SwiftUI State Management

Lecture 4

CIS 1951

Last week

SwiftUI Fundamentals

- What is SwiftUI
- Why do SwiftUI
- How to SwiftUI

Views! Modifiers! Lists!

Layouts! Scroll views!

- ***Any questions, comments, or feedback?***

State

State

Data that can change as the app runs

State Management

The process of **handling changes** to data to **ensure your UI reflects the current state** of the application

Makes your app **dynamic** and **interactive**!

Property Wrappers

Attributes that **add additional behavior** to properties

```
@PropertyWrapper var someValue = "Hello!"
```

Sidenote: *You can make your own!*

```
@propertyWrapper
struct PropertyWrapper<T> {
    var wrappedValue: T
}
```

@State

Declares a piece of state that the view owns

- Redraws when the value changes
- **Ownership:** View owns the data

@State

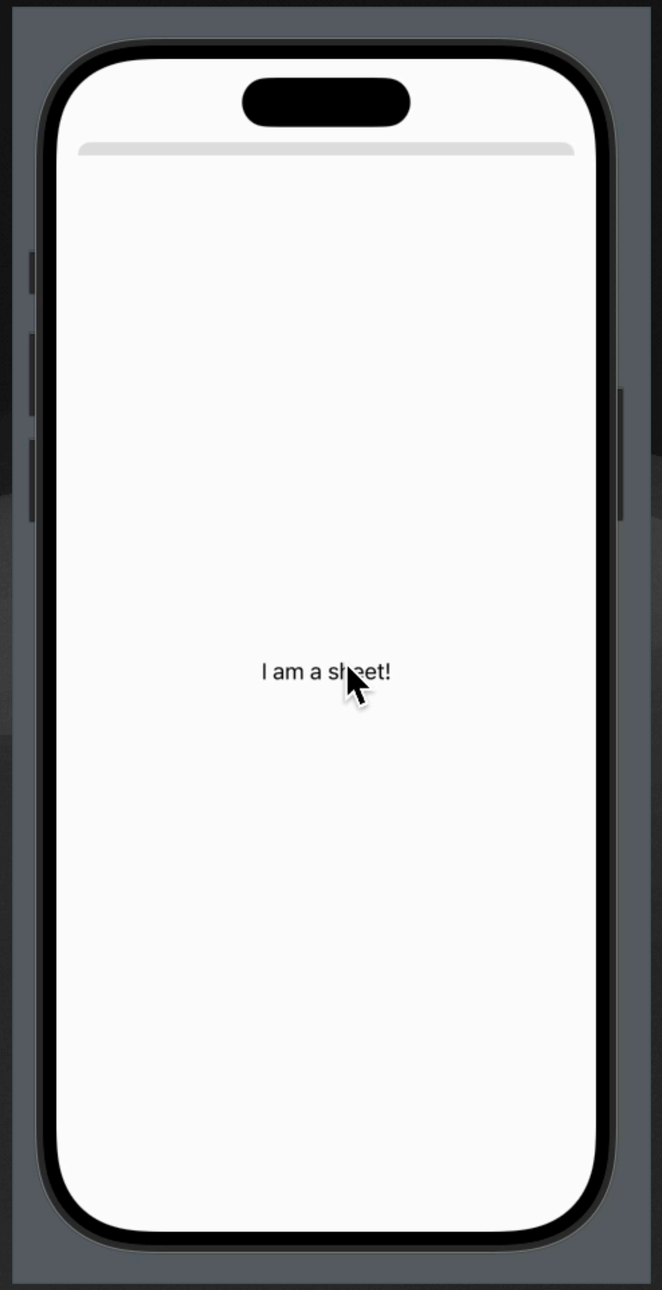
```
struct ContentView: View {
    @State var clicks = 0

    var body: some View {
        VStack {
            Text("\(clicks)")

            Button {
                clicks += 1
            } label: {
                Text("Click me")
            }
        }
        .padding()
    }
}
```

Toggle

```
struct ViewWithToggle: View {
    @State var isOn = false

    var body: some View {
        Toggle("Switch", isOn: $isOn)
            .fixedSize(horizontal: true,
                vertical: false)
    }
}
```

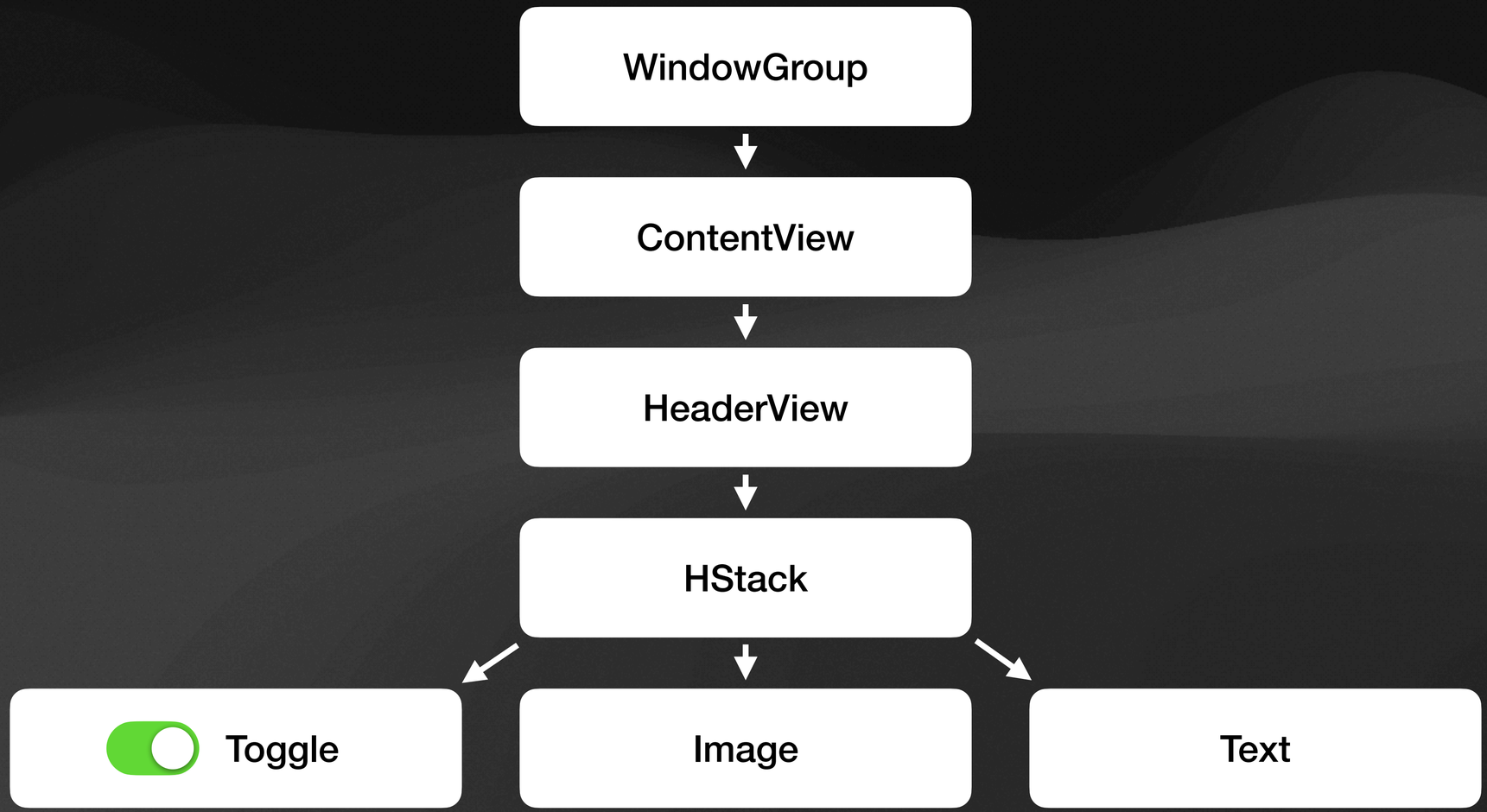
.sheet

```
struct ViewWithSheet: View {
    @State var isPresented = false

    var body: some View {
        Button {
            isPresented = true
        } label: {
            Text("Show sheet")
        }
        .sheet(isPresented: $isPresented) {
            Text("I am a sheet!")
        }
    }
}
```



View Hierarchy

View Hierarchy



https://velog.io/@mm723/SwiftUI

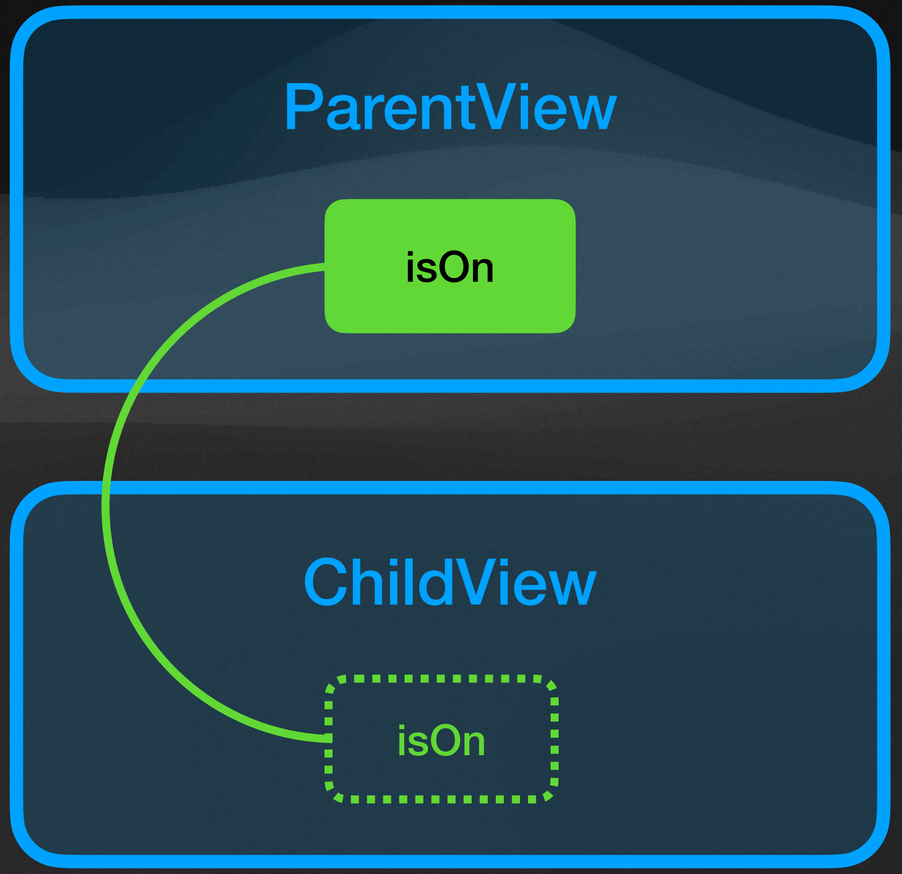
@Binding

Allows a view to mutate data owned by someone else

- Redraws when the underlying value changes
- Lets two views share the same state
  - e.g. letting a text field edit its parent's state
- **Ownership:** Someone else owns the data (e.g. a parent)

@Binding

```
struct ParentView: View {
    @State private var text = ""

    var body: some View {
        ChildView(text: $text)
    }
}

struct ChildView: View {
    @Binding var text: String

    var body: some View {
        TextField("Enter text", text: $text)
    }
}
```



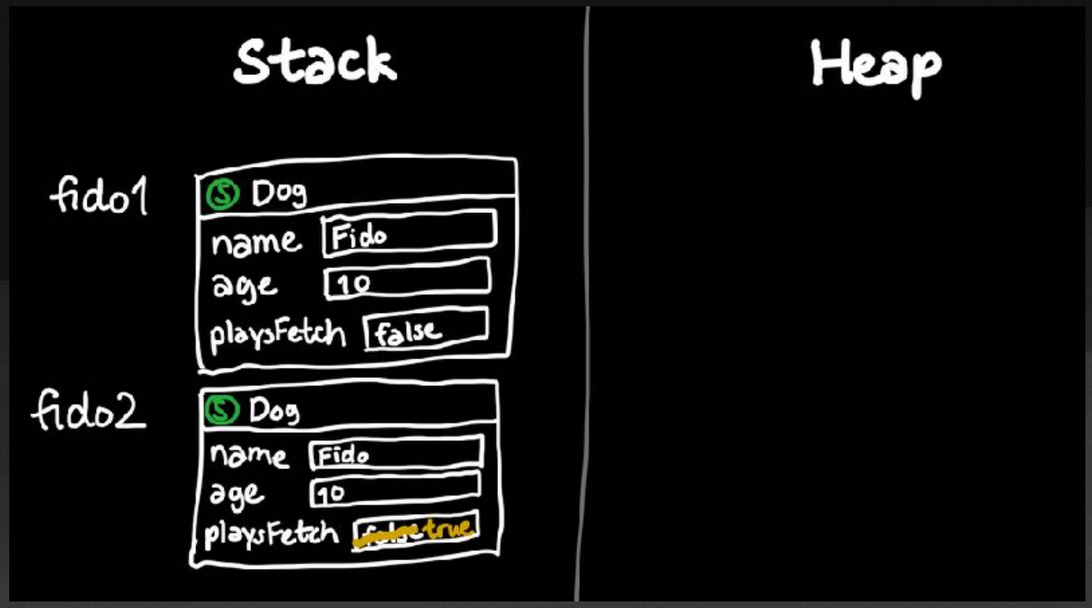
Sidenote

You only need to use @Binding when the child view needs to **edit** the data!

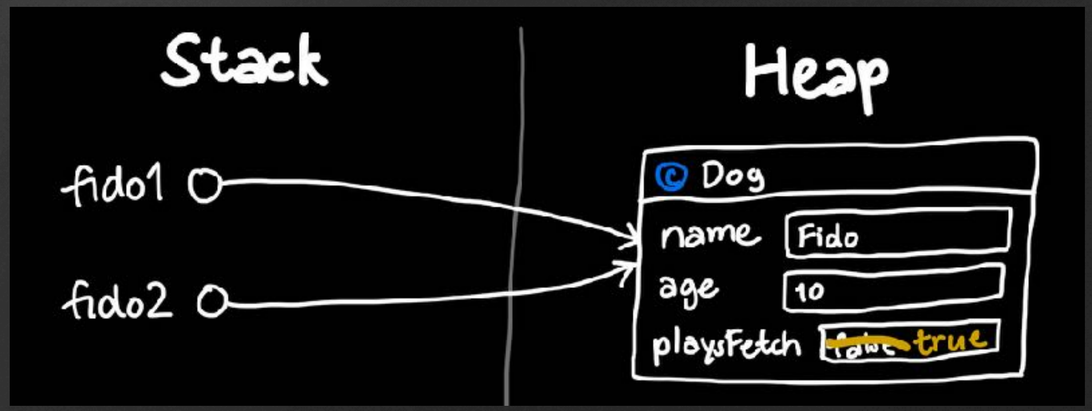
@Observable

Refresher

Structs



Classes



@Observable

When we want to use a class's property as state, we need to tag the class as **@Observable**

```
@Observable class MyViewModel
```

@Bindable

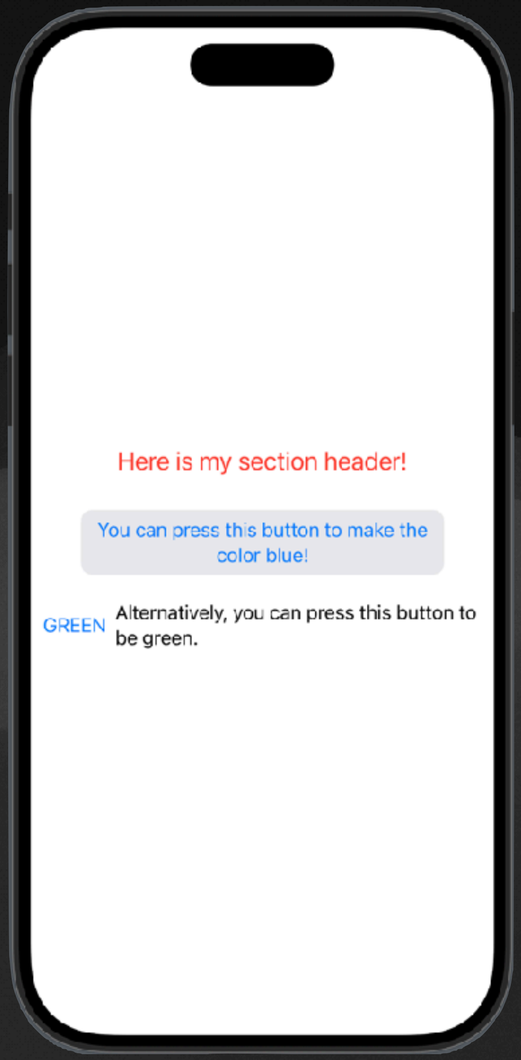
Bindings for Observables

- Similar to @Binding, we can pass a class marked with @Observable to a child view.
- Remember the relationship between @State (owns) and @Binding (allowed to change). The same relationship exists here.

ew!

```swift
@Observable
class MySettings: Identifiable {
    var color: Color = .red
    let internships = 0
}


struct PropDrilling: View {
    @State var settings = MySettings()


    var body: some View {
        SectionView(settings: settings)
    }
}
```

```swift
struct SectionView: View {
    @Bindable var settings: MySettings

    var body: some View {
        VStack {
            Text("Here is my section header!")
                .font(.title2)
                .foregroundStyle(settings.color)
            Button {
                settings.color = .blue
            } label: {
                Text("You can press this button to make the color blue!")
            }
            .buttonStyle(.bordered)
            .padding()
            SwitchView(settings: settings)
        }
    }
}

struct SwitchView: View {
    @Bindable var settings: MySettings

    var body: some View {
        HStack {
            Button {
                settings.color = .green
            } label: {
                Text("GREEN")
            }
            Text("Alternatively, you can press this button to be green.")
        }
    }
}
```



"prop drilling"

@Environment

Pass an object/property to **all** subviews

```
@Observable
class MySettings: Identifiable {
    var color: Color = .red
    let internships = 0
}


struct PropDrilling: View {
    @State var settings = MySettings()


    var body: some View {
        SectionView()
            .environment(settings)
    }
}
```

```
struct SectionView: View {
    // Will crash if not present
    @Environment(MySettings.self) var settings

    var body: some View {
        VStack {
            Text("Here is my section header!")
                .font(.title2)
                .foregroundStyle(settings.color)
            Button {
                settings.color = .blue
            } label: {
                Text("You can press this button to make the color blue!")
            }
                .buttonStyle(.bordered)
                .padding()
            SwitchView()
        }
    }
}
```

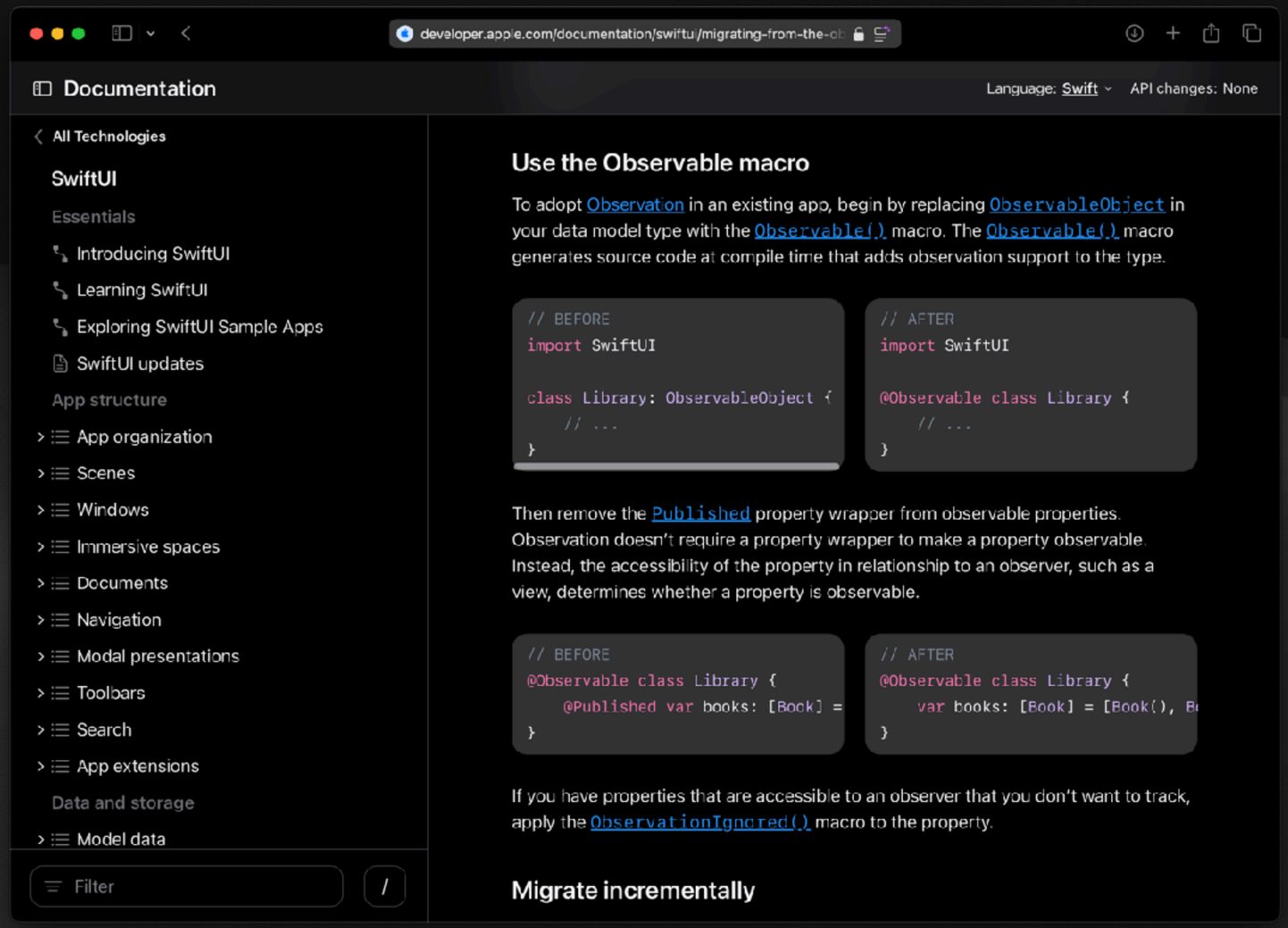
```
struct SwitchView: View {
    @Environment(MySettings.self) var settings

    var body: some View {
        HStack {
            Button {
                settings.color = .green
            } label: {
                Text("GREEN")
            }
            Text("Alternatively, you can press this button to be green.")
        }
    }
}
```

Note: using @Environment is a design decision. If you find yourself passing the same property/object to 2+ views (to be modified), consider using environment.

Brief aside

@ObservableObject

iOS 13-16



Useful Tools

.onAppear and .onDisappear

```
SomeGenericView()
.onAppear {
    print("I will be printed when the view
        appears!")
}
.onDisappear {
    print("I will be printed when the view
        disappears!")
}
```

.onChange

```
SomeGenericView()
    .onChange(of: clicks) {
        print("You have \(clicks) clicks!")
    }
```

Animations

Using withAnimation

```
Text("Loading")
    .opacity(isLoading ? 1 : 0)
```

or

```
Text("Loading")
    .transition(.opacity)
```

then:

```
withAnimation(.snappy) {
    isLoading = true
}
```

Live coding time!

https://github.com/cis1951/lec4-code

Conclusion

Today, we learned…

SwiftUI State Management

- View hierarchy
- State management
- Property wrappers
  - @State and @Binding
- @Observable
- .sheet, .onAppear, .onDisappear, .onChange
- Animations and transitions

Next time…

- Lecture 5: App Lifecycle and Structure