IN THIS CHAPTER

» Understanding what SwiftUI is

» Getting the tools for SwiftUI

» Comparing UIKit to SwiftUI

» Using the preview canvas and Live Preview

» Understanding the various files in a SwiftUI project

Chapter 1

Introducing SwiftUI

COPYRIGHTED MATERIAL

I know the feeling of being on the verge of learning something new. If you're anything like me, you're eager to try things out and see how it feels. And that's exactly what you do in this chapter!

In this chapter, I explain what SwiftUI is, show you how SwiftUI has changed the user interface (UI) development paradigm, and explain how SwiftUI makes the process easier going forward. Then I tell you how you can get started with the necessary tools. Finally, with the tools that you've installed, you create your first iOS application using SwiftUI, and learn how the various components in your project work together as a whole.

Understanding What SwiftUI Is

SwiftUI is a declarative programming framework for developing UIs for iOS, iPadOS, watchOS, tvOS, and macOS applications. In fact, SwiftUI was invented by the watchOS group at Apple.

Before SwiftUI was introduced, most developers used UIKit and Storyboard (which is still supported by Apple in the current version of Xcode, as of this writing [version 11.4.1]) to design a UI. Using UIKit and Storyboard, developers drag and drop View controls onto View Controllers and connect them to outlets and actions on the View Controller classes. This model of building UIs is known as *Model View Controller* (MVC), which creates a clean separation between UI and business logic.

The following shows a simple implementation in UIKit and Storyboard. Here, a `Button` and `Label` view have been added to the View Controller in Storyboard; two outlets and an action have been created to connect to them:

```
class ViewController: UIViewController {

    @IBOutlet weak var lbl: UILabel!
    @IBOutlet weak var button: UIButton!
    @IBAction func btnClicked(_ sender: Any) {
        lbl.text = "Button tapped"
    }
```

For laying out the views, you use auto-layout to position the button and label in the middle of the screen (both horizontally and vertically).

To customize the look and feel of the button, you can code it in the `loadView()` method, like this:

```
    override func loadView() {
        super.loadView()

        // background color
        button.backgroundColor = UIColor.yellow

        // button text and color
        button.setTitle("Submit", for: .normal)
        button.setTitleColor(.black, for: .normal)

        // padding
        button.contentEdgeInsets = UIEdgeInsets(
            top: 10, left: 10, bottom: 10, right: 10)

        // border
        button.layer.borderColor =
            UIColor.darkGray.cgColor
        button.layer.borderWidth = 3.0
```

```
        // text font
        button.titleLabel!.font =
            UIFont.systemFont(ofSize: 26, weight:
            UIFont.Weight.regular)

        // rounder corners
        button.layer.cornerRadius = 10

        // auto adjust button size
        button.sizeToFit()
    }
```

Figure 1-1 shows the button that has customized. UIKit is an *event-driven* framework, where you can reference each view in your view controller, update its appearance, or handle an event through *delegates* when some events occurred.

FIGURE 1-1: UIKit is event driven, and it uses delegates to handle events.

In contrast, SwiftUI is a *state-driven*, declarative framework. In SwiftUI, you can implement all the above with the following statements (I explain how to build all these later in this chapter):

```
struct ContentView: View {
    @State private var label = "label"

    var body: some View {
        VStack {
            Button(action: {
                self.label = "Button tapped"
            }) {
                Text("Submit")
                    .padding(EdgeInsets(
                        top: 10, leading: 10,
                        bottom: 10, trailing: 10))
                    .background(Color.yellow)
                    .foregroundColor(Color.black)
                    .border(Color.gray, width: 3)
                    .font(Font.system(size: 26.0))
                    .overlay(
                        RoundedRectangle(cornerRadius: 10)
                            .stroke(Color.gray,
                                lineWidth: 5)
                    )
            }
            Text(label)
                .padding()
        }
    }
}
```

Notice that all the views are now created declaratively using code — no more drag-and-drop in Storyboard. Layouts are now also specified declaratively using code (the `VStack` in this example stacks all the views vertically). Delegates are now replaced with closures. More important, views are now a function of state (and not a sequence of events) — the text displayed by the `Text` view is now bound to the state variable `label`. When the button is tapped, you change the value of the `label` state variable, which automatically updates the text displayed in the `Text` view. This programming paradigm is known as *reactive programming.*

Figure 1-2 shows the various views in action.

FIGURE 1-2: SwiftUI is a state-driven declarative framework.

Getting the Tools

To start developing using SwiftUI, you need the following:

- Xcode version 11 or later
- A deployment target (Simulator or real device) of iOS 13 or later
- macOS Mojave (10.14) or later (Note that if you're running macOS Mojave, you won't be able to use Live Preview and design canvas features; full features are available only in macOS Catalina (10.15) and later.)

To install Xcode, you can install it from the App Store on your Mac (see Figure 1-3).

Alternatively, if you have a paid Apple developer account (you need this if you want to make your apps available on the App Store, but this is not a requirement for trying out the examples in this book), head over to `https://developer.apple.com`, sign in, and download Xcode directly.

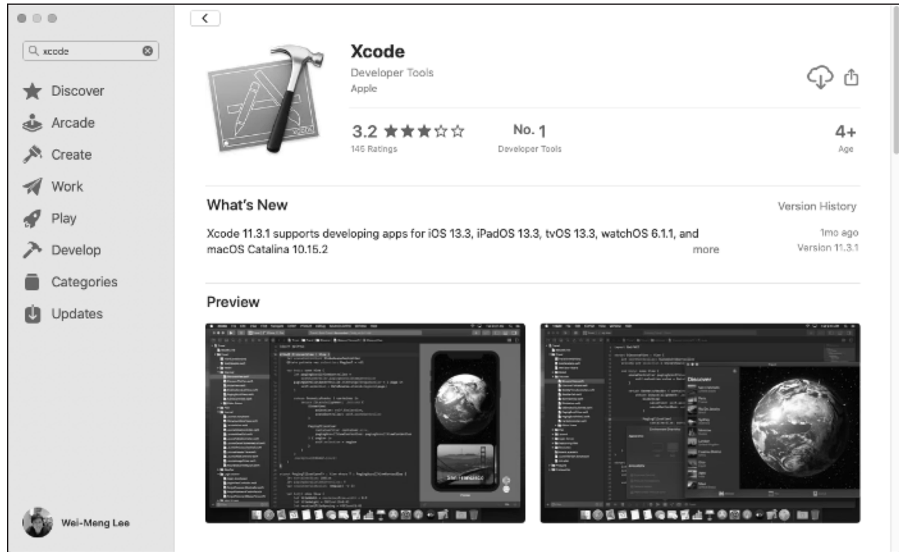

FIGURE 1-3: Installing Xcode from the Mac App Store.

TIP

For this book, our focus is on developing iOS applications for the iPhone. Developing iPad, watchOS, tvOS, and macOS applications using SwiftUI is beyond the scope of this book.

Hello, SwiftUI

After you've installed Xcode, you'll probably be very eager to try out SwiftUI. So, let's take a dive into SwiftUI and see how it works! Follow these steps:

1. **Launch Xcode.**
2. **Click Create a new Xcode project (see Figure 1-4).**
3. **Select Single View App and click Next (see Figure 1-5).**
4. **In the Product Name field, enter** HelloSwiftUI **(see Figure 1-6).**
5. **In the Organization Name field, enter your name.**
6. **In the Organization Identifier field, enter a unique identifier, such as the reverse domain name of your company.**

7. **From the User Interface drop-down list, select SwiftUI.**
8. **Click Next and save the project to a location on your Mac.**

   You should see the project created for you (see Figure 1-7). The `ContentView.swift` file contains the UI for your application's main screen.

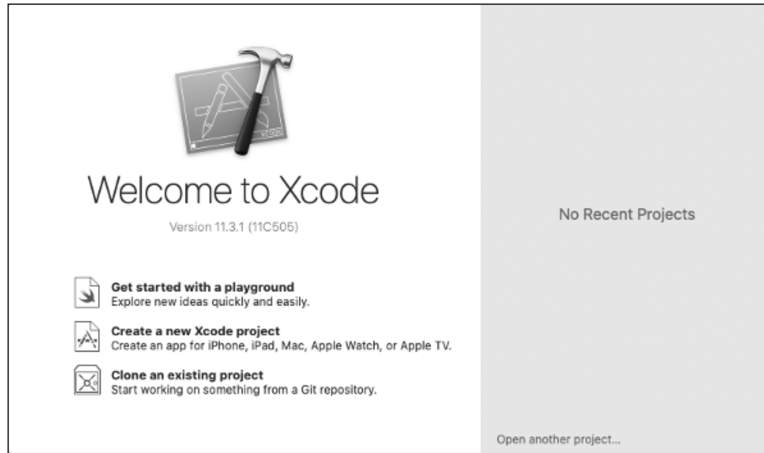

FIGURE 1-4: Launching Xcode.

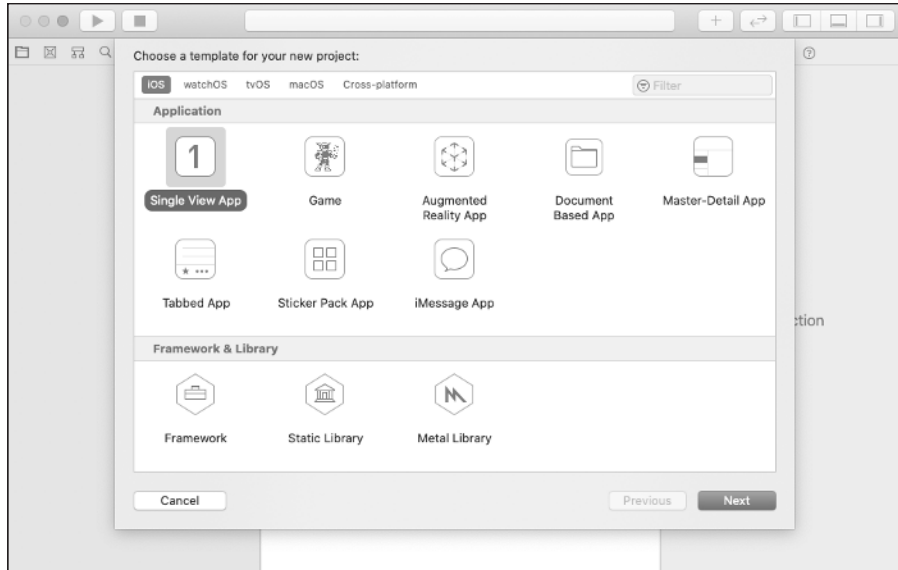

FIGURE 1-5: Selecting the Single View App project type.

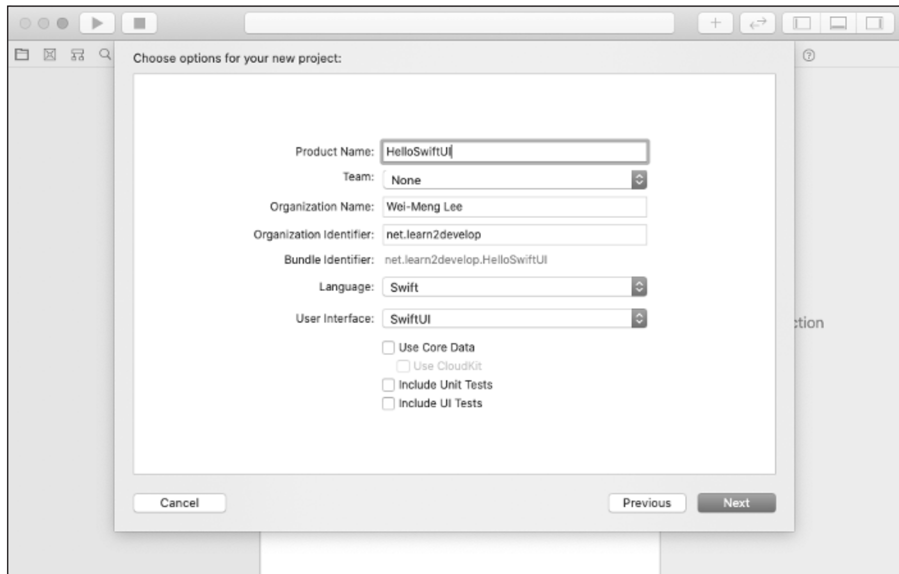

FIGURE 1-6: Naming the project.

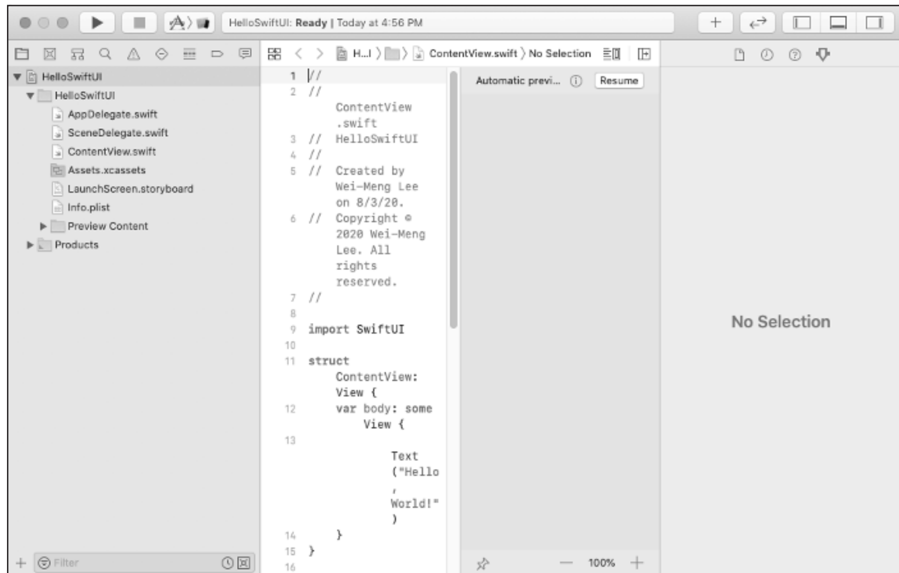

FIGURE 1-7: Viewing the project that you've created.

Automatically previewing your user interface using the canvas

By default, you should see the Inspector window on the right side of the Xcode window. For building your UI using SwiftUI, you usually don't need the Inspector window, so you can dismiss it to gain more screen estate for previewing your UI

using the canvas. To dismiss the Inspector window, click the button on the upper-right corner of Xcode (see Figure 1-8).

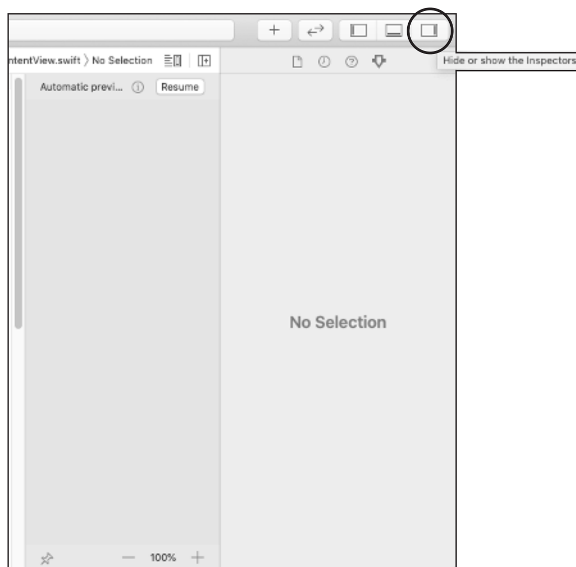

FIGURE 1-8: Dismissing the Inspector window.

With the Inspector window dismissed, you should now see the canvas on the right side of Xcode (see Figure 1-9). The canvas lets you preview the UI of your application without needing to run the application on the iPhone Simulator or a real device.

TIP

If you don't see the canvas, you can bring it up again through the Editor ➪ Canvas menu.

To preview your UI, click the Resume button on the canvas. You should now be able to see the preview (see Figure 1-10).

TIP

If you don't see the Resume button, make sure you're running macOS Catalina (10.15) or later.

Now let's modify the `ContentView.swift` file with the code that you've seen earlier (see Figure 1-11).

You may notice that the automatic preview has paused. This sometimes happen when the file you're previewing has some changes that caused the containing module to be rebuilt. When that happens, click the Restore button, and you should see the preview again (see Figure 1-12).

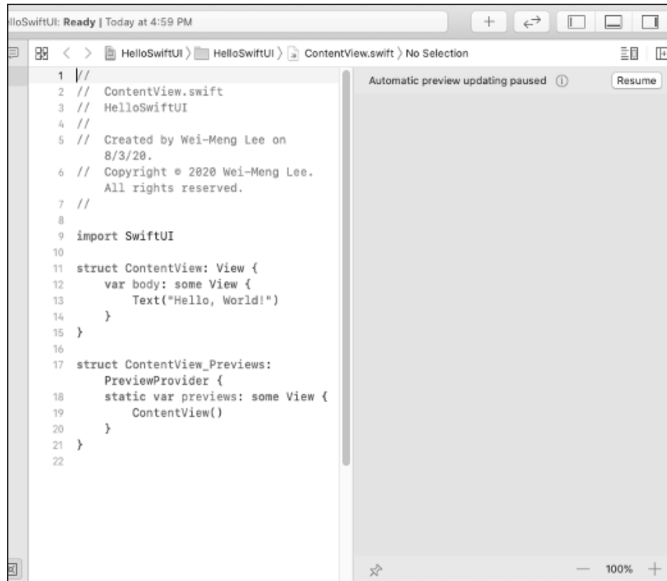

FIGURE 1-9: The canvas allows you to preview your application without deploying it on the iPhone Simulator or a real device.

FIGURE 1-10: Previewing your app on the canvas.

FIGURE 1-11: Modifying the `ContentView.swift` file.

FIGURE 1-12: The preview is updated to reflect the changes in the code.

If you change the color of the `Text` view (within the `Button` view) to blue, you should see the changes automatically reflected in the preview:

```
                Text("Submit")
                    .padding(EdgeInsets(
                        top: 10, leading: 10,
                        bottom: 10, trailing: 10))
                    .background(Color.blue)
```

TIP

The automatic update feature of preview doesn't always work. There are times where you have to click Try Again to rebuild the preview (see Figure 1-13).

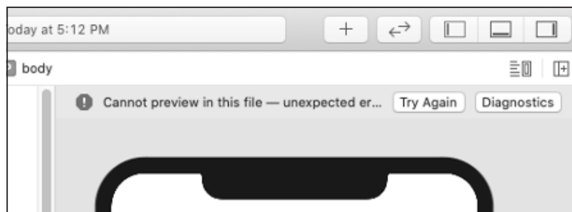

FIGURE 1-13: Occasionally you have to click the Try Again button to update the preview.

Working with Live Preview

Your code will change the text on the label when the button is clicked (or tapped on a real device). However, if you try clicking the button on the preview canvas, there is no reaction. This is because the preview canvas only allows previewing your UI — it doesn't run your application. To run the application, you need to click the Live Preview button (see Figure 1-14).

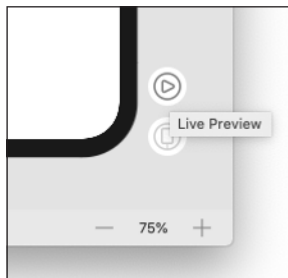

FIGURE 1-14: Clicking the Live Preview button allows you to run your application directly on the canvas.

When the Live Preview mode is turned on, the background of the simulator will turn dark (see the left side of Figure 1-15). You can now click on the button and the text on the label will be updated (see the right side of Figure 1-15).

FIGURE 1-15: Testing your application in Live Preview mode.

Generating different previews

Notice this block of code at the bottom of `ContentView.swift`?

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

The `ContentView_Previews` struct conforms to the `PreviewProvider` protocol. This protocol produces view previews in Xcode so that you can preview your UI created in SwiftUI without needing to explicitly run the application on the iPhone Simulator or

real devices. Essentially, it controls what you see on the preview canvas. As an example, if you want to preview how your UI will look like on an iPhone SE device, you can modify the `ContentView_Previews` struct as follows (see Figure 1-16):

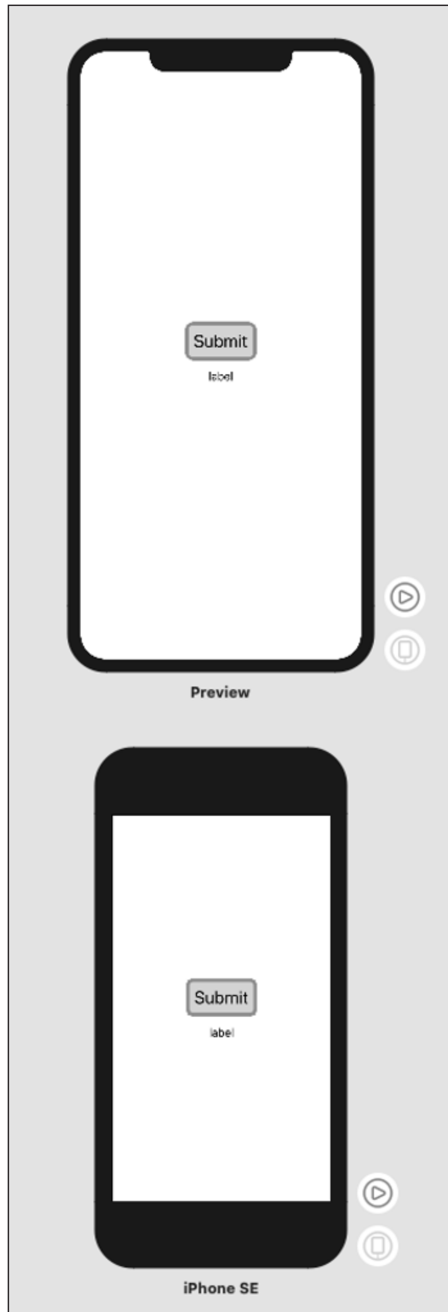

FIGURE 1-16: Previewing the UI on two iOS devices — the latest iPhone and an iPhone SE.

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

The Gory Details

Now that you've seen how to get started with SwiftUI, let's take a moment to examine the various files created in the project and see how the various parts connect.

In your project, notice that you have the following files created (see Figure 1-17):

- `AppDelegate.swift`
- `SceneDelegate.swift`
- `ContentView.swift` (this is the file that you've been modifying to create the UI of your iOS application)
- `Info.plist`

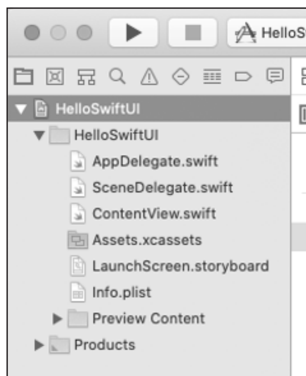

FIGURE 1-17: The content of the project created.

Info.plist

Let's take a look at the `Info.plist` file first (see Figure 1-18). In particular, look at the key named `Application Scene Manifest`.

Within the `Application Scene Manifest` key, you have the following keys:

- `Enable Multiple Windows`: This is set to `NO` by default. You can set this to `YES` if you're building apps for iPadOS and macOS.
- `Application Session Role`: An array that contains a list of dictionary objects. The default object contains a key named `Delegate Class Name` that points to the `SceneDelegate.swift` file.

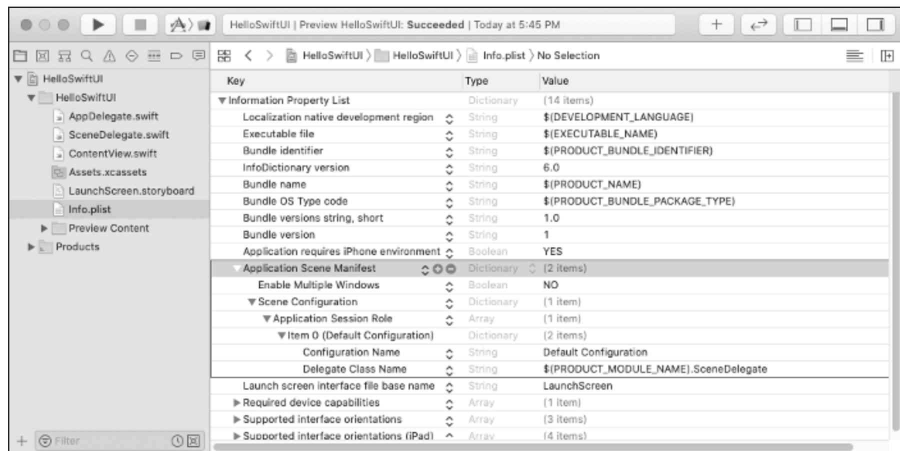

FIGURE 1-18: Examining the items in the `Info.plist` file.

AppDelegate.swift

`AppDelegate.swift` is the place where you write code to handle an application's launch, going into the background, coming to the foreground, and other activities.

`AppDelegate.swift` has three main functions:

- `application(:didFinishLaunchingWithOptions) -> Bool`: This function is called when the application is launched. You can use this function to perform your setup for the app when it's launched.

- `application(: configurationForConnecting:options:) -> UISceneConfiguration`: This function is called whenever your app is needed to supply a new scene. Here, it returns the default item in the dictionary named `Default Configuration`:

```
func application(_ application: UIApplication,
    configurationForConnecting connectingSceneSession:
    UISceneSession, options: UIScene.ConnectionOptions) ->
    UISceneConfiguration {

    // Called when a new scene session is being
    //created. Use this method to select a
    //configuration to create the new scene with.
    return UISceneConfiguration(
        name: "Default Configuration",
        sessionRole: connectingSceneSession.role)
}
```

TIP

A *scene* is an object that represents one instance of your app's user interface.

- `application(:didDiscardSceneSessions:)`: This function is called whenever a user discards a scene (such as swiping it away in the multitasking window).

SceneDelegate.swift

Whereas the `AppDelegate.swift` file is responsible for handling your app life cycle, the `SceneDelegate.swift` file is responsible for your scene's life cycle.

The `SceneDelegate.swift` file contains the following default functions:

- `scene(_:willConnectTo:options:)`
- `sceneDidDisconnect(_:)`
- `sceneDidBecomeActive(_:)`
- `sceneWillResignActive(_:)`
- `sceneWillEnterForeground(_:)`
- `sceneDidEnterBackground(_:)`

The `scene(_:willConnectTo:options:)` function is called when a scene is added to the app (in simple terms, when your UI is shown). Here, you load the content of

the file named ContentView (which is what you've modified earlier in the ContentView.swift file):

```
func scene(_ scene: UIScene, willConnectTo session:
    UISceneSession, options connectionOptions:
    UIScene.ConnectionOptions) {

    let contentView = ContentView()
    if let windowScene = scene as? UIWindowScene {
        let window = UIWindow(windowScene:
            windowScene)
        window.rootViewController =
            UIHostingController(rootView: contentView)
        self.window = window
        window.makeKeyAndVisible()
    }
}
```

In short, you use AppDelegate.swift to perform setup needed for the duration of the app. You also use it to handle events that focus on the app, as well as registered for external services like push notifications. The SceneDelegate.swift, on the other hand, is designed to handle events for multi-window OS (iPadOS), which supports multiple instances of your app's UI.