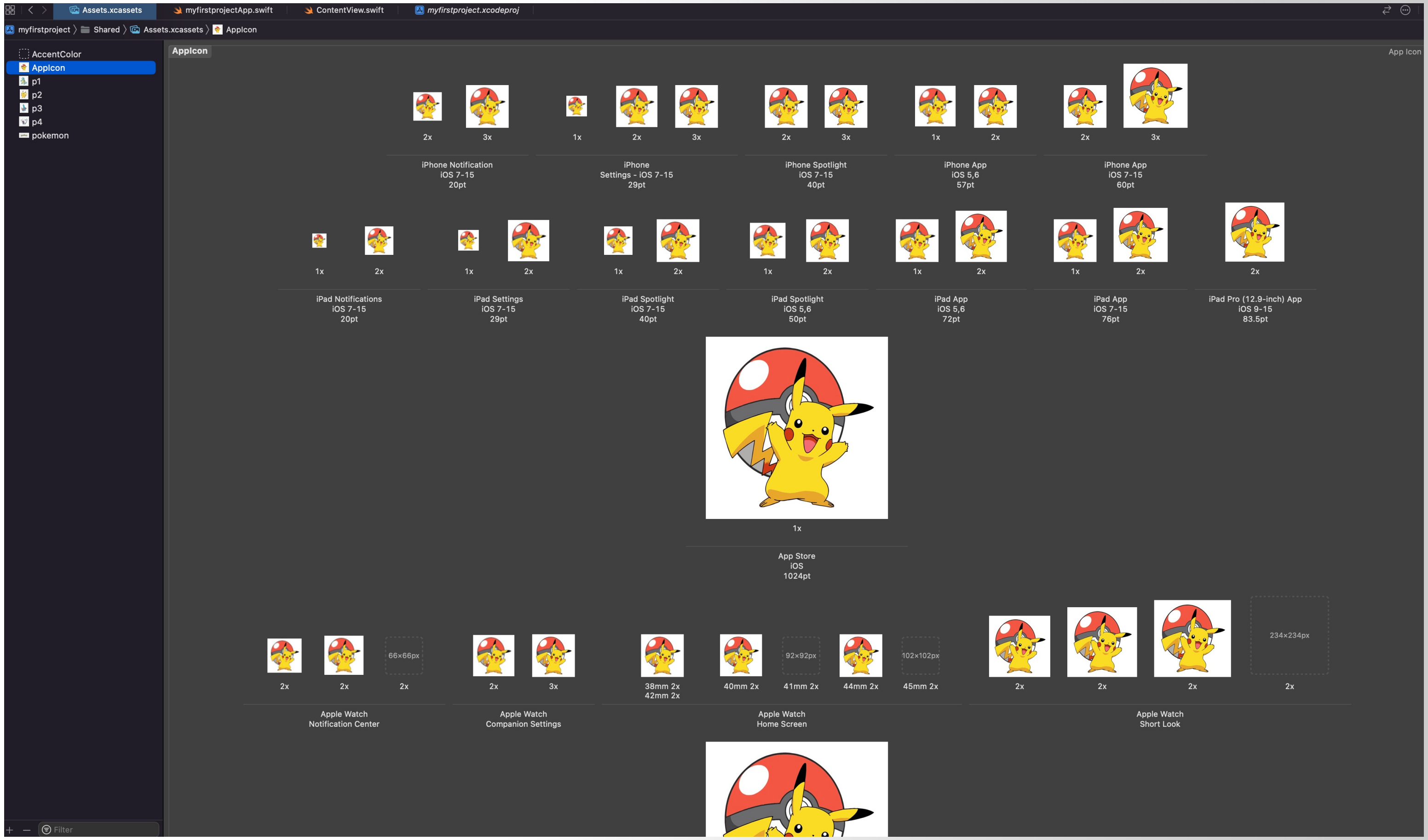
Class 3

CSC 496: iOS App Development
SwiftUI

Si Chen (schen@wcupa.edu)

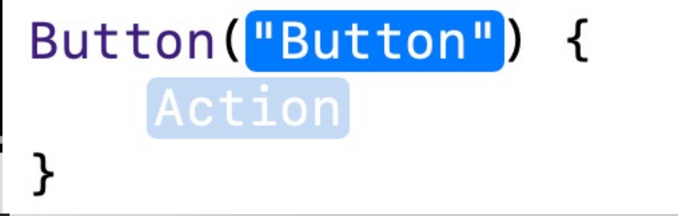
Build the first App



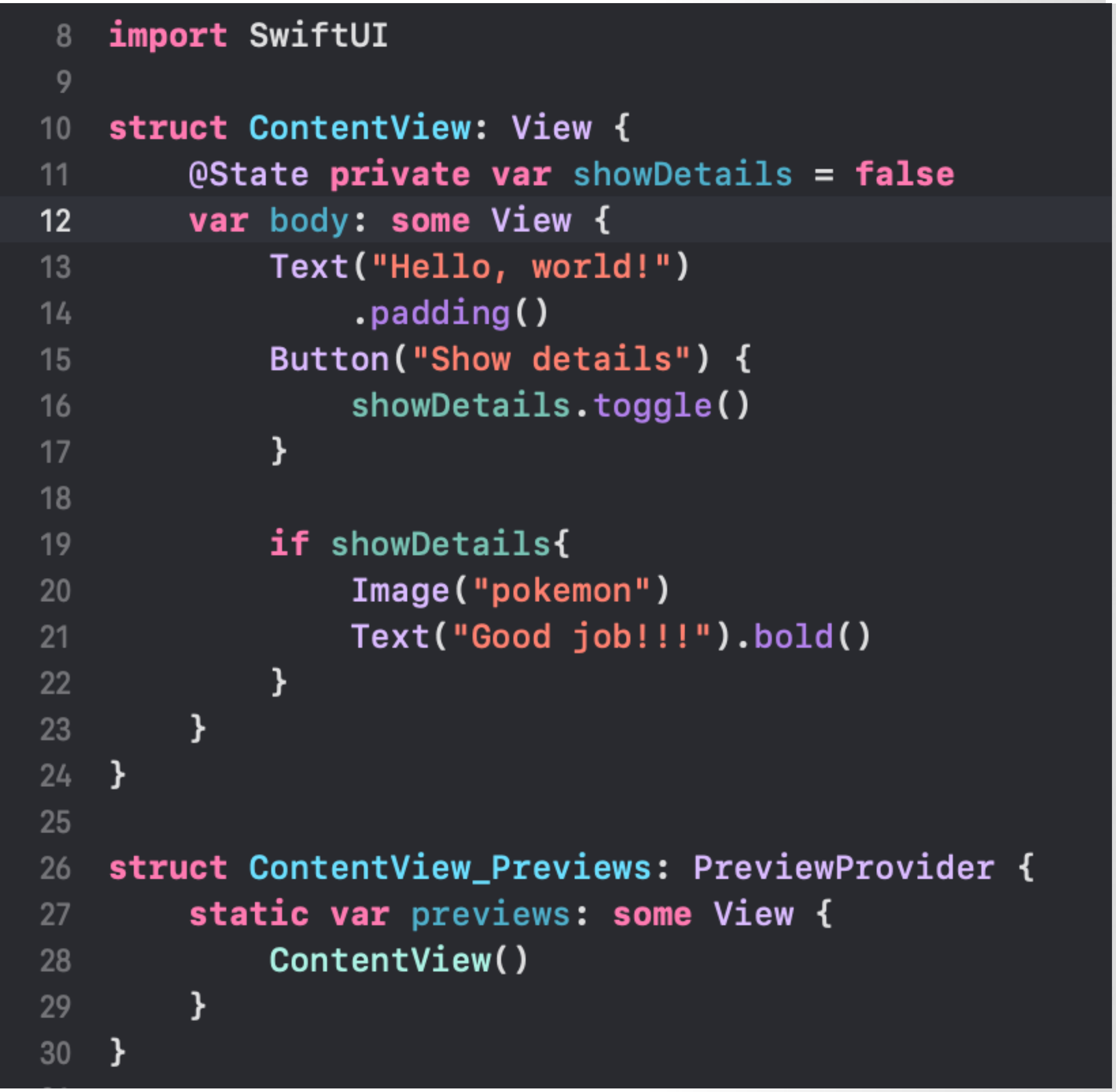
assets.xcassets → AppIcon

generate app icon: https://appicon.co/#app-icon

Create a Button

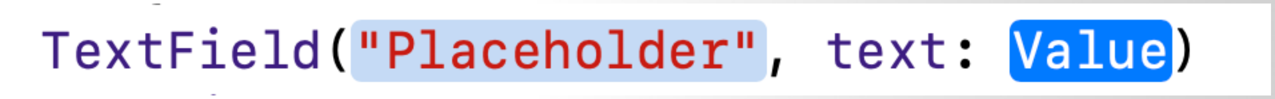


```
Button("Button") {
    Action
}
```

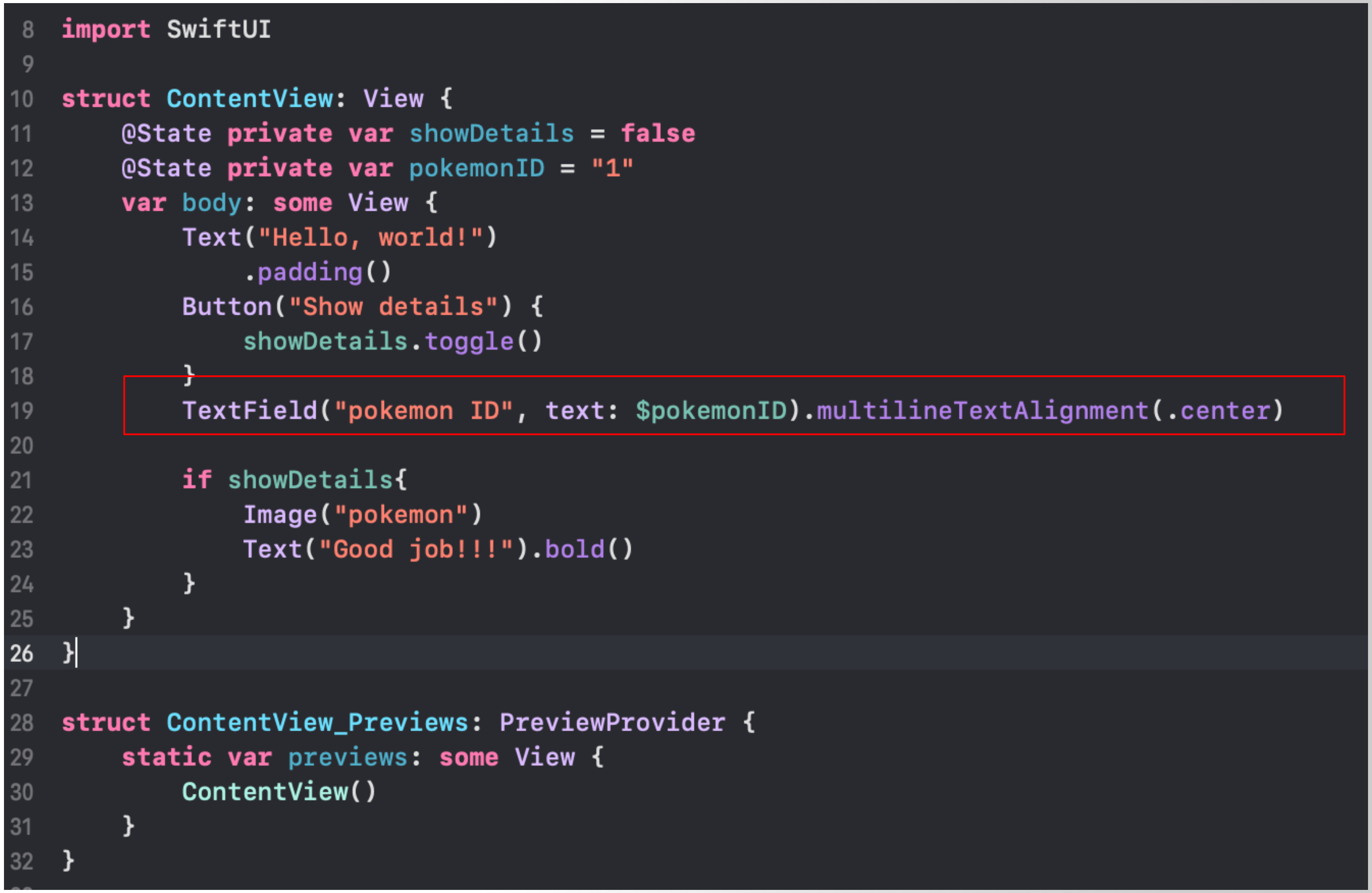


```swift
import SwiftUI

struct ContentView: View {
    @State private var showDetails = false
    var body: some View {
        Text("Hello, world!")
            .padding()
        Button("Show details") {
            showDetails.toggle()
        }

        if showDetails{
            Image("pokemon")
            Text("Good job!!!").bold()
        }
    }
}

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

Create a TextField



```
TextField("Placeholder", text: Value)
```

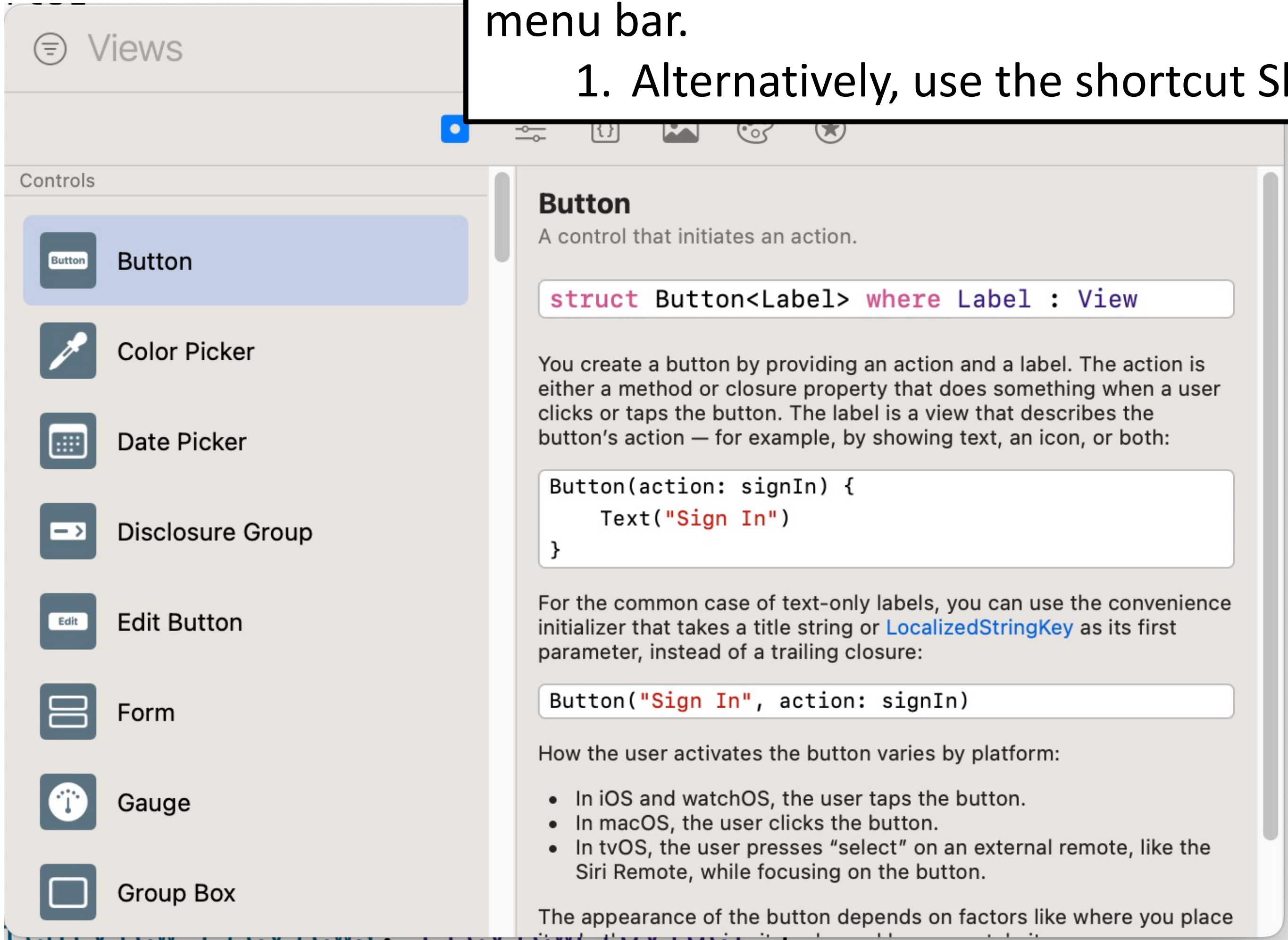


```
import SwiftUI

struct ContentView: View {
    @State private var showDetails = false
    @State private var pokemonID = "1"
    var body: some View {
        Text("Hello, world!")
            .padding()
        Button("Show details") {
            showDetails.toggle()
        }
        TextField("pokemon ID", text: $pokemonID).multilineTextAlignment(.center)

        if showDetails{
            Image("pokemon")
            Text("Good job!!!").bold()
        }
    }
}

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

Views Library Panel

Steps to Open the Library Panel

1. **Open Your Xcode Project**: Launch Xcode and open your SwiftUI view file.
2. **Navigate to Menu**: Go to View -> Show Library in the Xcode menu bar.
    1. Alternatively, use the shortcut Shift + Command + L.



Button

A control that initiates an action.

```
struct Button<Label> where Label : View
```

You create a button by providing an action and a label. The action is either a method or closure property that does something when a user clicks or taps the button. The label is a view that describes the button's action — for example, by showing text, an icon, or both:

```
Button(action: signIn) {
    Text("Sign In")
}
```

For the common case of text-only labels, you can use the convenience initializer that takes a title string or LocalizedStringKey as its first parameter, instead of a trailing closure:

```
Button("Sign In", action: signIn)
```

How the user activates the button varies by platform:

- In iOS and watchOS, the user taps the button.
- In macOS, the user clicks the button.
- In tvOS, the user presses "select" on an external remote, like the Siri Remote, while focusing on the button.

The appearance of the button depends on factors like where you place

An Overview of SwiftUI

- First announced at Apple’s Worldwide Developer Conference in **2019**, SwiftUI is an entirely new approach to developing apps for all Apple operating system platforms.
- Many of the advantages of SwiftUI originate from the fact that it is both ***declarative*** and ***data driven***, topics which will be explained in this class.

Before SwiftUI

- Up until the introduction of SwiftUI, iOS apps were built entirely using **UIKit** together with a collection of associated frameworks that make up the iOS Software Development Kit (SDK).

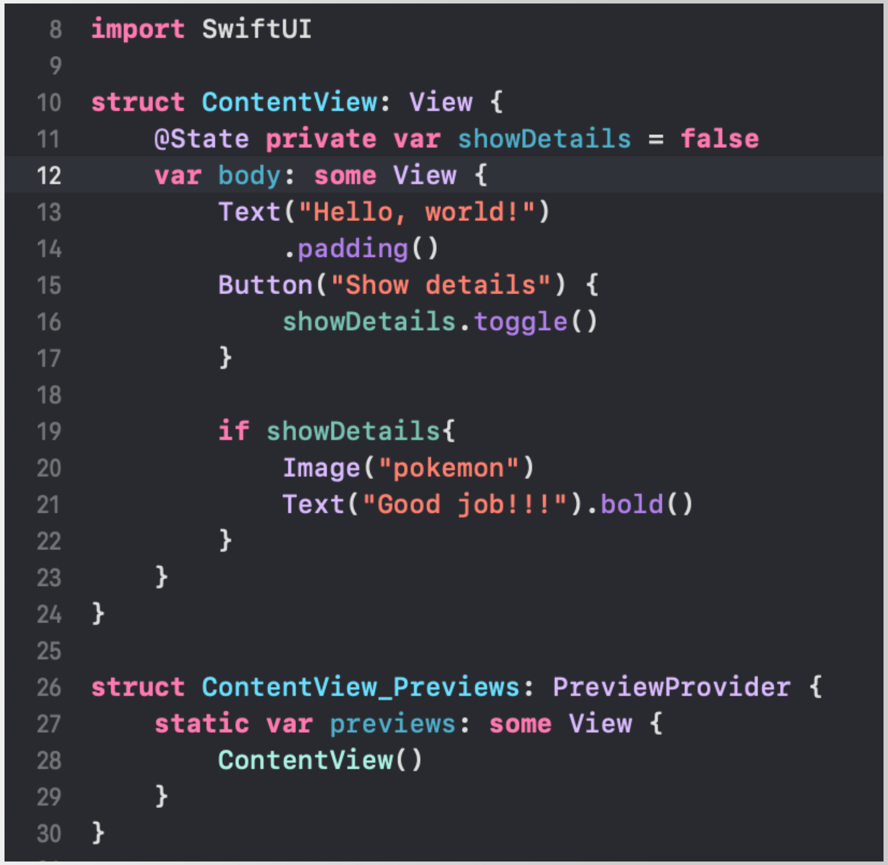
- Back in that time, Xcode includes a tool called **Interface Builder**. :
  - The user interface layout of a scene is designed within Interface Builder by **dragging** components (such as buttons, labels, text fields and sliders).
  - Any components that need to respond to user events (such as a button tap or slider motion) are connected to methods in the app source code where the event is handled.

SwiftUI is Data Driven

- SwiftUI provides several ways to **bind the data model of an app to the user interface components and logic** that provide the app functionality.
- The data model publishes data variables to which other parts of the app can then **subscribe**. Using this approach, changes to the published data are automatically reported to all subscribers.

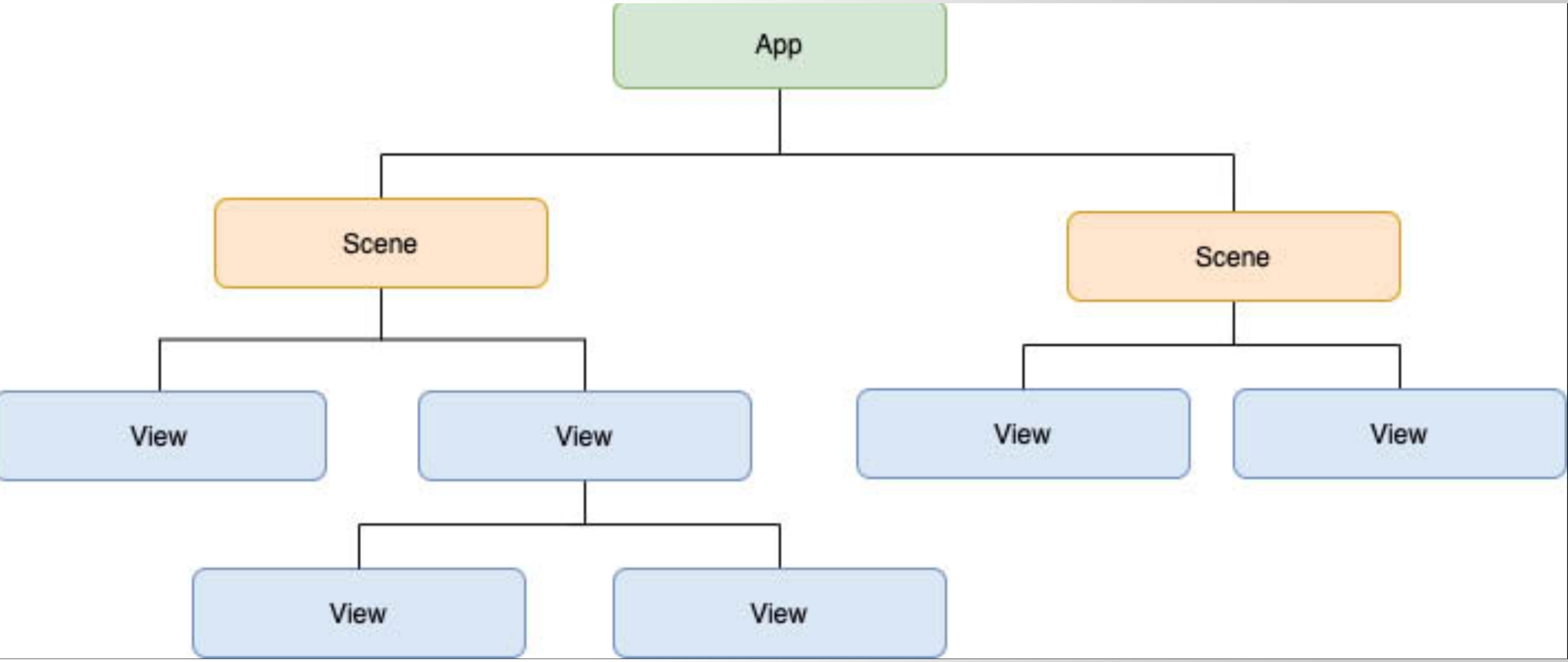
Summary

- **SwiftUI** introduces a different approach to app development than that offered by **UIKit** and **Interface Builder**.
  - SwiftUI allows the user interface to be declared in descriptive terms and then does all the work of deciding the best way to perform the rendering when the app runs.
  - It is also data driven in that data changes drive the behavior and appearance of the app. This is achieved through a publisher and subscriber model.



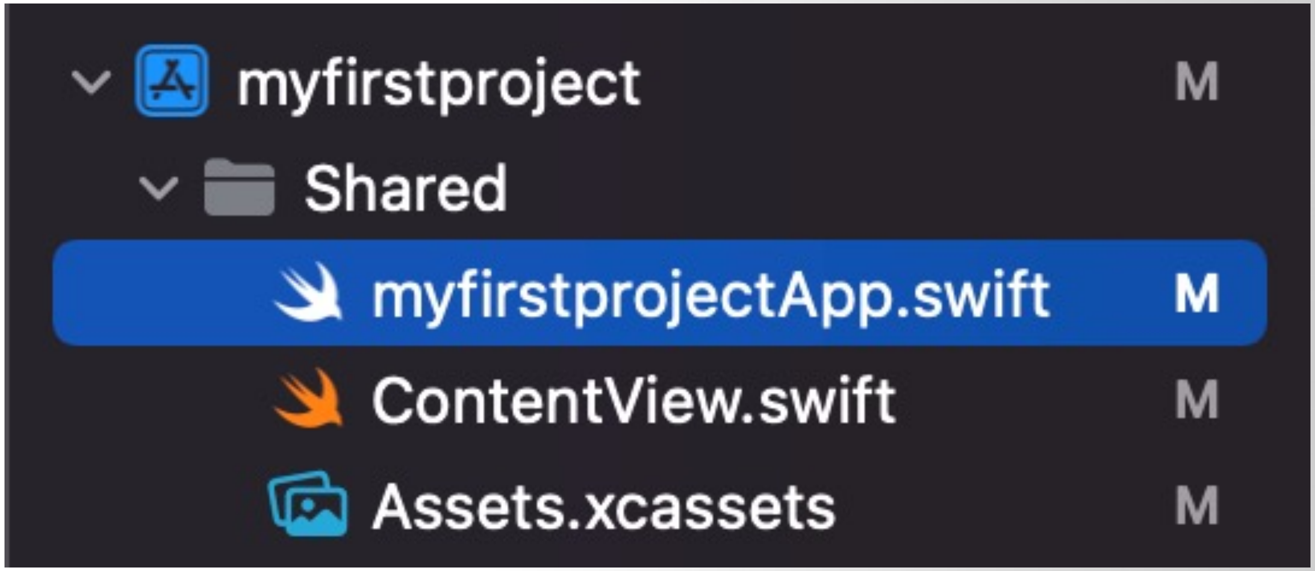
```swift
import SwiftUI

struct ContentView: View {
    @State private var showDetails = false
    var body: some View {
        Text("Hello, world!")
            .padding()
        Button("Show details") {
            showDetails.toggle()
        }

        if showDetails{
            Image("pokemon")
            Text("Good job!!!").bold()
        }
    }
}

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

SwiftUI Architecture



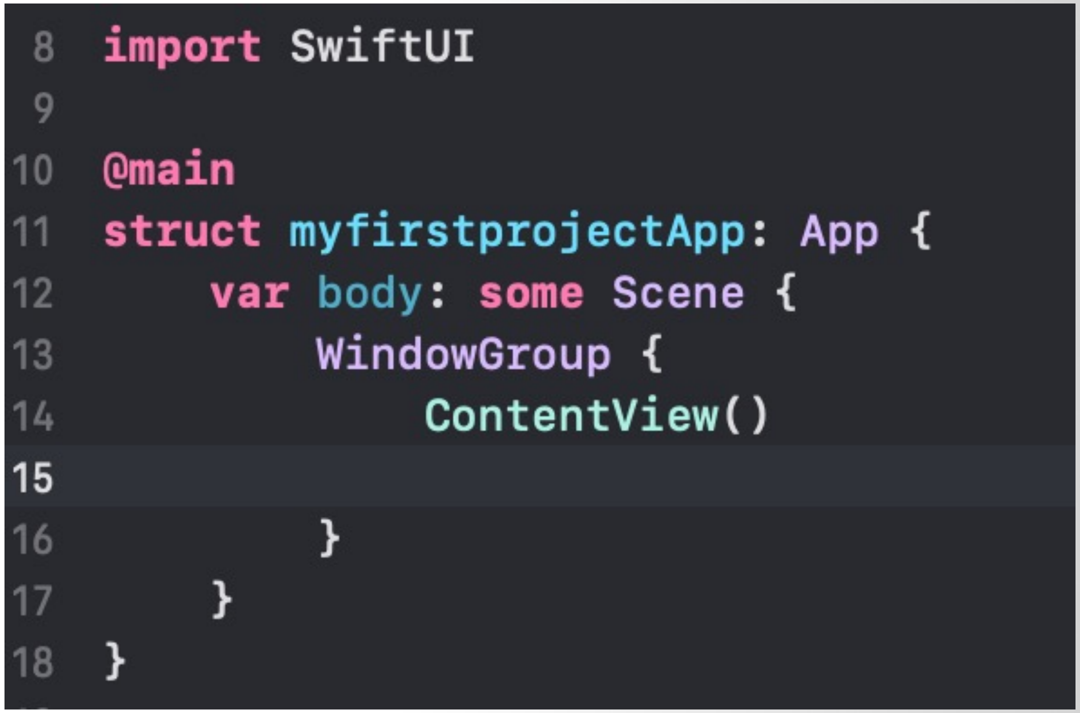
App: the app
Scene: Usually a window takes up the entire device screen
Views: buttons, labels, and text fields…

The Anatomy of a Basic SwiftUI Project



The myfirstprojectApp.swift File

- This file contains the declaration for the App objects

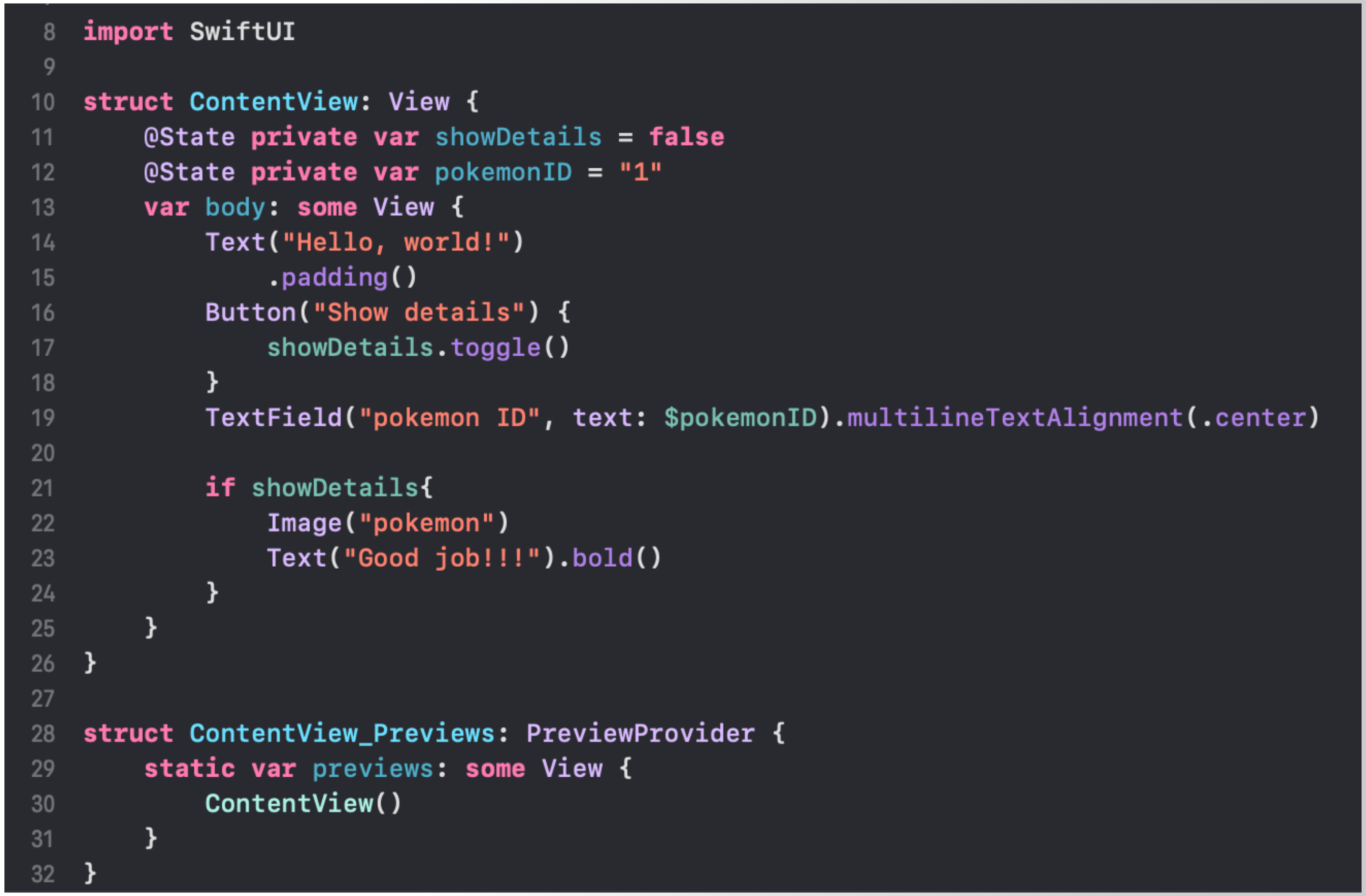


```swift
import SwiftUI

@main
struct myfirstprojectApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()

        }
    }
}
```

As implemented, the declaration returns a Scene consisting of a WindowGroup containing the View defined in the ContentView.swift file. Note that the declaration is prefixed with **@main**. This indicates to SwiftUI that this is the **entry point** for the app when it is launched on a device

The ContentView.swift File

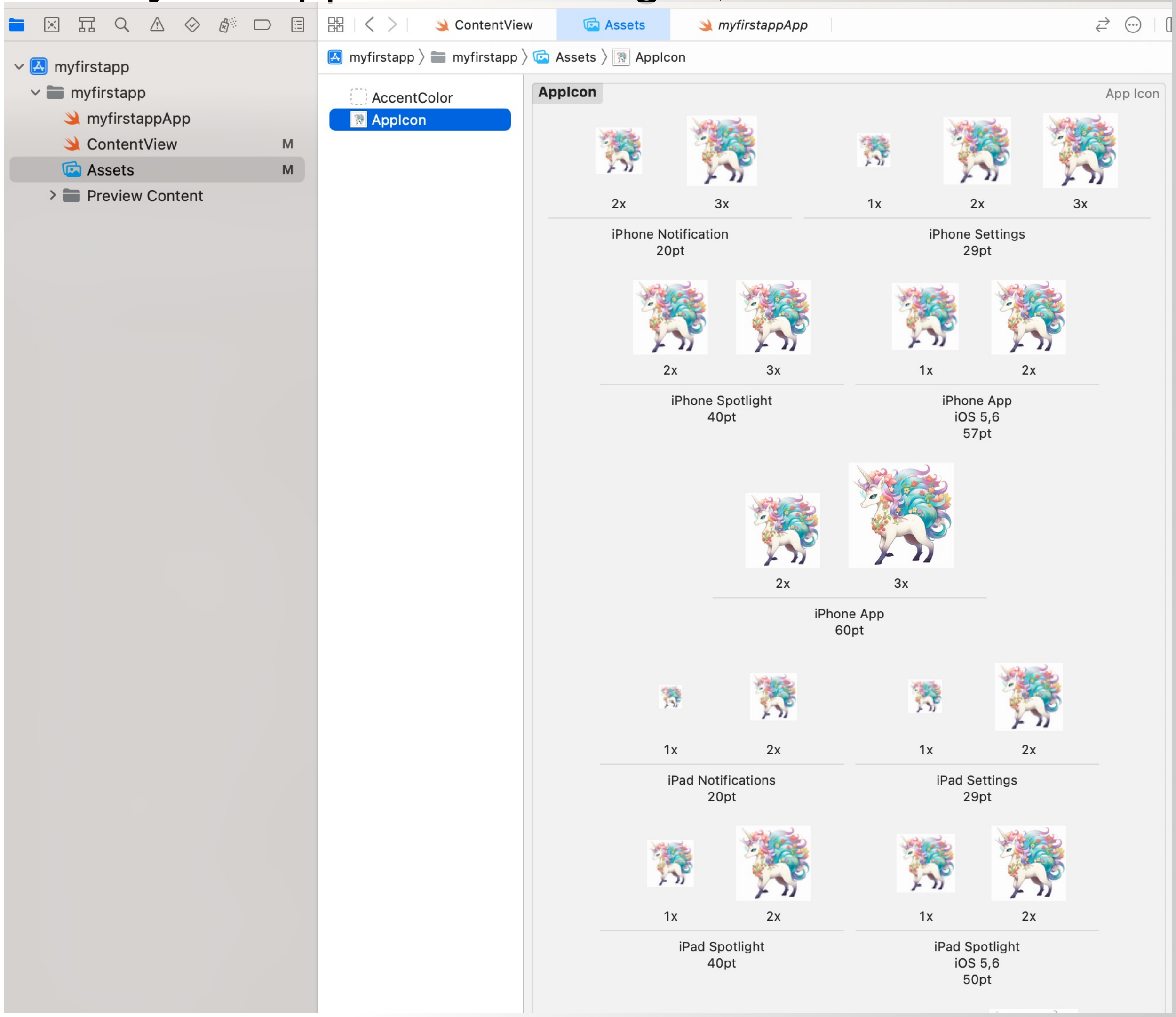
- This file serves as the initial SwiftUI View, which by default displays the first screen that users see upon launching the app. It acts as the primary workspace for most of your SwiftUI development tasks. Initially, it comes pre-populated with a single Text view that displays the message 'Hello, world!'.



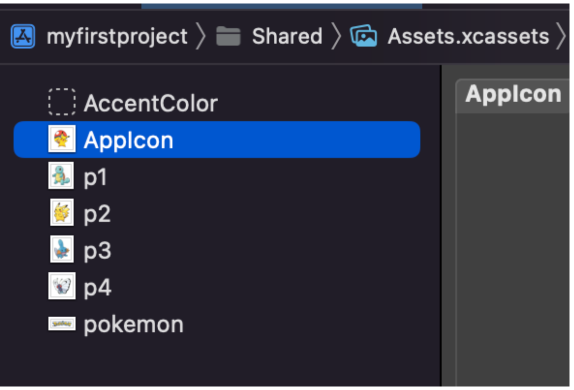
```swift
import SwiftUI

struct ContentView: View {
    @State private var showDetails = false
    @State private var pokemonID = "1"
    var body: some View {
        Text("Hello, world!")
            .padding()
        Button("Show details") {
            showDetails.toggle()
        }
        TextField("pokemon ID", text: $pokemonID).multilineTextAlignment(.center)

        if showDetails{
            Image("pokemon")
            Text("Good job!!!").bold()
        }
    }
}

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

Assets.xcassets

- The Assets.xcassets folder contains the asset catalog that is used to store resources used by the app such as images, icons and colors.

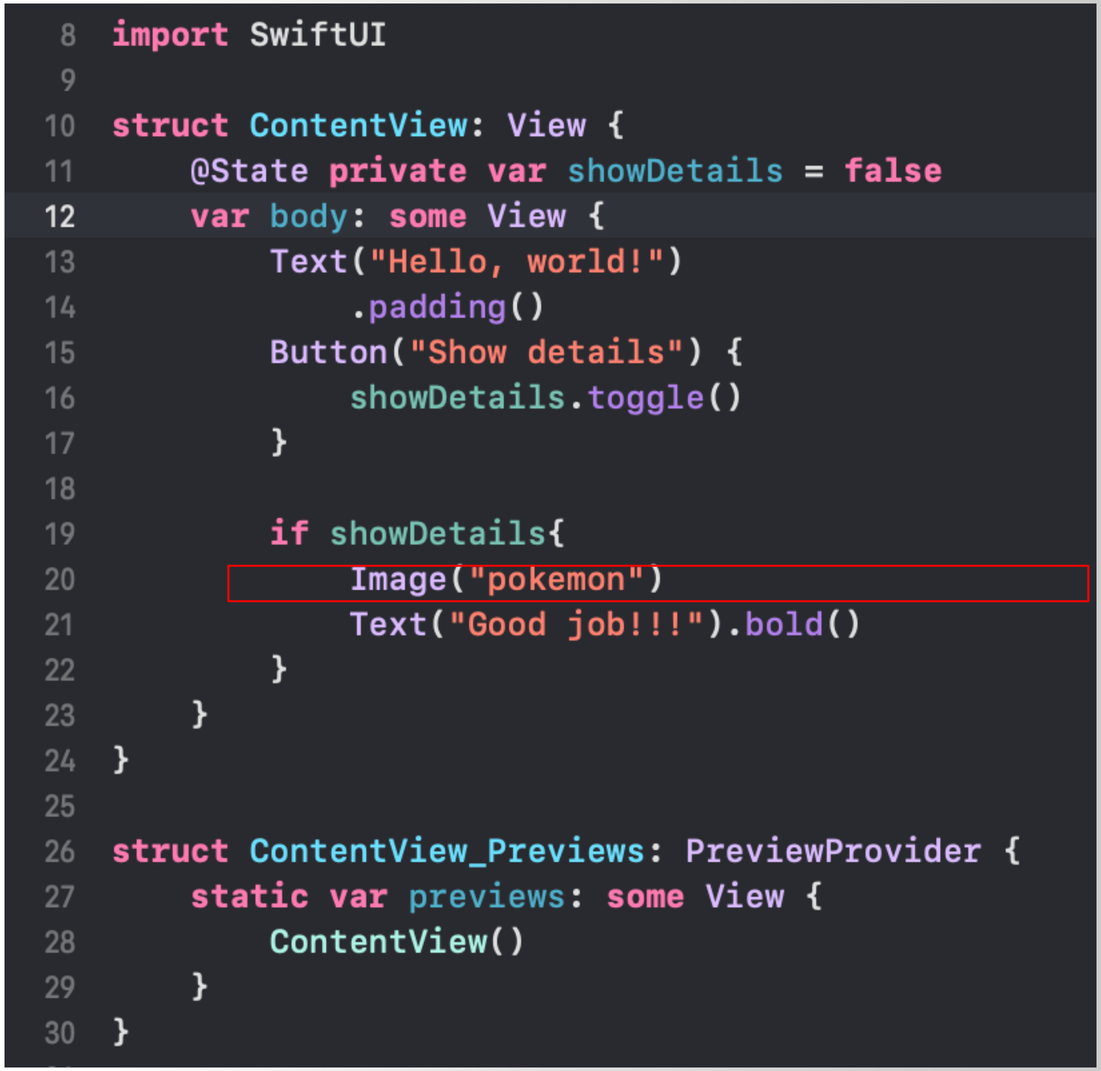


Add Image



myfirstproject 〉 Shared 〉 Assets.xcassets 〉

- AccentColor
- AppIcon
- p1
- p2
- p3
- p4
- pokemon



```
import SwiftUI

struct ContentView: View {
    @State private var showDetails = false
    var body: some View {
        Text("Hello, world!")
            .padding()
        Button("Show details") {
            showDetails.toggle()
        }

        if showDetails{
            Image("pokemon")
            Text("Good job!!!").bold()
        }
    }
}

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

Creating a Basic View

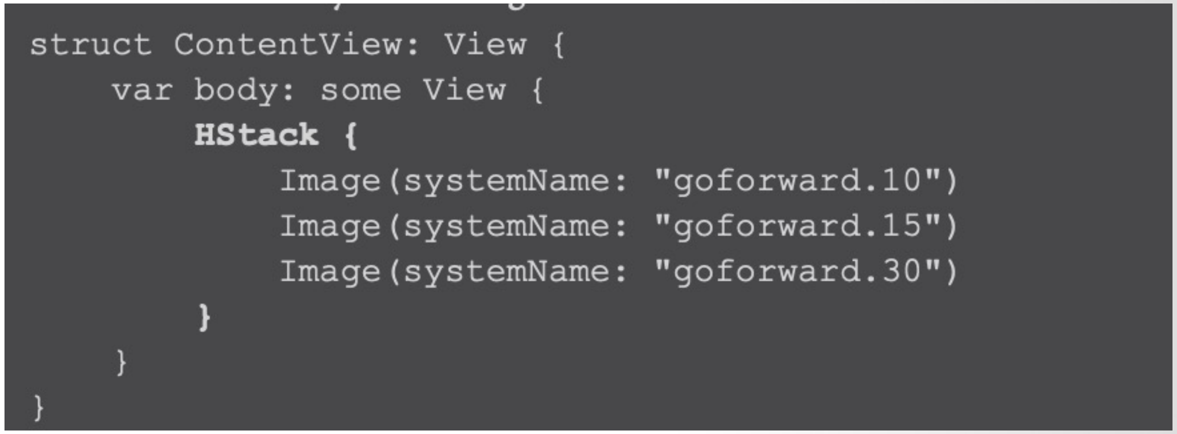
```
import SwiftUI

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

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

The view is named ContentView and is declared as conforming to the View protocol. It also includes the mandatory body property which, in turn contains an instance of the built-in Text view component which is initialized with a string which reads “Hello, world!”.

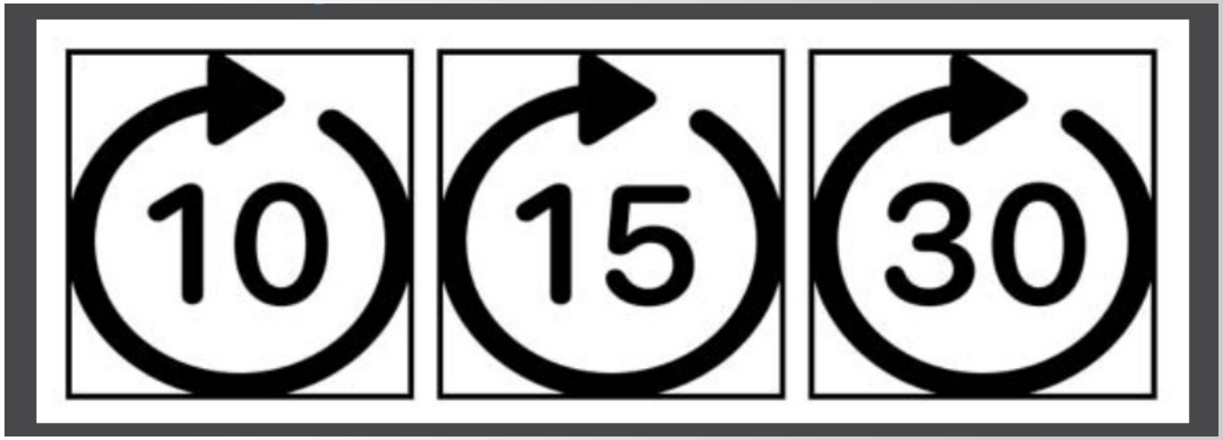
The second structure in the file is needed to **create an instance** of ContentView so that it **appears in the preview canvas**, a topic which will be covered in detail in later chapters.”

SwiftUI Stacks

- SwiftUI includes three stack layout views in the form of VStack (vertical), HStack (horizontal) and ZStack (views are layered on top of each other).

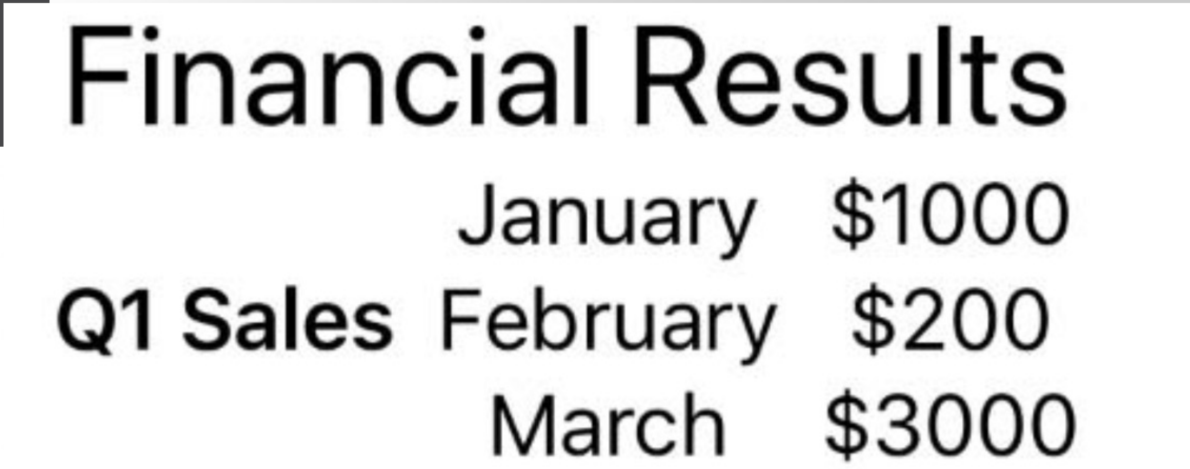


```
struct ContentView: View {
    var body: some View {
        HStack {
            Image(systemName: "goforward.10")
            Image(systemName: "goforward.15")
            Image(systemName: "goforward.30")
        }
    }
}
```



SwiftUI Stacks

```
VStack {
    Text("Financial Results")
        .font(.title)

    HStack {
        Text("Q1 Sales")
            .font(.headline)

        VStack {
            Text("January")
            Text("February")
            Text("March")
        }

        VStack {
            Text("$1000")
            Text("$200")
            Text("$3000")
        }
    }
}
```


Financial Results

	January	$1000
Q1 Sales	February	$200
	March	$3000

Spacers, Alignment and Padding

- To add space between views, SwiftUI includes the Spacer component. When used in a stack layout, the spacer will flexibly expand and contract along the axis of the containing stack (in other words either horizontally or vertically) to provide a gap between views positioned on either side, for example:

```
HStack(alignment: .top) {

    Text("Q1 Sales")
        .font(.headline)
    Spacer()
    VStack(alignment: .leading) {
        Text("January")
        Text("February")
        Text("March")
    }
    Spacer()
```

Spacers, Alignment and Padding

- In terms of aligning the content of a stack, this can be achieved by specifying an alignment value when the stack is declared, for example:

```
VStack(alignment: .center) {
            Text("Financial Results")
                .font(.title)
```

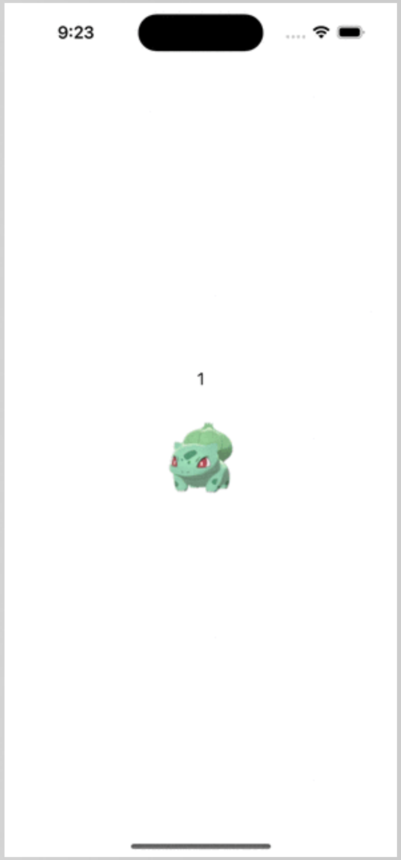
- Alignments may also be specified with a corresponding spacing value:

```
VStack(alignment: .center, spacing: 15) {
            Text("Financial Results")
                .font(.title)
```

Exercise: Build an App to display Pokemons (Pokédex version 1)

Exercise: Create a Pokémon Display App (Pokédex Version 1)
Objective:
In this exercise, you will build an iOS app that allows users to view Pokémon images based on their ID numbers.
Instructions:
1. **Download Resources**: Download the Pokemon.zip file from the class website.
2. **Unzip the File**: Extract the contents of the Pokemon.zip file to access the Pokémon profile pictures.
3. **Add to Xcode**: Import the unzipped Pokémon profile pictures into your Xcode project.
4. **App Development**: Create an app that performs the following functions:
    1. Allows users to input a Pokémon ID number.
    2. Displays the corresponding Pokémon image based on the entered ID.

By the end of this exercise, you should have a functional app that serves as a basic Pokédex, allowing users to view Pokémon based on their ID numbers.



Bonus: Display an extra rare star (rare.png) for pokemon with ID 150 and 151

Q & A