Day 16:

ContentView is the struct that gets recreated by default in a new SwiftUI application. ContentView conforms to View Protocol. View protocol required a body parameter that is of type ***some View***

Any element can contain a maximum upto 10 children. For example, a Form view can hold 10 children elements. Due to this limitation, we can wrap the children into Group/Section view in order to hold more data.

ContentView_Previews is used to create a Preview on the canvas and is not part of the final xcode build.

NavigationView can be used to add a navigation bar.

```
struct ContentView: View {
    var body: some View {
        NavigationView {
            Form {
                Section {
                    Text("Hello, world!")
                }
            }
            .navigationTitle("SwiftUI")
            .navigationBarTitleDisplayMode(.inline)
        }
    }
}

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

State variables can be used inside the code by mentioning the @State keyword. This is because the ContentView is of type Struct and hence the variables cannot be updated unless they are marked with @State. This make Swift UI store these state variables in a different location.

```
struct ContentView: View {
    @State private var tapCount = 0

    var body: some View {
        Button("Tap Count: \(tapCount)") {
            tapCount += 1
        }
    }
}
```

We can similarly use state variables for textfields to store the value that gets entered into the field. We add $ before the variable name denoting that two way binding is in place i.e, value is updated and also returned

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

    var body: some View {
        Form {
            TextField("Enter your name", text: $name)
            Text("Your name is \(name)")
        }

    }
}
```

Day 17:

Reading text from the user with TextField

When using a value of different type in a textfield or text elements, you need to use the value property along with format property to mention the format of the value.

```
var currencyCode: String {
        if #available(iOS 16, *) {
            return Locale.current.currency?.identifier ?? "USD"
        } else {
            return Locale.current.currencyCode ?? "USD"
        }
    }

    var body: some View {
        Form {
            Section {
                TextField("Amount", value: $checkAmount, format: .currency(code:
currencyCode))
                    .keyboardType(.decimalPad)
            }

            Section {
                Text(checkAmount, format: .currency(code: currencyCode))
            }
        }
    }
```

Creating pickers in a form

```
Picker("Number of people", selection: $numberOfPeople) {
                        ForEach(2..<100) {
                            Text("\($0) people")
                        }
                    }
```

Adding a segment control for tip percentage

```
                Section {
                     Picker("Tip percentage", selection: $tipPercentage) {
                         ForEach(tipPercentages, id: \.self) {
                             Text($0, format: .percent)
                         }
                     }
                     .pickerStyle(.segmented)
                 } header: {
                     Text("How much tip do you want to leave?")
                 }
```

Hiding the keyboard

In order to achieve this we have to first add a @FocusState variable to track and update the focus state of the textfield.

```
    @FocusState private var amountIsFocused: Bool
```

Add this focusstate to textfield by using the focused modifier

```
    TextField("Amount", value: $checkAmount, format: .currency(code: currencyCode))
     .keyboardType(.decimalPad)
     .focused($amountIsFocused)
```

Then add a toolbar item to the NavigationView with a done button to toggle this FocusState to false.

```
.toolbar {
                 ToolbarItemGroup(placement: .keyboard) {
                     Spacer()

                     Button("Done") {
                         amountIsFocused = false
                     }
                 }
             }
```

Day 20:

VStack, HStack and ZStack

Use stacks to arrange views in Vertical, Horizontal, Z-Axis stacks.

```
    var body: some View {
        VStack {
            HStack(alignment: .center, spacing: 20) {
                Text("Hello world!")
            }
            ZStack {
                Text("Hello world!")
                Text("This is another text view")
            }
        }
    }
```

Stack will always fit its content. Use Spacers to arrange the other view accordingly

Colors and Frames

We can use Color views in stacks and assign frames

```
ZStack {
            Color.red
                .frame(width: 200, height: 200)
            Text("Hello world!")
        }
```

We can use Color semantics as well. For example `Color.primary`

Gradients

Linear gradients

A simple gradient can we created using colors array.

```
LinearGradient(gradient: Gradient(colors: [.white, .black]), startPoint: .top, endPoint:
.bottom)
```

A gradient with stop points can created by mentioning the Gradient stops

```
        LinearGradient(gradient: Gradient(stops: [
            Gradient.Stop(color: .white, location: 0.45),
            Gradient.Stop(color: .black, location: 0.55)
        ]), startPoint: .top, endPoint: .bottom)
```

Radial Gradients

Radial gradients can be used to create gradients with a circular effect.

```
RadialGradient(gradient: Gradient(colors: [.blue, .black]), center: .center,
startRadius: 20, endRadius: 200)
```

Angular Gradients

Angular gradient can be used to create gradient with multiple colors at the same time.

```
AngularGradient(gradient: Gradient(colors: [.red, .yellow, .green, .blue,
.purple, .red]), center: .center)
```

Buttons and Images

A simple button can be created by using the Button view, giving it a title and closure for click event.
We can pass a function as well directly to the Button definition.

```
    var body: some View {
        Button("Delete selection", action: executeDelete)
    }

    func executeDelete() {
        print("Button clicked")
    }
```

You can set the button role to destructive or cancel if needed. We can set button style to .bordered or .borderedProminent

```
        Button {
            print("Button tapped")
        } label: {
            Text("Tap me!")
                .padding()
                .foregroundColor(.white)
                .background(.red)
        }
```

```
Button {
            print("Button tapped")
        } label: {
            Label("Edit", systemImage: "pencil")
        }
```

Alerts

```
    @State private var showingAlert = false

    var body: some View {
        Button("Show alert") {
            showingAlert = true
        }
        .alert("Important message", isPresented: $showingAlert) {
            Button("Delete", role: .destructive) { }
            Button("Cancel", role: .cancel) { }
        } message: {
            Text("Please read this")
        }
    }
```