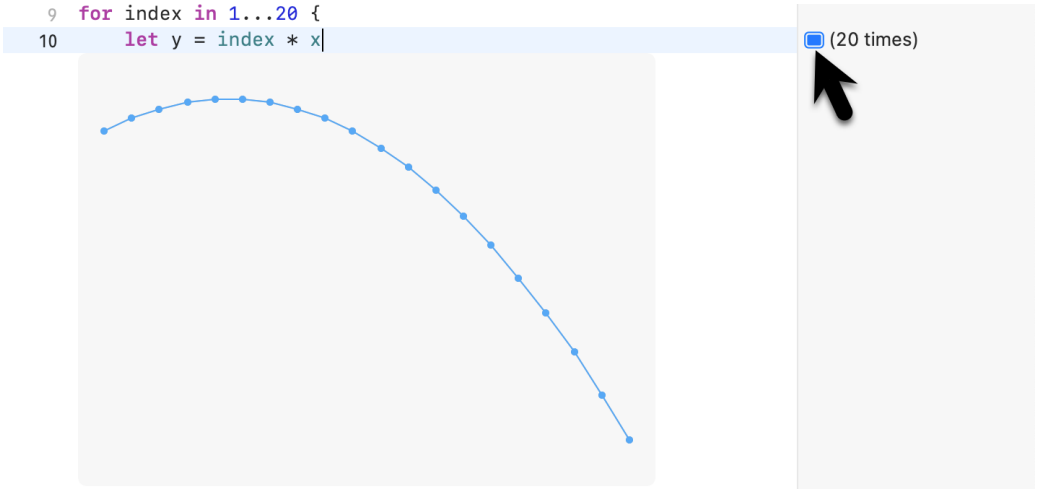

Figure 5-11

5.5 Adding Rich Text Comments

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

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

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

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

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

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

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

As the comment content is added, it is said to be displayed in *raw markup* format. To display in *rendered markup* format, either select the *Editor -> Show Rendered Markup* menu option or enable the *Render Documentation* option located under *Playground Settings* in the Inspector panel (marked A in Figure 5-2). If the Inspector panel is not currently visible, click on the button indicated by the right-most arrow in Figure 5-1 to display it. Once rendered, the above rich text should appear as illustrated in Figure 5-12:

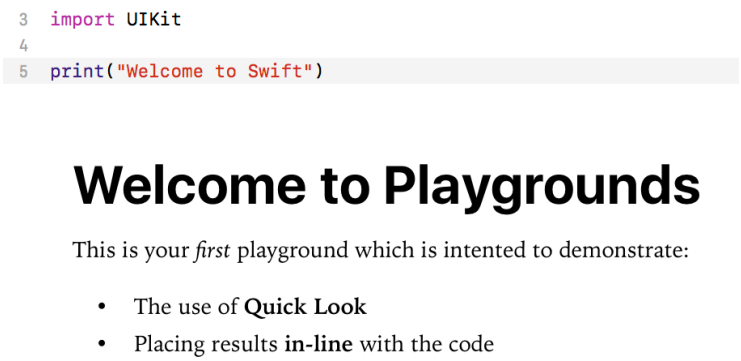

Figure 5-12

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

5.6 Working with Playground Pages

A playground can consist of multiple pages, with each page containing its own code, resources and, rich text comments. So far, the playground used in this chapter contains a single page. Add a page to the playground now by selecting the LearnSwift entry at the top of the Navigator panel, right-clicking, and selecting the *New Playground Page* menu option. If the Navigator panel is not currently visible, click the button indicated by the left-most arrow in Figure 5-1 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 *UIKit Examples* as outlined in Figure 5-13:

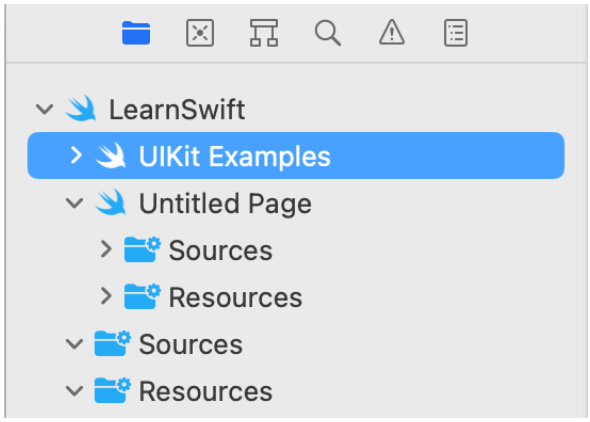

Figure 5-13

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

5.7 Working with UIKit in Playgrounds

The playground environment is not restricted to simple Swift code statements. Much of the power of the iOS SDK is also available for experimentation within a playground.

When working with UIKit within a playground page, we first need to import the iOS UIKit Framework. The UIKit Framework contains most of the classes required to implement user interfaces for iOS apps and is an area that will be covered in considerable detail throughout the book. A compelling feature of playgrounds is that it is possible to work with UIKit and many other frameworks that comprise the iOS SDK.

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

```
import UIKit

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

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

Enter this code into the playground editor on the UIKit Examples page (the existing code can be removed) and run the code. This code provides an excellent demonstration of how the Quick Look feature can be helpful.

Each line of the example Swift code configures a different aspect of the appearance of the UILabel instance. For example, clicking the Quick Look button for the first line of code will display an empty view (since the label exists but has yet to be given any visual attributes). Clicking on the Quick Look button in the line of code which sets the background color, on the other hand, will show the red label:

Figure 5-14

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

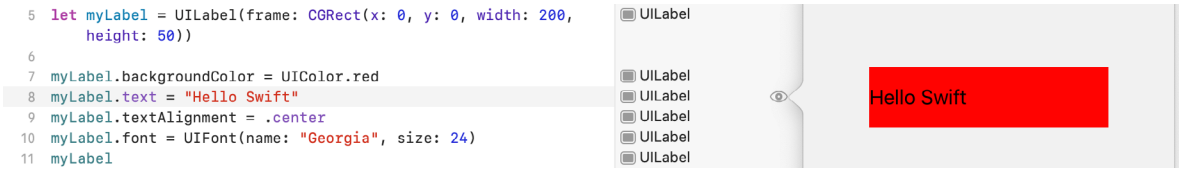

Figure 5-15

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

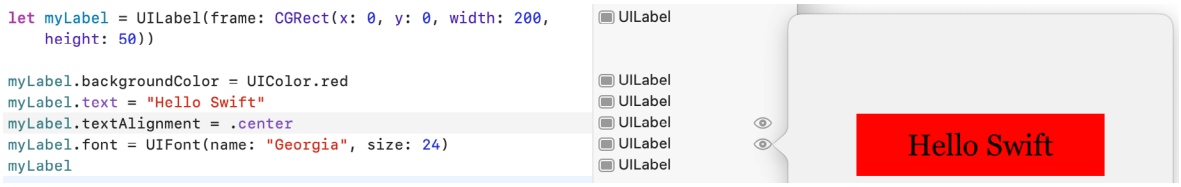

Figure 5-16

5.8 Adding Resources to a Playground

Another helpful feature of playgrounds is the ability to bundle and access resources such as image files in a playground. For example, within the Navigator panel, click on the right-facing arrow (known as a *disclosure arrow*) to the left of the UIKit Examples page entry to unfold the page contents (Figure 5-17) and note the presence of a folder named *Resources*:

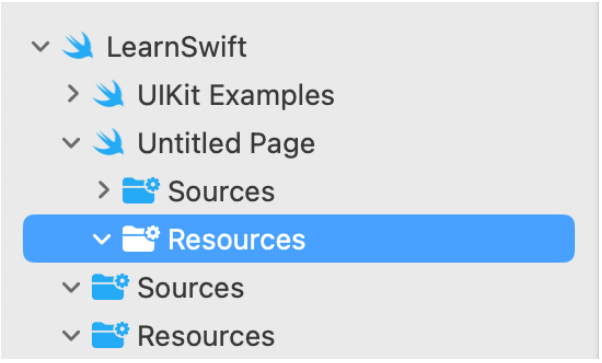

Figure 5-17

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

https://www.ebookfrenzy.com/retail/ios16/

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

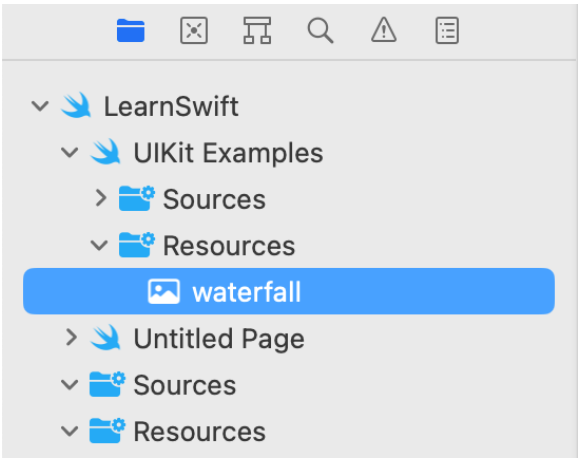

Figure 5-18

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

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

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

Figure 5-19

5.9 Working with Enhanced Live Views

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

```
import UIKit
import PlaygroundSupport
```

The PlaygroundSupport module provides several useful playground features, including a live view within the playground timeline.

Beneath the import statements, add the following code:

```
import UIKit
import PlaygroundSupport

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

container.addSubview(square)

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

The code creates a UIView object as a container view and assigns it a white background color. A smaller view is then drawn and positioned in the center of the container view and colored red. The second view is then added as a child of the container view. Animation is then used to change the smaller view's color to blue and rotate it 360 degrees. If you are new to iOS programming, rest assured that these areas will be covered in detail in later chapters. At this point, the code is provided to highlight the capabilities of live views.

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

The PlaygroundSupport module includes a class named PlaygroundPage that allows playground code to interact with the pages that make up a playground. This is achieved through a range of methods and properties of the class, one of which is the *current* property. This property, in turn, provides access to the current playground page. To execute the live view within the playground timeline, the *liveView* property of the current page needs to be set to our new container. To enable the timeline, enable the Xcode *Editor -> Live View* menu option as shown in Figure 5-20:

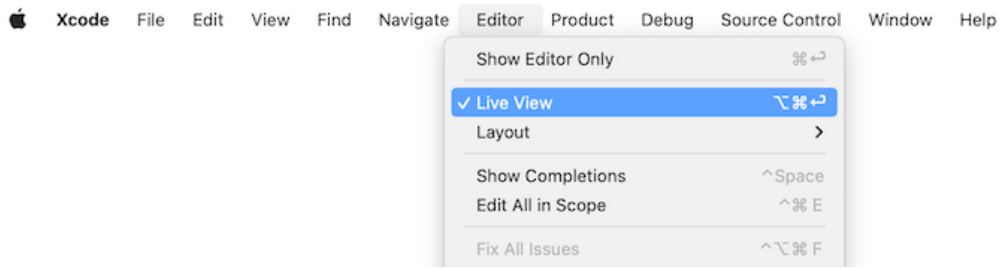

Figure 5-20

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

```
import UIKit
import PlaygroundSupport

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

PlaygroundPage.current.liveView = container

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

container.addSubview(square)

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

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

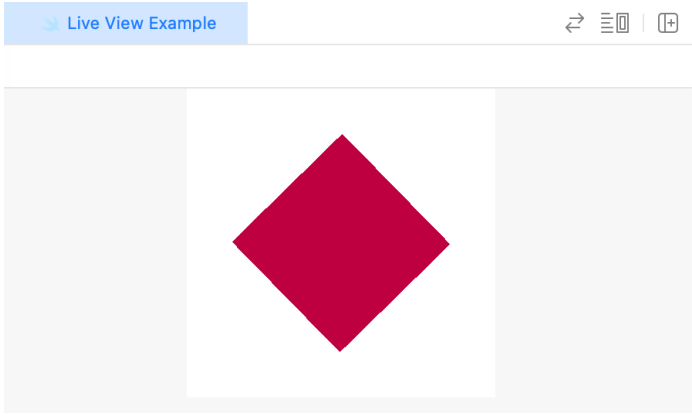

Figure 5-21

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

5.10 When to Use Playgrounds

Swift Playgrounds provide an ideal environment for learning to program using the Swift programming language, and the use of playgrounds in the Swift introductory chapters is recommended.

It is also essential to remember that playgrounds will remain useful long after the basics of Swift have been learned and will become increasingly useful when moving on to more advanced areas of iOS development.

The iOS SDK is a vast collection of frameworks and classes. As a result, it is not unusual for even experienced developers to experiment with unfamiliar aspects of iOS development before adding code to a project. Historically this has involved creating a temporary iOS Xcode project and then repeatedly looping through the somewhat cumbersome edit, compile, and run cycle to arrive at a programming solution. Rather than fall into this habit, consider having a playground on standby to conduct experiments during your project development work.

5.11 Summary

This chapter has introduced the concept of playgrounds. Playgrounds provide an environment in which Swift code can be entered, and the results of that code are viewed dynamically. This provides an excellent environment for learning the Swift programming language and experimenting with many of the classes and APIs included in the iOS SDK without creating Xcode projects and repeatedly editing, compiling, and running code.

Chapter 6

6. Swift Data Types, Constants and Variables

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

Due entirely to the popularity of iOS, Objective-C had become one of the more widely used programming languages. With origins firmly rooted in the 40-year-old C Programming Language, however, and 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.

6.1 Using a Swift Playground

Both this and the following few chapters are intended to introduce the basics of the Swift programming language. As outlined in the previous chapter, entitled *"An Introduction to Xcode 14 Playgrounds"* the best way to learn Swift is to experiment within a Swift playground environment. Before starting this chapter, therefore, create a new playground and use it to try out the code in both this and the other Swift introduction chapters that follow.

6.2 Swift Data Types

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

Humans, of course, don't think in binary. We work with decimal numbers, letters and words. In order for a human to easily (easily being a 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.

6.2.1 Integer Data Types

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

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

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

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

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

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

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

6.2.2 Floating Point Data Types

The Swift floating point data types are able to store values containing decimal places. For example, 4353.1223 would be stored in a floating-point data type. Swift provides two floating point data types in the form of *Float* and *Double*. Which type to use depends on the size of value to be stored and the level of precision required. The Double type can be used to store up to 64-bit floating point numbers with a level of precision of 15 decimal places or greater. The Float data type, on the other hand, is limited to 32-bit floating point numbers and offers a level of precision as low as 6 decimal places depending on the native platform on which the code is running. Alternatively, the Float16 type may be used to store 16-bit floating point values. Float16 provides greater performance at the expense of lower precision.

6.2.3 Bool Data Type

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

6.2.4 Character Data Type

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

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

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

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

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

6.2.5 String Data Type

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

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

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

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

print(message)
```

When executed, the code will output the following message:

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

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

```
var multiline = """

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

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

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

"""

print(multiline)
```

The above code will generate the following output when run:

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

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

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

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

6.2.6 Special Characters/Escape Sequences

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

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

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

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

Commonly used special characters supported by Swift are as follows:

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

6.3 Swift Variables

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

6.4 Swift Constants

A constant is 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.

6.5 Declaring Constants and Variables

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

```
var userCount = 10
```

Constants are declared using the *let* keyword.

```
let maxUserCount = 20
```

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

6.6 Type Annotations and Type Inference

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

There are two ways in which the type of a constant or variable will be identified. One approach is to use a *type annotation* at the point the variable or constant is declared in the code. This is achieved by placing a colon after the constant or variable name followed by the type declaration. The following line of code, for example, declares a variable named userCount as being of type Int:

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

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

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

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

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

```
let bookTitle = "iOS 16 App Development Essentials"
```

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

```
let bookTitle: String
.
.
if iosBookType {
    bookTitle = "iOS 16 App Development Essentials"
} else {
    bookTitle = "Android Studio Development Essentials"
}
```

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

6.7 The Swift Tuple

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

6.8 The Swift Optional Type

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

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

```
var index: Int?
```

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

An optional can easily be tested (typically using an if statement) to identify whether 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 in more detail in the chapter entitled "*Working with Array and Dictionary Collections in Swift*"). If the index optional variable has a value assigned to it, the tree name at that location in the array is printed to the console. Since the index is an optional type, the value has been unwrapped by placing an exclamation mark after the variable name:

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

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

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

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

```
if let constantname = optionalName {

}

if var variablename = optionalName {

}
```

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

```
var index: Int?

index = 3

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

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

In this case the value assigned to the index variable is unwrapped and assigned to a temporary (also referred to as *shadow*) 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 {
.
.
```

When considering the above example, the use of the temporary value begins to seem redundant. Fortunately, the Swift development team arrived at the same conclusion and introduced the following shorthand if-let syntax in Swift 5.7:

```
var index: Int?

index = 3
```

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

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

Using this approach it is no longer necessary to assign the optional to a temporary value.

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 shorthand if-let syntax is also available when working with multiple optionals and test conditions avoiding the need to use temporary values:

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

}
```

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

```
var pet1: String?
var pet2: String?

pet1 = "cat"
pet2 = "dog"

if let pet1, let pet2 {
    print(pet1)
    print(pet2)
} else {
    print("insufficient pets")
}
```

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

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

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

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

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

index = 3

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

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

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

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

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

6.9 Type Casting and Type Checking

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

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

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

In fact, there are two types of casting which are referred to as *upcasting* and *downcasting*. Upcasting occurs when an object of a particular class is cast to one of its superclasses. Upcasting is performed using the *as* keyword and is also referred to as *guaranteed conversion* since the compiler can tell from the code that the cast will be successful. The UIButton class, for example, is a subclass of the UIControl class as shown in the fragment of the UIKit class hierarchy shown in Figure 6-1:

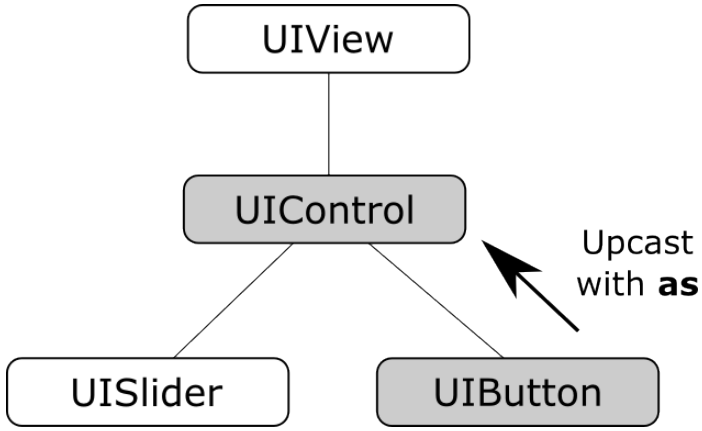

Figure 6-1

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

```
let myButton: UIButton = UIButton()


let myControl = myButton as UIControl
```

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

Downcasting usually involves converting from a class to one of its subclasses. Downcasting is performed using the *as!* keyword syntax and is also referred to as *forced conversion*. Consider, for example, the UIKit UIScrollView class which has as subclasses both the UITableView and UITextView classes as shown in Figure 6-2:

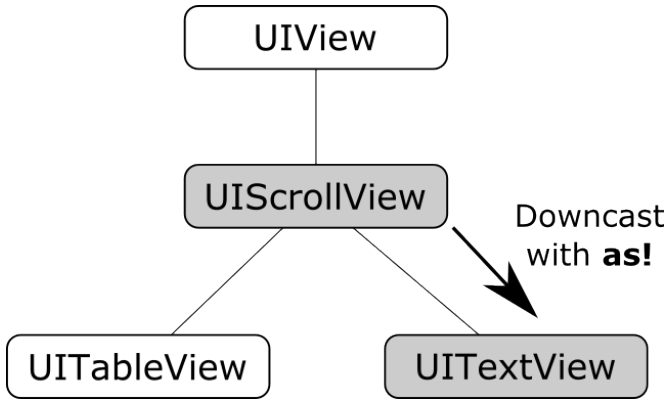

Figure 6-2

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

```
let myScrollView: UIScrollView = UIScrollView()


let myTextView = myScrollView as UITextView
```

The above code will result in the following error:

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

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

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

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

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

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

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

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

6.10 Summary

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

Chapter 7

7. Swift Operators and Expressions

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

7.1 Expression Syntax in Swift

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

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

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

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

7.2 The Basic Assignment Operator

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

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

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

7.3 Swift Arithmetic Operators

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

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

The following table lists the primary Swift arithmetic operators:

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

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

Table 7-1

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

For example:

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

7.4 Compound Assignment Operators

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

```
x = x + y
```

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

```
x += y
```

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

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

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

Table 7-2

7.5 Comparison Operators

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

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

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

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

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

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

result = x < y
```

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

Operator	Description
x == y	Returns true if x is equal to y
x > y	Returns true if x is greater than y
x >= y	Returns true if x is greater than or equal to y
x < y	Returns true if x is less than y
x <= y	Returns true if x is less than or equal to y
x != y	Returns true if x is not equal to y

Table 7-3

7.6 Boolean Logical Operators

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

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

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

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

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

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

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

7.7 Range Operators

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

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

x…y

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

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

```
x..<y
```

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

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

x…

or

…y

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

2…

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

…6

7.8 The Ternary Operator

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

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

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

```
let x = 10
let y = 20


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

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

```
Largest number is 20
```

7.9 Nil Coalescing Operator

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

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

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

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

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

7.10 Bitwise Operators

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

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

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

```
10101011
```

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

```
00000011
```

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

7.10.1 Bitwise NOT

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

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

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

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

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

7.10.2 Bitwise AND

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

```
10101011 AND
```

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

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

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

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

7.10.3 Bitwise OR

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

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

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

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

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

7.10.4 Bitwise XOR

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

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

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

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

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

7.10.5 Bitwise Left Shift

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

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

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

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

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

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

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

7.10.6 Bitwise Right Shift

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

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

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

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

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

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

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

7.11 Compound Bitwise Operators

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

Operator	Description
x &= y	Perform a bitwise AND of x and y and assign result to x
x |= y	Perform a bitwise OR of x and y and assign result to x
x ^= y	Perform a bitwise XOR of x and y and assign result to x
x <<= n	Shift x left by n places and assign result to x
x >>= n	Shift x right by n places and assign result to x

Table 7-4

7.12 Summary

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

Chapter 8

8. 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.

8.1 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.

8.2 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
}
```

8.2.1 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.

8.3 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)
```

8.4 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.

8.5 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).

8.6 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.

8.7 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.

8.8 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 10!")
}
```

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

8.9 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*".

8.10 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.

8.11 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. The Swift Switch Statement

In *"Swift Control Flow"* we looked at how to control program execution flow using the *if* and *else* statements. While these statement constructs work well for testing a limited number of conditions, they quickly become unwieldy when dealing with larger numbers of possible conditions. To simplify such situations, Swift has inherited the *switch* statement from the C programming language. Those familiar with the switch statement from other programming languages should be aware, however, that the Swift switch statement has some key differences from other implementations. In this chapter we will explore the Swift implementation of the *switch* statement in detail.

9.1 Why Use a switch Statement?

For a small number of logical evaluations of a value the *if ... else if ...* construct is perfectly adequate. Unfortunately, any more than two or three possible scenarios can quickly make such a construct both time consuming to write and difficult to read. For such situations, the *switch* statement provides an excellent alternative.

9.2 Using the switch Statement Syntax

The syntax for a basic Swift *switch* statement implementation can be outlined as follows:

```
switch expression
{
     case match1:
          statements

     case match2:
          statements

     case match3, match4:
          statements

     default:
          statements
}
```

In the above syntax outline, *expression* represents either a value, or an expression which returns a value. This is the value against which the *switch* operates.

For each possible match a *case* statement is provided, followed by a *match* value. Each potential match must be of the same type as the governing expression. Following on from the *case* line are the Swift statements that are to be executed in the event of the value matching the case condition.

Finally, the *default* section of the construct defines what should happen if none of the case statements present a match to the *expression*.

9.3 A Swift switch Statement Example

With the above information in mind we may now construct a simple *switch* statement:

```
let value = 4

switch (value)
{
      case 0:
        print("zero")

      case 1:
        print("one")

      case 2:
        print("two")

      case 3:
        print("three")

      case 4:
        print("four")

      case 5:
        print("five")

      default:
        print("Integer out of range")
}
```

9.4 Combining case Statements

In the above example, each case had its own set of statements to execute. Sometimes a number of different matches may require the same code to be executed. In this case, it is possible to group case matches together with a common set of statements to be executed when a match for any of the cases is found. For example, we can modify the switch construct in our example so that the same code is executed regardless of whether the value is 0, 1 or 2:

```
let value = 1

switch (value)
{
      case 0, 1, 2:
        print("zero, one or two")

      case 3:
        print("three")

      case 4:
        print("four")

      case 5:
```

```
        print("five")

    default:
        print("Integer out of range")
}
```

9.5 Range Matching in a switch Statement

The case statements within a switch construct may also be used to implement range matching. The following switch statement, for example, checks a temperature value for matches within three number ranges:

```
let temperature = 83

switch (temperature)
{
    case 0...49:
        print("Cold")

    case 50...79:
        print("Warm")

    case 80...110:
        print("Hot")

    default:
        print("Temperature out of range")
}
```

9.6 Using the where statement

The *where* statement may be used within a switch case match to add additional criteria required for a positive match. The following switch statement, for example, checks not only for the range in which a value falls, but also whether the number is odd or even:

```
let temperature = 54

switch (temperature)
{
    case 0...49 where temperature % 2 == 0:
        print("Cold and even")

    case 50...79 where temperature % 2 == 0:
        print("Warm and even")

    case 80...110 where temperature % 2 == 0:
        print("Hot and even")

    default:
        print("Temperature out of range or odd")
}
```

9.7 Fallthrough

Those familiar with switch statements in other languages such as C and Objective-C will notice that it is no longer necessary to include a *break* statement after each case declaration. Unlike other languages, Swift automatically breaks out of the statement when a matching case condition is met. The fallthrough effect of other switch implementations (whereby the execution path continues through the remaining case statements) can be emulated using the *fallthrough* statement:

```
let temperature = 10

switch (temperature)
{
      case 0...49 where temperature % 2 == 0:
        print("Cold and even")
        fallthrough

      case 50...79 where temperature % 2 == 0:
        print("Warm and even")
        fallthrough

      case 80...110 where temperature % 2 == 0:
        print("Hot and even")
        fallthrough

      default:
        print("Temperature out of range or odd")
}
```

Although *break* is less commonly used in Swift switch statements, it is useful when no action needs to be taken for the default case. For example:

```
.
.
.
default:
     break
}
```

9.8 Summary

While the *if.. else..* construct serves as a good decision-making option for small numbers of possible outcomes, this approach can become unwieldy in more complex situations. As an alternative method for implementing flow control logic in Swift when many possible outcomes exist as the result of an evaluation, the *switch* statement invariably makes a more suitable option. As outlined in this chapter, however, developers familiar with switch implementations from other programming languages should be aware of some subtle differences in the way that the Swift switch statement works.

Chapter 10

10. 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.

10.1 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*.

10.2 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.

10.3 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)")
}
```

10.4 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
}
```

10.5 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()
```

10.6 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)
```

10.7 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.

10.8 Declaring Default Function Parameters

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

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

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

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

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

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

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

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

10.9 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)
```

10.10 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")
```

10.11 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))
```

10.12 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
```

10.13 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 Feet
```

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

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

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

10.14 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)
    }
})
```

10.15 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.

10.16 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.

10.17 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 24

24. Using Storyboards in Xcode 14

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

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

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

24.1 Creating the Storyboard Example Project

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

24.2 Accessing the Storyboard

Upon creating the new project, Xcode will have created what appears to be the usual collection of files for a single-view app, including a storyboard named file *Main.storyboard*. Select this file in the project navigator panel to view the storyboard canvas as illustrated in Figure 24-1.

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

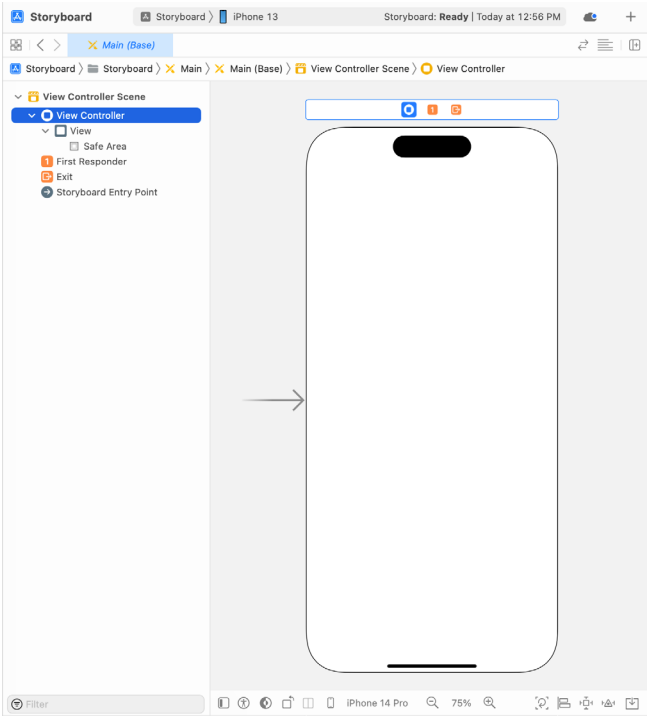

Figure 24-1

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

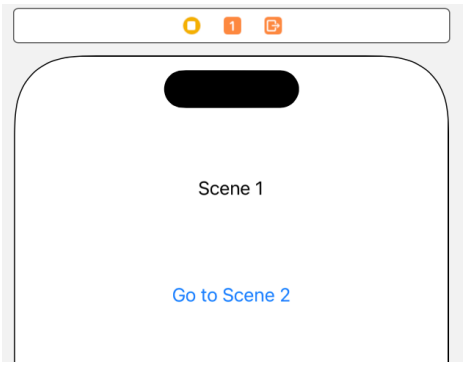

Figure 24-2

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

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

```
import UIKit

class ViewController: UIViewController {

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

24.3 Adding Scenes to the Storyboard

To add a second scene to the storyboard, drag a View Controller object from the Library panel onto the canvas. Figure 24-3 shows a second scene added to a storyboard:

Figure 24-3

Drag and drop a label and a button into the second scene and configure the objects so that the view appears as shown in Figure 24-4. Then, repeat the steps performed for the first scene to configure Auto Layout constraints on the two views.

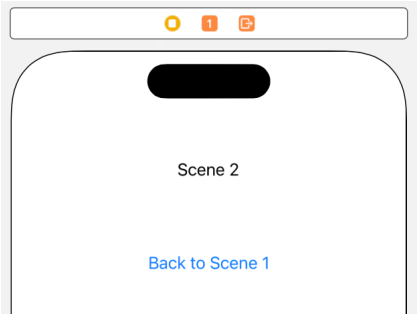

Figure 24-4

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

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

24.4 Configuring Storyboard Segues

As previously discussed, a segue is a transition from one scene to another within a storyboard. Within the example app, touching the *Go To Scene 2* button will segue to scene 2. Conversely, the button on scene 2 is intended to return the user to scene 1. To establish a segue, hold down the Ctrl key on the keyboard, click over a control (in this case, the button on scene 1), and drag the resulting line to the scene 2 view. Upon releasing the mouse button, a menu will appear. Select the *Present Modally* menu option to establish the segue. Once the segue has been added, a connector will appear between the two scenes, as highlighted in Figure 24-5:

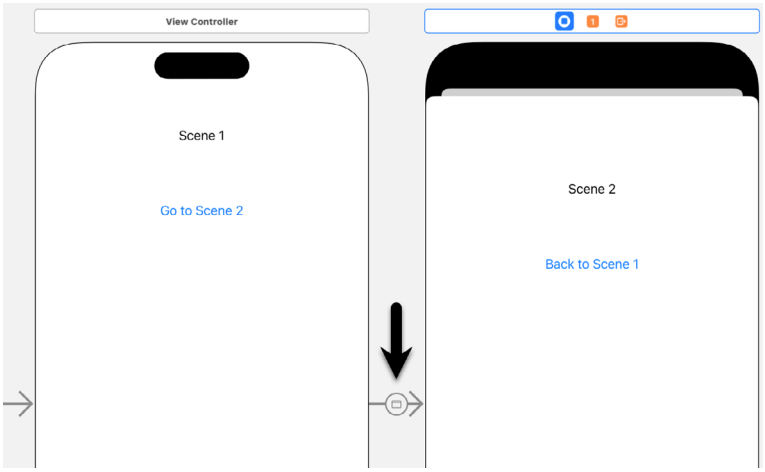

Figure 24-5

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

24.5 Configuring Storyboard Transitions

Xcode allows changing the visual appearance of the transition that occurs during a segue. To change the transition, select the corresponding segue connector, display the Attributes Inspector, and modify the *Transition* setting. For example, in Figure 24-6, the transition has been changed to *Cross Dissolve:*

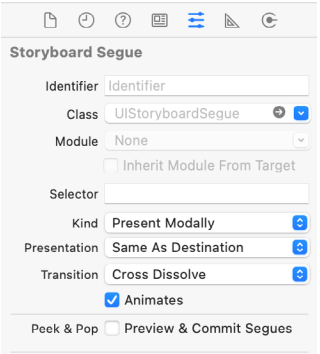

Figure 24-6

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

24.6 Associating a View Controller with a Scene

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

Select the *Main.storyboard* file in the project navigator panel and click the View Controller button located in the panel above the Scene 2 view, as shown in Figure 24-7:

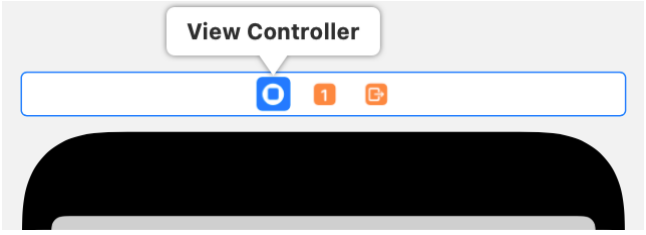

Figure 24-7

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

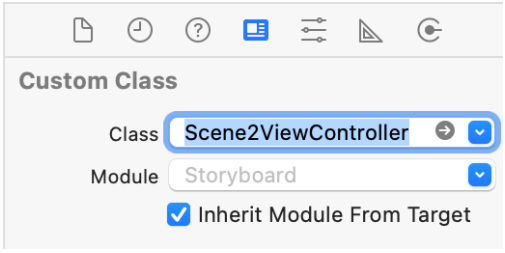

Figure 24-8

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

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

24.7 Passing Data Between Scenes

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

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

```
import UIKit

class Scene2ViewController: UIViewController {

    @IBOutlet weak var scene2Label: UILabel!

    var labelText: String?
.
.
.
```

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

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

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

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

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

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

24.8 Unwinding Storyboard Segues

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

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

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

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

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

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

The next step is to establish the unwind segue. To achieve this, locate scene 2 within the storyboard canvas and right-click and drag from the button view to the "exit" icon (the orange button with the white square and the right-facing arrow pointing outward shown in Figure 24-9) in the panel located along the top edge of the scene view. Release the line and select the *returnedWithSegue* method from the resulting menu:

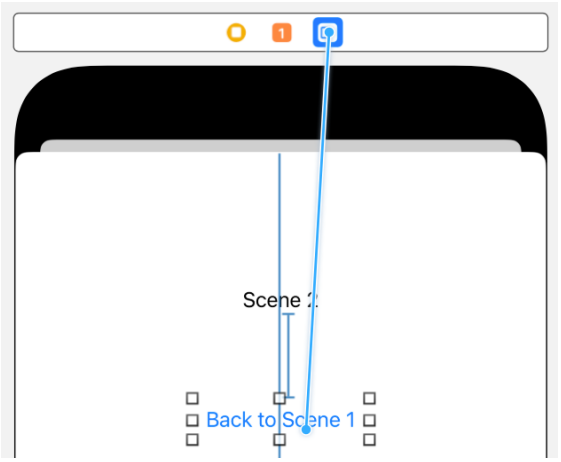

Figure 24-9

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

24.9 Triggering a Storyboard Segue Programmatically

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

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

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

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

24.10 Summary

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

Chapter 35

35. An Overview of Swift Structured Concurrency

Concurrency can be defined as the ability of software to perform multiple tasks in parallel. Many app development projects will need to use concurrent processing at some point, and concurrency is essential for providing a good user experience. Concurrency, for example, allows an app's user interface to remain responsive while performing background tasks such as downloading images or processing data.

In this chapter, we will explore the *structured concurrency* features of the Swift programming language and explain how these can be used to add multi-tasking support to your app projects.

35.1 An Overview of Threads

Threads are a feature of modern CPUs and provide the foundation of concurrency in any multitasking operating system. Although modern CPUs can run large numbers of threads, the actual number of threads that can be run in parallel at any one time is limited by the number of CPU cores (depending on the CPU model, this will typically be between 4 and 16 cores). When more threads are required than there are CPU cores, the operating system performs thread scheduling to decide how the execution of these threads is to be shared between the available cores.

Threads can be thought of as mini-processes running within a main process, the purpose of which is to enable at least the appearance of parallel execution paths within application code. The good news is that although structured concurrency uses threads behind the scenes, it handles all of the complexity for you, and you should never need to interact with them directly.

35.2 The Application Main Thread

When an app is first started, the runtime system will typically create a single thread in which the app will run by default. This thread is generally referred to as the *main thread*. The primary role of the main thread is to handle the user interface in terms of UI layout rendering, event handling, and user interaction with views in the user interface.

Any additional code within an app that performs a time-consuming task using the main thread will cause the entire application to appear to lock up until the task is completed. This can be avoided by launching the tasks to be performed in separate threads, allowing the main thread to continue unhindered with other tasks.

35.3 Completion Handlers

As outlined in the chapter entitled *"Swift Functions, Methods and Closures"*, Swift previously used completion handlers to implement asynchronous code execution. In this scenario, an asynchronous task would be started, and a completion handler would be assigned to be called when the task finishes. In the meantime, the main app code would continue to run while the asynchronous task is performed in the background. On completion of the asynchronous task, the completion handler would be called and passed any results. The body of the completion handler would then execute and process those results.

Unfortunately, completion handlers tend to result in complex and error-prone code constructs that are difficult to write and understand. Completion handlers are also unsuited to handling errors thrown by asynchronous

tasks and generally result in large and confusing nested code structures.

35.4 Structured Concurrency

Structured concurrency was introduced into the Swift language with Swift version 5.5 to make it easier for app developers to implement concurrent execution safely and in a logical and easy way to both write and understand. In other words, structured concurrency code can be read from top to bottom without jumping back to completion handler code to understand the logic flow. Structured concurrency also makes it easier to handle errors thrown by asynchronous functions.

Swift provides several options for implementing structured concurrency, each of which will be introduced in this chapter.

35.5 Preparing the Project

Launch Xcode and select the option to create a new iOS *App* project named ConcurrencyDemo. Once created, select the Main.storyboard file, display the Library, and drag a Button object onto the center of the scene layout. Next, double-click on the button and change the text to read "Async Test", then use the Align menu to add constraints to center the button horizontally and vertically in the container:

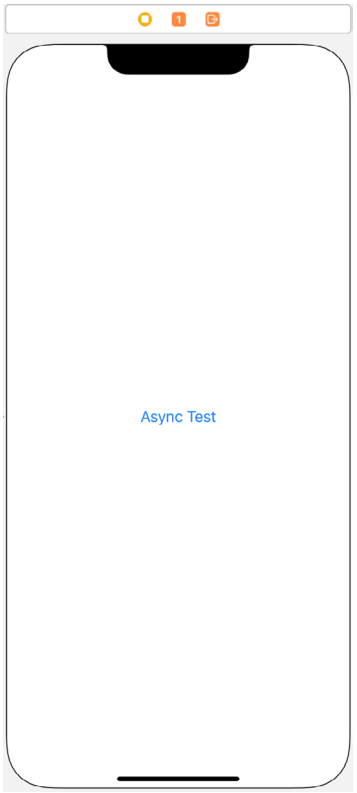

Figure 35-1

Display the Assistant Editor and establish an action connection from the button to a method named *buttonClick()*. Next, edit the *ViewController.swift* file and add two additional functions that will be used later in the chapter. Finally, modify the buttonCLick() method to call the doSomething function:

```
import UIKit
```

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    @IBAction func buttonClick(_ sender: Any) {
        doSomething()
    }

    func doSomething() {
    }

    func takesTooLong() {
    }
}
```

35.6 Non-Concurrent Code

Before exploring concurrency, we will first look at an example of non-concurrent code (also referred to as *synchronous code*) execution. Begin by adding the following code to the two stub functions. The changes to the *doSomething()* function print out the current date and time before calling the *takesTooLong()* function. Finally, the date and time are output once again before the *doSomething()* function exits.

The *takesTooLong()* function uses the system *sleep()* method to simulate the effect of performing a time-consuming task that blocks the main thread until it is complete before printing out another timestamp:

```
func doSomething() {
    print("Start \(Date())")
    takesTooLong()
    print("End \(Date())")
}

func takesTooLong() {
    sleep(5)
    print("Async task completed at \(Date())")
}
```

Run the app on a device or simulator and click on the "Async Test" button. Output similar to the following should appear in the Xcode console panel:

```
Start 2023-02-05 17:43:10 +0000
Async task completed at 2023-02-05 17:43:15 +0000
End 2023-02-05 17:43:15 +0000
```

The key point to note in the above timestamps is that the end time is 5 seconds after the start time. This tells us not only that the call to *takesTooLong()* lasted 5 seconds as expected but that any code after the call was made within the *doSomething()* function was not able to execute until after the call returned. During that 5 seconds, the app would appear to the user to be frozen.

The answer to this problem is implementing a Swift *async/await* concurrency structure. Before looking at *async/*

await, we will need a Swift concurrency-compatible alternative to the *sleep()* call used in the above example. To achieve this, add a new function to the *ViewController.swift* file that reads as follows:

```
func taskSleep(_: Int) async {
    do {
        try await Task.sleep(until: .now + .seconds(5), clock: .continuous)
    } catch { }
}
```

Ironically, the new function used Swift concurrency before we covered the topic. Rest assured, the techniques used in the above code will be explained in the remainder of this chapter.

35.7 Introducing async/await Concurrency

The foundation of structured concurrency is the *async/await* pair. The *async* keyword is used when declaring a function to indicate that it will be executed asynchronously relative to the thread from which it was called. We need, therefore, to declare both of our example functions as follows (any errors that appear will be addressed later):

```
func doSomething() async {
    print("Start \(Date())")
    takesTooLong()
    print("End \(Date())")
}

func takesTooLong() async {
    await taskSleep(5)
    print("Async task completed at \(Date())")
}
```

Marking a function as async achieves several objectives. First, it indicates that the code in the function needs to be executed on a different thread to the one from which it was called. It also notifies the system that the function itself can be suspended during execution to allow the system to run other tasks. As we will see later, these *suspend points* within an async function are specified using the *await* keyword.

Another point to note about async functions is that they can generally only be called from within the scope of other async functions though, as we will see later in the chapter, the Task object can be used to provide a bridge between synchronous and asynchronous code. Finally, if an async function calls other async functions, the parent function cannot exit until all child tasks have also been completed.

Most importantly, once a function has been declared asynchronous, it can only be called using the await keyword. Before looking at the *await* keyword, we must understand how to call async functions from synchronous code.

35.8 Asynchronous Calls from Synchronous Functions

The rules of structured concurrency state that an async function can only be called from within an asynchronous context. If the entry point into your program is a synchronous function, this raises the question of how any async functions can ever get called. The answer is to use the Task object from within the synchronous function to launch the async function. Suppose we have a synchronous function named *main()* from which we need to call one of our async functions and attempt to do so as follows:

```
func main() {
    doSomething()
}
```

The above code will result in the following error notification in the code editor:

```
'async' call in a function that does not support concurrency
```

The only options we have are to make *main()* an async function or to launch the function in an unstructured task. Assuming that declaring *main()* as an async function is not a viable option, in this case, the code will need to be changed as follows:

```
func main() {
    Task {
        await doSomething()
    }
}
```

35.9 The await Keyword

As we previously discussed, the await keyword is required when making a call to an async function and can only usually be used within the scope of another async function. Attempting to call an async function without the await keyword will result in the following syntax error:

```
Expression is 'async' but is not marked with 'await'
```

To call the *takesTooLong()* function, therefore, we need to make the following change to the *doSomething()* function:

```
func doSomething() async {
    print("Start \(Date())")
    await takesTooLong()
    print("End \(Date())")
}
```

One more change is now required because we are attempting to call the async *doSomething()* function from a synchronous context (in this case, the buttonClick action method). To resolve this, we need to use the Task object to launch the *doSomething()* function:

```
@IBAction func buttonClick(_ sender: Any) {
    Task {
        await doSomething()
    }
}
```

When tested now, the console output should be similar to the following:

```
Start 2023-02-05 17:53:03 +0000
Async task completed at 2023-02-05 17:53:08 +0000
End 2023-02-05 17:53:08 +0000
```

This is where the await keyword can be a little confusing. As you have probably noticed, the *doSomething()* function still had to wait for the *takesTooLong()* function to return before continuing, giving the impression that the task was still blocking the thread from which it was called. In fact, the task was performed on a different thread, but the await keyword told the system to wait until it completed. The reason for this is that, as previously mentioned, a parent async function cannot complete until all of its sub-functions have also completed. This means that the call has no choice but to wait for the async *takesTooLong()* function to return before executing the next line of code. The next section will explain how to defer the wait until later in the parent function using the *async-let* binding expression. Before doing that, however, we need to look at another effect of using the await keyword in this context.

In addition to allowing us to make the async call, the await keyword has also defined a *suspend point* within the *doSomething()* function. When this point is reached during execution, it tells the system that the *doSomething()* function can be temporarily suspended and the thread on which it is running used for other purposes. This allows the system to allocate resources to any higher priority tasks and will eventually return control to the *doSomething()* function so that execution can continue. By marking suspend points, the *doSomething()* function is essentially being forced to be a good citizen by allowing the system to briefly allocate processing resources to other tasks. Given the speed of the system, it is unlikely that a suspension will last more than fractions of a second and will not be noticeable to the user while benefiting the overall performance of the app.

35.10 Using async-let Bindings

In our example code, we have identified that the default behavior of the await keyword is to wait for the called function to return before resuming execution. A more common requirement, however, is to continue executing code within the calling function while the async function is executing in the background. This can be achieved by deferring the wait until later in the code using an async-let binding. To demonstrate this, we first need to modify our *takesTooLong()* function to return a result (in this case, our task completion timestamp):

```
func takesTooLong() async -> Date {
    await taskSleep(5)
    return Date()
}
```

Next, we need to change the call within *doSomething()* to assign the returned result to a variable using a *let* expression but also marked with the *async* keyword:

```
func doSomething() async {
    print("Start \(Date())")
    async let result = takesTooLong()
    print("End \(Date())")
}
```

Now, we need to specify where within the doSomething() function we want to wait for the result value to be returned. We do this by accessing the result variable using the await keyword. For example:

```
func doSomething() async {
    print("Start \(Date())")
    async let result = takesTooLong()
    print("After async-let \(Date())")
    // Additional code to run concurrently with async function goes here
    print ("result = \(await result)")
    print("End \(Date())")
}
```

When printing the result value, we are using await to let the system know that execution cannot continue until the async *takesTooLong()* function returns with the result value. At this point, execution will stop until the result is available. Any code between the async-let and the await, however, will execute concurrently with the *takesTooLong()* function.

Execution of the above code will generate output similar to the following:

```
Start 2023-02-05 17:56:00 +0000
After async-let 2023-02-05 17:56:00 +0000
result = 2023-02-05 17:56:05 +0000
End 2023-02-05 17:56:05 +0000
```

Note that the "After async-let" message has a timestamp that is 5 seconds earlier than the "result =" call return stamp confirming that the code was executed while *takesTooLong()* was also running.

35.11 Handling Errors

Error handling in structured concurrency uses the throw/do/try/catch mechanism previously covered in the chapter entitled *"Understanding Error Handling in Swift 5"*. The following example modifies our original async *takesTooLong()* function to accept a sleep duration parameter and to throw an error if the delay is outside of a specific range:

```
enum DurationError: Error {
    case tooLong
    case tooShort
}
.
.
func takesTooLong(delay: Int) async throws {

    if delay < 5 {
        throw DurationError.tooShort
    } else if delay > 20 {
        throw DurationError.tooLong
    }

    await taskSleep(delay)
    print("Async task completed at \(Date())")
}
```

Now when the function is called, we can use a do/try/catch construct to handle any errors that get thrown:

```
func doSomething() async {
    print("Start \(Date())")
    do {
        try await takesTooLong(delay: 25)
    } catch DurationError.tooShort {
        print("Error: Duration too short")
    } catch DurationError.tooLong {
        print("Error: Duration too long")
    } catch {
        print("Unknown error")
    }
    print("End \(Date())")
}
```

When executed, the resulting output will resemble the following:

```
Start 2022-03-30 19:29:43 +0000
Error: Duration too long
End 2022-03-30 19:29:43 +0000
```

35.12 Understanding Tasks

Any work that executes asynchronously runs within an instance of the Swift *Task* class. An app can run multiple tasks simultaneously and structures these tasks hierarchically. When launched, the async version of our *doSomething()* function will run within a Task instance. When the *takesTooLong()* function is called, the system creates a *sub-task* within which the function code will execute. In terms of the task hierarchy tree, this sub-task is a child of the *doSomething()* parent task. Any calls to async functions from within the sub-task will become children of that task, and so on.

This task hierarchy forms the basis on which structured concurrency is built. For example, child tasks inherit attributes such as priority from their parents, and the hierarchy ensures that a parent task does not exit until all descendant tasks have been completed.

As we will see later in the chapter, tasks can be grouped to enable the dynamic launching of multiple asynchronous tasks.

35.13 Unstructured Concurrency

Individual tasks can be created manually using the Task object, a concept referred to as unstructured concurrency. As we have already seen, a common use for unstructured tasks is to call async functions from within synchronous functions.

Unstructured tasks also provide more flexibility because they can be externally canceled at any time during execution. This is particularly useful if you need to provide the user with a way to cancel a background activity, such as tapping on a button to stop a background download task. This flexibility comes with extra cost in terms of having to do a little more work to create and manage tasks.

Unstructured tasks are created and launched by calling the Task initializer and providing a closure containing the code to be performed. For example:

```
Task {
    await doSomething()
}
```

These tasks also inherit the configuration of the parent from which they are called, such as the actor context, priority, and task local variables. Tasks can also be assigned a new priority when they are created, for example:

```
Task(priority: .high) {
    await doSomething()
}
```

This provides a hint to the system about how the task should be scheduled relative to other tasks. Available priorities ranked from highest to lowest are as follows:

- .high / .userInitiated
- .medium
- .low / .utility
- .background

When a task is manually created, it returns a reference to the Task instance. This can be used to cancel the task or to check whether the task has already been canceled from outside the task scope:

```
let task = Task(priority: .high) {
    await doSomething()
```

```
}
.
.
if (!task.isCancelled) {
    task.cancel()
}
```

35.14 Detached Tasks

Detached tasks are another form of unstructured concurrency, but they differ in that they do not inherit any properties from the calling parent. Detached tasks are created by calling the *Task.detached()* method as follows:

```
Task.detached {
    await doSomething()
}
```

Detached tasks may also be passed a priority value, and checked for cancellation using the same techniques as outlined above:

```
let detachedTask = Task.detached(priority: .medium) {
    await doSomething()
}
.
.
if (!detachedTask.isCancelled) {
    detachedTask.cancel()
}
```

35.15 Task Management

Whether you are using structured or unstructured tasks, the Task class provides a set of static methods and properties that can be used to manage the task from within the scope.

A task may, for example, use the *currentPriority* property to identify the priority assigned when it was created:

```
Task {
    let priority = Task.currentPriority
    await doSomething()
}
```

Unfortunately, this is a read-only property so cannot be used to change the priority of the running task.

It is also possible for a task to check if it has been canceled by accessing the *isCancelled* property:

```
if Task.isCancelled {
    // perform task cleanup
}
```

Another option for detecting cancellation is to call the *checkCancellation()* method, which will throw a CancellationError error if the task has been canceled:

```
do {
    try Task.checkCancellation()
} catch {
   // Perform task cleanup
}
```

A task may cancel itself at any time by calling the *cancel()* Task method:

```
Task.cancel()
```

Finally, if there are locations within the task code where execution could safely be suspended, these can be declared to the system via the *yield()* method:

```
Task.yield()
```

35.16 Working with Task Groups

So far in this chapter, our examples have involved creating one or two tasks (a parent and a child). In each case, we knew how many tasks were required in advance of writing the code. Situations often arise, however, where several tasks need to be created and run concurrently based on dynamic criteria. We might, for example, need to launch a separate task for each item in an array or within the body of a *for* loop. Swift addresses this by providing *task groups*.

Task groups allow a dynamic number of tasks to be created and are implemented using either the *withThrowingTaskGroup()* or *withTaskGroup()* functions (depending on whether or not the async functions in the group throw errors). The looping construct to create the tasks is then defined within the corresponding closure, calling the group *addTask()* function to add each new task.

Modify the two functions as follows to create a task group consisting of five tasks, each running an instance of the *takesTooLong()* function:

```
func doSomething() async {
    await withTaskGroup(of: Void.self) { group in
        for i in 1...5 {
            group.addTask {
                let result = await self.takesTooLong()
                print("Completed Task \(i) = \(result)")
            }
        }
    }
}

func takesTooLong() async -> Date {
    await taskSleep(5)
    return Date()
}
```

When executed, there will be a 5-second delay while the tasks run before output similar to the following appears:

```
Completed Task 1 = 2022-03-31 17:36:32 +0000
Completed Task 2 = 2022-03-31 17:36:32 +0000
Completed Task 5 = 2022-03-31 17:36:32 +0000
Completed Task 3 = 2022-03-31 17:36:32 +0000
Completed Task 4 = 2022-03-31 17:36:32 +0000
```

Note that the tasks all show the same completion timestamp indicating that they were executed concurrently. It is also interesting to notice that the tasks did not complete in the order in which they were launched. When working with concurrency, it is important to remember that there is no guarantee that tasks will be completed in the order they were created.

In addition to the *addTask()* function, several other methods and properties are accessible from within the task group, including the following:

- **cancelAll()** - Method call to cancel all tasks in the group
- **isCancelled** - Boolean property indicating whether the task group has already been canceled.
- **isEmpty** - Boolean property indicating whether any tasks remain within the task group.

35.17 Avoiding Data Races

In the above task group example, the group did not store the results of the tasks. In other words, the results did not leave the scope of the task group and were not retained when the tasks ended. For example, let's assume we want to store the task number and result timestamp for each task within a Swift dictionary object (with the task number as the key and the timestamp as the value). When working with synchronous code, we might consider a solution that reads as follows:

```
func doSomething() async {

    var timeStamps: [Int: Date] = [:]

    await withTaskGroup(of: Void.self) { group in
        for i in 1...5 {
            group.addTask {
                timeStamps[i] = await self.takesTooLong()
            }
        }
    }
}
```

Unfortunately, the above code will report the following error on the line where the result from the *takesTooLong()* function is added to the dictionary:

```
Mutation of captured var 'timeStamps' in concurrently-executing code
```

The problem is that we have multiple tasks concurrently accessing the data and risk encountering a data race condition. A data race occurs when multiple tasks attempt to access the same data concurrently, and one or more of these tasks is performing a write operation. This generally results in data corruption problems that can be hard to diagnose.

One option is to create an *actor* in which to store the data. Another solution is to adapt our task group to return the task results sequentially and add them to the dictionary. We originally declared the task group as returning no results by passing Void.self as the return type to the *withTaskGroup()* function as follows:

```
await withTaskGroup(of: Void.self) { group in
.
.
```

The first step is to design the task group so that each task returns a tuple containing the task number (Int) and timestamp (Date) as follows. We also need a dictionary in which to store the results:

```
func doSomething() async {

    var timeStamps: [Int: Date] = [:]
```

```
    await withTaskGroup(of: (Int, Date).self) { group in
        for i in 1...5 {
            group.addTask {
                return(i, await self.takesTooLong())
            }
        }
    }
}
```

Next, we need to declare a second loop to handle the results as they are returned from the group. Because the results are being returned individually from async functions, we cannot simply write a loop to process them all at once. Instead, we need to wait until each result is returned. For this situation, Swift provides the *for-await* loop.

35.18 The for-await Loop

The *for-await* expression allows us to step through sequences of values that are being returned asynchronously and *await* the receipt of values as they are returned by concurrent tasks. The only requirement for using for-await is that the sequential data conforms to the AsyncSequence protocol (which should always be the case when working with task groups).

In our example, we need to add a for-await loop within the task group scope, but after the addTask loop as follows:

```
func doSomething() async {

    var timeStamps: [Int: Date] = [:]

    await withTaskGroup(of: (Int, Date).self) { group in

        for i in 1...5 {
            group.addTask {
                return(i, await self.takesTooLong())
            }
        }

        for await (task, date) in group {
            timeStamps[task] = date
        }
    }
}
```

As each task returns, the for-await loop will receive the resulting tuple and store it in the timeStamps dictionary. To verify this, we can add some code to print the dictionary entries after the task group exits:

```
func doSomething() async {
.
.
        for await (task, date) in group {
            timeStamps[task] = date
        }
    }
```

```
    for (task, date) in timeStamps {
        print("Task = \(task), Date = \(date)")
    }
}
```

When executed, the output from the completed example should be similar to the following:

```
Task = 2, Date = 2023-02-05 18:54:06 +0000
Task = 3, Date = 2023-02-05 18:54:06 +0000
Task = 4, Date = 2023-02-05 18:54:06 +0000
Task = 5, Date = 2023-02-05 18:54:06 +0000
Task = 1, Date = 2023-02-05 18:54:06 +0000
```

35.19 Asynchronous Properties

In addition to async functions, Swift also supports async properties within class and struct types. Asynchronous properties are created by explicitly declaring a getter and marking it as async as demonstrated in the following example. Currently, only read-only properties can be asynchronous.

```
struct MyStruct {
    var myResult: Date {
        get async {
            return await self.getTime()
        }
    }
    func getTime() async -> Date {
        sleep(5)
        return Date()
    }
}
.
.
func doSomething() async {

    let myStruct = MyStruct()

    Task {
        let date = await myStruct.myResult
        print(date)
    }
}
```

35.20 Summary

Modern CPUs and operating systems are designed to execute code concurrently, allowing multiple tasks to be performed simultaneously. This is achieved by running tasks on different *threads*, with the *main thread* primarily responsible for rendering the user interface and responding to user events. By default, most code in an app is also executed on the main thread unless specifically configured to run on a different thread. If that code performs tasks that occupy the main thread for too long, the app will appear to freeze until the task completes. To avoid this, Swift provides the structured concurrency API. When using structured concurrency, code that would block

the main thread is instead placed in an asynchronous function (async properties are also supported) so that it is performed on a separate thread. The calling code can be configured to wait for the async code to complete before continuing using the *await* keyword or to continue executing until the result is needed using *async-let*.

Modern CPUs and operating systems are designed to execute code concurrently allowing multiple tasks to be performed at the same time. This is achieved by running tasks on different *threads* with the *main thread* being primarily responsible for rendering the user interface and responding to user events. By default, most code in an app is also executed on the main thread unless specifically configured to run on a different thread. If that code performs tasks that occupy the main thread for too long the app will appear to freeze until the task completes. To avoid this, Swift provides the structured concurrency API. When using structured concurrency, code that would block the main thread is instead placed in an asynchronous function (async properties are also supported) so that it is performed on a separate thread. The calling code can be configured to wait for the async code to complete before continuing using the *await* keyword, or to continue executing until the result is needed using *async-let*.

Tasks can be run individually or as groups of multiple tasks. The for-await loop provides a useful way to asynchronously process the results of asynchronous task groups.

Chapter 41

41. Using iCloud Storage in an iOS 16 App

The two preceding chapters of this book were intended to convey the knowledge necessary to begin implementing iCloud-based document storage in iOS apps. Having outlined the steps necessary to enable iCloud access in the chapter entitled *"Preparing an iOS 16 App to use iCloud Storage"* and provided an overview of the UIDocument class in *"Managing Files using the iOS 16 UIDocument Class"*, the next step is to begin to store documents using the iCloud service.

Within this chapter, the *iCloudStore* app created in the previous chapter will be re-purposed to store a document using iCloud storage instead of the local device-based file system. The assumption is also made that the project has been enabled for iCloud document storage following the steps outlined in *"Preparing an iOS 16 App to use iCloud Storage"*.

Before starting on this project, it is important to note that membership to the Apple Developer Program will be required as outlined in *"Joining the Apple Developer Program"*.

41.1 iCloud Usage Guidelines

Before implementing iCloud storage in an app, a few rules must first be understood. Some of these are mandatory rules, and some are simply recommendations made by Apple:

- Apps must be associated with a provisioning profile enabled for iCloud storage.
- The app projects must include a suitably configured entitlements file for iCloud storage.
- Apps should not make unnecessary use of iCloud storage. Once a user's initial free iCloud storage space is consumed by stored data, the user will either need to delete files or purchase more space.
- Apps should, ideally, provide the user with the option to select which documents are to be stored in the cloud and which are to be stored locally.
- When opening a *previously created* iCloud-based document, the app should never use an absolute path to the document. The app should instead search for the document by name in the app's iCloud storage area and then access it using the result of the search.
- Documents stored using iCloud should be placed in the app's *Documents* directory. This gives the user the ability to delete individual documents from the storage. Documents saved outside the *Documents* folder can only be deleted in bulk.

41.2 Preparing the iCloudStore App for iCloud Access

Much of the work performed in creating the local storage version of the *iCloudStore* app in the previous chapter will be reused in this example. The user interface, for example, remains unchanged, and the implementation of the UIDocument subclass will not need to be modified. The only methods that need to be rewritten are the *saveDocument* and *viewDidLoad* methods of the view controller.

Load the *iCloudStore* project into Xcode and select the *ViewController.swift* file. Locate the *saveDocument*

method and remove the current code from within the method so that it reads as follows:

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

Next, locate the *loadFile* method and modify it accordingly to match the following fragment:

```
func loadFile() {
}
```

41.3 Enabling iCloud Capabilities and Services

Before writing any code, we need to add the iCloud capability to our project, enable the iCloud Documents service, and create an iCloud container.

Begin by selecting the iCloudStore target located at the top of the Project Navigator panel (marked A in Figure 41-1) so that the main panel displays the project settings. From within this panel, select the *Signing & Capabilities* tab (B) followed by the CoreDataDemo target entry (C):

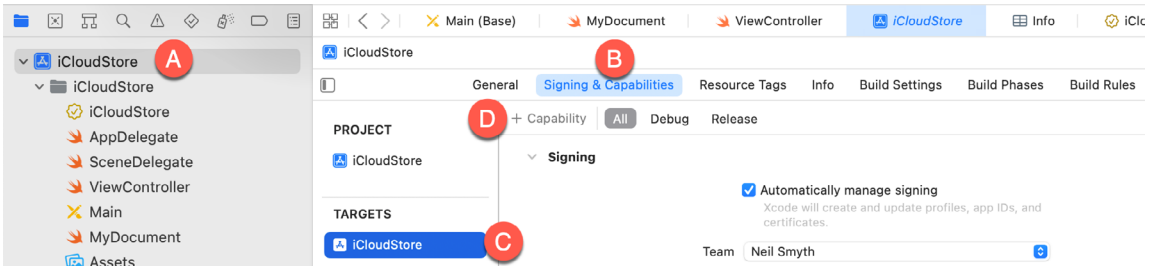

Figure 41-1

Click on the "+ Capability" button (D) to display the dialog shown in Figure 41-2. Enter iCloud into the filter bar, select the result and press the keyboard enter key to add the capability to the project:

Figure 41-2

If iCloud is not listed as an option, you will need to pay to join the Apple Developer program as outlined in the chapter entitled *"Joining the Apple Developer Program"*. If you are already a member, use the steps outlined in the chapter entitled *"Installing Xcode 14 and the iOS 16 SDK"* to ensure you have created a *Developer ID Application* certificate.

Within the iCloud entitlement settings, make sure that the iCloud Documents service is enabled before clicking on the "+" button indicated by the arrow in Figure 41-3 below to add an iCloud container for the project:

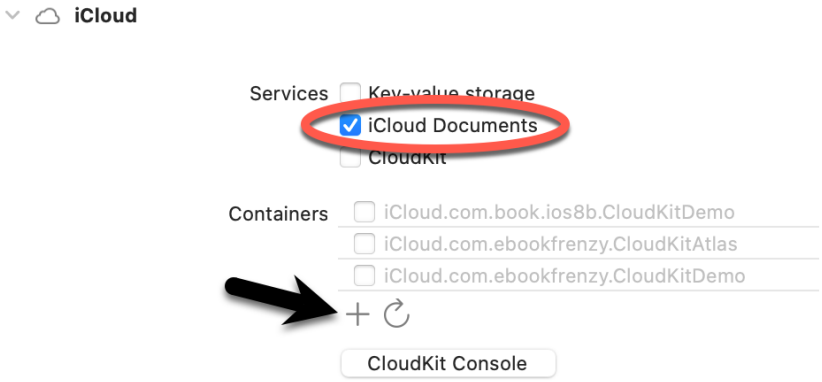

Figure 41-3

After clicking the "+" button, the dialog shown in Figure 41-4 will appear containing a text field into which you need to enter the container identifier. This entry should uniquely identify the container within the CloudKit ecosystem, generally includes your organization identifier (as defined when the project was created), and should be set to something similar to *iCloud.com.yourcompany.iCloudStore*.

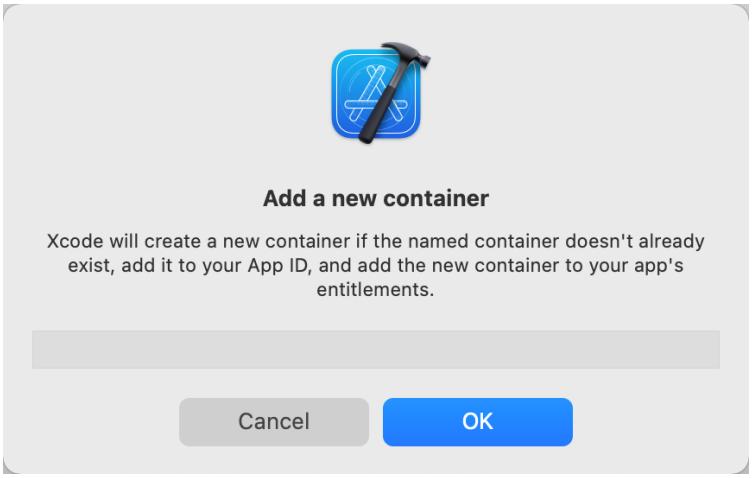

Figure 41-4

Once you have entered the container name, click the OK button to add it to the app entitlements. Returning to the *Signing & Capabilities* screen, make sure that the new container is selected:

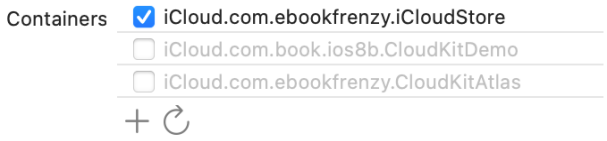

Figure 41-5

41.4 Configuring the View Controller

Before writing any code, several variables need to be defined within the view controller's *ViewController.swift* file in addition to those implemented in the previous chapter.

Creating a URL to the document location in the iCloud storage will also be necessary. When a document is stored on iCloud, it is said to be *ubiquitous* since the document is accessible to the app regardless of the device on which it is running. Therefore, the object used to store this URL will be named *ubiquityURL*.

As previously stated, when opening a stored document, an app should search for it rather than directly access it using a stored path. An iCloud document search is performed using an *NSMetaDataQuery* object which needs to be declared in the view controller class, in this instance, using the name *metaDataQuery*. Note that declaring the object locally to the method in which it is used will result in the object being released by the automatic reference counting system (ARC) before it has completed the search.

To implement these requirements, select the *ViewController.swift* file in the Xcode project navigator panel and modify the file as follows:

```
import UIKit

class ViewController: UIViewController {

    @IBOutlet weak var textView: UITextView!

    var document: MyDocument?
    var documentURL: URL?
    var ubiquityURL: URL?
    var metaDataQuery: NSMetadataQuery?
.
.
.
}
```

41.5 Implementing the loadFile Method

The purpose of the code in the view controller *loadFile* method is to identify the URL for the ubiquitous file version to be stored using iCloud (assigned to *ubiquityURL*). The ubiquitous URL is constructed by calling the *url(forUbiquityContainerIdentifier:)* method of the FileManager passing through *nil* as an argument to default to the first container listed in the entitlements file.

```
ubiquityURL = filemgr.url(forUbiquityContainerIdentifier: nil)
```

The app will only be able to obtain the ubiquityURL if the user has configured a valid Apple ID within the iCloud page of the iOS Settings app. Therefore, some defensive code must be added to notify the user and return from the method if a valid ubiquityURL cannot be obtained. For testing in this example, we will output a message to the console before returning:

```
guard ubiquityURL != nil else {
    print("Unable to access iCloud Account")
    print("Open the Settings app and enter your Apple ID into iCloud settings")
    return
}
```

Since it is recommended that documents be stored in the *Documents* sub-directory, this needs to be appended

to the URL path along with the file name:

```
ubiquityURL =
       ubiquityURL?.appendingPathComponent("Documents/savefile.txt")
```

The final task for the *loadFile* method is to initiate a search in the app's iCloud storage area to find out if the *savefile.txt* file already exists and to act accordingly, subject to the result of the search. The search is performed by calling the methods on an instance of the *NSMetaDataQuery* object. This involves creating the object, setting a predicate to indicate the files to search for, and defining a ubiquitous search scope (in other words instructing the object to search within the Documents directory of the app's iCloud storage area). Once initiated, the search is performed on a separate thread and issues a notification when completed. For this reason, it is also necessary to configure an observer to be notified when the search is finished. The code to perform these tasks reads as follows:

```
metaDataQuery = NSMetadataQuery()

metaDataQuery?.predicate =
    NSPredicate(format: "%K like 'savefile.txt'",
        NSMetadataItemFSNameKey)
metaDataQuery?.searchScopes = [NSMetadataQueryUbiquitousDocumentsScope]

NotificationCenter.default.addObserver(self,
    selector: #selector(
           ViewController.metadataQueryDidFinishGathering),
    name: NSNotification.Name.NSMetadataQueryDidFinishGathering,
    object: metaDataQuery!)

metaDataQuery?.start()
```

Once the *start* method is called, the search will run and call the *metadataQueryDidFinishGathering* method when the search is complete. The next step, therefore, is to implement the *metadataQueryDidFinishGathering* method. Before doing so, however, note that the *loadFile* method is now complete, and the full implementation should read as follows:

```
func loadFile() {

    let filemgr = FileManager.default

    ubiquityURL = filemgr.url(forUbiquityContainerIdentifier: nil)

    guard ubiquityURL != nil else {
        print("Unable to access iCloud Account")
        print("Open the Settings app and enter your Apple ID into iCloud
settings")
        return
    }

    ubiquityURL = ubiquityURL?.appendingPathComponent(
                                    "Documents/savefile.txt")
```

```
    metaDataQuery = NSMetadataQuery()

    metaDataQuery?.predicate =
        NSPredicate(format: "%K like 'savefile.txt'",
            NSMetadataItemFSNameKey)
    metaDataQuery?.searchScopes =
              [NSMetadataQueryUbiquitousDocumentsScope]

    NotificationCenter.default.addObserver(self,
        selector: #selector(
               ViewController.metadataQueryDidFinishGathering),
        name: NSNotification.Name.NSMetadataQueryDidFinishGathering,
        object: metaDataQuery!)

    metaDataQuery?.start()
}
```

41.6 Implementing the metadataQueryDidFinishGathering Method

When the metadata query was triggered in the *loadFile* method to search for documents in the Documents directory of the app's iCloud storage area, an observer was configured to call a method named *metadataQueryDidFinishGathering* when the initial search was completed. The next logical step is to implement this method. The first task of the method is to identify the query object that caused this method to be called. This object must then disable any further query updates (at this stage, the document either exists or doesn't exist, so there is nothing to be gained by receiving additional updates) and stop the search. Finally, removing the observer that triggered the method call is also necessary. When combined, these requirements result in the following code:

```
let query: NSMetadataQuery = notification.object as! NSMetadataQuery

query.disableUpdates()

NotificationCenter.default.removeObserver(self,
        name: NSNotification.Name.NSMetadataQueryDidFinishGathering,
        object: query)

query.stop()
```

The next step is to make sure at least one match was found and to extract the URL of the first document located during the search:

```
if query.resultCount == 1 {
       let resultURL = query.value(ofAttribute: NSMetadataItemURLKey,
                    forResultAt: 0) as! URL
```

In all likelihood, a more complex app would need to implement a *for* loop to iterate through more than one document in the array. Given that the iCloudStore app searched for only one specific file name, we can check the array element count and assume that if the count is one, then the document already exists. In this case, the ubiquitous URL of the document from the query object needs to be assigned to our *ubiquityURL* member property and used to create an instance of our MyDocument class called *document*. The *document object's open(completionHandler:) method* is then called to open the document in the cloud and read the contents. This

will trigger a call to the *load(fromContents:)* method of the *document* object, which, in turn, will assign the contents of the document to the *userText* property. Assuming the document read is successful, the value of *userText* needs to be assigned to the *text* property of the text view object to make it visible to the user. Bringing this together results in the following code fragment:

```
    document = MyDocument(fileURL: resultURL as URL)

    document?.open(completionHandler: {(success: Bool) -> Void in
        if success {
            print("iCloud file open OK")
            self.textView.text = self.document?.userText
            self.ubiquityURL = resultURL as URL
        } else {
            print("iCloud file open failed")
        }
    })
} else {
}
```

Suppose the document does not yet exist in iCloud storage. In that case, the code needs to create the document using the *save(to:)* method of the *document* object passing through the value of *ubiquityURL* as the destination path on iCloud:

```
.
.
} else {
    if let url = ubiquityURL {
        document = MyDocument(fileURL: url)

        document?.save(to: url,
                       for: .forCreating,
                       completionHandler: {(success: Bool) -> Void in
                        if success {
                            print("iCloud create OK")
                        } else {
                            print("iCloud create failed")
                        }
        })
    }
}
```

The individual code fragments outlined above combine to implement the following *metadataQueryDidFinishGathering* method, which should be added to the *ViewController.swift* file:

```
@objc func metadataQueryDidFinishGathering(notification: NSNotification)
     -> Void
{
    let query: NSMetadataQuery = notification.object as! NSMetadataQuery

    query.disableUpdates()
```

```
    NotificationCenter.default.removeObserver(self,
            name: NSNotification.Name.NSMetadataQueryDidFinishGathering,
            object: query)

    query.stop()

    if query.resultCount == 1 {
        let resultURL = query.value(ofAttribute: NSMetadataItemURLKey,
                                    forResultAt: 0) as! URL

        document = MyDocument(fileURL: resultURL as URL)

        document?.open(completionHandler: {(success: Bool) -> Void in
            if success {
                print("iCloud file open OK")
                self.textView.text = self.document?.userText
                self.ubiquityURL = resultURL as URL
            } else {
                print("iCloud file open failed")
            }
        })
    } else {
        if let url = ubiquityURL {
            document = MyDocument(fileURL: url)

            document?.save(to: url,
                           for: .forCreating,
                           completionHandler: {(success: Bool) -> Void in
                            if success {
                                print("iCloud create OK")
                            } else {
                                print("iCloud create failed")
                            }
            })
        }
    }
}
```

41.7 Implementing the saveDocument Method

The final task before building and running the app is implementing the *saveDocument* method. This method needs to update the *userText* property of the *document* object with the text entered into the text view and then call the *saveToURL* method of the *document* object, passing through the *ubiquityURL* as the destination URL using the *.forOverwriting* option:

```
@IBAction func saveDocument(_ sender: Any) {
    document?.userText = textView.text
```

```
    if let url = ubiquityURL {
        document?.save(to: url,
                for: .forOverwriting,
                completionHandler: {(success: Bool) -> Void in
                 if success {
                     print("Save overwrite OK")
                 } else {
                     print("Save overwrite failed")
                 }
        })
    }
}
```

All that remains now is to build and run the iCloudStore app on an iOS device, but first, some settings need to be checked.

41.8 Enabling iCloud Document and Data Storage

When testing iCloud on an iOS Simulator session, it is important to ensure that the simulator is configured with a valid Apple ID within the Settings app. Launch the simulator, load the Settings app, and click on the iCloud option to configure this. If no account information is configured on this page, enter a valid Apple ID and corresponding password before proceeding with the testing.

Whether or not apps are permitted to use iCloud storage on an iOS device or Simulator is controlled by the iCloud settings. To review these settings, open the Settings app on the device or simulator, select your account at the top of the settings list and, on the resulting screen, select the *iCloud* category. Scroll down the list of various iCloud-related options and verify that the *iCloud Drive* option is set to *On*:

Figure 41-6

41.9 Running the iCloud App

Once you have logged in to an iCloud account on the device or simulator, test the iCloudStore app by clicking the run button. Once running, edit the text in the text view and touch the *Save* button. Next, in the Xcode toolbar,

click on the stop button to exit the app, followed by the run button to re-launch the app. On the second launch, the previously entered text will be read from the document in the cloud and displayed in the text view object.

41.10 Making a Local File Ubiquitous

In addition to writing a file directly to iCloud storage, as illustrated in this example app, it is also possible to transfer a pre-existing local file to iCloud storage, making it ubiquitous. This can be achieved using the *setUbiquitous* method of the FileManager class. For example, assuming that *documentURL* references the path to the local copy of the file and *ubiquityURL* the iCloud destination, a local file can be made ubiquitous using the following code:

```
do {
    try filemgr.setUbiquitous(true, itemAt: documentUrl,
                destinationURL: ubiquityURL)
} catch let error {
    print("setUbiquitous failed: \(error.localizedDescription)")
}
```

41.11 Summary

The objective of this chapter was to work through the process of developing an app that stores a document using the iCloud service. Both techniques of directly creating a file in iCloud storage and making an existing locally created file ubiquitous were covered. In addition, some important guidelines that should be observed when using iCloud were outlined.

Chapter 84

84. Using Create ML to Build an Image Classification Model

This chapter will demonstrate the use of Create ML to build an image classification model trained to classify images based on whether the image contains one of four specific items (an apple, banana, cat, or a mixture of fruits).

The tutorial will include the training and testing of an image recognition machine learning model using an Xcode playground. In the next chapter, this model will be integrated into an app to perform image classifications when a user takes a photo or selects an image from the photo library.

84.1 About the Dataset

As explained in the previous chapter, an image classification model is trained by providing it with a range of images that have already been categorized. Once the training process has been performed, the model is then tested using a set of images that were not previously used in the training process. Once the testing achieves a high enough level of validation, the model is ready to be integrated into an app project.

For this example, a dataset containing images of apples and bananas will be used for training. The dataset is contained within a folder named *CreateML_dataset,* which is included with the source code download available at the following URL:

https://www.ebookfrenzy.com/retail/ios16/

The dataset includes a *Training* folder containing images divided into two subfolders organized as shown in Figure 84-1:

Figure 84-1

The dataset also includes a *Testing* folder containing images for each classification, none of which were used during the training session. Once the dataset has been downloaded, take some time to browse the folders and images to understand the data's structure.

84.2 Creating the Machine Learning Model

With the dataset prepared, the next step is to create the model using the Create ML tool. Begin by launching Xcode, then select the *Xcode -> Open Developer Tool -> Create ML* menu option. When Create ML has loaded, it will display a Finder window where you can choose an existing model or create a new one. Within this window, click on the New Document button to display the template selection screen shown in Figure 84-2:

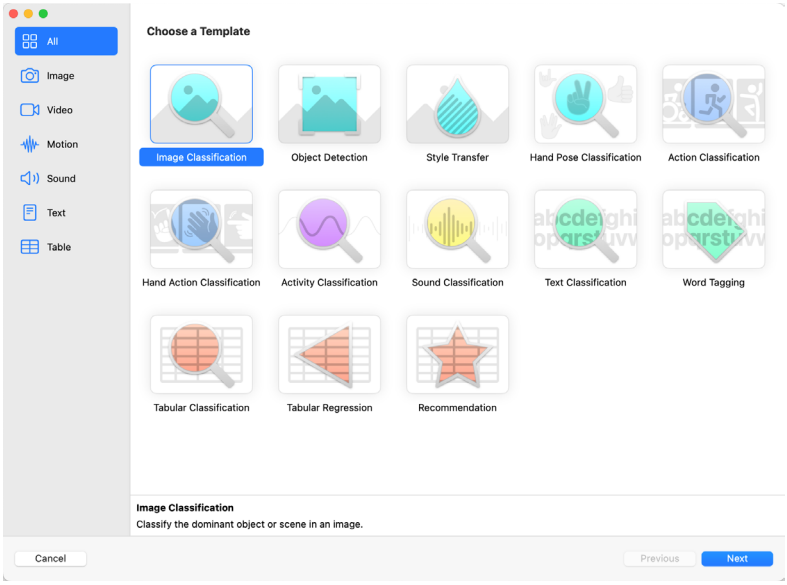

Figure 84-2

For this example, we will work with images, so select the *Image Classification* template followed by the Next button. Then, continue through the remaining screens, naming the project MyImageClassifier and selecting a suitable folder into which to create the model.

Once the model has been created, the screen shown in Figure 84-3 will appear ready for the training and tested data to be imported:

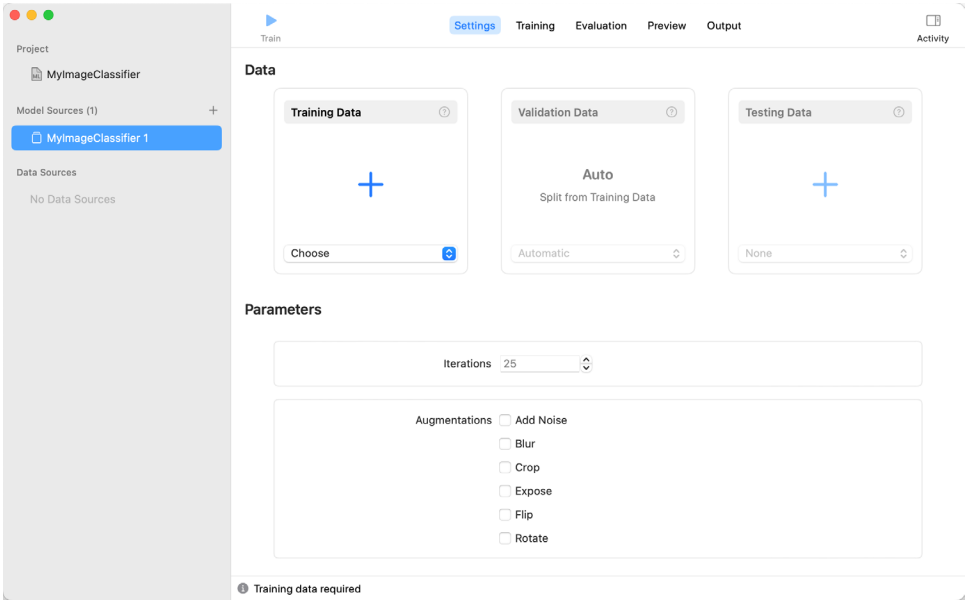

Figure 84-3

In the left-hand panel, click on the “MyImageClassifier 1” entry listed under Model Sources and change the name to MyImageClassifier.

84.3 Importing the Training and Testing Data

Click on the box labeled Training Data to display the finder dialog and navigate to and open the Training data set folder. Next, repeat these steps to import the test folder into the Testing Data box. Once the data has been imported, the Data section of the Create ML screen should resemble Figure 84-4:

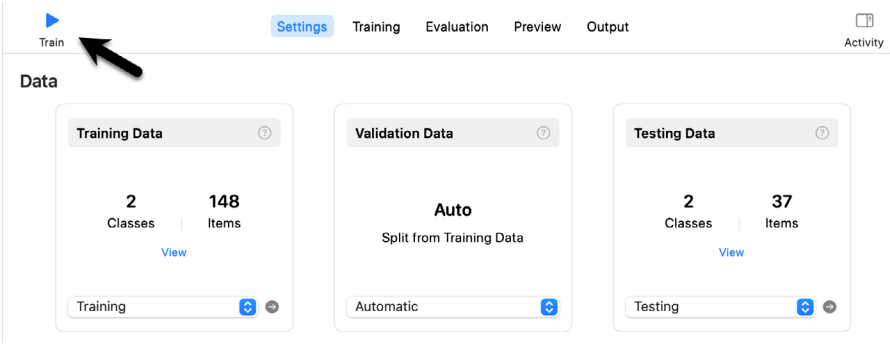

Figure 84-4

Next, increase the number of training iterations to 45 and set some augmentations to train the model to deal with image variations:

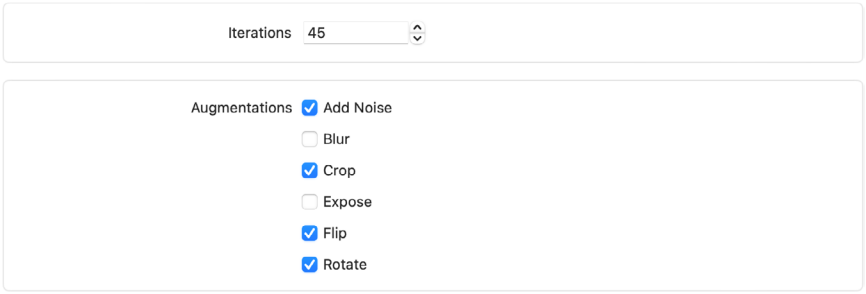

Figure 84-5

84.4 Training and Testing the Model

Now that we have loaded the data, we are ready to start training and testing the model by clicking on the Train button indicated by the arrow in Figure 84-4 above. Once the process is complete, select the Training tab to display a graph showing how accuracy improved with each training iteration:

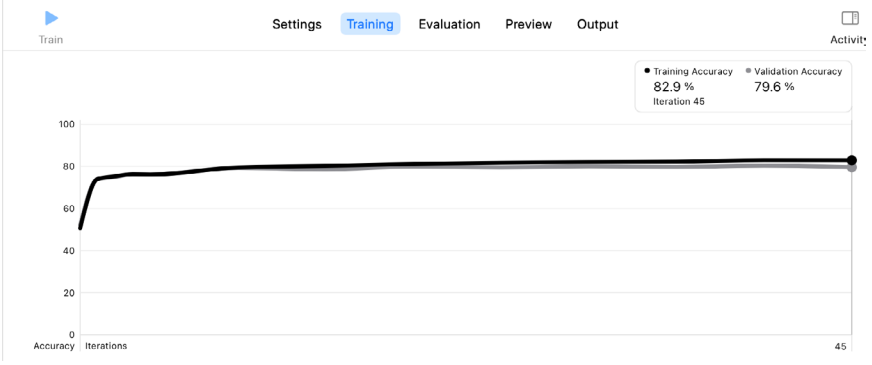

Figure 84-6

Next, switch to the Evaluation screen to review a more detailed breakdown of the training:

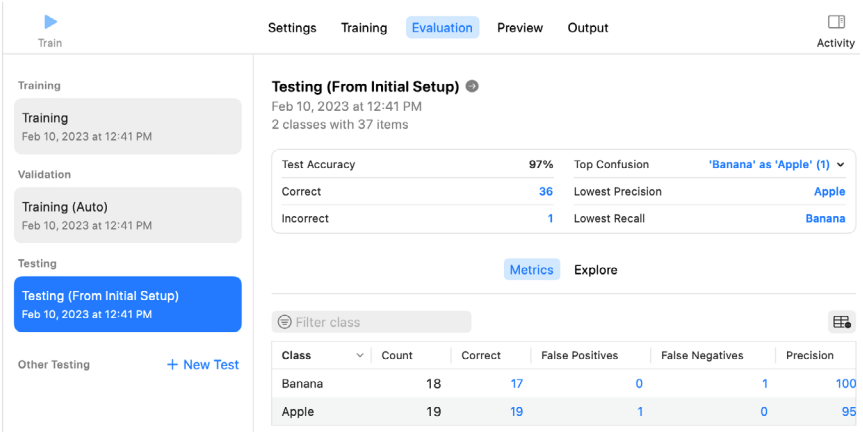

Figure 84-7

Finally, switch to the Preview screen and drag and drop images for each category that were not part of the training or testing data and see which are successfully classified by the model:

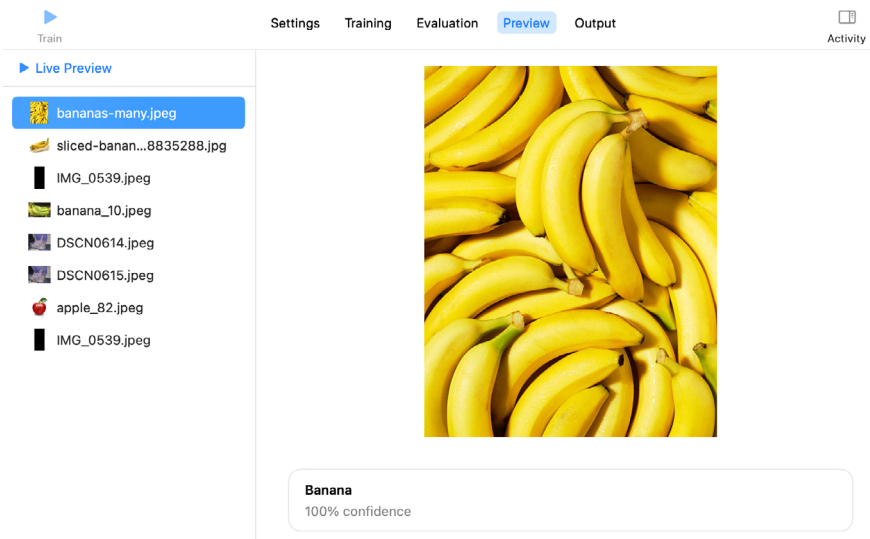

Figure 84-8

We now need to save the model in preparation for loading it into an Xcode project. To save the model, display the Output screen, click on the Get button highlighted in Figure 84-9, and select a name and location for the model file:

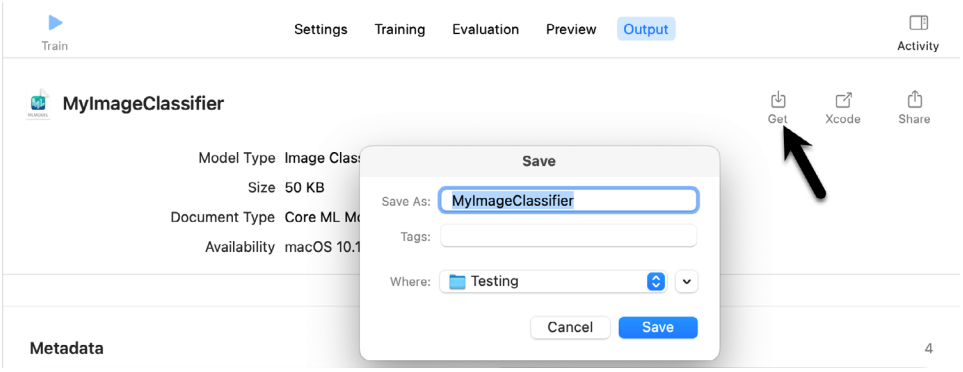

Figure 84-9

Before exiting from Create ML tool, save the project using the *File -> Save...* menu option.

84.5 Summary

A key component of any machine learning implementation is a well-trained model, and the accuracy of that model is dependent on the dataset used during the model training. Once comprehensive training data has been gathered, Create ML makes creating and testing machine learning models easy. This involves the use of the Create ML classifier builder. This chapter demonstrated the use of the Image classifier builder to create and save a model designed to identify several different object types within images. This model will be used in the next chapter to perform image identification within an existing iOS app.

Chapter 92

92. An Introduction to iOS 16 Sprite Kit Programming

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

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

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

92.1 What is Sprite Kit?

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

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

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

92.2 The Key Components of a Sprite Kit Game

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

92.2.1 Sprite Kit View

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

92.2.2 Scenes

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

92.2.3 Nodes

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

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

92.2.4 Physics Bodies

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

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

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

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

developer.

92.2.5 Physics World

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

92.2.6 Actions

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

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

92.2.7 Transitions

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

92.2.8 Texture Atlas

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

92.2.9 Constraints

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

92.3 An Example Sprite Kit Game Hierarchy

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

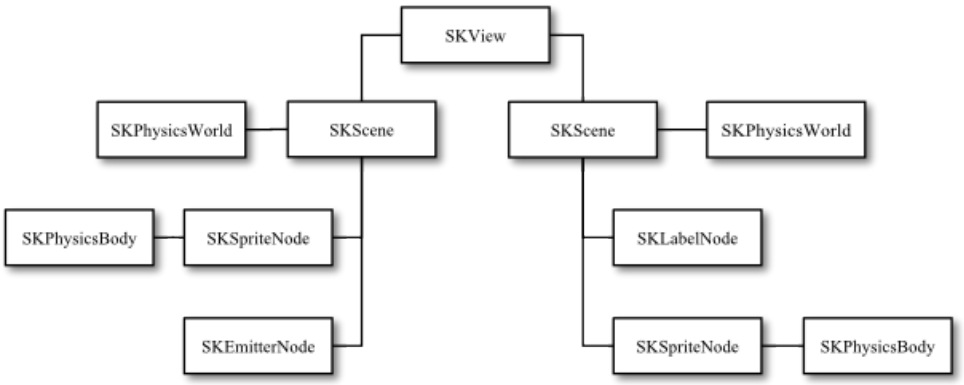

Figure 92-1

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

92.4 The Sprite Kit Game Rendering Loop

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

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

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

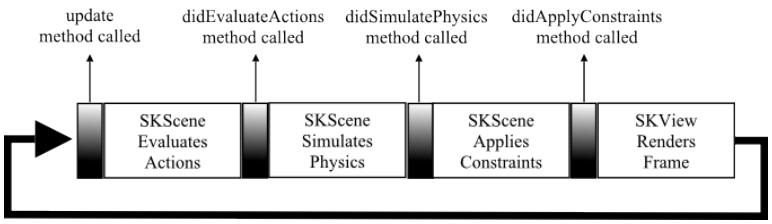

Figure 92-2

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

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

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

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

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

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

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

92.5 The Sprite Kit Level Editor

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

92.6 Summary

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

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

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

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

Chapter 93

93. An iOS 16 Sprite Kit Level Editor Game Tutorial

In this chapter of iOS 16 App Development Essentials, many of the Sprite Kit Framework features outlined in the previous chapter will be used to create a game-based app. In particular, this tutorial will demonstrate the practical use of scenes, textures, sprites, labels, and actions. In addition, the app created in this chapter will also use physics bodies to demonstrate the use of collisions and simulated gravity.

This tutorial will also demonstrate using the Xcode Sprite Kit Level, Live, and Action editors combined with Swift code to create a Sprite Kit-based game.

93.1 About the Sprite Kit Demo Game

The game created in this chapter consists of a single animated character that shoots arrows across the scene when the screen is tapped. For the game's duration, balls fall from the top of the screen, with the objective being to hit as many balls as possible with the arrows.

The completed game will comprise the following two scenes:

- **GameScene** – The scene which appears when the game is first launched. The scene will announce the game's name and invite the user to touch the screen to begin the game. The game will then transition to the second scene.
- **ArcheryScene** – The scene where the game-play takes place. Within this scene, the archer and ball sprites are animated, and the physics behavior and collision detection are implemented to make the game work.

In terms of sprite nodes, the game will include the following:

- **Welcome Node** – An SKLabelNode instance that displays a message to the user on the Welcome Scene.
- **Archer Node** – An SKSpriteNode instance to represent the archer game character. The animation frames that cause the archer to load and launch an arrow are provided via a sequence of image files contained within a texture atlas.
- **Arrow Node** – An SKSpriteNode instance used to represent the arrows as the archer character shoots them. This node has associated with it a physics body so that collisions can be detected and to make sure it responds to gravity.
- **Ball Node** – An SKSpriteNode represents the balls that fall from the sky. The ball has associated with it a physics body for gravity and collision detection purposes.
- **Game Over Node** – An SKLabelNode instance that displays the score to the user at the end of the game.

The overall architecture of the game can be represented hierarchically, as outlined in Figure 93-1:

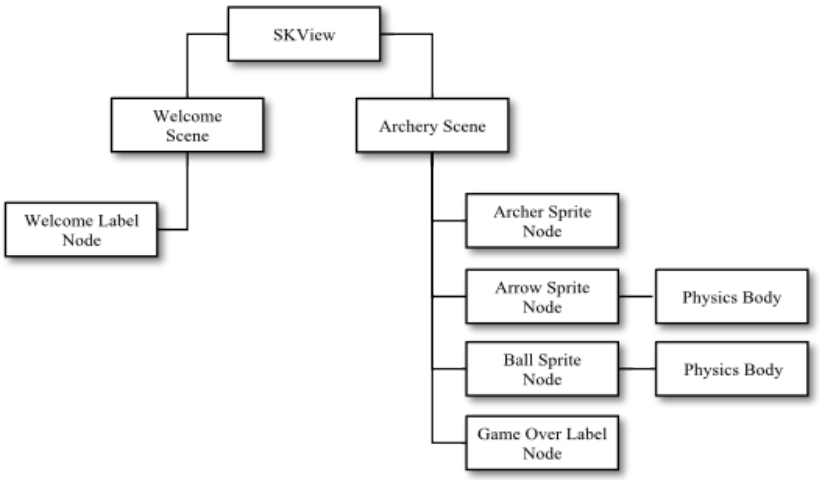

Figure 93-1

In addition to the nodes outlined above, the Xcode Live and Action editors will be used to implement animation and audio actions, which will be triggered from within the app's code.

93.2 Creating the SpriteKitDemo Project

To create the project, launch Xcode and select the *Create a new Xcode project* option from the welcome screen (or use the *File -> New -> Project…*) menu option. Next, on the template selection panel, choose the iOS *Game* template option. Click on the *Next* button to proceed and on the resulting options screen, name the product *SpriteKitDemo* and choose *Swift* as the language in which the app will be developed. Finally, set the Game Technology menu to *SpriteKit*. Click *Next* and choose a suitable location for the project files. Once selected, click *Create* to create the project.

93.3 Reviewing the SpriteKit Game Template Project

The selection of the SpriteKit Game template has caused Xcode to create a template project with a demonstration incorporating some pre-built Sprite Kit behavior. This template consists of a View Controller class (*GameViewController.swift*), an Xcode Sprite Kit scene file (*GameScene.sks*), and a corresponding GameScene class file (*GameScene.swift*). The code within the *GameViewController.swift* file loads the scene design contained within the *GameScene.sks* file and presents it on the view to be visible to the user. This, in turn, triggers a call to the *didMove(to view:)* method of the GameScene class as implemented in the *GameScene.swift* file. This method creates an SKLabelNode displaying text that reads "Hello, World!".

The GameScene class also includes a variety of touch method implementations that create SKShapeNode instances into which graphics are drawn when triggered. These nodes, in turn, are displayed in response to touches and movements on the device screen. To see the template project in action, run it on a physical device or the iOS simulator and perform tapping and swiping motions on the display.

As impressive as this may be, given how little code is involved, this bears no resemblance to the game that will be created in this chapter, so some of this functionality needs to be removed to provide a clean foundation on which to build. Begin the tidying process by selecting and editing the *GameScene.swift* file to remove the code to create and present nodes in the scene. Once modified, the file should read as follows:

```
import SpriteKit
import GameplayKit
```

```
class GameScene: SKScene {

    override func didMove(to view: SKView) {

    }

    override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {

    }

    override func update(_ currentTime: TimeInterval) {
        // Called before each frame is rendered
    }
}
```

With these changes, it is time to start creating the SpriteKitDemo game.

93.4 Restricting Interface Orientation

The game created in this tutorial assumes that the device on which it is running will be in landscape orientation. Therefore, to prevent the user from attempting to play the game with a device in portrait orientation, the *Device Orientation* properties for the project need to be restricted. To achieve this, select the *SpriteKitDemo* entry located at the top of the Project Navigator and, in the resulting *General* settings panel, change the device orientation settings so that only the *Landscape* options are selected both for iPad and iPhone devices:

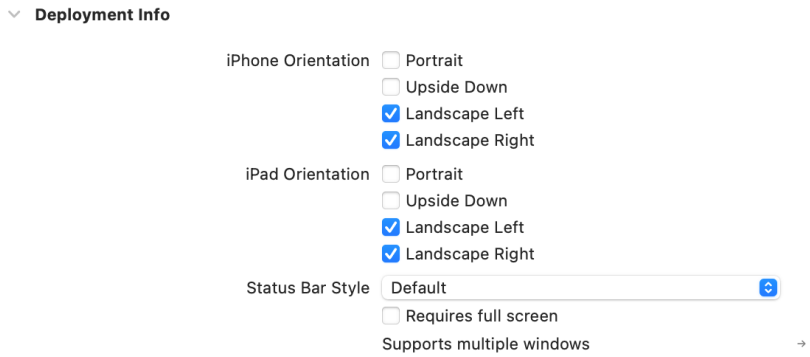

Figure 93-2

93.5 Modifying the GameScene SpriteKit Scene File

As previously outlined, Xcode has provided a SpriteKit scene file (*GameScene.sks*) for a scene named GameScene together with a corresponding class declaration contained within the *GameScene.swift* file. The next task is to repurpose this scene to act as the welcome screen for the game. Begin by selecting the *GameScene.sks* file so that it loads into the SpriteKit Level Editor, as shown in Figure 93-3:

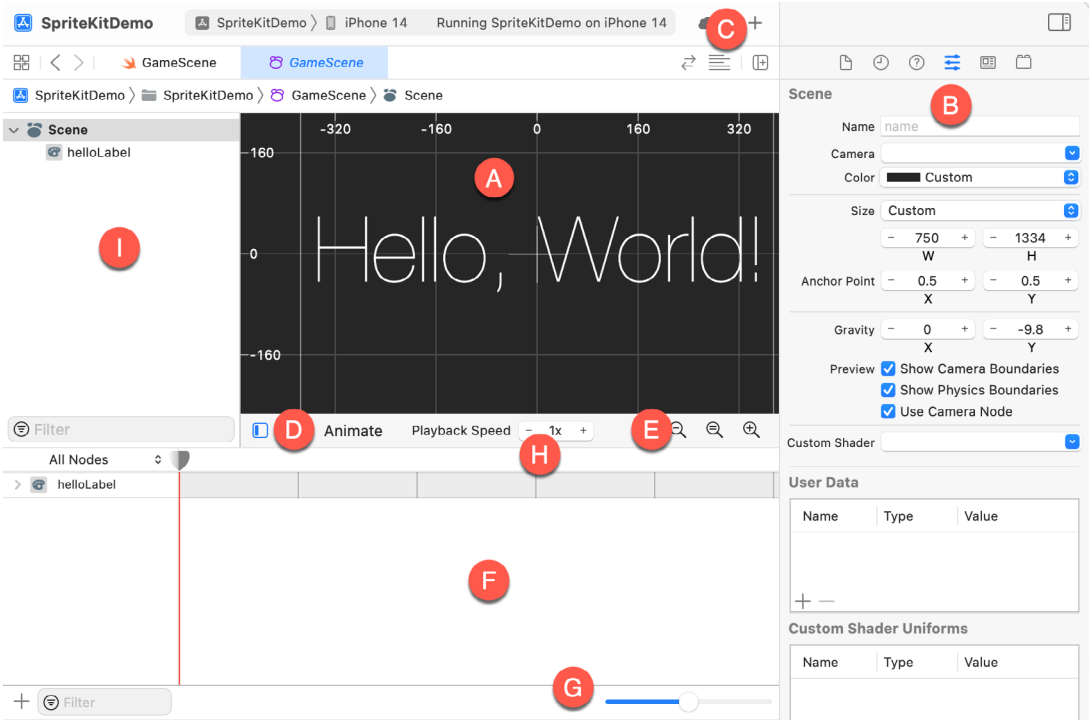

Figure 93-3

When working with the Level Editor to design SpriteKit scenes, there are several key areas of importance, each of which has been labeled in the above figure:

- **A – Scene Canvas** - This is the canvas onto which nodes may be placed, positioned, and configured.
- **B – Attribute Inspector Panel** - This panel provides a range of configuration options for the currently selected item in the editor panel. This allows SKNode and SKAction objects to be customized within the editor environment.
- **C – Library Button** – This button displays the Library panel containing a range of node and effect types that can be dragged and dropped onto the scene.
- **D – Animate/Layout Button** - Toggles between the editor's simulation and layout editing modes. Simulate mode provides a useful mechanism for previewing the scene behavior without compiling and running the app.
- **E – Zoom Buttons** – Buttons to zoom in and out of the scene canvas.
- **F – Live Editor** – The live editor allows actions and animations to be placed within a timeline and simulated within the editor environment. It is possible, for example, to add animation and movement actions within the live editor and play them back live within the scene canvas.
- **G – Timeline View Slider** – Pans back and forth through the view of the live editor timeline.
- **H – Playback Speed** – When in Animation mode, this control adjusts the playback speed of the animations and actions contained within the live editor panel.
- **I – Scene Graph View** – This panel provides an overview of the scene's hierarchy and can be used to select,

delete, duplicate and reposition scene elements within the hierarchy.

Within the scene editor, click on the "Hello, World!" Label node and press the keyboard delete key to remove it from the scene. With the scene selected in the scene canvas, click on the *Color* swatch in the Attribute Inspector panel and use the color selection dialog to change the scene color to a shade of green. Remaining within the Attributes Inspector panel, change the Size setting from *Custom* to *iPad 9.7"* in *Landscape* mode:

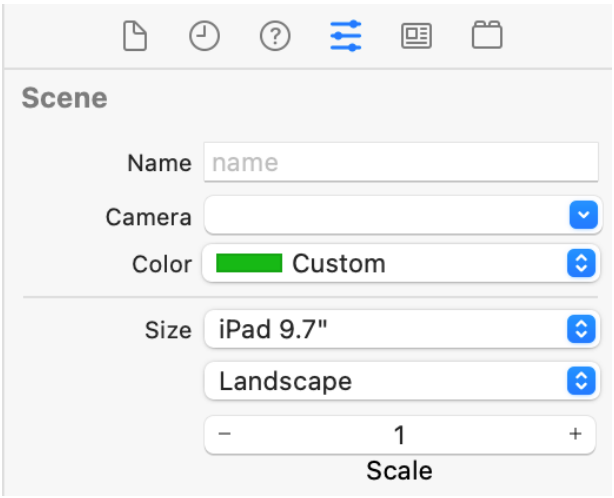

Figure 93-4

Click on the button (marked C in Figure 93-3 above) to display the Library panel, locate the Label node object, and drag and drop an instance onto the center of the scene canvas. With the label still selected, change the *Text* property in the inspector panel to read "SpriteKitDemo – Tap Screen to Play". Remaining within the inspector panel, click on the T next to the font name and use the font selector to assign a 56-point *Marker Felt Wide* font to the label from the *Fun* font category. Finally, set the *Name* property for the label node to "welcomeNode". Save the scene file before proceeding.

With these changes complete, the scene should resemble that of Figure 93-5:

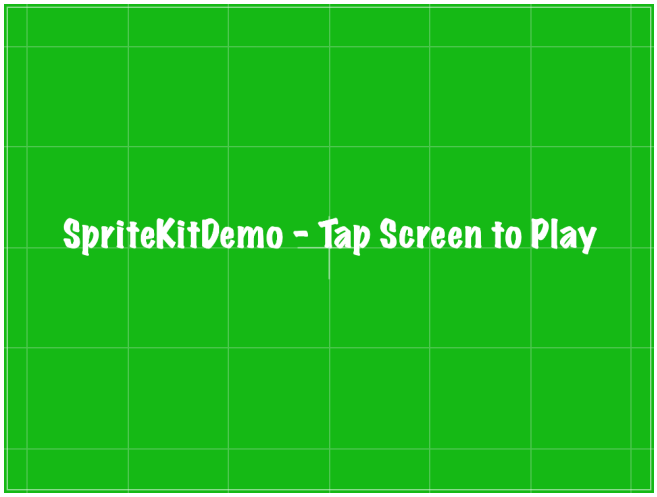

Figure 93-5

93.6 Creating the Archery Scene

As previously outlined, the game's first scene is a welcome screen on which the user will tap to begin playing within a second scene. Add a new class to the project to represent this second scene by selecting the *File -> New -> File…* menu option. In the file template panel, make sure that the *Cocoa Touch Class* template is selected in the main panel. Click on the *Next* button and configure the new class to be a subclass of *SKScene* named *ArcheryScene*. Click on the *Next* button and create the new class file within the project folder.

The new scene class will also require a corresponding SpriteKit scene file. Select *File -> New -> File…* once again, this time selecting *SpriteKit Scene* from the Resource section of the main panel (Figure 93-6). Click *Next*, name the scene *ArcheryScene* and click the *Create* button to add the scene file to the project.

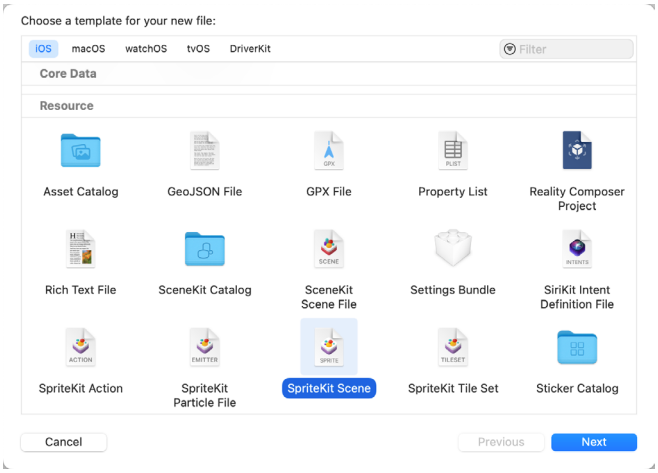

Figure 93-6

Edit the newly added *ArcheryScene.swift* file and modify it to import the SpriteKit Framework as follows:

```
import UIKit
import SpriteKit

class ArcheryScene: SKScene {

}
```

93.7 Transitioning to the Archery Scene

Clearly, having instructed the user to tap the screen to play the game, some code needs to be written to make this happen. This behavior will be added by implementing the *touchesBegan* method in the GameScene class. Rather than move directly to ArcheryScene, some effects will be added as an action and transition.

When implemented, the SKAction will cause the node to fade from view, while an SKTransition instance will be used to animate the transition from the current scene to the archery scene using a "doorway" style of animation. Implement these requirements by adding the following code to the *touchesBegan* method in the *GameScene.swift* file:

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {
    if let welcomeNode = childNode(withName: "welcomeNode") {
        let fadeAway = SKAction.fadeOut(withDuration: 1.0)
```

```
        welcomeNode.run(fadeAway, completion: {
            let doors = SKTransition.doorway(withDuration: 1.0)
            if let archeryScene = ArcheryScene(fileNamed: "ArcheryScene") {
                self.view?.presentScene(archeryScene, transition: doors)
            }
        })
    }
}
```

Before moving on to the next steps, we will take some time to provide more detail on the above code.

From within the context of the *touchesBegan* method, we have no direct reference to the *welcomeNode* instance. However, we know that when it was added to the scene in the SpriteKit Level Editor, it was assigned the name "welcomeNode". Using the *childNode(withName:)* method of the scene instance, therefore, a reference to the node is being obtained within the *touchesBegan* method as follows:

```
if let welcomeNode = childNode(withName: "welcomeNode") {
```

The code then checks that the node was found before creating a new SKAction instance configured to cause the node to fade from view over a one-second duration:

```
let fadeAway = SKAction.fadeOut(withDuration: 1.0)
```

The action is then executed on the welcomeNode. A completion block is also specified to be executed when the action completes. This block creates an instance of the ArcheryScene class preloaded with the scene contained within the *ArcheryScene.sks* file and an appropriately configured SKTransition object. The transition to the new scene is then initiated:

```
let fadeAway = SKAction.fadeOut(withDuration: 1.0)

welcomeNode.run(fadeAway, completion: {
    let doors = SKTransition.doorway(withDuration: 1.0)
    if let archeryScene = ArcheryScene(fileNamed: "ArcheryScene") {
        self.view?.presentScene(archeryScene, transition: doors)
    }
})
```

Compile and run the app on an iPad device or simulator in landscape orientation. Once running, tap the screen and note that the label node fades away and that after the transition to the ArcheryScene takes effect, we are presented with a gray scene that now needs to be implemented.

93.8 Adding the Texture Atlas

Before textures can be used on a sprite node, the texture images must first be added to the project. Textures take the form of image files and may be added individually to the project's asset catalog. However, for larger numbers of texture files, it is more efficient (both for the developer and the app) to create a texture atlas. In the case of the archer sprite, this will require twelve image files to animate an arrow's loading and subsequent shooting. A texture atlas will be used to store these animation frame images. The images for this project can be found in the sample code download, which can be obtained from the following web page:

https://www.ebookfrenzy.com/retail/ios16/

Within the code sample archive, locate the folder named *sprite_images*. Located within this folder is the *archer.atlas* sub-folder, which contains the animation images for the archer sprite node.

To add the atlas to the project, select the *Assets* catalog file in the Project Navigator to display the image assets panel. Locate the *archer.atlas* folder in a Finder window and drag and drop it onto the asset catalog panel so that it appears beneath the existing AppIcon entry, as shown in the following figure:

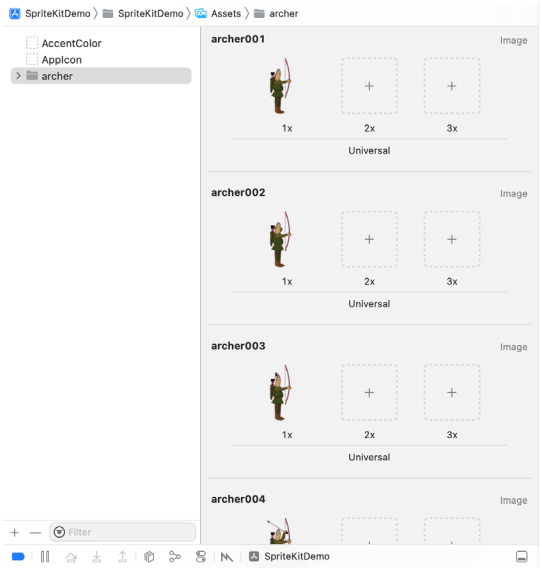

Figure 93-7

93.9 Designing the Archery Scene

The layout for the archery scene is contained within the *ArcheryScene.sks* file. Select this file so that it loads into the Level Editor environment. With the scene selected in the canvas, use the Attributes Inspector panel to change the color property to white and the Size property to landscape *iPad 9.7*".

From within the SpriteKit Level Editor, the next task is to add the sprite node representing the archer to the scene. Display the Library panel, select the Media Library tab as highlighted in Figure 93-8 below, and locate the *archer001.png* texture image file:

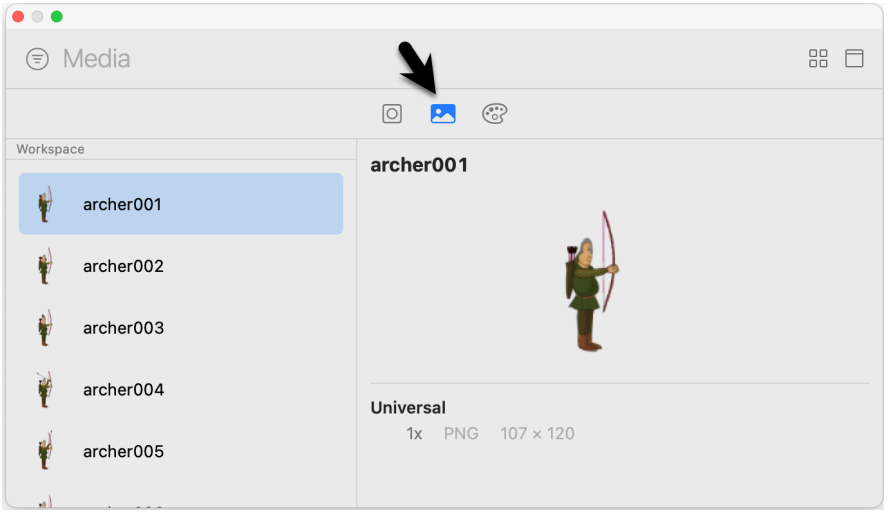

Figure 93-8

Once located, change the Size property in the Attributes Inspector to iPad 9.7", then drag and drop the texture onto the canvas and position it so that it is located in the vertical center of the scene at the left-hand edge, as shown in the following figure:

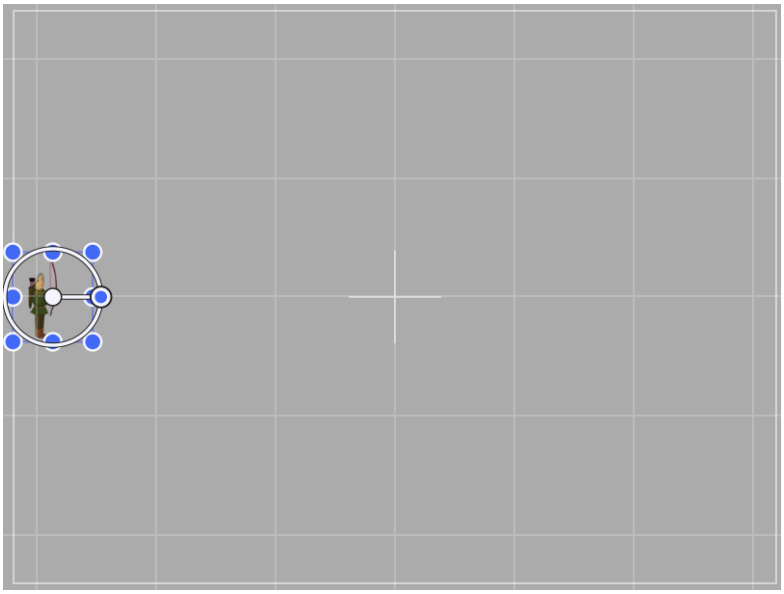

Figure 93-9

With the archer node selected, use the Attributes Inspector panel to assign the name "archerNode" to the sprite. The next task is to define the physical outline of the archer sprite. The SpriteKit system will use this outline when deciding whether the sprite has been involved in a collision with another node within the scene. By default, the physical shape is assumed to be a rectangle surrounding the sprite texture (represented by the blue boundary around the node in the scene editor). Another option is to define a circle around the sprite to represent the physical shape. A much more accurate approach is to have SpriteKit define the physical shape of the node based on the outline of the sprite texture image. With the archer node selected in the scene, scroll down within the Attribute Inspector panel until the *Physics Definition* section appears. Then, using the *Body Type* menu, change the setting to *Alpha mask*:

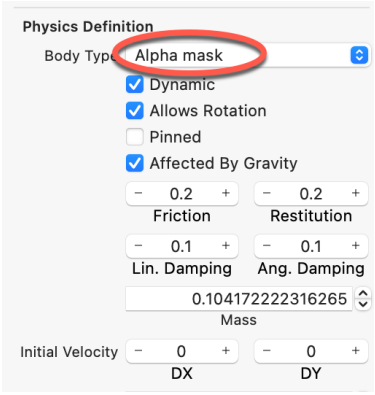

Figure 93-10

Before proceeding with the next phase of the development process, test that the scene behaves as required by clicking on the *Animate* button located along the bottom edge of the editor panel. Note that the archer slides

down and disappears off the bottom edge of the scene. This is because the sprite is configured to be affected by gravity. For the game's purposes, the archer must be pinned to the same location and not subject to the laws of gravity. Click on the *Layout* button to leave simulation mode, select the archer sprite and, within the *Physical Definition* section, turn the *Pinned* option on and the *Dynamic*, *Allows Rotation,* and *Affected by Gravity* options off. Re-run the animation to verify that the archer sprite now remains in place.

93.10 Preparing the Archery Scene

Select the *ArcheryScene.swift* file and modify it as follows to add some private variables and implement the *didMove(to:)* method:

```
import UIKit
import SpriteKit

class ArcheryScene: SKScene {

    var score = 0
    var ballCount = 20

    override func didMove(to view: SKView) {
        let archerNode = self.childNode(withName: "archerNode")
        archerNode?.position.y = 0
        archerNode?.position.x = -self.size.width/2 + 40
        self.initArcheryScene()
    }
.
.
}
```

When the archer node was added to the ArcheryScene, it was positioned using absolute X and Y coordinates. This means the node will be positioned correctly on an iPad with a 9.7" screen but not on any other screen sizes. Therefore, the first task performed by the didMove method is to position the archer node correctly relative to the screen size. Regarding the scene, position 0, 0 corresponds to the screen's center point. Therefore, to position the archer node in the vertical center of the screen, the y-coordinate is set to zero. The code then obtains the screen's width, performs a basic calculation to identify a position 40 points in from the screen's left-hand edge, and assigns it to the x-coordinate of the node.

The above code then calls another method named *initArcheryScene* which now needs to be implemented as follows within the *ArcheryScene.swift* file ready for code which will be added later in the chapter:

```
func initArcheryScene() {
}
```

93.11 Preparing the Animation Texture Atlas

When the user touches the screen, the archer sprite node will launch an arrow across the scene. For this example, we want the sprite character's loading and shooting of the arrow to be animated. The texture atlas already contains the animation frames needed to implement this (named sequentially from *archer001.png* through to *archer012.png*), so the next step is to create an action to animate this sequence of frames. One option would be to write some code to perform this task. A much easier option, however, is to create an animation action using the SpriteKit Live Editor.

Begin by selecting the *ArcheryScene.sks* file so that it loads into the editor. Once loaded, the first step is to add an

AnimateWithTextures action within the timeline of the live editor panel. Next, within the Library panel, scroll down the list of objects until the *AnimateWithTextures Action* object comes into view. Once located, drag and drop an instance of the object onto the live editor timeline for the archerNode as indicated in Figure 93-11:

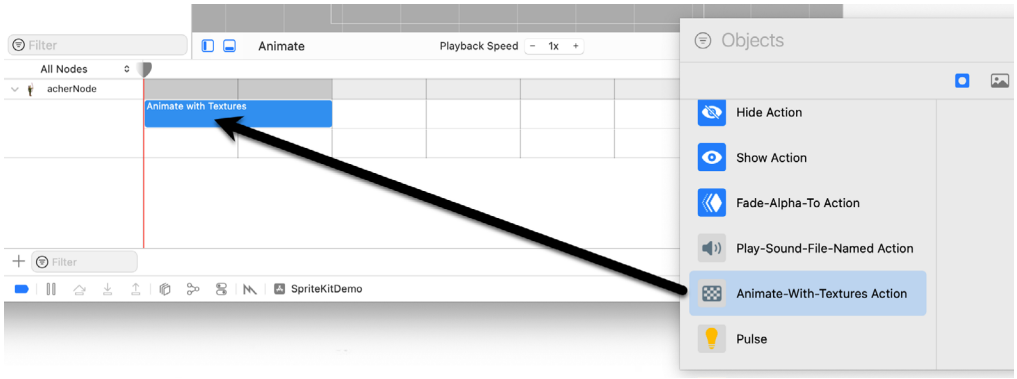

Figure 93-11

With the animation action added to the timeline, the action needs to be configured with the texture sequence to be animated. With the newly added action selected in the timeline, display the Media Library panel so that the archer texture images are listed. Next, use the Command-A keyboard sequence to select all of the images in the library and then drag and drop those images onto the *Textures* box in the *Animate with Textures* attributes panel, as shown in Figure 93-12:

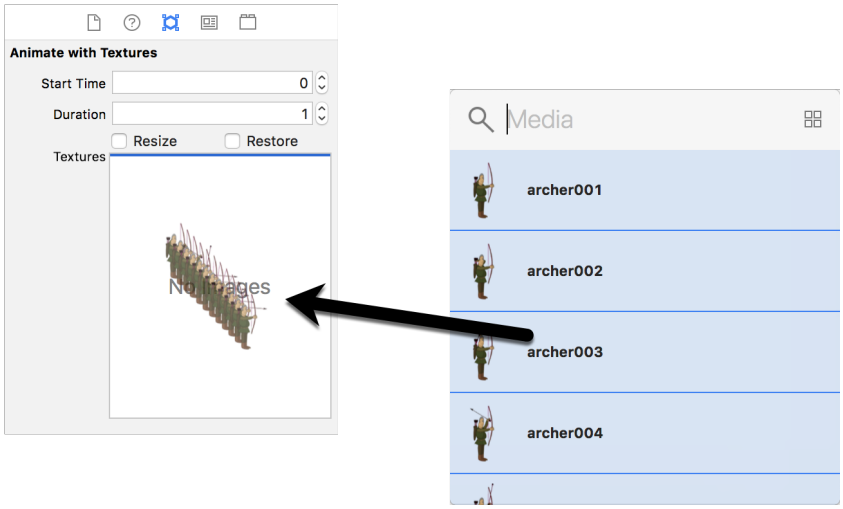

Figure 93-12

Test the animation by clicking on the *Animate* button. The archer sprite should animate through the sequence of texture images to load and shoot the arrow.

Compile and run the app and tap on the screen to enter the archery scene. On appearing, the animation sequence will execute once. The animation sequence should only run when the user taps the screen to launch an arrow. Having this action within the timeline, therefore, does not provide the required behavior for the game. Instead, the animation action needs to be converted to a *named action reference*, placed in an action file, and triggered from within the *touchesBegan* method of the archer scene class.

93.12 Creating the Named Action Reference

With the *ArcherScene.sks* file loaded into the level editor, right-click on the *Animate with Textures* action in the timeline and select the *Convert to Reference* option from the popup menu:

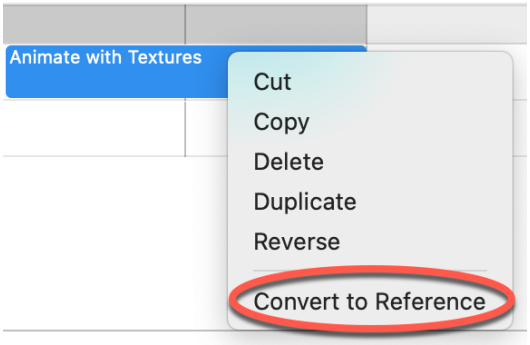

Figure 93-13

In the *Create Action* panel, name the action *animateArcher* and change the *File* menu to *Create New File*. Next, click on the *Create* button and, in the *Save As* panel, navigate to the *SpriteKitDemo* subfolder of the main project folder and enter *ArcherActions* into the *Save As:* field before clicking on *Create*.

Since the animation action is no longer required in the timeline of the archer scene, select the *ArcherScene.sks* file, right-click on the *Animate with Texture* action in the timeline, and select *Delete* from the menu.

93.13 Triggering the Named Action from the Code

With the previous steps completed, the project now has a named action (named *animateArcher*) which can be triggered each time the screen is tapped by adding some code to the *touchesBegan* method of the *ArcheryScene.swift* file. With this file selected in the Project Navigator panel, implement this method as follows:

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {

    if let archerNode = self.childNode(withName: "archerNode"),
        let animate = SKAction(named: "animateArcher") {
        archerNode.run(animate)
    }
}
```

Run the app and touch the screen within the Archery Scene. Each time a touch is detected, the archer sprite will run through the animation sequence of shooting an arrow.

93.14 Creating the Arrow Sprite Node

At this point in the tutorial, the archer sprite node goes through an animation sequence of loading and shooting an arrow, but no actual arrow is being launched across the scene. To implement this, a new sprite node must be added to the ArcheryScene. This node will be textured with an arrow image and placed to the right of the archer sprite at the end of the animation sequence. Then, a physics body will be associated with the arrow, and an impulse force will be applied to it to propel it across the scene as though shot by the archer's bow. This task will be performed entirely in code to demonstrate the alternative to using the action and live editors.

Begin by locating the *ArrowTexture.png* file in the *sprite_images* folder of the sample code archive and drag and drop it onto the left-hand panel of the *Assets* catalog screen beneath the *archer* texture atlas entry. Next, add a

new method named *createArrowNode* within the *ArcheryScene.swift* file so that it reads as follows:

```
func createArrowNode() -> SKSpriteNode {

    let arrow = SKSpriteNode(imageNamed: "ArrowTexture.png")

    if let archerNode = self.childNode(withName: "archerNode"),
        let archerPosition = archerNode.position as CGPoint?,
         let archerWidth = archerNode.frame.size.width as CGFloat? {

        arrow.position = CGPoint(x: archerPosition.x + archerWidth,
                                 y: archerPosition.y)

        arrow.name = "arrowNode"
        arrow.physicsBody = SKPhysicsBody(rectangleOf:
                                arrow.frame.size)
        arrow.physicsBody?.usesPreciseCollisionDetection = true
    }
    return arrow
}
```

The code creates a new SKSpriteNode object, positions it to the right of the archer sprite node, and assigns the name *arrowNode*. A physics body is then assigned to the node, using the node's size as the boundary of the body and enabling precision collision detection. Finally, the node is returned.

93.15 Shooting the Arrow

A physical force needs to be applied to propel the arrow across the scene. The arrow sprite's creation and propulsion must be timed to occur at the end of the archer animation sequence. This timing can be achieved via some minor modifications to the *touchesBegan* method:

```
override func touchesBegan(_ touches: Set<UITouch>, with event: UIEvent?) {

    if let archerNode = self.childNode(withName: "archerNode"),
        let animate = SKAction(named: "animateArcher") {
        let shootArrow = SKAction.run({
            let arrowNode = self.createArrowNode()
            self.addChild(arrowNode)
            arrowNode.physicsBody?.applyImpulse(CGVector(dx: 60, dy: 0))
        })

        let sequence = SKAction.sequence([animate, shootArrow])

        archerNode.run(sequence)
    }
}
```

A new SKAction object is created, specifying a block of code to be executed. This run block calls the *createArrowNode* method, adds the new node to the scene, and then applies an impulse force of 60.0 on the X-axis of the scene. An SKAction sequence comprises the previously created animation action and the new run

block action. This sequence is then run on the archer node.

When executed with these changes, touching the screen should now cause an arrow to be launched after the archer animation completes. Then, as the arrow flies across the scene, it gradually falls toward the bottom of the display. This behavior is due to gravity's effect on the physics body assigned to the node.

93.16 Adding the Ball Sprite Node

The game's objective is to score points by hitting balls with arrows. So, the next logical step is adding the ball sprite node to the scene. Begin by locating the *BallTexture.png* file in the *sprite_images* folder of the sample code package and drag and drop it onto the *Assets.xcassets* catalog.

Next, add the corresponding *createBallNode* method to the *ArcheryScene.swift* file as outlined in the following code fragment:

```
func createBallNode() {
    let ball = SKSpriteNode(imageNamed: "BallTexture.png")

    let screenWidth = self.size.width

    ball.position = CGPoint(x: CGFloat.random(
                    in: -screenWidth/2 ..< screenWidth/2-100),
                        y: self.size.height-50)

    ball.name = "ballNode"
    ball.physicsBody = SKPhysicsBody(circleOfRadius:
                       (ball.size.width/2))

    ball.physicsBody?.usesPreciseCollisionDetection = true
    self.addChild(ball)
}
```

This code creates a sprite node using the ball texture and then sets the initial position at the top of the scene but a random position on the X-axis. Since position 0 on the X-axis corresponds to the horizontal center of the screen (as opposed to the far left side), some calculations are performed to ensure that the balls can fall from most of the screen's width using random numbers for the X-axis values.

The node is assigned a name and a circular physics body slightly less than the radius of the ball texture image. Finally, precision collision detection is enabled, and the ball node is added to the scene.

Next, modify the *initArcheryScene* method to create an action to release a total of 20 balls at one-second intervals:

```
func initArcheryScene() {

    let releaseBalls = SKAction.sequence([SKAction.run({
    self.createBallNode() }),
    SKAction.wait(forDuration: 1)])

    self.run(SKAction.repeat(releaseBalls,
                             count: ballCount))
}
```

Run the app and verify that the balls now fall from the top of the scene. Then, attempt to hit the balls as they fall by tapping the background to launch arrows. Note, however, that when an arrow hits a ball, it simply bounces off:

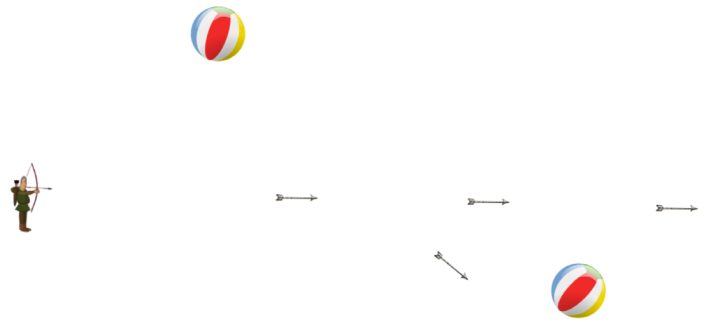

Figure 93-14

The goal for the completed game is to have the balls burst with a sound effect when hit by the arrow and for a score to be presented at the end of the game. The steps to implement this behavior will be covered in the next chapters.

The balls fall from the top of the screen because they have been assigned a physics body and are subject to the simulated forces of gravity within the Sprite Kit physical world. To reduce the effects of gravity on both the arrows and balls, modify the *didMove(to view:)* method to change the current gravity setting on the scene's *physicsWorld* object:

```
override func didMove(to view: SKView) {
    let archerNode = self.childNode(withName: "archerNode")
    archerNode?.position.y = 0
    archerNode?.position.x = -self.size.width/2 + 40
    self.physicsWorld.gravity = CGVector(dx: 0, dy: -1.0)
    self.initArcherScene()
}
```

93.17 Summary

The goal of this chapter has been to create a simple game for iOS using the Sprite Kit framework. In creating this game, topics such as using sprite nodes, actions, textures, sprite animations, and physical forces have been used to demonstrate the use of the Xcode Sprite Kit editors and Swift code.

In the next chapter, this game example will be further extended to demonstrate the detection of collisions.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y