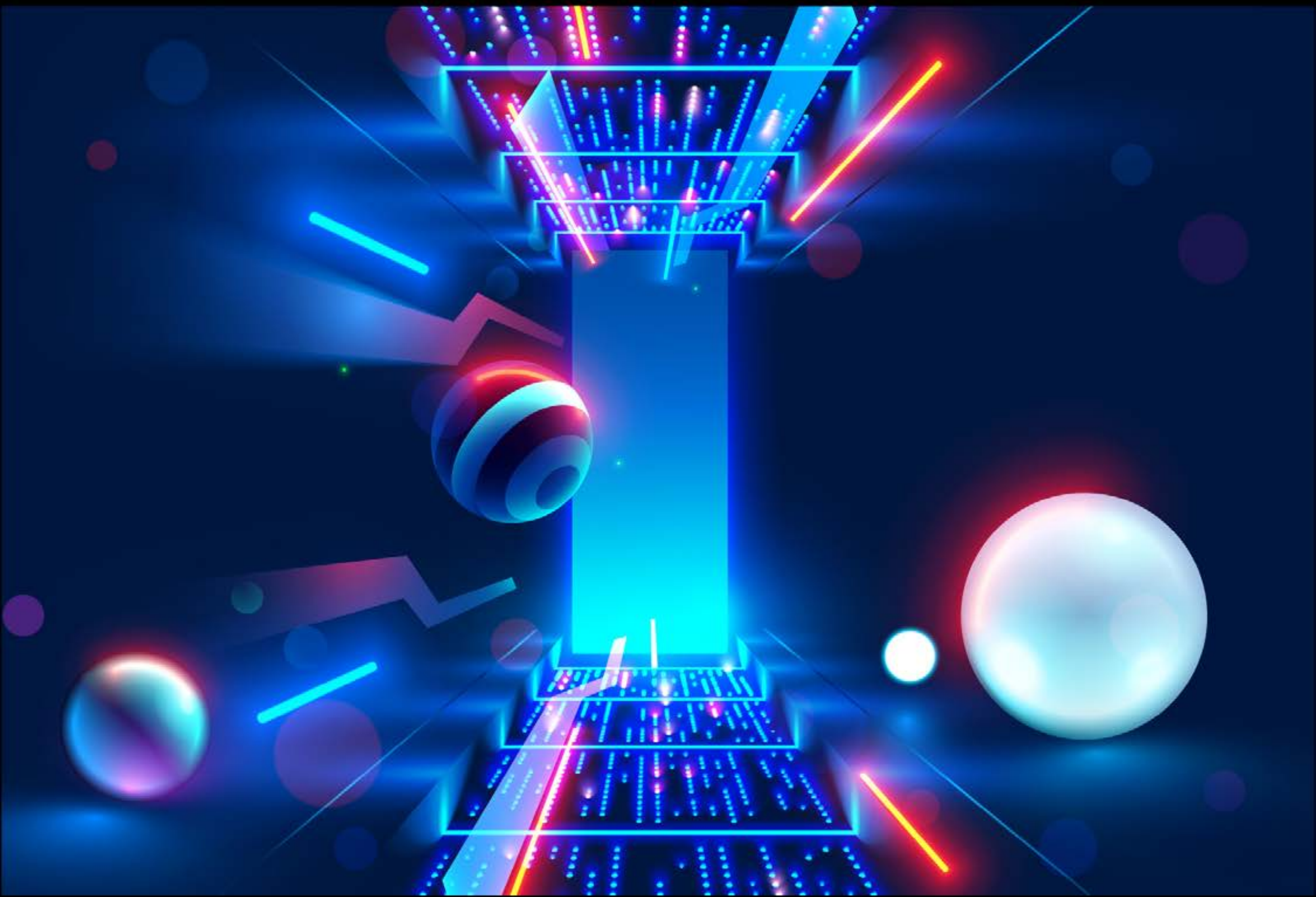
iOS 18 App Development Essentials



Payload publishing

iOS 18 App Development Essentials

iOS 18 App Development Essentials

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

Rev: 1.0

Payload publishing

https://www.payloadbooks.com

Contents

Table of Contents

3. Installing Xcode 16 and the iOS 18 SDK

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

All of the examples in this book are based on Xcode version 15 and use features unavailable in earlier Xcode versions. This chapter will cover the steps involved in installing Xcode 16 and the iOS 18 SDK on macOS.

3.1 Identifying Your macOS Version

When developing with Xcode 16, a system running macOS Sonoma 14.5 or later or later is required. If you are unsure of the version of macOS on your Mac, you can find this information by clicking on the Apple menu in the top left-hand corner of the screen and selecting the *About This Mac* option from the menu. In the resulting dialog, check the *macOS* line:

Figure 3-1

If the "About This Mac" dialog does not indicate that macOS Sonoma 14.5 or later is running, click on the *More Info...* button to open the Settings app and check for available operating system updates.

3.2 Installing Xcode 16 and the iOS 18 SDK

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

3.3 Starting Xcode

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

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

Figure 3-2

3.4 Adding Your Apple ID to the Xcode Preferences

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

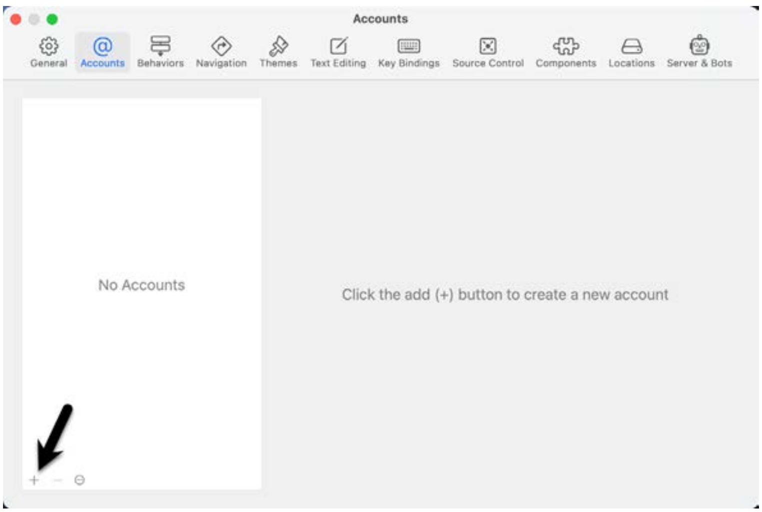


Figure 3-3

3.5 Developer and Distribution Signing Identities

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

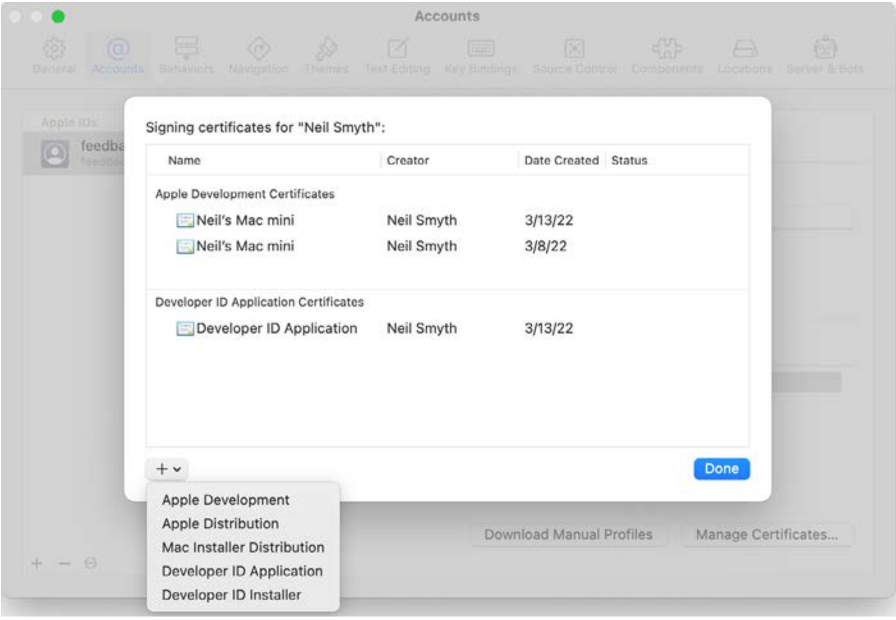


Figure 3-4

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

3.6 Summary

This book was written using Xcode 16 and the iOS 18 SDK running on macOS 13.5.2 (Ventura). Before beginning SwiftUI development, the first step is to install Xcode and configure it with your Apple ID via the accounts section of the Preferences screen. Once these steps have been performed, a development certificate must be generated which will be used to sign apps developed within Xcode. This will allow you to build and test your apps on physical iOS-based devices.

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

Having installed the iOS SDK and successfully launched Xcode 16, we can now look at Xcode in more detail, starting with Playgrounds.

Chapter 4

4. An Introduction to Xcode 16 Playgrounds

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

4.1 What is a Playground?

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

4.2 Creating a New Playground

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

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

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

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

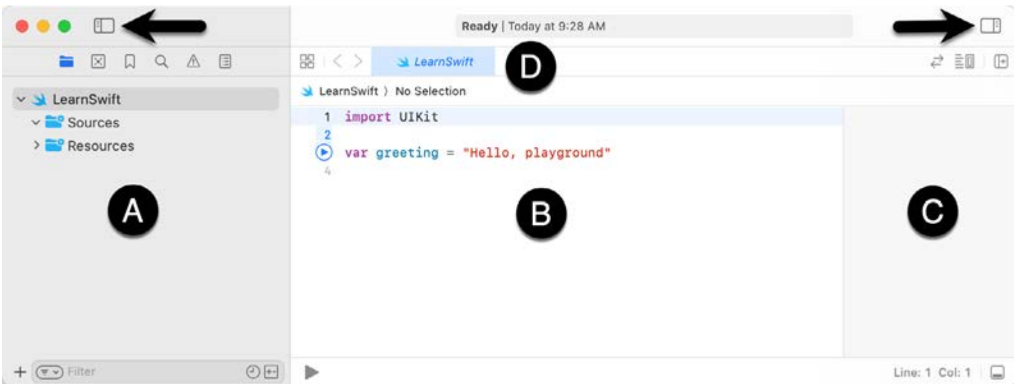


Figure 4-1

The panel on the left-hand side of the window (marked A in Figure 4-1) is the Navigator panel which provides access to the folders and files that make up the playground. To hide and show this panel, click on the button indicated by the left-most arrow. The center panel (B) is the *playground editor* where the lines of Swift code are entered. The right-hand panel (C) is referred to as the *results panel* and is where the results of each Swift expression entered into the playground editor panel are displayed. The tab bar (D) will contain a tab for each file currently open within the playground editor. To switch to a different file, simply select the corresponding tab. To close an open file, hover the mouse pointer over the tab and click on the "X" button when it appears to the left of the file name.

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

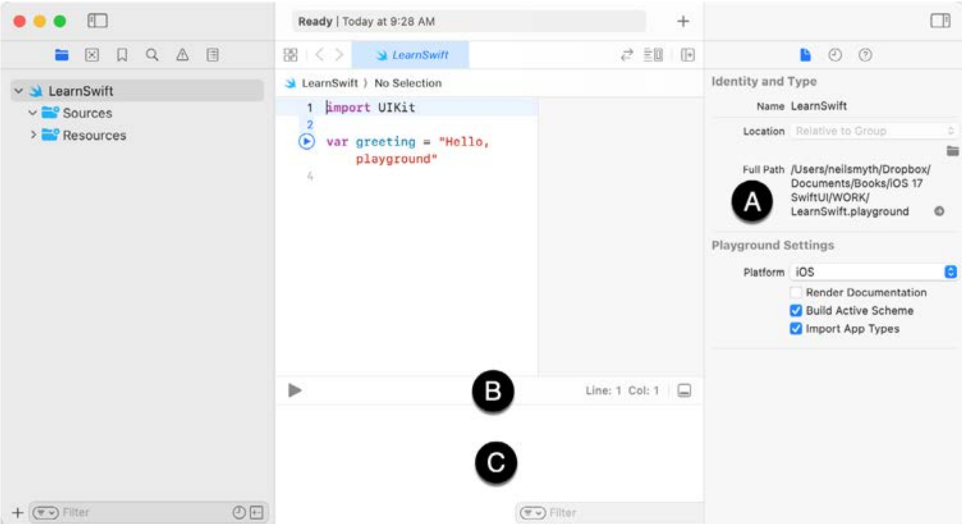


Figure 4-2

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

4.3 A Swift Playground Example

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

```
import UIKit

var greeting = "Hello, playground"

print("Welcome to Swift")
```

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

Note that although some extra code has been entered, nothing yet appears in the results panel. This is because the code has yet to be executed. One option to run the code is to click on the Execute Playground button located in the bottom left-hand corner of the main panel as indicated by the arrow in Figure 4-3:

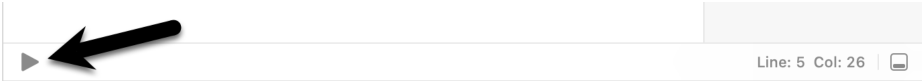


Figure 4-3

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

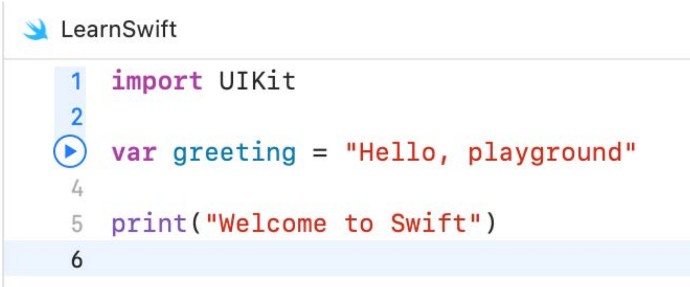


Figure 4-4

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

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

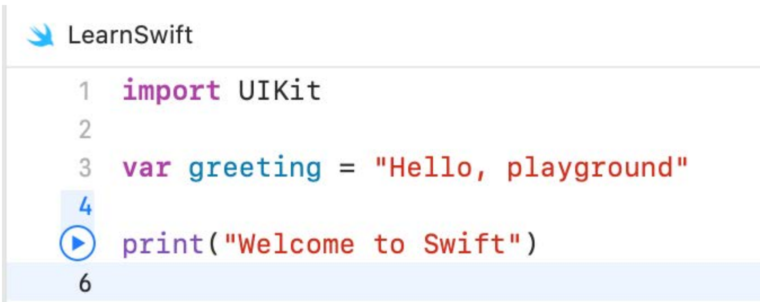


Figure 4-5

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

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

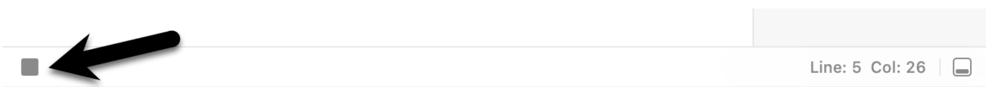


Figure 4-6

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

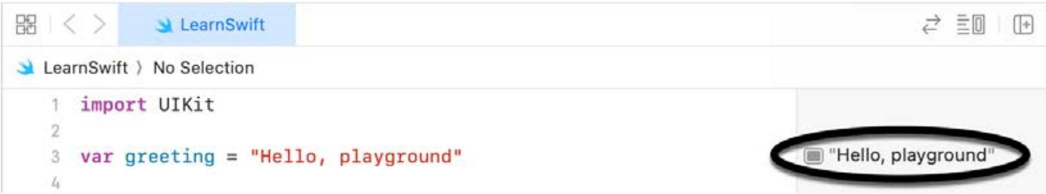


Figure 4-7

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

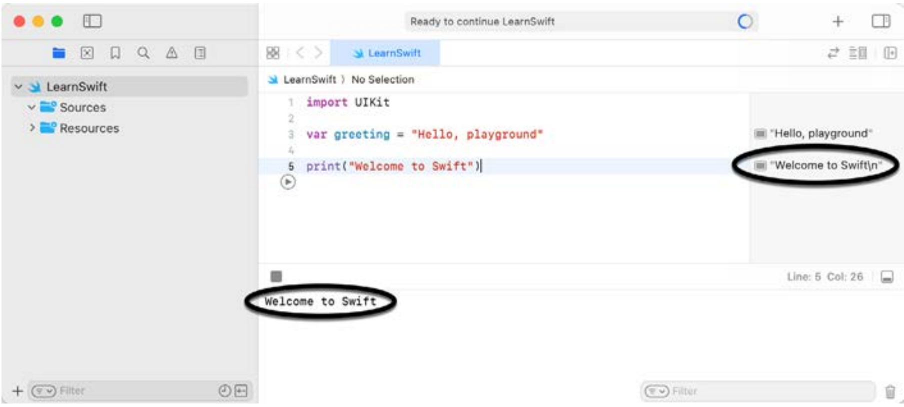


Figure 4-8

4.4 Viewing Results

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

```
var x = 10

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

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

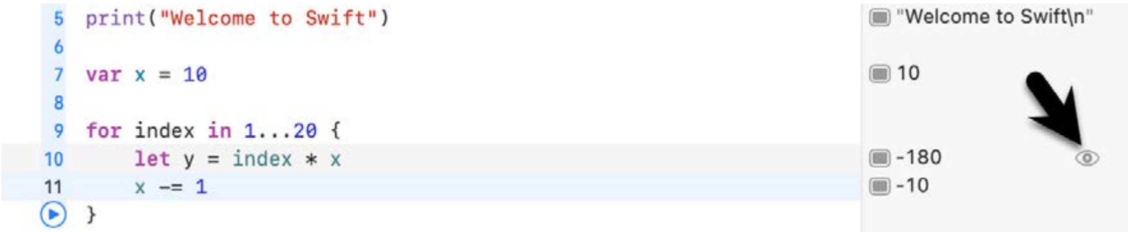


Figure 4-9

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

Figure 4-10

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

Figure 4-11

4.5 Adding Rich Text Comments

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

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

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

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

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

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

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

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

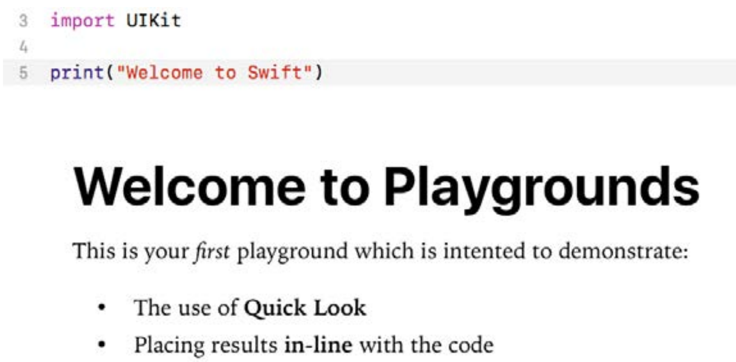


Figure 4-12

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

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

4.6 Working with Playground Pages

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

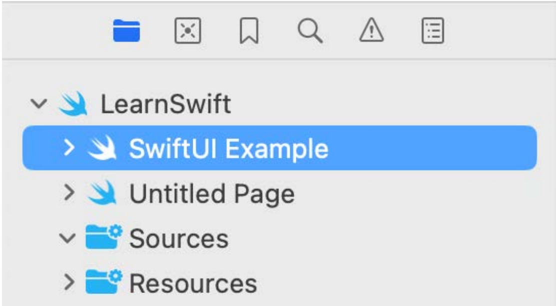


Figure 4-13

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

4.7 Working with SwiftUI and Live View in Playgrounds

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

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

```
import SwiftUI
import PlaygroundSupport
```

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

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

```
struct ExampleView: View {

    var body: some View {

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

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

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

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

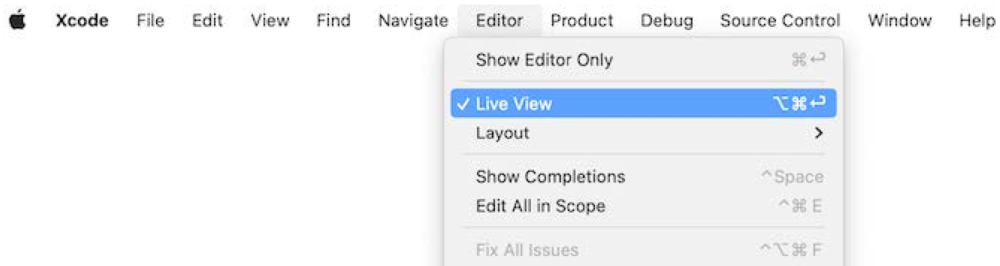


Figure 4-14

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

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

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

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

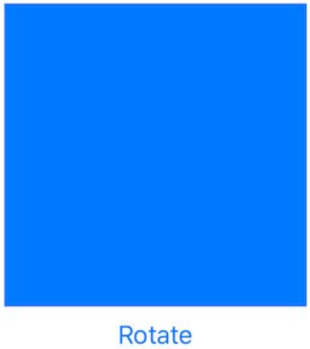


Figure 4-15

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

```
import SwiftUI
import PlaygroundSupport

struct ExampleView: View {

    @State private var rotation: Double = 0

    var body: some View {

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

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

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

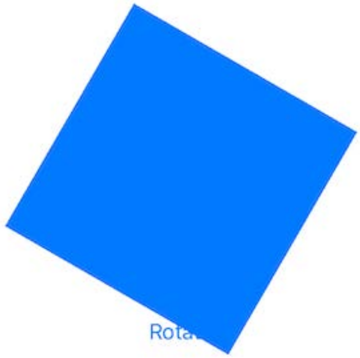


Figure 4-16

4.8 Summary

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

Chapter 9

9. Swift Functions, Methods, and Closures

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

9.1 What is a Function?

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

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

9.2 What is a Method?

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

9.3 How to Declare a Swift Function

A Swift function is declared using the following syntax:

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

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

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

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

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

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

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

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

9.4 Implicit Returns from Single Expressions

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

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

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

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

9.5 Calling a Swift Function

Once declared, functions are called using the following syntax:

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

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

```
sayHello()
```

9.6 Handling Return Values

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

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

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

When developing in Swift, situations may arise where the result returned by a method or function call is not used. When this is the case, the return value may be discarded by assigning it to '_'. For example:

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

9.7 Local and External Parameter Names

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

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

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

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

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

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

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

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

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

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

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

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

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

9.8 Declaring Default Function Parameters

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

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

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

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

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

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

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

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

9.9 Returning Multiple Results from a Function

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

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

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

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

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

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

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

Usage of this function might read as follows:

```
let lengthTuple = sizeConverter(20)

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

9.10 Variable Numbers of Function Parameters

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

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

```
    }
}

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

9.11 Parameters as Variables

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

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

    var length = length
    var width = width

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

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

9.12 Working with In-Out Parameters

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

```
var myValue = 10

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.13 Functions as Parameters

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

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

let toFeet = inchesToFeet
```

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

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

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

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

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

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

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

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

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

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

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

    return inches * 0.0833333
}

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
    }
}
```

9.14 Closure Expressions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

9.15 Shorthand Argument Names

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

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

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

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

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

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

9.16 Closures in Swift

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

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

    var counter = 0

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

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

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

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

9.17 Take the Knowledge Test

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

https://www.answertopia.com/uc2w

9.18 Summary

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

Chapter 18

18. SwiftUI Architecture

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

18.1 SwiftUI App Hierarchy

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

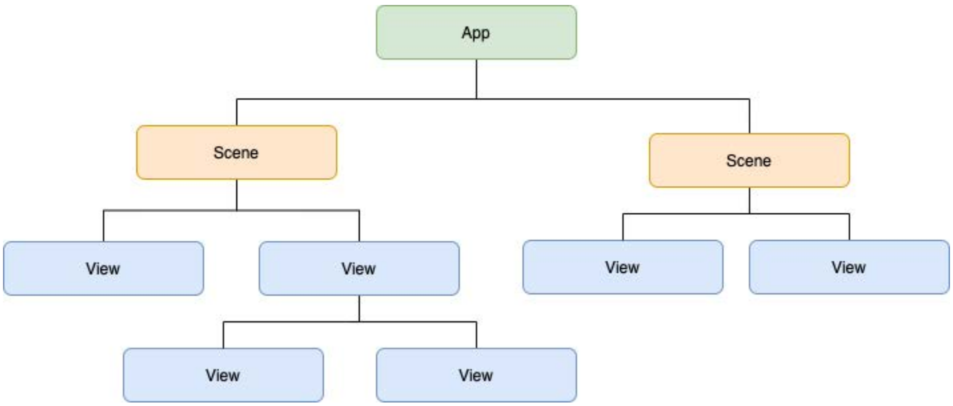


Figure 18-1

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

18.2 App

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

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

18.3 Scenes

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

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

18.4 Views

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

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

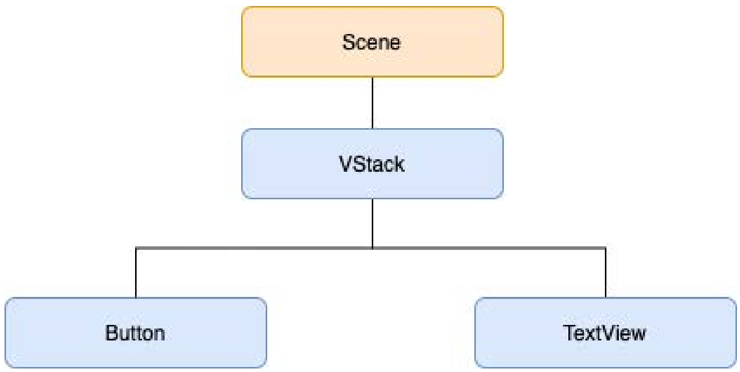


Figure 18-2

18.5 Take the Knowledge Test

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

https://www.answertopia.com/z871

18.6 Summary

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

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

Chapter 23

23. A SwiftUI Example Tutorial

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

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

23.1 Creating the Example Project

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

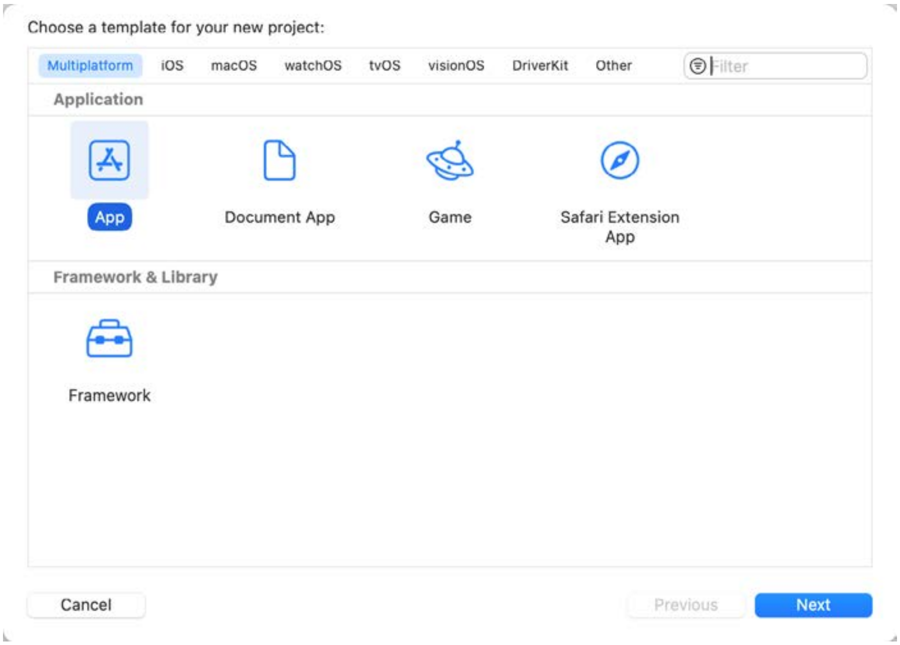


Figure 23-1

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

23.2 Reviewing the Project

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

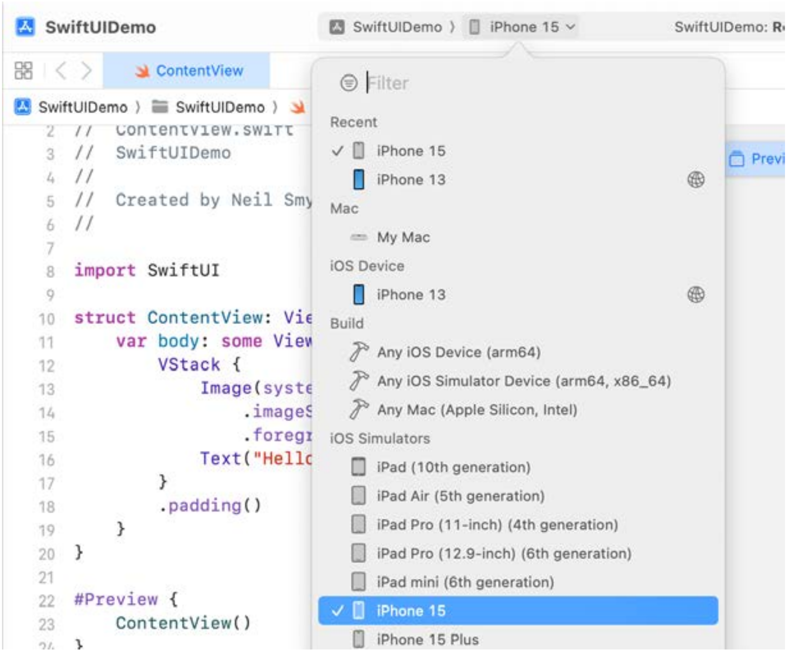


Figure 23-2

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

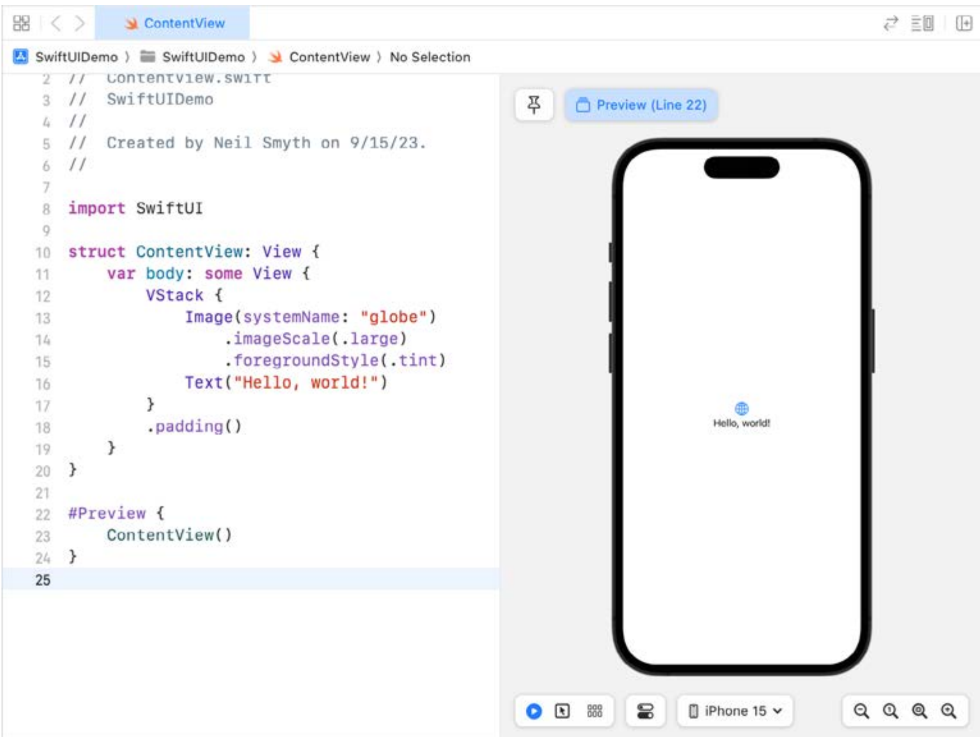


Figure 23-3

23.3 Modifying the Layout

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

```
import SwiftUI

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

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

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

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

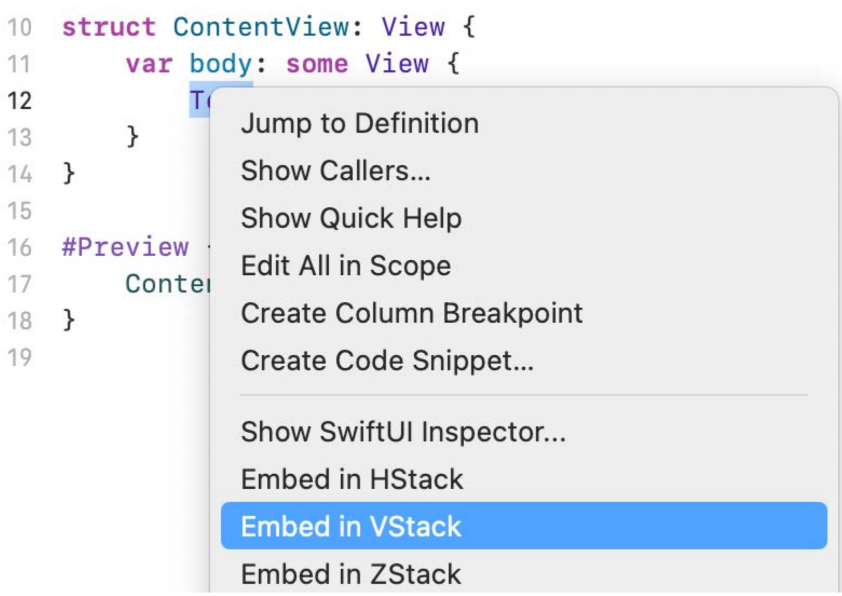


Figure 23-4

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

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

23.4 Adding a Slider View to the Stack

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

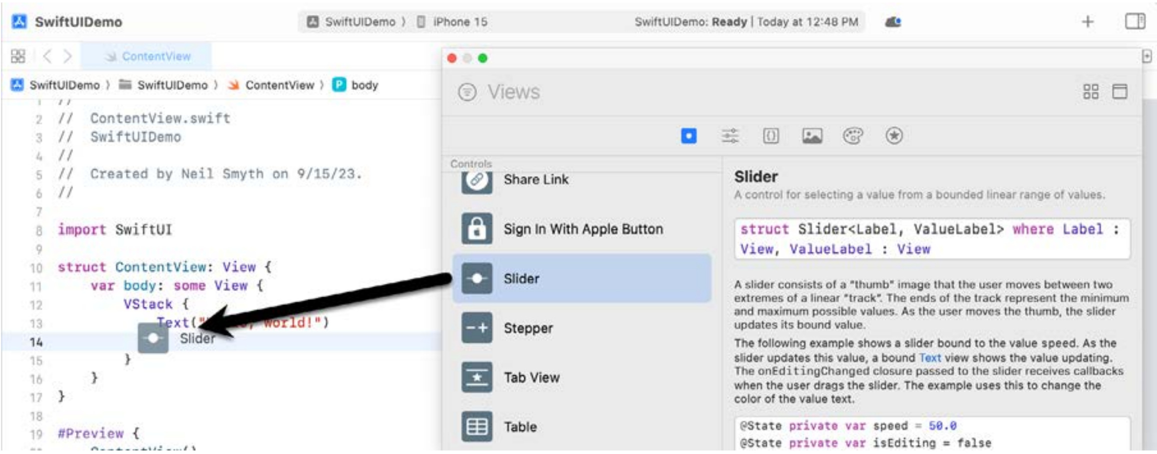


Figure 23-5

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

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

23.5 Adding a State Property

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

```
struct ContentView: View {
```

```
    @State private var rotation: Double = 0

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

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

23.6 Adding Modifiers to the Text View

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

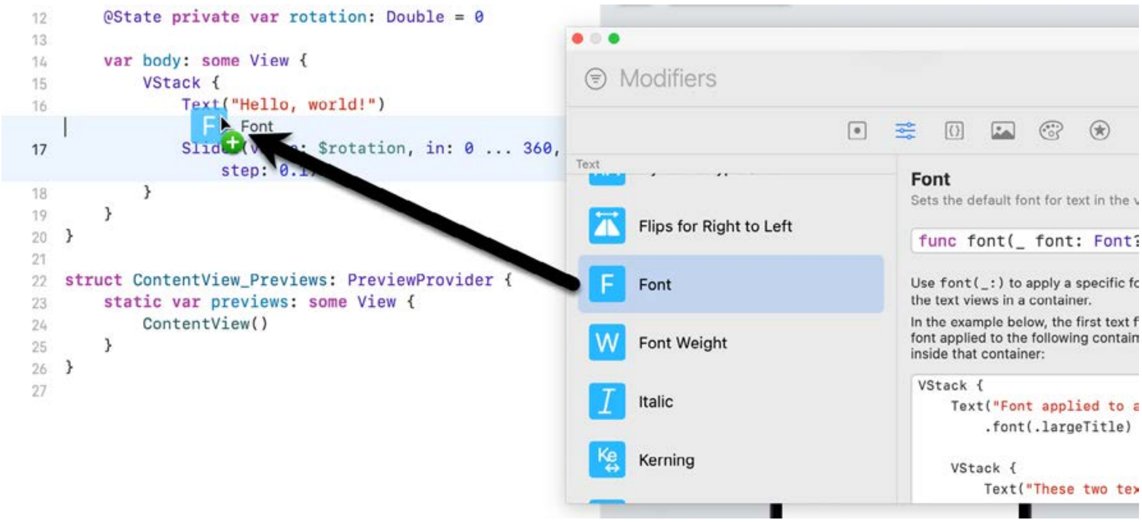


Figure 23-6

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

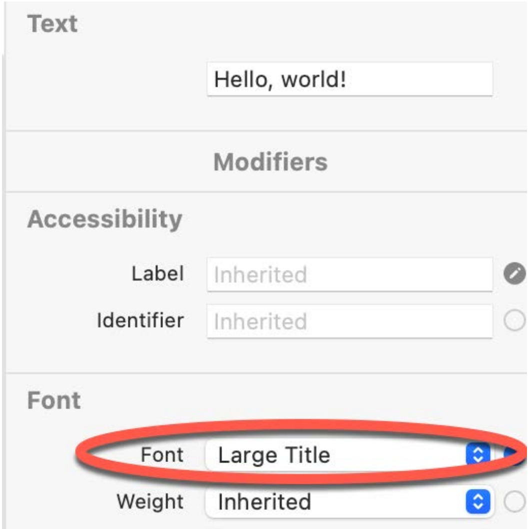


Figure 23-7

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

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

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

23.7 Adding Rotation and Animation

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

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

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

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

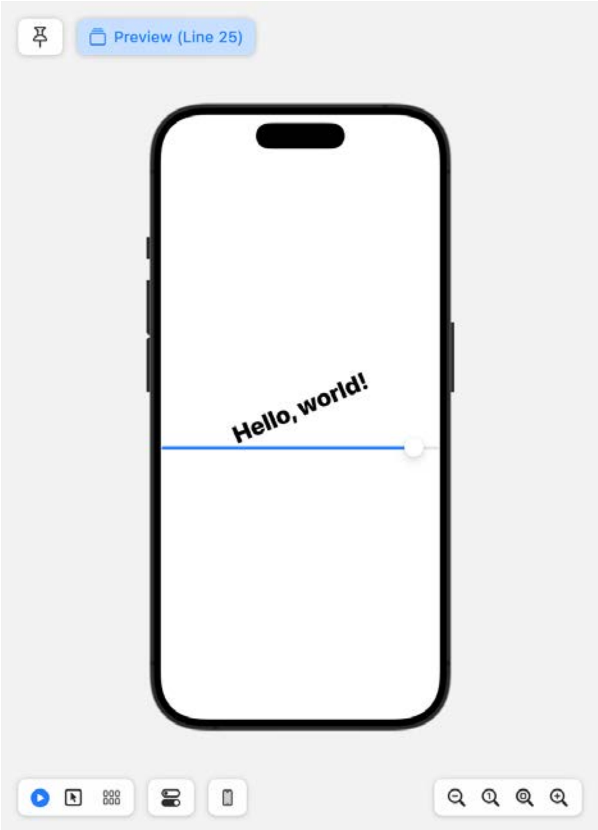


Figure 23-8

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

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

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

23.8 Adding a TextField to the Stack

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

```
struct ContentView: View {

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

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

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

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

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

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

23.9 Adding a Color Picker

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

```
import SwiftUI

struct ContentView: View {

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

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

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

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

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

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

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

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

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

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

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

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

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

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

23.10 Tidying the Layout

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

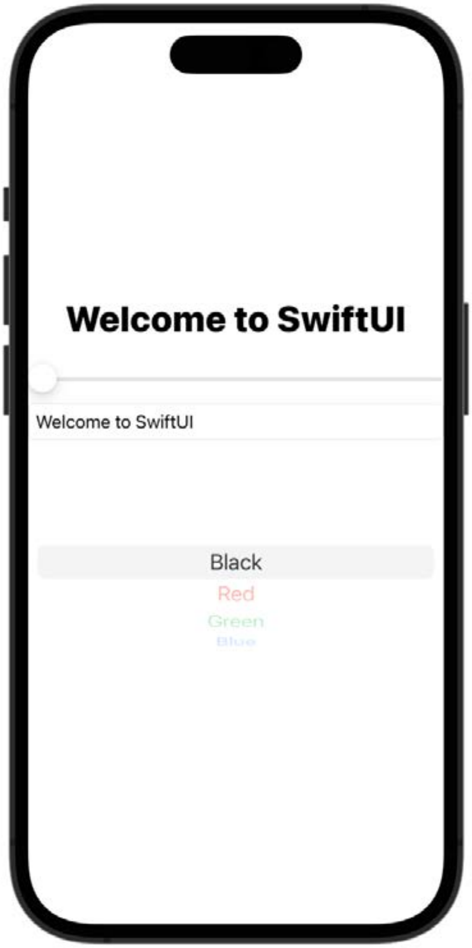


Figure 23-9

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

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

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

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

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

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

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

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

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

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

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

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

Figure 23-10

23.11 Take the Knowledge Test

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

https://www.answertopia.com/ukob

23.12 Summary

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

Chapter 31

31. SwiftUI Lists and Navigation

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

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

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

31.1 SwiftUI Lists

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

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

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

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

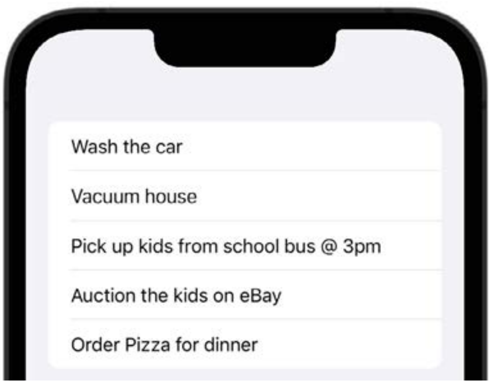


Figure 31-1

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

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

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

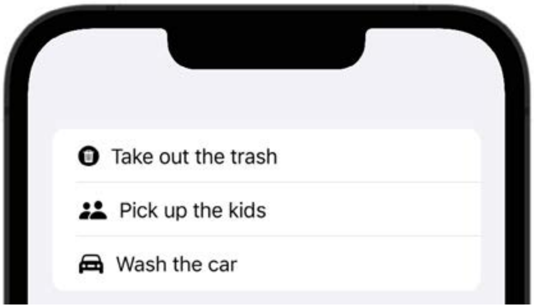


Figure 31-2

31.2 Modifying List Separators and Rows

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

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

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

31.3 SwiftUI Dynamic Lists

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

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

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

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

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

```
struct ContentView: View {

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

    var body: some View {

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

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

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

```
struct ContentView: View {

    @State private var toggleStatus = true
.
.
    var body: some View {

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

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

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

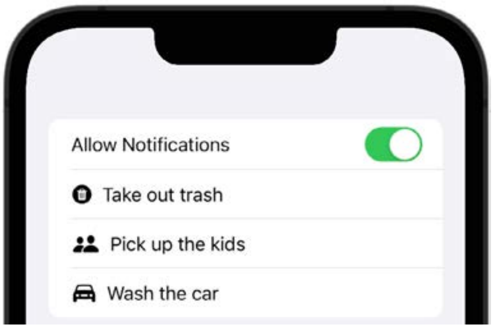


Figure 31-3

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

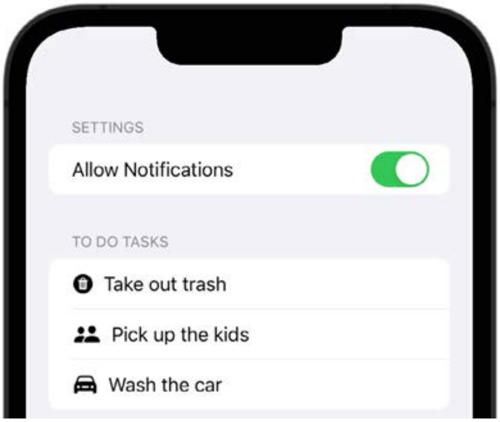


Figure 31-4

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

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

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

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

31.4 Creating a Refreshable List

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

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

```
            }
        }

        Section(header: Text("To Do Tasks")) {
            ForEach (listData) { item in
                HStack {
                    Image(systemName: item.imageName)
                    Text(item.task)
                }
            }
        }
}
.refreshable {
    listData = [
        ToDoItem(task: "Order dinner", imageName: "dollarsign.circle.fill"),
        ToDoItem(task: "Call financial advisor", imageName: "phone.fill"),
        ToDoItem(task: "Sell the kids", imageName: "person.2.fill")
        ]
}
```

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

Figure 31-5

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

31.5 SwiftUI NavigationStack and NavigationLink

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

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

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

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

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

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

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

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

Figure 31-6

31.6 Navigation by Value Type

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

```
NavigationStack {

    List {

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

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

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

```
.
```

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

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

31.7 Working with Navigation Paths

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

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

```
struct ContentView: View {

   @State private var stackPath = NavigationPath()

   var body: some View {

        NavigationStack(path: $stackPath) {
.
.
```

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

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

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

```
stackPath.append(value)
```

31.8 Navigation Bar Customization

The NavigationStack title bar may also be customized using modifiers on the List component to set the title and add buttons to perform additional tasks. The following example calls the *navigationTitle()* modifier to set the title to "To Do List." The code also adds a button labeled "Add" to the toolbar. This button is applied to the layout as the child of a ToolbarItem instance using the *toolbar()* modifier and configured to call a hypothetical method named *addTask()*:

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

31.9 Making the List Editable

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

Figure 31-7

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

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

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

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

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

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

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

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

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

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

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

Figure 31-8

31.10 Hierarchical Lists

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

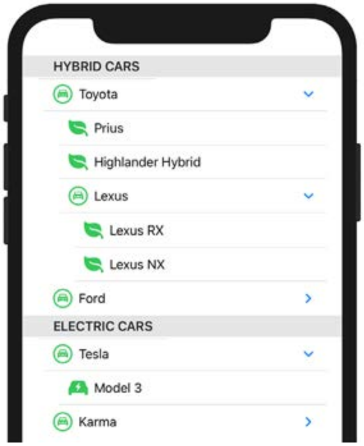


Figure 31-9

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

31.11 Multicolumn Navigation

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

31.12 Take the Knowledge Test

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

https://www.answertopia.com/utji

31.13 Summary

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

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

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

Chapter 55

55. An Introduction to SwiftData

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

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

55.1 Introducing SwiftData

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

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

55.2 Model Classes

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

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

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

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

```
import SwiftData

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

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

55.3 Model Container

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

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

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

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

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

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

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

55.4 Model Configuration

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

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

55.5 Model Context

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

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

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

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

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

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

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

55.6 Predicates and FetchDescriptors

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

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

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

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

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

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

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

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

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

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

55.7 The @Query Macro

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

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

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

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

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

55.8 Model Relationships

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

```
@Model
```

```
class CallDate {

    var date: Date

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

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

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

    @Relationship var calls = [CallDate]()

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

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

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

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

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

The full list of deletion rules is as follows:

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

55.9 Model Attributes

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

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

    @Relationship var calls = [CallDate]()

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

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

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

55.10 Take the Knowledge Test

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

https://www.answertopia.com/a7fx

55.11 Summary

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

Chapter 62

62. An Overview of Live Activities in SwiftUI

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

62.1 Introducing Live Activities

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

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

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

62.2 Creating a Live Activity

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

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

62.3 Live Activity Attributes

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

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

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

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

62.4 Designing the Live Activity Presentations

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

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

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

        } dynamicIsland: { context in

            DynamicIsland {

                DynamicIslandExpandedRegion(.leading) {

                }
                DynamicIslandExpandedRegion(.trailing) {

                }
                DynamicIslandExpandedRegion(.bottom) {

                }
                DynamicIslandExpandedRegion(.center) {

                }
            } compactLeading: {

            } compactTrailing: {

            } minimal: {
```

```
            }
        }
    }
}
```

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

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

62.4.1 Lock Screen/Banner

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

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

            }
        } dynamicIsland: { context in
.
.
```

62.4.2 Dynamic Island Expanded Regions

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

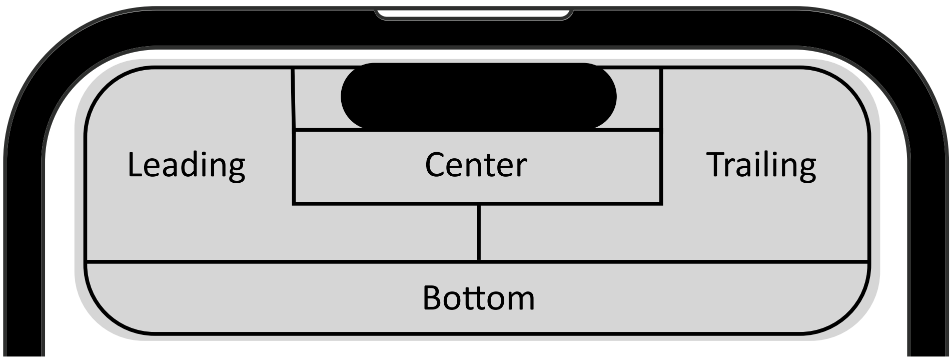


Figure 62-1

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

```
.
.
```

```
} dynamicIsland: { context in

    DynamicIsland {

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

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

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

62.4.3 Dynamic Island Compact Regions

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

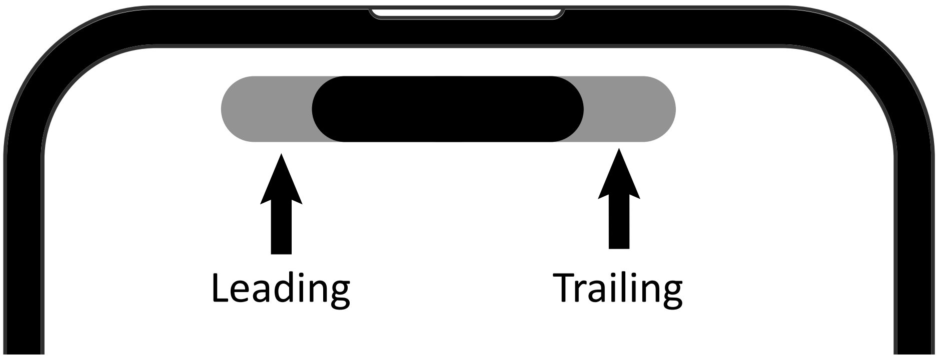


Figure 62-2

An example compact declaration might read as follows:

```
.
```

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

62.4.4 Dynamic Island Minimal

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

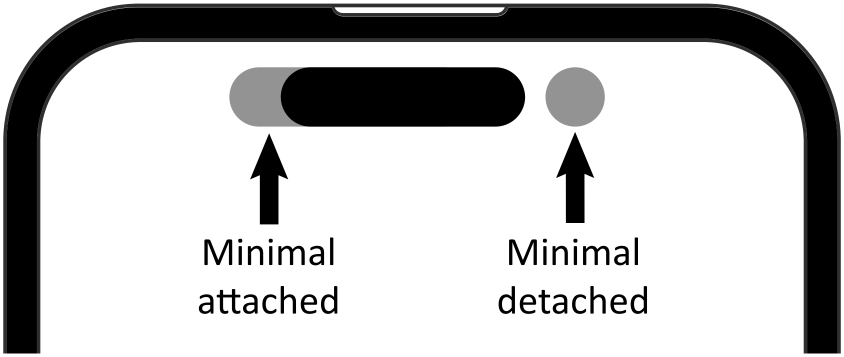


Figure 62-3

For example:

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

62.5 Starting a Live Activity

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

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

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

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

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

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

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

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

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

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

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

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

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

62.6 Updating a Live Activity

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

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

Task {

    await activity?.update(
```

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

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

62.7 Activity Alert Configurations

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

```
var alertConfig: AlertConfiguration? = nil

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

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

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

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

62.8 Stopping a Live Activity

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

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

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

```
)
```

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

62.9 Take the Knowledge Test

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

https://www.answertopia.com/oh2m

62.10 Summary

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

Index

Symbols

A

B

C

D

G

H

I

L

M

N

O

P

R

T

U

Y

Z