Chapter 20/22

Views

- declared as structures that conform to the `View` protocol
    - must contain a `body` computed property

Last Parameter Call Syntax

```
VStack(content: {
    () -> Void in
    Text("Hello World")
}
```

```
VStack(content: {
    Text("Hello World") // removed, () -> Void, since no parameters
}

VStack() {
    Text("Hello World") // last parameter call syntax
}

VStack {
    Text("Hello World") // removed, (), since no parameters
}
```

Assigning Views to Properties

```
struct ContentView: View {
    let carStack = HStack {
        Text("Car Image")
        Image(systemName: "car.fill")
    }

    var body: some View {
        VStack {
            Text("Main Title")
                .font(.largeTitle)
            carStack
        }
    }
}
```

Custom Container Views (ViewBuilders)

```
struct MyVStack<Content: View>: View {
    let content: () -> Content

    init(@ViewBuilder content: @escaping () -> Content) {
        self.content = content
    }

    var body: some View {
        VStack(spacing: 10) {
            content()
        }.font(.largeTitle)
    }
}

MyVStack {
    Text("Text 1")
    Text("Text 2")
    HStack {
        Image(systemName: "star.fill")
        Image(systemName: "star.fill")
        Image(systemName: "star")
    }
}
```

State Objects and Bindings

State Properties

- stores the state of a ***primitive*** property that is local to a view using `@State`
- creates a ***two-way binding*** with the `$` symbol
    - any changes **synchronizes** the values between the ***state property*** and its ***bounded view*** object
- used when you want **user input**

```
@State private var name: String = "" // allows SwiftUI to create a binding w/ that variable
TextField("Name", text: $name) // enables a two-way binding
Text(name) // only references the state property (one-way binding)
```

- ***Subviews (State-Binding Relationship)***
    - any changes **synchronizes** the values between ***sub-views*** the ***state properties*** using `@Binding`

    ```
    var body: some View {
        VStack {
            Information(name: $name, address: $address)
        }
    }

    struct Information: View {
        @Binding var name: String
        @Binding var address: String
    }
    ```

Observable Objects

- represent persistent data ***(non-primitive)*** that is both external/accessible to multiple views
- conforms to the `ObservableObject` protocol
- `@StateObject`
    - any changes updates, **synchronizes**, the values between the ***state object*** and its ***bounded view*** object

> enable binding to a property declared in the view and the property uses a custom type such as classes you defined yourself

- `@Published`
    - permits SwiftUI to listen for changes on the specified property
- `@ObservedObject`
    - any changes updates, **synchronizes**, the values between the ***state object*** and ***sub-views***
    - parameters are not prefixed with the `$` symbol because classes are passed by reference

> enable binding to a property declared in a different view, such as when passed to subviews or navigation links and the property uses a custom type such as classes you defined yourself

Environment Objects

- conforms to the `ObservableObject` protocol
- all **state objects** can be assessed by all its **child views**

> enable binding to a property declared in a containing view via subviews or navigation views and the property is not passed as a

> parameter and is accessible to any contained view using this annotation

- `.environmentObject(...)`
    - initializes the environment object instance which by inserts it into the view hierarchy
    - provided by all views

```
class SpeedSetting: ObservableObject {
    @Published var speed = 0.0
}

struct ContentView: View {
    @StateObject var speedsetting = SpeedSetting()

    var body: some View {
        VStack {
            SpeedControlView()
            SpeedDisplayView()
        }.environmentObject(speedsetting)
    }
}

struct SpeedControlView: View {
    @EnvironmentObject var speedsetting: SpeedSetting

    var body: some View {
        Slider(value: $speedsetting.speed, in: 0...100)
    }
}

struct SpeedDisplayView: View {
    @EnvironmentObject var speedsetting: SpeedSetting

    var body: some View {
        Text("Speed = \(speedsetting.speed)")
    }
}
```

ForEach Structure

- conforms to the `Identifiable` protocol

```
ForEach(<list>) {
    <name> in <View>
}
```

Frames and Geometry Readers

Frames

- allows us to define the dimensions of *SwiftUI views*

Geometry Readers

- allows us to retrieve the entire screen details ***(dimensions/orientation)***
- results in adaptive layouts for devices with different screen sizes and orientations

Customizations

Labels

```
Label("Welcome to SwiftUI", systemImage: "person.circle.fill")
    .font(.largeTitle)
```

Modifiers

- can be chained/wrapped (each returning a compounded modified view with respect to their orders)

```
Text("SafeWalk Volunteer")
    .font(.headline) // headline is an enumeration value
    .foregroundColor(Color.white)
    .padding()
    .background(Color.black)
    .cornerRadius(10)
```

- ***Custom Modifiers***
    - declared as structures that conform to the `ViewModifier` protocol

    - applies the same modifiers on any given view
    - provides a consistent user-interface

```
struct SafeWalkText: ViewModifier {
   func body(content: Content) -> some View {
        content
           .font(.custom("Courier New", size: 30))
           .foregroundColor(Color.white)
           .padding()
           .background(Color.black)
           .cornerRadius(10)
    }
}

Text("SafeWalk Volunteer")
    .modifier(SafeWalkText())
```

Negative Space

- refers to the space around your views
- `Spacer()`
    - expands the negative space relative to other views

```
HStack(spacing: 0) { // describes the default horizontal space between views
    Spacer() // pushes elements all to the right
    Text("Name")
        .frame(width: 100)
        .border(Color.black)
    Spacer() // pushes elements both left and right
    TextField("Name", text: $name)
        .frame(width: 100)
        .border(Color.black)
        .padding(.leading, 20) // controls the amount of space before each view to provide
                               // negative space
    Spacer() // pushes elements all to the left
}
```

Navigations

NavigationView

- defines the scope of the view that is subjected to be replaced by a new view provided as the destination parameter in `NavigationLink`

NavigationLink

- a way to change what is shown on that navigation view

```
// the destination view is shown when the user clicks on the link
NavigationLink(destination: some View) {
    Text("Click Here")
}
```