SwiftUI

UIKit

• You’ve been using UIKit for everything
  – contains UIButton, UIView, UILabel, etc

• UIKit is imperative – you use commands to add, remove, modify interface

• What are common issues with UIKit?
  – Storyboards don’t work with Git and have hidden code
  – Programmatic interface gets messy
Introducing SwiftUI

- **Declarative** – code succinctly defines the interface
- **You can still drag in views using the + button**

- **No more storyboards**
  - Instead, there is side-by-side code and preview
  - Editing one changes the other as well
  - Good for Git!

What does it look like?

- **A list of views inside the body view**
- **Simple views**
  - Text
  - Image
  - Button
  - Color
- **Use Stacks to combine views**

```swift
struct ContentView: View {
    var body: some View {
        VStack {
            Text("Hello, World!")
                .font(.title)
            Image("washu")
                .resizable() 
                .scaledToFit() 
                .frame(width: 200, height: 200)
            Text("This is a subtitle")
                .font(.caption)
        }
    }
}
```
A closer look at a view

- View with optional parameters
  - Here it's the name of the image
- A list of modifiers
  - Just functions that return a modified view
  - Modifiers change properties like font, size, background color, etc

```swift
Image("washu")
  .resizable()
  .scaledToFit()
  .frame(width: 200, height: 200)
```

View composition

- Many views can take more views as children
  - Ex Stacks, Views, Buttons
- Simply list the views that are children and they will be added in the correct order

```swift
Button(action: {}) {
  Text("I am a button")
  Image(systemName: "plus")
}
```
HANDLING DATA – STATE AND BINDINGS

State

• A state variable is just a property, marked with @State

• When the variable changes value, anything that uses the variable will refresh
  – SwiftUI automatically determines what in a view needs to be refreshed, so don’t be afraid to hinder performance with state changes

• A one-way relationship

```swift
struct ContentView: View {
    @State var count: Int = 0

    var body: some View {
        VStack {
            Text("The count is \(count)")
            Button(action: {
                self.count += 1
            }){
                Text("Increase count")
                .padding()
            }
        }
    }
}
```
LISTS

• Replace UITableViews

• Automatically handles most of the boilerplate
  – no more datasource methods

• A List is tied to an array
  • When the array changes, the list changes
List Demo

- **Notice a few things:**
  - Animation happens for free
  - List requires a way to identify each item so it can determine when to update – here we use “.self” to identify strings by their own value
    - With a more complex object, you might use an id or name field

- And most importantly...
- **The list is always true to the data object it represents!**
  - No more `tableView.reloadData()`

Bindings

- We have covered going from data->view, but what about the other way around?

- **A binding creates a two-way connection between a variable and a view object**
  - If one changes, the other does as well

- Define using ‘$’ before a variable name

- Useful for:
  - Toggles
  - Textfields
  - DatePickers
  - and more
Binding demo

```swift
struct ContentView: View {
    @State var imageShowing = false

    var body: some View {
        VStack(alignment: .leading, spacing: 20) {
            Toggle(isOn: $imageShowing.animation()) {
                Text("Show image")
            }
            if imageShowing {
                Image("washu")
                    .resizable()
                    .scaledToFit()
                Spacer()
            }
        }
    }
}
```

There’s the binding

Complex State

- A lot of times, your state will be more complex than a few variables
- The solution:
  - Make a Model object that conforms to `ObservableObject`
  - In the View, make an instance of the object
    - `@ObservedObject var scores = Scores()`
  - Use properties of the object
    - `Text(scores.currentScore)`
  - When the property of the object changes, the view will be updated
MVVM

- The previous example is an example of the Model-View-ViewModel pattern (as opposed to MVC)

- The key difference is the ViewModel
  - An object that turns the raw data into a form that’s easy for the view to use
  - Very simple example: An object that holds firstName and lastName but only exposes the property name to the view.

- One-directional ownership
  - View -> ViewModel -> Model

Navigation

- NavigationView – wraps any view that will navigate to other views
- NavigationLink – wraps a view that, when touched, will transition to a new view
  - Takes the destination, an instance of another view

- This is equivalent to a Show segue in UIKit
  - For a modal segue, look into the .sheet modifier

```swift
struct ContentView: View {
    var body: some View {
        NavigationView{
            NavigationLink(destination: ViewTwo()){
              Text("tap me")
            }
        }
    }
}
```
DEMO: REMAKING LAB 1 IN SWIFTUI

More to explore

• Universal interfaces
  – SwiftUI can use the same code to produce an iOS, macOS, and watchOS interface

• Declaring and using your own views
  – SwiftUI is meant to be extendable. You can make your own views, modifiers, etc

• Dark Mode
  – when you use system colors, you get dark mode support automatically

• SF Symbols
  – ready-to-use icons

• Combine
  – a complex framework for publishing/subscribing to messages (how Observable Object works)
Resources

  - The official documentation is pretty minimal at this point, so I highly recommend one of the ones below

- [https://www.hackingwithswift.com/](https://www.hackingwithswift.com/)
- [https://swiftui-lab.com](https://swiftui-lab.com)