CHAPTER EIGHTEEN: INTRODUCTION TO SWIFTUI

I. INTRODUCTION

SwiftUI was announced at Apple WWDC 2019. It provides a modern way to declare user interfaces for any Apple platform. Developers can create beautiful, dynamic applications faster and easier than ever before.

So far in this book, we have been using storyboards to define user interfaces (views). These storyboards are actually XML files that can be displayed and manipulated in a GUI provided by Xcode. When the storyboard was first announced, it became quite popular really fast. But as the usage became more and more demanding and the views in mobile iOS apps become larger and more complex, storyboards have some serious performance issues (even with the latest powerful flagship iPhones). Developers can have trouble implementing the layouts when there are a lot of element with interconnected logical relations using storyboards. Sometimes storyboards can lead to test and deployment UI issues when the UI tests aren't reliable enough.

In some UI frameworks like Qt in Python/C++ or Swing in Java, the UI is mostly designed by code. It is a little harder to create the UI but it's definitely much more reliable and

controllable. This is the idea of SwiftUI: instead of using storyboards, we define user interfaces by code in some Swift files.

There are some great online tutorials to get you started in SwiftUI. The one that I found that is the easiest to understand is the official SwiftUI tutorial by Apple. You can go to their website to check it out:

https://developer.apple.com/tutorials/swiftui

In this chapter we will build a Landmarks application step by step. You will need the latest Xcode version that supports SwiftUI. Go to **menu bar > Xcode > About Xcode** to check it out. I tested it out with the version 11.2.1. If your Xcode is version 10 or older, you should consider updating your IDE. If your macOS is earlier than Catalina 10.15, you should also updating your OS.

II. CREATING AND COMBINING VIEWS

Let's create a SwiftUI project from scratch! Open Xcode and either click Create a new Xcode project in Xcode's startup window, or choose File > New > Project.

In the template selector, select **iOS** as the platform, select the **Single View App** template, and then click **Next**.

Enter “Landmarks” as the Product Name, select **SwiftUI** for the user interface, and click **Next**. Choose a location to save the Landmarks project on your Mac.

In the Project navigator, select ContentView.swift. By default, SwiftUI view files declare two structures. The first structure conforms to the View protocol and describes the view’s content and layout. The second structure declares a preview for that view.

In the canvas, click Resume to display the preview. If the canvas isn’t visible, select Editor > Editor and Canvas to show it. You will see a blank page with a “Hello, World!” text in it.

Let's take took at the code: at line 5, we declare a text view of text "Hello SwiftUI" in the body of a content view. Change it to "Hello SwiftUI!" and the preview should automatically update to reflect your changes like this:

In the storyboards, we are able to modify components' attributes in the identity inspector. In SwiftUI, it's much easier to do! You can customize a view's display by changing your code, or by using the inspector to discover what's available and to help you write code.

As you build the Landmarks app, you can use any combination of editors: the source editor, the canvas, or the inspectors. Your code stays updated, regardless of which tool you use.

Now, you'll customize the text view using the inspector. In the preview, Command-click the greeting to bring up the structured editing popover, and choose **Inspect**. The popover shows different attributes that you can customize, depending on the type of view you inspect.

Use the inspector to change the text to "Turtle Rock", the name of the first landmark you'll show in your app.

Change the Font modifier to **Title**. This applies the system font to the text so that it responds correctly to the user's preferred font sizes and settings.

Look at the code in ContentView.swift, Xcode automatically added the font and updated the text for the Text component. To customize a SwiftUI view, you call methods called *modifiers*. Modifiers wrap a view to change its display or other properties. Each modifier returns a new view, so it's common to chain multiple modifiers, stacked vertically.

For example, edit the code by hand to add the foregroundColor(.green) modifier; this changes the text's color to green. The content view declaration should look like this:

```
struct ContentView: View {
    var body: some View {
        Text("Turtle Rock")
            .font(.title)
            .foregroundColor(.green)
    }
}
```

The greatest thing about SwiftUI is that your code is always the source of truth for the view. When you use the inspector to change or remove a modifier, Xcode updates your code immediately to match.

This time, open the inspector by Command-clicking on the Text declaration in the code editor, and then choose **Inspect** from the popover. Click the **Color** pop-up menu and choose **Inherited** to change the text color to black again.

Notice that Xcode updates your code automatically to reflect the change, removing the foregroundColor(.green) modifier.

III. COMBINE VIEWS USING STACKS

Beyond the title view you created in the previous section, you'll add text views to contain details about the landmark, such as the name of the park and state it's in. When creating a SwiftUI view, you describe its content, layout, and behavior in the view's body property; however, the body property only returns a single view. You can combine and embed multiple views in *stacks*, which group views together horizontally, vertically, or back-to-front.
In this section, you'll use a vertical stack to place the title above a horizontal stack that contains details about the park.

You can use Xcode's structured editing support to embed a view in a container view, open an inspector, or help with other useful changes.

Command-click the text view's initializer to show the structured editing popover, and then choose **Embed in VStack**.

Next, you'll add a text view to the stack by dragging a Text view from the library. Open the library by clicking the plus button (+) at the top-right of the Xcode window, and then drag a Text view to the place in your code immediately after the "Turtle Rock" text view.

Replace the Text view's placeholder text with **Joshua Tree National Park**. Now let's customize the location to match the desired layout. Set the location's font to subheadline by adding **.font(.subline)** after the text declaration.

By default, stacks center their contents along their axis and provide context-appropriate spacing. Edit the VStack initializer to align the views by their leading edges by adding the parameter **“alignment: .leading”** to the argument of the VStack constructor. The content view should look like this:

```
struct ContentView: View {
    var body: some View {
        VStack(alignment: .leading) {
            Text("Turtle Rock")
                .font(.title)
            Text("Joshua Tree National Park")
                .font(.subheadline)
        }
    }
}
```

Next, you’ll add another text view to the right of the location, this for the park’s state. In the canvas, Command-click **Joshua Tree National Park**, and choose **Embed in HStack**. Add a new text view after the location, change the placeholder text to the park’s state, and then set its font to subheadline.

To direct the layout to use the full width of the device, separate the park and the state by adding a Spacer to the horizontal stack holding the two text views. A *spacer* expands to make its containing view use all of the space of its parent view, instead of having its size defined only by its contents.

Finally, use the padding() modifier method to give the landmark’s name and details a little more space.

IV. CREATE A CUSTOM IMAGE VIEW

With the name and location views all set, the next step is to add an image for the landmark. Instead of adding more code in this file, you'll create a custom view that applies a mask, border, and drop shadow to the image.

Let's start by adding an image to the project's asset catalog. Find turtlerock.png in the project files' **Resources** folder; drag it into the asset catalog's editor. Xcode creates a new image set for the image.

Next, you'll create a new SwiftUI view for your custom image view. Choose **File > New > File** to open the template selector again. In the **User Interface** section, select **SwiftUI View** and click **Next**. Name the file CircleImage.swift and click **Create**.

In the created file, you can see that the ImageView same structure as ContentView. Let's insert the image and modify its display to match the desired design. Replace the text view with the image of Turtle Rock by using the Image(_:) initializer.

Add a call to clipShape(Circle()) to apply the circular clipping shape to the image. The Circle type is a shape that you can use as a mask, or as a view by giving the circle a stroke or fill. Create another circle with a gray stroke, and then add it as an overlay to give the image a border. Next, add a shadow with a 10 point radius. Finally, switch the border color to white.

You will have a nicely rounded image of turtle rock:

V. USE UIKIT AND SWIFTUI VIEWS TOGETHER

Now you're ready to create a map view. You can use the MKMapView class from MapKit to render the map. To use UIView subclasses from within SwiftUI, you wrap the other view in a SwiftUI view that conforms to the UIViewRepresentable protocol. SwiftUI includes similar protocols for WatchKit and AppKit views.

To get started, you'll create a new custom view that can present an MKMapView. Choose **File > New > File**, select **iOS** as the platform, select the **SwiftUI View** template, and click **Next**. Name the new file MapView.swift and click **Create**.

Add an import statement for MapKit, and declare UIViewRepresentable conformance for the MapView type. Don't worry about the error that Xcode displays; you'll fix that in the next few steps.

The UIViewRepresentable protocol has two requirements you need to add: a **makeUIView(context:)** method that creates an MKMapView, and an **updateUIView(_:context:)** method that configures the view and responds to any changes.

Replace the body property with a makeUIView(context:) method that creates and returns an empty MKMapView.

```
import SwiftUI
import MapKit

struct MapView: UIViewRepresentable {
    func makeUIView(context: Context) -> MKMapView {
        MKMapView(frame: .zero)
    }
}

struct MapView_Previews: PreviewProvider {
    static var previews: some View {
        MapView()
    }
}
```

Create an updateUIView(_:context:) method that sets the map view's region to the correct coordinates to center the map on Turtle Rock.

```
func updateUIView(_ uiView: MKMapView, context: Context) {
    let coordinate = CLLocationCoordinate2D(
        latitude: 34.011286, longitude: -116.166868)
    let span = MKCoordinateSpan(latitudeDelta: 2.0, longitudeDelta: 2.0)
    let region = MKCoordinateRegion(center: coordinate, span: span)
    uiView.setRegion(region, animated: true)
}
```

When previews are in static mode, they only fully render SwiftUI views. Because MKMapView is a UIView subclass, you'll need to switch to a live preview to see the map. Click the **Live Preview** button to switch the preview to live mode. You might need to click the **Try Again** or **Resume** button above your preview. In a moment, you'll see a map of Joshua Tree National Park, home of Turtle Rock.

VI. COMPOSE THE DETAIL VIEW

You now have all of the components you need — the name and place, a circular image, and a map for the location. With the tools you've used so far, combine your custom views to create the final design for the landmark detail view.

In the Project navigator, select the ContentView.swift file. Embed a VStack that holds the three text views in another VStack. Add your custom MapView to the top of the stack. Set the size of the MapView with frame(width:height:). When you specify only the height parameter, the view automatically sizes to the width of its content. In this case, MapView expands to fill the available space.

Then click the **Live Preview** button to see the rendered map in the composed view. You can continue editing the view while showing a Live Preview.

Next, add the CircleImage view to the stack. To layer the image view on top of the map view, give the image an offset of -130 points vertically, and padding of -130 points from the bottom of the view. These adjustments make room for the text by moving the image upwards.

```
CircleImage()
    .offset(y: -130)
    .padding(.bottom, -130)
```

Add a spacer at the bottom of the outer VStack to push the content to the top of the screen. Finally, to allow the map content to extend to the top edge of the screen, add the edgesIgnoringSafeArea(.top) modifier to the map view.

```
VStack {
    MapView()
        .edgesIgnoringSafeArea(.top)
        .frame(height: 300)

    CircleImage()
        .offset(y: -130)
        .padding(.bottom, -130)

    VStack(alignment: .leading) {
        Text("Turtle Rock")
            .font(.title)
        HStack(alignment: .top) {
            Text("Joshua Tree National Park")
                .font(.subheadline)
            Spacer()
            Text("California")
                .font(.subheadline)
        }
    }
    .padding()

    Spacer()
}
```

SUMMARY

In this chapter, we have worked on these topics:

- Create of views in SwiftUI
- Combine views using stacks
- Customize views
- Compose views

If any of these is unclear to you, please make sure to go back and read the related part or parts before moving on the the next chapter.

For reference, you can download the complete Xcode project from

https://github.com/CarlistleZ/MyiOSTutorial/blob/master/Landmark.zip